SPI Programming Manual

Abstract

This manual describes the operating system procedures that programmers call to process the Subsystem Programmatic Interface (SPI) messages. It presents conventions that regulate message content and interpretation, provides programming guidelines and examples, and describes the common ZSPI data definitions.

Product Version

SPI H02

Supported Release Version Updates (RVUs)

This publication supports J01 and all subsequent J-series RVUs and H02 and all subsequent H-series RVUs until otherwise indicated by its replacement publication.

Part Number	Published
427506-007	February 2012

Document History

Part Number	Product Version	Published
427506-003	SPI D40 and G05	September 2003
427506-004	SPI D40 and G05	December 2003
427506-005	SPI G05	August 2004
427506-006	SPI G05 and H01	February 2006
427506-007	SPI H02	February 2012

Legal Notices

Printed in the US

SPI Programming Manual

6. SPI Programming in C

7. SPI Programming in COBOL

8. SPI Programming in TACL

9. SPI Programming in TAL

Examples

Figures

Tables

What's New in This Manual

Manual Information

Abstract

This manual describes the operating system procedures that programmers call to process the Subsystem Programmatic Interface (SPI) messages. It presents conventions that regulate message content and interpretation, provides programming guidelines and examples, and describes the common ZSPI data definitions.

Product Version

SPI H02

Supported Release Version Updates (RVUs)

This publication supports J01 and all subsequent J-series RVUs and H02 and all subsequent H-series RVUs until otherwise indicated by its replacement publication.

Part Number	Published
427506-007	February 2012

Document History

Part Number	Product Version	Published
427506-003	SPI D40 and G05	September 2003
427506-004	SPI D40 and G05	December 2003
427506-005	SPI G05	August 2004
427506-006	SPI G05 and H01	February 2006
427506-007	SPI H02	February 2012

New and Changed Information

Changes in the 427506-007 Manual:

- Added Product Version SPI H02 to the title page.
- Added the Note on page 3-6
- Added the Note on page 3-9.

Changes in the 427506-006 Manual:

Added details of these subsystems in Table D-1, NonStop Kernel Subsystem Numbers, on page D-1 and Table D-2, NonStop Kernel Subsystem Abbreviations, on page D-12:

- 259, Constellation IP (CIP)
- 262, Fibre Channel Storage Monitor (FSM)
- 263, HP NonStop Operating System - Complex Manager Auxiliary Process (CMP)
- 264, XA Broker Subsystem (ZXA)
- 265, Java Logging Subsystem (L4J)
- 266, Matrix SMLC CBB 4.2 Subsystem (CPS)
- 267, Deadlock Detector Subsystem (DLD)

Changes in the G06.24 Manual:

Added details of subsystems, 260 and 261 in Table D-1, NonStop Kernel Subsystem Numbers, on page D-1.

About This Manual

This manual describes the Subsystem Programmatic Interface (SPI):

Section	Description
Section 1, Introduction to SPI	Provides an overview of SPI.
Section 2, SPI Concepts and Protocol	Presents topics central to understanding SPI.
Section 3, The SPI Procedures	Describes the syntax of the SPI procedures and the operations they perform.
Section 4, ZSPI Data Definitions	Provides descriptions of the ZSPI data definitions and the Data Definition Language (DDL) code used to define them.
Section 5, General SPI Programming Guidelines	Offers general programming guidelines for SPI requesters and servers.
Section 6, SPI Programming in C	Provides language-specific information for SPI programming in C.
Section 7, SPI Programming in COBOL	Provides language-specific information for SPI programming in COBOL.
Section 8, SPI Programming in TACL	Provides language-specific information for SPI programming in the HP Tandem Advanced Command Language (TACL), including descriptions of the built-ins that correspond to the SPI procedures.
Section 9, SPI Programming in TAL	Provides language-specific information for SPI programming in the HP Transaction Application Language (TAL)
Appendix A, Errors	Describes error numbers returned by the SPI procedures and other common errors encountered in SPI processing.
Appendix B, Summary of DDL for SPI	Summarizes features of the Data Definition Language for HP NonStop™ servers that are particularly relevant to SPI data definitions.
Appendix C, SPI Internal Structures	Describes the internal structure of common SPI data structures to facilitate debugging.
Appendix D, NonStop Kernel Subsystem Numbers and Abbreviations	Presents tables that list HP NonStop Kernel subsystems by their subsystem number and by their subsystem abbreviation.
Appendix E, SPI Programming Examples	Presents TAL and C source code for several working SPI programs.

Audience

This manual is written for those who maintain or develop programs that communicate using the SPI and for anyone interested in the contents of SPI messages. SPI is used

by many NonStop Kernel subsystems and by customer-developed applications that manage these subsystems.

Related Manuals

For D-series and G-series RVUs, SPI is part of the Distributed Systems Management (DSM) architecture. For more information, see the *Distributed Systems Management (DSM) Manual*.

If the subsystems you are managing expect requests to be routed through an SCP process, read the *Subsystem Control Point (SCP) Management Programming Manual*.

Many NonStop Kernel subsystems that use SPI also implement the SPI extensions described in the *SPI Common Extensions Manual*.

For any specific subsystems you are working with, you can also see the management programming manual.

Information about the language used to define tokens and related data elements is in the *Data Definition Language (DDL) Reference Manual*.

Notation Conventions

General Syntax Notation

This list summarizes the notation conventions for syntax presentation in this manual:

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

```
MAXATTACH
```

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

```
file-name
```

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

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

INT[ERRUPTS]
```

A group of items enclosed in brackets is a list from which you can choose one item or none. The items in the list may be arranged either vertically, with aligned brackets on

each side of the list, or horizontally, enclosed in a pair of brackets and separated by vertical lines. For example:

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

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

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

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

ALLOWSU { ON | OFF }
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Notation for Messages

This list summarizes the notation conventions for the presentation of displayed messages in this manual:

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

```
Backup Up.
```

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

```
p-register

process-name
```

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

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

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

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

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

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

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

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

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

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

```
%005400
```

```
%B101111
```

```
%H2F
```

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

Notation for Management Programming Interfaces

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

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

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

```
token-type
```

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

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

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

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

Change Bar Notation

Change bars are used to indicate substantive differences between this manual and its preceding version. 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 COBOL 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.

HP Encourages Your Comments

HP encourages your comments concerning this document. We are committed to providing documentation that meets your needs. Send any errors found, suggestions for improvement, or compliments to docsfeedback@hp.com.

Include the document title, part number, and any comment, error found, or suggestion for improvement you have concerning this document.

1 Introduction to SPI

This section provides an overview of the Subsystem Programmatic Interface:

The Components of SPI

The Subsystem Programmatic Interface (SPI), a central component of the Distributed System Management (DSM) architecture, is the path that management applications and subsystems use to exchange command-response messages and event messages.

A Programmatic Interface

HP provides interactive text interfaces to most of its subsystems for the HP NonStop Operating System. TMFCOM, PATHCOM, and the Subsystem Control Facility (SCF) are examples. These text-based interfaces provide needed information in a form readable by human operators on terminals, printers, and other display devices. When a program communicates with a human operator, it is appropriate for the communication to be based on native-language text. However, text-based messages are not well suited for processing by a program. Programs can be written to parse text strings (and many programs process messages intended to be read by an operator), but it is more efficient for them to process messages containing data in readily processed representations.

SPI

SPI is a programmatic interface designed to facilitate management of subsystems without operator intervention:

- Programmers do not need to be concerned with the internals of message format but can concentrate instead on requesting and retrieving relevant data.
- Subsystems can add features without changing the basic interface—existing applications interact successfully with new versions of subsystems.
- A programmer who uses the interface in one programming language to communicate with one subsystem then knows most of what is necessary to use the interface in other languages and with other subsystems.
- Application programs can use any interprocess communication features (such as nowait or waited I/O) that are available in the language in which they are written.

SPI consists of:

- A standard message format
- A standard message protocol
- A standard unit of information: the token
- Procedures for composing and decoding messages
- Data definitions for commonly used data structures
- Rules and guidelines governing message content and protocol

SPI and EMS

The Event Management Service (EMS) is based on SPI, and EMS messages are one of two types of SPI messages (the other being control and inquiry messages). This manual concentrates on the control-and-inquiry role of SPI. For information about event messages, see the *EMS Manual*.

Subsystem Objects

An object is a well-defined logical or physical entity such as a device, communications line, logical subdevice, process, processor, file, or transaction. Most objects are controlled by subsystems, and a subsystem itself can be treated as an object. SPI is designed to allow programmatic management of subsystem objects.

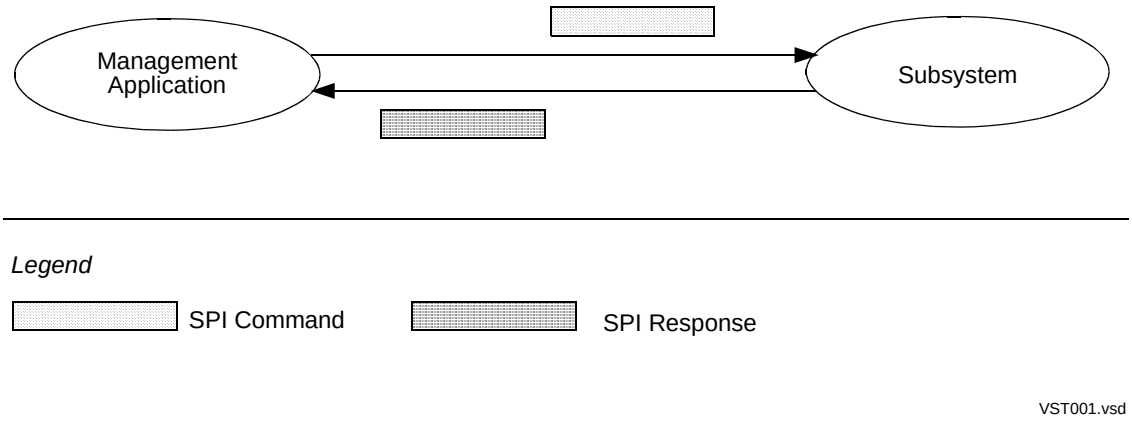
Management Applications

Management applications configure, control, monitor, and report the status of subsystem objects. The primary tasks of a management application differ from those of other applications. Rather than using a subsystem's basic services, a management application monitors and controls the subsystem itself. Because such management tasks can be complex, repetitive, or time-consuming, they lend themselves to programmatic solutions. Hence, the value of management applications, which can be designed to manage individual subsystem objects or entire subsystems.

Figure 1-1 on page 1-3 shows a management application communicating with a subsystem that consists of a single process.

Figure 1-1. SPI Communication With a Single-Process Subsystem

A subsystem might consist of a single process, in which case this process both provides the subsystem services and serves as the SPI subsystem manager.



Management applications use SPI to:

- Start or stop an object (such as a Pathway terminal)
- Change an object attribute value (such as the speed of a communications line)
- Add a new object to the system (such as defining a logical subdevice on a communications line)
- Inquire whether an object is stopped or running

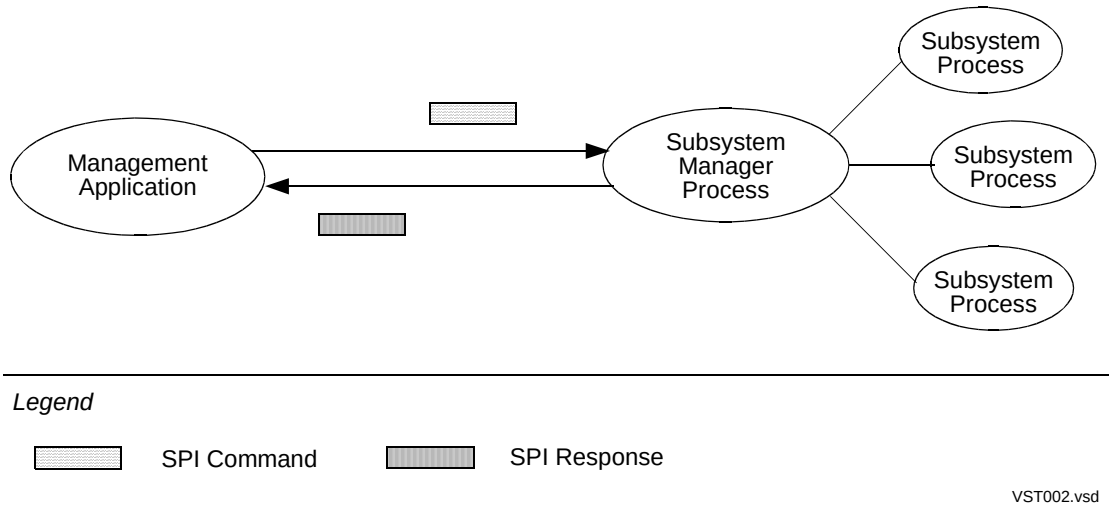
Subsystem Manager

Every subsystem that has an SPI control-inquiry interface includes a process that is responsible for supporting the interface. This process is called the subsystem manager process or subsystem manager. Management applications send SPI commands to this process and receive responses from it.

In Figure 1-1, a single subsystem process provides subsystem services and acts as subsystem manager. Figure 1-2 on page 1-4 shows a management application communicating with the subsystem manager of a multiprocess subsystem. Communication between the management application and the subsystem manager follows the SPI protocol. Communication between the subsystem manager and the other subsystem processes can follow any protocol (including SPI) selected by the subsystem.

Figure 1-2. SPI Communication With a Multiprocess Subsystem

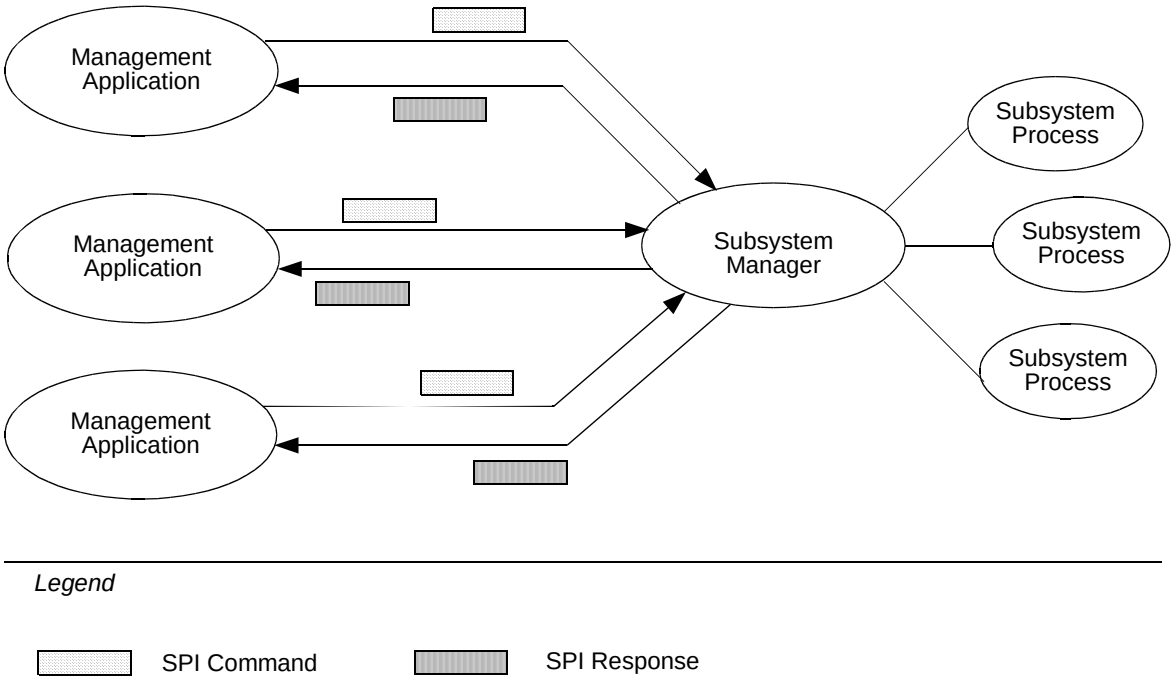
A subsystem might consist of several processes. One of these, the manager process, supports the SPI interface to management applications.



Multiple management applications can communicate with a single subsystem. Figure 1-3 on page 1-5 shows several management applications communicating with the manager process of a multiprocess subsystem.

Figure 1-3. Subsystem Manager Communicating With Multiple Management Applications

A subsystem manager can support concurrent SPI communications with multiple management applications.



A subsystem does not need to communicate directly with management applications. Instead, a subsystem can have its commands and responses routed through an intermediate process, as shown in Figure 1-4 on page 1-6. The SPI common extensions provide such a process: the Subsystem Control Point. (See the *SPI Common Extensions Manual*.)

In any configuration, SPI provides the management interface between the management application, the subsystem manager process, and any intermediate process.

SPI Message Protocol

The basic SPI protocol specifies:

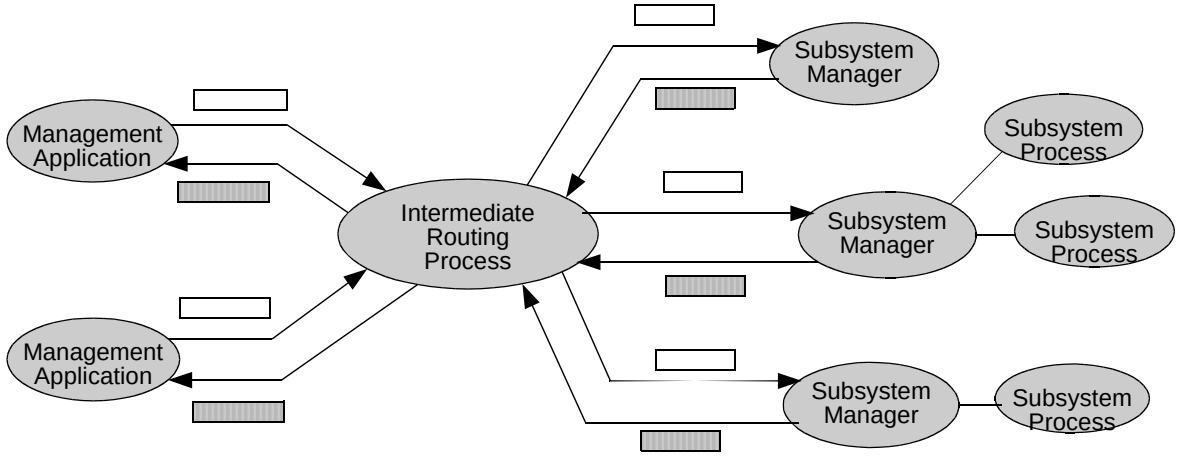
- How SPI requesters and servers process messages
- What information must appear in a message
- Whether or not that information must be sent in a specific token

- How requesters and servers should respond to error conditions

Communication between a management application and a subsystem follows the standard HP requester-server model, with the management application in the role of requester and the subsystem manager process in the role of server.

Figure 1-4. SPI Communication Through an Intermediate Process

A subsystem can have its commands and responses routed through an intermediate process.



Legend

SPI Command SPI Response

VST004.vsd

SPI is a general interface governed by many general rules and guidelines. This manual describes the basic SPI protocol, the SPI procedures used to build and decode messages, and the common data definitions used in SPI messages. An SPI interface to a particular subsystem is usually based on a very specific implementation of these generalities. For detailed information about the SPI interface to a particular subsystem, see the management programming manual for that subsystem.

After composing an SPI message, an application (requester) uses standard interprocess communication to send the message to the subsystem manager (server).

SPI Message Format

SPI messages have a common structure. Each message consists of a message header followed by as many tokens as are necessary to convey information relevant to the message. (The number of tokens is limited by the size of the message buffer.)

SPI messages are built in buffers initialized by the SSINIT procedure. The messages contain tokens (and are sometimes called tokenized messages). The interface itself is described as token-based. Programs use SPI procedures to format message buffers, assign values to tokens, and put tokens into and retrieve tokens from message buffers. An application working with an SPI message refers to a token by its symbolic name, and does not need to be concerned with the address of the token in the message buffer.

Tokens

Tokens are self-identifying data items; a token's name indicates the function of the token or identifies the information that the token contains. A typical token in an SPI message carries with it an identifying number, the data type of its value, the length of its value, and the value itself. Some of the SPI guidelines govern how to name tokens.

Most tokens have an associated value. Programs use the SSPUT procedure to assign a values to tokens and put the tokens into a message buffer.

The four basic token types (described in Section 2, SPI Concepts and Protocol) are:

Simple	Value can be a simple data item or a data structure. In either case, the type and structure of the value are fixed for the life of the token.
Extensible structured	Can be extended by adding fields in new releases of the server.
Zero-length	Has no associated value.
Header	Appears in every SPI message; has values but no corresponding token code in the buffer.

SPI Procedures

The five basic SPI procedures used to compose and decode SPI messages are:

SSINIT	Creates a new SPI message and assigns values to selected header tokens.
SSNULL	Initializes the fields of an extensible structure to null values.
SSPUT	Either places a token in a message buffer or performs a special operation on a buffer, depending on the token specified in the procedure call.
SSGET	Either extracts a token from a message or retrieves information about a message, depending on the token specified in the procedure call.
SSMOVE	Copies a token or group of tokens from one message buffer to another.

Three related procedures, SSPUTTKN, SSGETTKN, and SSMOVETKN, let tokens be specified in the procedure call by value rather than by reference.

Similarly named built-ins allow access to the SPI procedures from TACL.

The SPI procedures to format and decode messages are available:

- From C using the **tal** interface declaration
- From COBOL using the ENTER TAL construct
- From TACL using special built-in functions
- From TAL

SPI Data Definitions

Tokens, values, command numbers, message headers, and error numbers are some of the SPI-defined data items that are used to build SPI messages. These items are defined using the DDL and are provided to programmers in definition files with names beginning with ZSPI. A definition file specifically suited to each programming language is supported by SPI:

Programming Language	Definition File
C	ZSPIC
COBOL	ZSPICOB
TACL	ZSPITACL
TAL	SZPITAL

You can use these predefined items by adding the definitions from the appropriate definition file to their source code. In the case of TACL, load the SPI TACL definitions or attach the ZSPISEGF segment file.

NonStop Kernel subsystems supply additional definitions in files with names beginning with Z*sss*, where *sss* is a subsystem abbreviation. (See Appendix D, NonStop Kernel Subsystem Numbers and Abbreviations.) All items defined in these files also have names beginning with Z*sss*, where *sss* is the subsystem abbreviation. User-defined definitions begin with a letter other than *Z* to avoid possible conflicts with HP definitions.

SPI data definitions are developed using the DDL. See Appendix B, Summary of DDL for SPI. For a complete description of DDL, see the *Data Definition Language (DDL) Reference Manual*.

SPI and EMS

The event messages created using the Event Management Service (EMS) are a form of SPI message, and the EMS procedures with which event messages are manipulated are based on the SPI procedures. The *EMS Manual* describes event messages and EMS.

SPI Debugging

Inspect, a HP debugging tool, supports two methods of displaying the contents of SPI messages. See the *Inspect Manual* and the *DSM Template Services Manual*.

A Basic Interface and Extensions

This manual describes the basic SPI interface. Many NonStop Kernel subsystems are based on an extended version of SPI. These extensions, based on ZCOM data definitions, and a common implementation of the extensions, based on ZCMK definitions, are described in the *SPI Common Extensions Manual*.

Table 1-1. Comparison: SPI Basic and Extended Features (page 1 of 2)

	Basic SPI	Extended SPI
Procedures	Provides procedures for creating and decoding SPI messages.	Uses the basic SPI procedures.
Data definitions	Defines basic message structures and the data types on which all tokens are based (ZSPI).	Defines many additional tokens, structures, error numbers, event numbers, and other data items (ZCOM).
Commands	Defines a single command: GETVERSION.	Defines over 30 additional commands, and specifies the contents of the command and response messages.
Errors	Defines the basic mechanism for reporting errors, defines errors returned by the SPI procedures, and defines some common errors returned in SPI responses.	Using the basic SPI error reporting mechanism, defines common errors reported by subsystems under specific circumstances and specifies the contents of the associated error lists.
Events	Provides the basic mechanism for event messages (event messages are one type of SPI message).	Using the basic SPI event reporting mechanism, defines common events to be generated by subsystems under specific circumstances.
Object names	Discusses object names in general terms.	Makes specific recommendations for naming objects, and provides specific tokens for conveying object names in messages.
Security	Provides a basic buffer checksum feature for detecting corrupted messages.	In addition to the checksum feature, regulates command security based on requester authorization and the distinction between sensitive and nonsensitive commands.
General protocol	Defines basic message protocol, but leaves many implementation decisions for the subsystem.	Standardizes many behaviors that basic SPI leaves up to the subsystem.

Table 1-1. Comparison: SPI Basic and Extended Features (page 2 of 2)

	Basic SPI	Extended SPI
Message routing	Does not regulate message routing.	Provides a message routing process (SCP) that also provides security and other features.
Compatibility	Provides basic mechanisms for determining versions of requesters, servers, and the definitions they use.	The SCP process performs some reconciliation of incompatibilities between different versions of the extended interface.
Manual	*SPI Programming Manual*	*SPI Common Extensions Manual*

The extended SPI interface is based on and assumes an understanding of the features of the basic SPI interface described in this manual.

The Environment for D-Series RVUs

For D-series RVUs, the features of the NonStop operating system prompted several changes relevant to management applications. The operating system incorporates numerous changes intended to improve CPU utilization and allow larger I/O configurations. In part, these improvements were brought about by raising the limits on a number of system parameters to allow:

- More concurrent processes per CPU
- More I/O devices per node
- More I/O subdevices per device
- More opens (concurrent access paths) per device and subdevice
- More pending operations (outstanding messages)

These changes require the introduction of new and expanded data structures, including:

- New process identifiers
- A new file name format
- New SPI and EMS tokens corresponding to these new data structures

These structures are supported by a set of system procedures and associated error numbers and error lists.

Implications for Management Applications

In general, an application on a node running a C-series RVU runs without modification at a low PIN on a node running a D-series RVU. In particular, if all the processes involved are running at low PINs, management applications currently on a node running a C-series RVU should run identically on a node running a D-series RVU, or

on a node running a C-series RVU in a network with a node running a D-series RVU. The SCP and subsystem processes that communicate with an unconverted management application should be run at low PINs. This approach provides the best way, other than converting the application, to avoid the problems described here.

Obsolete Tokens in Responses

Even if a converted subsystem returns new tokens to convey data from a D-series RVU, it continues to include the same information in previously used tokens from C-series RVUs if the data fits in the old tokens. If the returned values do not exceed the range supported by the old tokens, an unconverted requester sees no change. (The requester can ignore the additional new tokens included in the response.)

However, if a converted subsystem cannot fit a value in a token from a C-series RVU, it omits the token from the response and stores the value in a new token for the D-series RVU. The subsystem does not return an error. The requester does not discover the problem until it encounters the error ZSPI-ERR-MISTKN when it tries to get the old token from the buffer.

For example, a subsystem that previously returned PIN values in an 8-bit token must now include a larger token to accommodate PINs greater than 255. When the subsystem deals with low-PIN processes, it can continue to use the 8-bit token in addition to the new one. But if it has to return a high PIN, the 8-bit token is inadequate and is not included in the response.

Obsolete Fields in Structured Tokens

Similar considerations apply to the fields of structured tokens. If data from a D-series RVU fits in a field for a C-series RVU, the subsystem returns the field for the C-series RVU in the structured token.

If the data does not fit in the field for the C-series RVU, and the field has a defined null value, the null value is returned. If the token is an extensible structured token, the value might be returned in a new field appended to the structure. Otherwise, the value is returned in a new simple token.

If the data from a D-series RVU does not fit in the old field, and no null value is defined, the entire structured token is omitted from the response. A new structured token with expanded fields is returned in its place.

For information about how a specific subsystem adapts to the system running the D-series RVU, see the management programming manual for the subsystem in question.

△ **Caution.** Subsystems do not return an error to indicate that a token from a C-series RVU has been omitted from a response.

Interprocess Communications Restrictions

These restrictions apply to communication among processes on a node running a D-series RVU or among processes in a mixed network of C-series and D-series RVUs and are relevant to all instances of SPI requester-server communications:

- No process on a node running a C-series RVU can be opened by a high-PIN process on a node running a D-series RVU. For example, a management application running at a high PIN cannot open an SCP process on a node running a C-series RVU, nor can an SCP running at a high PIN open a subsystem manager process on a node running a C-series RVU.
- A high-PIN process on a node running a D-series RVU can open an unconverted low-PIN process on a node running a D-series RVU if the low-PIN process has been recompiled to set the HIGHREQUESTERS object-file attribute.

For more information about these restrictions and how to overcome them, see the *Guardian Application Conversion Guide*.

Converting Management Applications

Existing management applications do not need to be converted in order to run on a system running a D-series RVU or in a mixed network of C-series and D-series RVUs if all related processes run at low PINs. (Related processes include the management application, the managed subsystem processes, and any intermediate process such as SCP.)

However, if you decide to convert a management application in order to take advantage of the new features of the operating system for D-series RVUs:

1. Consult the *Guardian Application Conversion Guide* for an overview of conversion and its benefits. This guide provides help with the details of the conversion process.
2. Modify your management application to use any new tokens for D-series RVUs described in the management programming manual for the managed subsystem.
3. Convert the sections of your application that process event messages. (See the *EMS Manual.*)

2 SPI Concepts and Protocol

This section defines the basic concepts on which the Subsystem Programmatic Interface (SPI) is based:

SPI Message Protocol

Communication between a management application and a subsystem follows the standard HP requester-server model, with the management application in the role of requester and the subsystem manager process in the role of server. The requester is an application program that can run as a stand-alone process or a process pair. The subsystem manager process is the server process that accepts SPI requests and prepares SPI responses.

This subsection overviews the protocol followed by SPI requesters and servers when exchanging messages. Details are presented in the remainder of this manual.

Requester Initializes a Buffer

The requester calls the SPI procedure SSINIT to initialize a message buffer in which it composes a command message. The message contains the subsystem ID (SSID) of the target subsystem and is at least as large as the buffer size recommended by the subsystem. (Both the SSID and a recommended buffer size are provided by the subsystem as part of its SPI data definitions.)

Requester Composes a Command Message

The requester uses the SSPUT procedure to add tokens and values to the message. Before adding an extensible structured token to the message, the requester calls the SSNULL procedure to initialize the structure, then places values in the fields of the structure, and finally adds the structured token to the message using SSPUT.

An entire command must always be sent in one command message. SPI does not support commands continued across multiple messages, nor does it support multiple commands per message. However, a command can be applied to multiple objects.

If the requester needs to perform the same command on several objects, the subsystem might accept commands with multiple object-name tokens, each with a different object name as its value, or it might accept special forms of object names (such as object-name templates that contain wild cards) that specify a collection of objects. Not all subsystems support these features, and each subsystem has its own rules for acceptable object name forms.

Requester Sends a Command Message

The requester then uses a language-dependent mechanism, usually involving the file system, to send the command and receive the response. In TAL, for example, a requester program calls the WRITEREAD procedure.

Server Validates the Received Message

When it receives the command message, the server resets the buffer and then verifies that the message is an SPI command message, that it is long enough to be a valid message but not so long that it overflows the read buffer, and that the used portion of the buffer is not longer than the buffer length recorded in the message header. The server then verifies that it is the intended recipient of the message by checking for its SSID in the message header, and checks that no field in an extensible structured token has a version greater than its own. Finally, the server examines the buffer’s contents for missing, required, invalid, and unrecognized tokens and token values.

Server Applies the Command to Objects

If the message is a valid command, the server locates the target object, applies the command to that object, and composes a response record describing the outcome of that processing. If the command is directed to more than one object, the server verifies that there is room in the buffer for another response record before it applies the command to the next object.

Server Composes a Response Message

The response record contains at least a return code token with a value that summarizes the outcome of processing for the object. The response record might also contain additional tokens and, if there was a problem, one or more error lists.

If the command is directed to more than one object, the subsystem returns its response information in multiple response records, each containing all the information that results from performing the command on one object. If not all of the response records fit in the buffer, the server stops processing objects when it runs out of buffer space and adds a context token to tell the requester that not all objects were

processed. The server and requester then follow the response continuation protocol in Continued Responses on page 2-34.

By default, the subsystem returns a single response record per response message. However, if requested, many NonStop Kernel subsystems return multiple response records per response message. The requester sets the value of a token in the command buffer to indicate the maximum number of response records per response the requester is willing to accept, or to ask for as many response records as will fit in the buffer. The subsystem then determines how many response records per response it will send, up to the maximum number specified by the requester.

As in the case of a single response record per response, the subsystem returns a context token in the response if not all response records for the command fit into a single response message. The requester asks for the next group of response records by resending a copy of the original command that includes this context token.

Server Returns the Response Message

The server returns the response message to the requester. If the server is context-free, it retains no information about the processing it performed, and from the server's perspective, this response completes the requester-server interaction. If not all objects were processed (because of lack of buffer space, for example), the server stores enough information in the context token to resume processing with the next object based on the information returned in the context token and a copy of the original command.

Requester Examines the Response

Upon receiving the response, the requester resets the buffer using the special SSPUT operation ZSPI-TKN-RESET-BUFFER to avoid problems arising from differing requester and server buffer sizes.

The requester checks the value of the return code in each response record. If an error occurs, the response code is nonzero and the response record contains an error list with an error token containing the same nonzero error number.

If the response contains a context token, the requester can continue processing by copying the context token to a copy of the original command and returning it to the server, which then resumes processing.

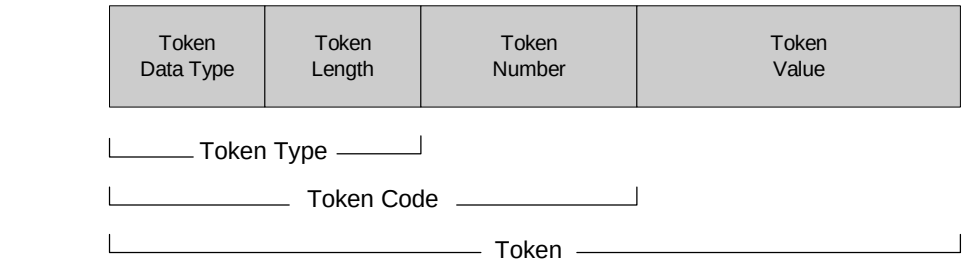
Tokens

All information in SPI messages is in the form of tokens and their values. Aside from error checking and message validation, most of the SPI-related processing performed by requesters and servers involves placing tokens in messages and retrieving token values from messages. The primary function of the SPI procedures is to manipulate tokens: the SSPUT procedure adds tokens to a message, SSGET retrieves token values from a message, SSMOVE copies tokens from one message buffer to another, and SSNULL initializes token values.

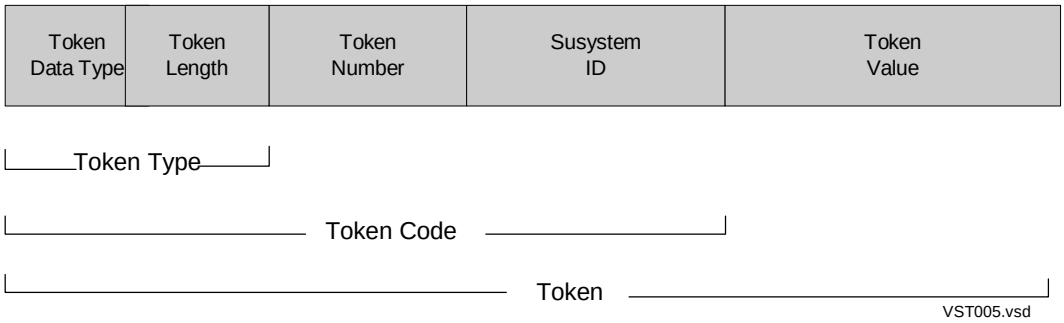
Tokens are self-describing data items; a typical token in an SPI message carries with it an identifying number, the data type of its value, the length of its value, and the value. Token number, data type, and length are known collectively as the token code, and a token is often viewed as consisting of two parts: a token code and a token value. The token data type and token length are known collectively as the token type. Token codes contain a subsystem ID.

Figure 2-1. The Basic Components of a Token

The components of an unqualified token:



The components of an SSID-qualified token:



Token Data Type

The token data type is the fundamental data type of the token's value. All token types are based on the token data types defined by SPI. Subsystems can define their own token types, but those types must be based on the ZSPI-defined token data types. Token data types have symbolic names of the form ZSPI-TDT-*desc*.

Token Length

For fixed-length token values up to 254 bytes long, the token length field contains the length of the value in bytes. In these cases, the token length is always a multiple of the length of the fundamental token data type. For token data type *x* and token length *y*, the value consists of as many items of token data type *x* as will fit within length *y*. For instance, a token of token data type ZSPI-TDT-INT (2 bytes) and a length of 8 bytes

has a value consisting of four integers. If the value is longer than 254 bytes, or if it is of variable length, the token length is set to 255, and the actual length of the value is stored in the first 2 bytes of the value itself. Both options are illustrated in Figure 2-2 on page 2-6.

Token Type

The token data type and token length together define the token type. Token types have symbolic names of the form *subsys*-TYP-*desc*. When writing an SPI server, you can use the ZSPI-defined token types or define your own types based on the fundamental ZSPI token data types.

Token Number

Following the token type in the token code is the token number. Each token code contains a token number that uniquely identifies that token within the set of tokens defined by a subsystem. Token numbers are signed integers in these ranges:

9999 through 32767	Reserved for tokens defined by NonStop operating system software
1 through 9998	Available for HP and user tokens
–512 through 0	Reserved for ZSPI tokens
–32768 through –513	Reserved for tokens defined by software for the NonStop system

Token numbers have symbolic names of the form *subsys*-TNM-*desc*. Different subsystems can define tokens with the same token number without causing a conflict, because tokens are always qualified, explicitly or implicitly, by a subsystem ID.

SSID

Tokens that are explicitly qualified by a subsystem ID (SSID) include that SSID as part of the token code.

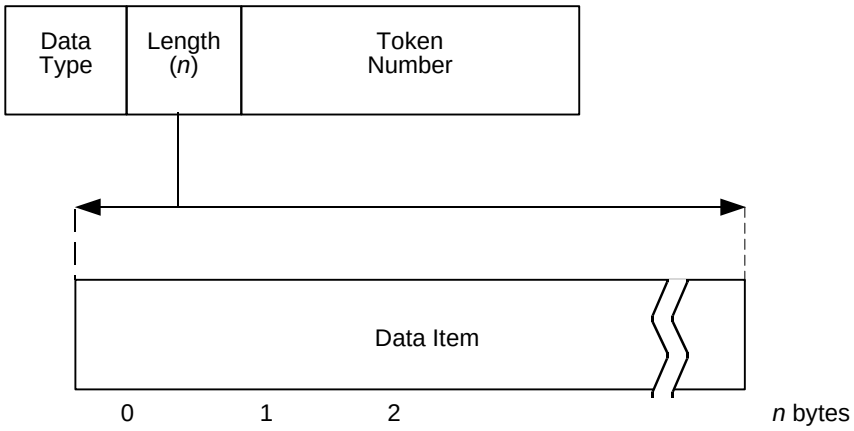
Token Code

The token number, token type, and optional subsystem ID are often treated as a single entity called the token code. Token codes have symbolic names of the form *subsys*-TKN-*desc*.

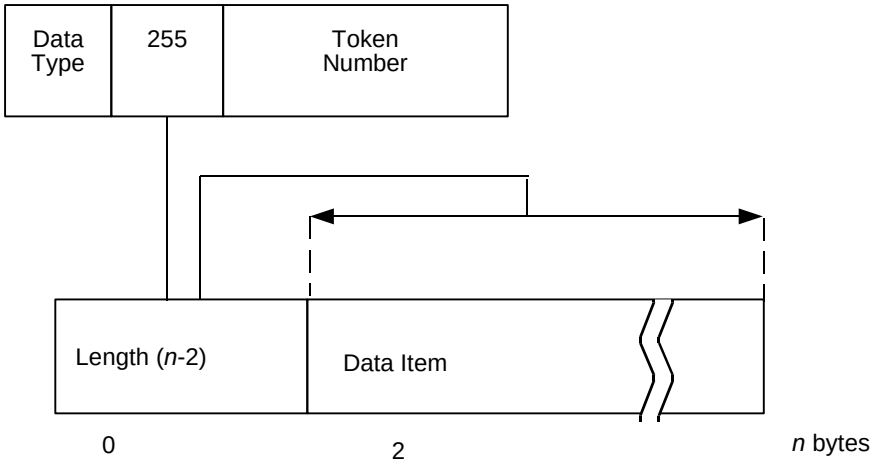
Figure 2-2 on page 2-6 shows two options. The first one represents a fixed-length token value less than 255 bytes, where the length is stored in the length field of the token code. The second option represents a variable-length token value or a value longer than 254 bytes. A length of 255 in the token code indicates that the length of the value is stored in the first two bytes of the value itself.

Figure 2-2. Token Length

Fixed-Length Token



Variable-Length Token



VST006.vsd

Token Names

Every token code has a symbolic token name, a meaningful name of the form *subsys*-TKN-*name*, where *subsys* identifies the subsystem responsible for the definition and *name* describes the content or function of the token. Programs refer to tokens by these names, not by their token numbers. Some tokens are defined by SPI (*subsys* = ZSPI). ZSPI tokens are described in Section 4, ZSPI Data Definitions. Other tokens are defined by the Event Management Service (*subsys* = ZEMS) or by the common SPI extensions used by many NonStop Kernel subsystems (*subsys* = ZCOM or ZCMK). Still others are defined by individual subsystems.

Types of Tokens

Tokens are divided into four categories:

Simple tokens	Have a simple data item or fixed structure as a value
Extensible structured tokens	Have an extensible structure as a value
Zero-length tokens	Have no value and consist of a token code only
Header tokens	Have values that reside in the SPI message header

Simple Tokens

Tokens whose values are elementary data items or fixed structures are called simple tokens. After they are defined, the size, data type, and structure of a simple token cannot be changed.

These are examples of simple tokens defined by SPI or NonStop Kernel subsystems:

ZSPI-TKN-RETCODE	The standard SPI return token, whose value is a number indicating command success or an error
ZCOM-TKN-SUB	The subordinate objects command modifier token used by extended SPI subsystems
ZFUP-TKN-SOURCE-FILE	The file name of the source-file parameter for a FUP command
ZSPI-TKN-ERROR	A fixed structure containing a subsystem ID—itself a fixed structure—and an error number

Extensible Structured Tokens

Extensible structured tokens are tokens whose values are structures that can be extended by adding fields to the ends of the structures. Like a simple structured token, an extensible structured token lets a program place multiple data items in an SPI buffer (or to retrieve them) using a single procedure call. Unlike simple structured tokens, extensible structured tokens allow the addition of new data fields. The version of each field in the structure is recorded in an associated token map so that a specific version of a program sees only those fields of the structure that are defined for that version. Also, each field has a defined null value, which lets a process determine whether the message originator stored a value in the field.

Version and null value information is kept in a separate structure called a token map. The SSNULL procedure initializes extensible structures based on the information in the corresponding map, and the SPI procedures process extensible structured tokens through reference to the map. Token maps have names of the form *subsys*-MAP-*desc*.

To add an extensible structured token to a message, a program specifies the map in the SSPUT procedure call. The procedure then places the associated structure, not the

map itself, into the buffer. The message recipient retrieves the structure from the message by specifying its own version of the map in an SSGET call.

These are examples of extensible structured tokens:

ZPWY-MAP-DEF-PATHWAY	An extensible structure containing parameters for Pathway system configuration
ZFUP-MAP-LOAD-KEYSEQ-OPTS	An extensible structure containing options for key-sequenced destination files for the FUP LOAD command
ZX25-MAP-INFO-SU	An extensible structure containing subdevice information returned by X25AM

Token Maps

Every extensible structured token has an associated token map that contains the null value and version for each field in the structure.

A token map is an extended form of tag used by the SPI procedures to ensure compatibility between different versions of extensible structured tokens. It includes a token number and a token type indicating it is a variable-length extensible structure.

The SSNULL procedure uses token maps to set a structure to its null values, SSPUT uses token maps to set the maximum field version in the SPI message header for use in version compatibility checking, and SSGET uses token maps to truncate or pad the corresponding structure in the buffer to match the value expected by the caller.

The maximum field version in a token map is the most recent version associated with any of the fields defined in the map. When an application adds an extensible structured token to a buffer, the SPI procedure SSPUT updates the maximum field version (ZSPI-TKN-MAX-FIELD-VERSION) in the message header to reflect the most recent version of any non-null field in any extensible structured token in the buffer. (SSPUT does not update the maximum field version when it deletes an extensible token from a message, so following such an operation the maximum field version can actually be greater than the most recent field version actually in the buffer.)

Only the structure described by a token map is put into a message, never the map itself. The SSPUT procedure stores the extensible structured token in the buffer as a typical token with a token type of ZSPI-TYP-STRUCT (token data type ZSPI-TDT-STRUCT and token length 255) and a token number that matches the number of the corresponding map.

Null Values

Every field in an extensible structured token is assigned a null value as part of its DDL structure definition. As a result, any field not assigned a value by a program contains a known value, and the message recipient can determine whether the sender assigned a value to a field. The null values for a structure are recorded in the corresponding token

map and placed in the fields of the structure by the SSNULL and SSGET procedures. A null value is required for every field of an extensible structured token.

The null value specified in the DDL is a single-byte value, defined either as a character or as an integer in the range 0 to 255. The SPI procedures form the null value for the corresponding field by concatenating this single-byte value with itself as many times as necessary to fill the field. So, the null value in the field is seldom equal to the null value specified in the DDL. (They are equal only when the field is also 1 byte long.) For example, a DDL null value of 1 for a 16-bit field results in a structure field null value of 257 (1 << 8 + 1).

Because the null values are generated by repetition of a single-byte value, only some possible values of the field can be null values. For example, a 16-bit integer field cannot have 1 as its null value because the value 1 cannot be formed by repeating any single byte.

The fundamental data structures defined by SPI, and therefore also the data types that are based on them, all have defined null values. You can override these values when defining structures based on these data types.

Zero-Length Tokens

A zero-length token has no value and consist of a token code only. Some zero-length tokens are used as markers or delimiters within a message (ZSPI-TKN-LIST, ZSPI-TKN-SEGLIST, and ZSPI-TKN-ENDLIST, for example). Other zero-length tokens function as special operation codes in SPI procedure calls (ZSPI-TKN-DATA-FLUSH and ZSPI-TKN-CLEARERR, for example).

Header Tokens

SPI messages include a header containing a standard set of information. Some header token values are set by SSINIT when it initializes the buffer. Some header values can be set or retrieved using the SSGET and SSPUT procedures.

Header tokens differ from tokens in the body of the message in several ways:

- Header tokens are always present in the buffer after it has been initialized.
- Header tokens cannot be deleted or flushed from the buffer.
- Each header token occurs only once in the buffer.
- Header tokens cannot be copied using SSMOVE.
- Header tokens cannot be enclosed in a list.
- Neither the token codes nor the values of header tokens can be retrieved by scanning the buffer using the SSGET ZSPI-TKN-NEXTCODE and ZSPI-TKN-NEXTTOKEN operations.
- A program cannot set the current-token pointer to a header token.

- The values of header tokens can be retrieved at any time using SSGET, without changing and regardless of the current position of the buffer pointers.
- Certain special SSGET operations, such as ZSPI-TKN-ADDR and ZSPI-TKN-OFFSET, cannot be performed on header tokens.

The header tokens in Table 2-1 are described in Token Codes on page 4-31. For their use in procedure calls, see Section 3, The SPI Procedures.

Table 2-1. SPI Header Tokens

Header Token	Contents
ZSPI-TKN-BUFLEN	Buffer length
ZSPI-TKN-CHECKSUM	Checksum flag
ZSPI-TKN-COMMAND	Command number
ZSPI-TKN-HDRTYPE	Header type
ZSPI-TKN-LASTERR	Error number of the last nonzero SPI procedure error
ZSPI-TKN-LASTERRCODE	Token code from the last SPI call with a nonzero error
ZSPI-TKN-LASTPOSITION	Position of last token added by SSPUT
ZSPI-TKN-MAX-FIELD-VERSION	Most recent version of any non-null extensible structure field
ZSPI-TKN-MAXRESP	Maximum response records per message
ZSPI-TKN-OBJECT-TYPE	Object-type number
ZSPI-TKN-POSITION	Current token position for SSGET
ZSPI-TKN-SERVER-VERSION	Server version
ZSPI-TKN-SSID	Subsystem ID specified in SSINIT call
ZSPI-TKN-USEDLEN	Number of bytes used in the buffer

Data Definitions

Tokens and related data elements (token numbers, token types, values, structures, fundamental data types, token maps, token codes, and subsystem IDs, for example) are originally defined using the DDL. The DDL compiler translates these definitions into each of the programming languages that support SPI (C, COBOL, TACL, and TAL). HP supplies these files to its customers as part of the operating system.

Naming Conventions

SPI data definitions have names of this form.

```
subsys-type-description          (in DDL and COBOL)
subsys^type^description          (in TAL and TACL)
subsys_type_description           (in C)
```

subsys

is a four-character abbreviation that identifies the subsystem that defined the item. Some subsystems declare many of their own definitions in addition to the common definitions that are used by most subsystems (such as those defined by SPI and EMS). All *subsys* abbreviations beginning with an uppercase or lowercase Z are reserved for HP definitions. Some examples of *subsys* abbreviations are:

ZSPI	Common Subsystem Programmatic Interface definitions
ZCOM	Common SPI extended definitions
ZEMS	Common Event Management Service definitions
ZSCP	Subsystem Control Point definitions
ZEXP	Expand definitions
ZX25	X25AM definitions

type

is a three-character mnemonic that indicates the type of data object that the name represents: a value, object type, or command, for example. The most commonly used mnemonics and the types they represent are:

CMD	Command numbers
DDL	Data Definition Language (DDL) data-structure definitions
ENM	Level 89 items in an enumerated declaration
ERR	Error numbers
EVT	Event numbers
MAP	Extensible structured tokens
OBJ	Object-type numbers
SSN	Subsystem numbers
TDT	Token data types
TKN	Token codes

TNM	Token numbers
TYP	Token types
VAL	Token values

description

is an expression describing the function or meaning of the item. For example, OBJNAME has something to do with the name of an object, and PATH-SWITCH-CAUSE has something to do with the reason for a path change.

This part of the definition name can contain additional separators (-,^, or _).

Examples of Definition Names

ZSPI-CMD-GETVERSION

is the command number for the GETVERSION command that can be implemented by any subsystem:

subsys	ZSPI (a common SPI definition)
type	CMD (a command number)
description	GETVERSION (the GETVERSION command)

ZSCP^ERR^NO^BKUP

is a TAL implementation of the error number for the no-backup-process error implemented by the SCP subsystem:

subsys	ZSCP (a Subsystem Control Point definition)
type	ERR (an error number)
description	NO^BKUP (error related to absence of a backup process)

ZCDG-VAL-SSID

is a value representing the subsystem ID of the common communications diagnostics definitions:

subsys	ZCDG (a common communications diagnostics definition)
type	VAL (a value)
description	SSID (subsystem ID)

ZCOM-TKN-OBJNAME

is a common token used to convey the name an object:

subsys	ZCOM (an extended SPI definition)
type	TKN (a token code)
description	OBJNAME (value conveyed by token: an object name)

Definition Files Supplied by HP

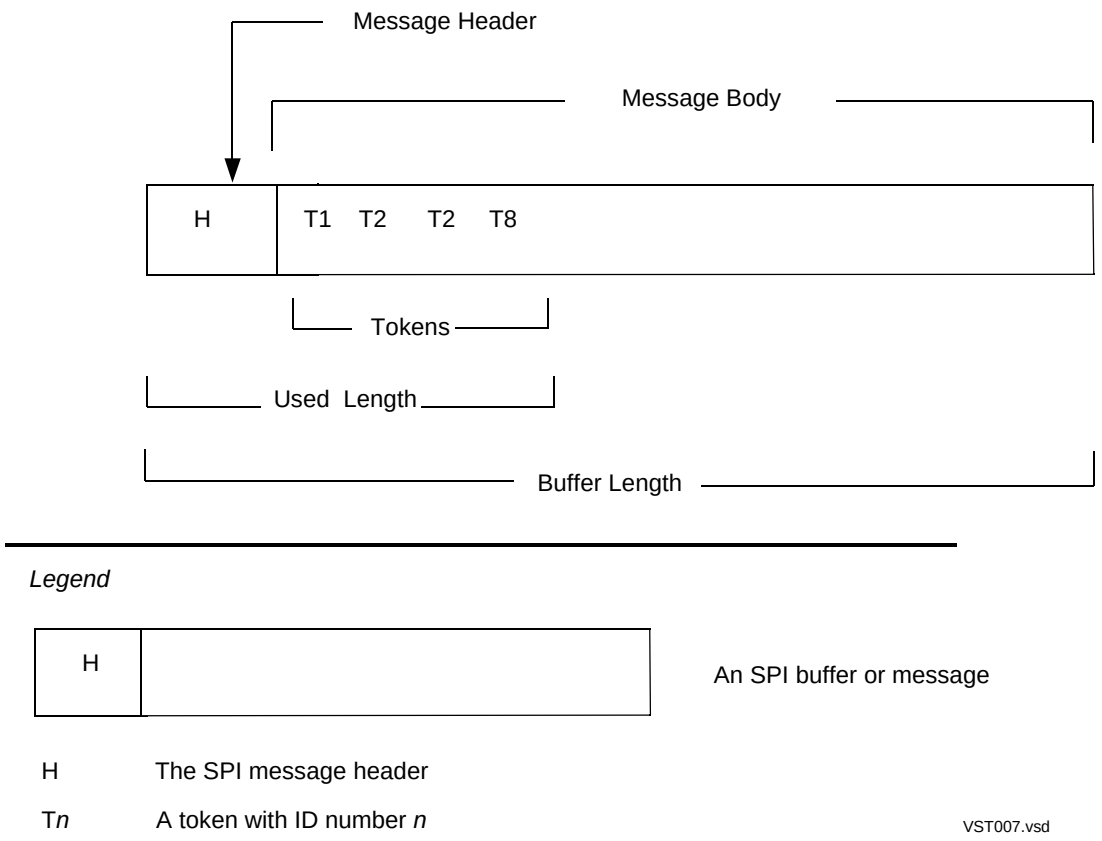
The data definitions provided by HP for the NonStop server are distributed in standard definition files, normally located in $*release-vol*.ZSPIDEF.* (although they can be placed elsewhere). The file names begin with the 4-character subsystem abbreviation *subsys*; for example, **ZSPI**DDL, **ZEMS**DDL, and **ZCOM**DDL. Language-specific files for TAL, COBOL, TACL, and C are generated from each DDL file. They have names like ZSPI**TAL**, ZSPI**COB**, ZSPI**TACL,** and ZSPI**C**. An Expand management application written in TAL might require these files:

ZSPIDEF.ZSPITAL	Definitions common to all subsystems
ZSPIDEF.ZCOMTAL	Definitions for common SPI extensions
ZSPIDEF.ZEMSTAL	Event (EMS) definitions common to all subsystems
ZSPIDEF.ZEXPTAL	Expand definitions

SPI Message Buffer

SPI messages are composed in a specially allocated block of memory called an SPI buffer. An SPI buffer consists of a header, which contains message information common to all command and response messages, and a variable-length body containing message tokens. The logical structure of an SPI buffer is shown in Figure 2-3 on page 2-14.

Figure 2-3. The SPI Buffer



The SPI procedure SSINIT initializes an SPI buffer and places values in header tokens. SSPUT puts tokens into a buffer, and SSGET retrieves token values from a buffer. Section 3, The SPI Procedures, describes the SPI procedures.

All modifications of an SPI buffer must be performed using the SPI procedures. SPI buffers contain many specialized data structures and can be corrupted if manipulated in any other way.

Message Header

The two types of SPI messages are: messages containing commands or responses, and messages reporting events. A header token in each SPI message specifies whether the message is a command/response message or an event message. This manual describes command/response messages. For information about event messages and the event message header, see the *EMS Manual*.

The header portion of a command/response buffer contains:

- A header type indicating a command/response message
- The length of the message buffer

- The maximum version of any field in any extensible structure in the buffer
- The subsystem ID of the server
- The server version
- The maximum number of response records the server is allowed to return
- The command number
- The object-type to which the command is applied
- A flag indicating the status of checksum protection
- The number of the last nonzero error returned by an SPI procedure
- The token code involved in the last SPI call that returned a nonzero status
- The current token position for SSGET
- The position of the last token added by SSPUT
- The number of used bytes in the buffer

Each of these values is stored as the value of a header token. For a description of these special tokens, see Header Tokens on page 2-9.

Message Body

The body of the message contains tokens added by SSPUT. Although the same header tokens are present in every command/response message, the tokens in the message body vary depending on the command and on the specific implementation of that command by a particular subsystem. The exact contents of a command message and its associated response message for a particular subsystem are described in the subsystem's management programming manual.

A particular token can appear more than once in the message, though multiple occurrences are not necessarily contiguous.

Tokens in the message body can be grouped into lists. A list is a group of tokens that is bracketed by special list-delimiting tokens—a list starts with ZSPI-TKN-LIST, -DATALIST, -ERRLIST, or -SEGLIST, and ends with ZSPI-TKN-ENDLIST. Lists are used to group tokens that logically belong together, such as all the tokens containing response information about one object or all the tokens describing an error. For the four types of SPI lists, the generic list, the data list, the error list, and the segment list, see Lists on page 2-19.

Buffer Length

The initial size of an SPI buffer is determined by the *buffer-length* parameter of the SSINIT procedure, which initializes the specified number of bytes. The size can be modified using SSPUT with the header token ZSPI-TKN-BUFLEN or with the special operation code ZSPI-TKN-RESET-BUFFER. All NonStop Kernel subsystems

recommend a buffer size that can accommodate the largest command or response supported by the subsystem. These recommended buffer sizes have names of the form
subsys-VAL-BUFLEN.

Used Length

Few command and response messages occupy the entire allocated buffer. You can determine how many bytes actually contain message information by using the SSGET procedure to retrieve the value of the header token ZSPI-TKN-USEDLEN. This lets you, for example, send only the used portion of the buffer to a server.

Buffer Pointers

Four pointers track four important token positions in each buffer:

- The current token
- The next token
- The last-put token
- The current list

The Current-Token Pointer

The current-token pointer contains the position of the token most recently selected using SSGET. This pointer is stored in the header token ZSPI-TKN-POSITION. SSINIT sets the current token pointer to the beginning of the buffer, immediately following the header and preceding any nonheader token. Thereafter, the current-token pointer is updated by any successful call to SSGET or SSMOVE, and can be explicitly set by a call to SSPUT using one of the special token codes ZSPI-TKN-POSITION, ZSPI-TKN-INITIAL-POSITION, or ZSPI-TKN-RESET-BUFFER. Its value can be retrieved by calling SSGET with ZSPI-TKN-POSITION. The current-token pointer never points to a header token. The current-token pointer is not updated when SSGET retrieves a header token value or when a nonzero error is returned by an SPI procedure.

The Next-Token Pointer

The next-token pointer contains the position where SSGET starts scanning the buffer if it is called with an index value of 0. SSINIT sets the next-token pointer to the beginning of the buffer, immediately following the header. Thereafter, whenever a token value is retrieved, the next-token pointer is set to the token following the value retrieved. If SSGET is used to retrieve a token code rather than a token value—when SSGET is called using the ZSPI-TKN-NEXTCODE or ZSPI-TKN-NEXTTOKEN special operations—the next-token pointer is set to the retrieved token code so that an SSGET call specifying that token code retrieves that token's value. The next-token pointer never points to a header token, and is not changed when SSGET retrieves a header

token value. The next-token pointer is an internal buffer management position and cannot be explicitly retrieved by an application.

The Last-Put-Token Pointer

The last-put-token pointer contains the position of the last token added to the buffer by the SSPUT procedure, and is stored in the header token ZSPI-TKN-LASTPOSITION. Its value can be retrieved by calling SSGET with ZSPI-TKN-LASTPOSITION. An application can store this value and later return to the same location in the buffer by using SSPUT to restore this value to the ZSPI-TKN-POSITION header token.

The Current-List Pointer

The current-list pointer always points to the currently selected list. If no list is selected, this pointer is set to null.

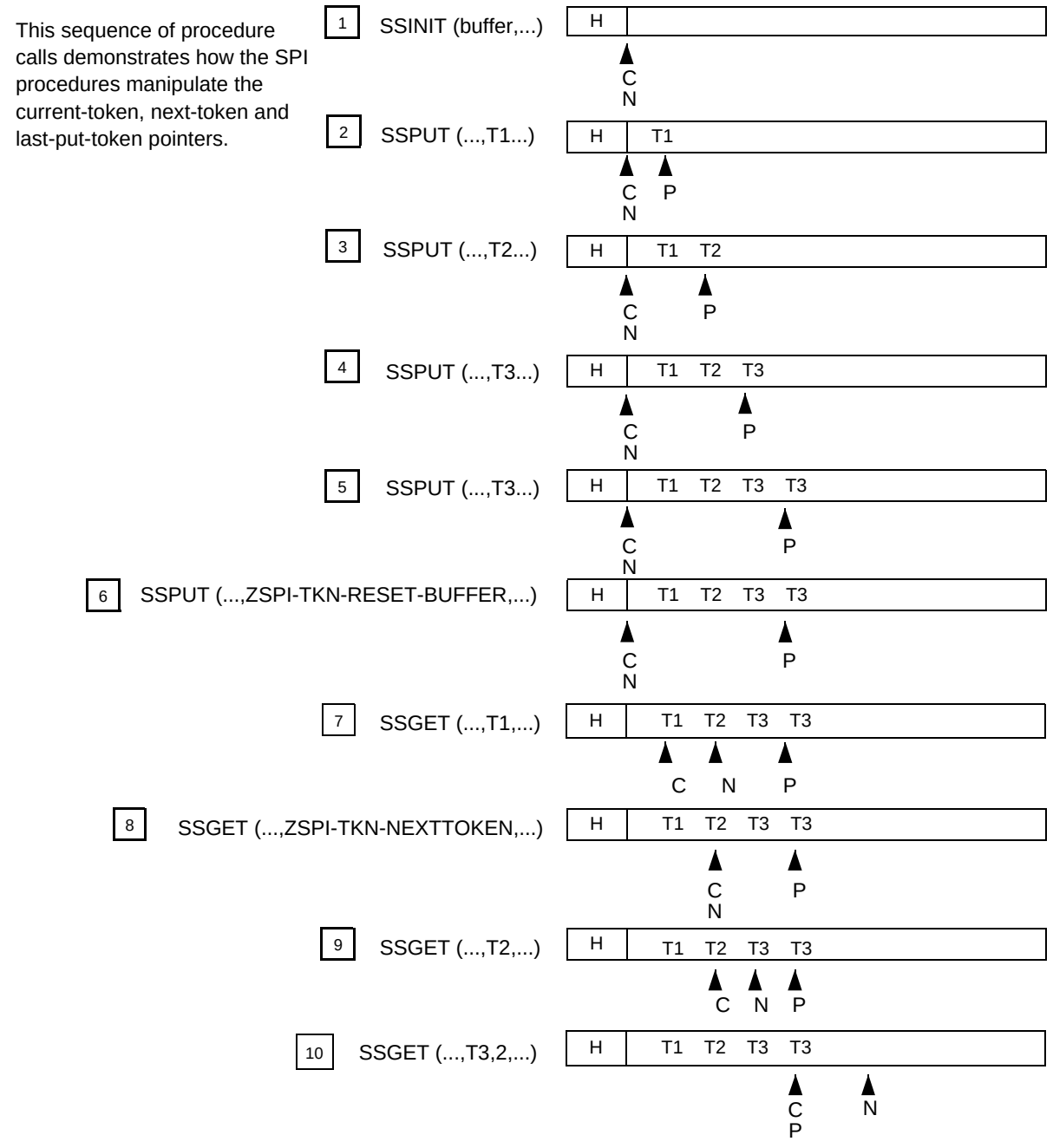
Pointer Manipulation

Figure 2-4 on page 2-18 demonstrates how basic procedure calls affect the buffer pointers. It shows a sequence of procedure calls and the resulting pointer movements:

1. The SSINIT call resets both the current-token and next-token pointers to the beginning of the buffer, and the last-put-token pointer is null.
2. Four consecutive calls to SSPUT add four tokens to the buffer (Token T3 appears twice.) After each call, the last-put-token pointer is updated.
3. When the buffer is reset, both the current-token and next-token pointers are reset to the beginning of the buffer, and the last-put-token pointer is null.
4. The first SSGET call retrieves the value of token T1. The next-token pointer is set to token T2. This is where the next SSGET starts searching the buffer the next time it is called, unless a positive index is specified.
5. The next SSGET call uses the special operation ZSPI-TKN-NEXTTOKEN to retrieve the token code of the next token in the buffer. Because only the token code—not the token value—is retrieved, the next-token pointer remains at token T2 with the current-token pointer. This allows the next call, which specifies the token code just retrieved, to get the value for that token.
6. The third SSGET call retrieves the value for token T2, and the next-token pointer is advanced.
7. The last SSGET call uses the index parameter to retrieve the value of the second occurrence of token T3. The next-token pointer is now past all tokens in the buffer, so another call to SSGET without any index parameter fails to find a token.

The procedure call series in Figure 2-4 is performed by the TAL program in Example E-1 on page E-5 and the C program in Example E-2 on page E-7.

Figure 2-4. Pointer Manipulation Examples



Buffer Checksum

The SPI buffer contains a checksum that the SPI procedures can use to detect corruption of the buffer contents.

When so directed, each SPI procedure updates the checksum after any modification of the buffer. Each time an SPI procedure is called, it first recomputes the checksum and compares it against the checksum stored in the buffer. If they do not match, the procedure reports a checksum error, indicating that the buffer has been corrupted since the last SPI procedure call.

By default, SPI does not calculate buffer checksums. To enable checksum protection, use SSINIT or SSPUT to set the header token ZSPI-TKN-CHECKSUM to a nonzero value.

Lists

Lists are used in SPI response messages to organize tokens into logical groups. Lists appear only in response messages, not in command messages. Lists can be nested—lists can contain other lists. The four types of lists are:

Data Lists	`(ZSPI-TKN-DATALIST…ZSPI-TKN-ENDLIST)`
Error lists	`(ZSPI-TKN-ERRLIST…ZSPI-TKN-ENDLIST)`
Segment lists	`(ZSPI-TKN-SEGLIST…ZSPI-TKN-ENDLIST)`
Generic Lists	`(ZSPI-TKN-LIST…ZSPI-TKN-ENDLIST)`

Normally, to access tokens in a list, a program must select the list token that marks the beginning of the list. To exit from a list, the program must select the corresponding ENDLIST token. (The exception is the SSGET NEXTTOKEN special operation.) For more information about lists, see Working With Lists on page 5-5.

Data Lists

Data lists are used in response messages to delimit response records. When a command is applied to more than one object, the tokens describing the results are returned in data lists, one list for each object. By default, response tokens for a command that is applied to a single object are not returned in a data list. However, even in this case a requester can have the response record returned in a data list by using SSINIT or SSPUT to assign a value of 1 to the header token ZSPI-TKN-MAXRESP. A data list consists of all tokens between ZSPI-TKN-DATALIST and the corresponding ZSPI-TKN-ENDLIST.

Error Lists

Error lists contain tokens describing an error that occurred during command processing. SPI requires that certain information be included in error lists. For a full description of error list structure and contents, see Errors and Warnings on page 2-47. An error list consists of all tokens between ZSPI-TKN-ERRLIST and the corresponding ZSPI-TKN-ENDLIST.

Segment Lists

SPI servers use segment lists to build segmented responses. Tokens in a segmented response are divided into repeating and nonrepeating groups, and each repeating group is enclosed in a segment list. For more information, see Segmented Responses on page 2-38. A segment list consists of all tokens between ZSPI-TKN-SEGLIST and the corresponding ZSPI-TKN-ENDLIST.

Generic Lists

Generic lists are provided for use in customer-developed subsystems. NonStop Kernel subsystems provided by HP do not use generic lists in commands or responses. An SPI server can use generic lists whenever necessary to organize response tokens. A generic list consists of all tokens between ZSPI-TKN-LIST and the corresponding ZSPI-TKN-ENDLIST.

Pointer Manipulation and Lists

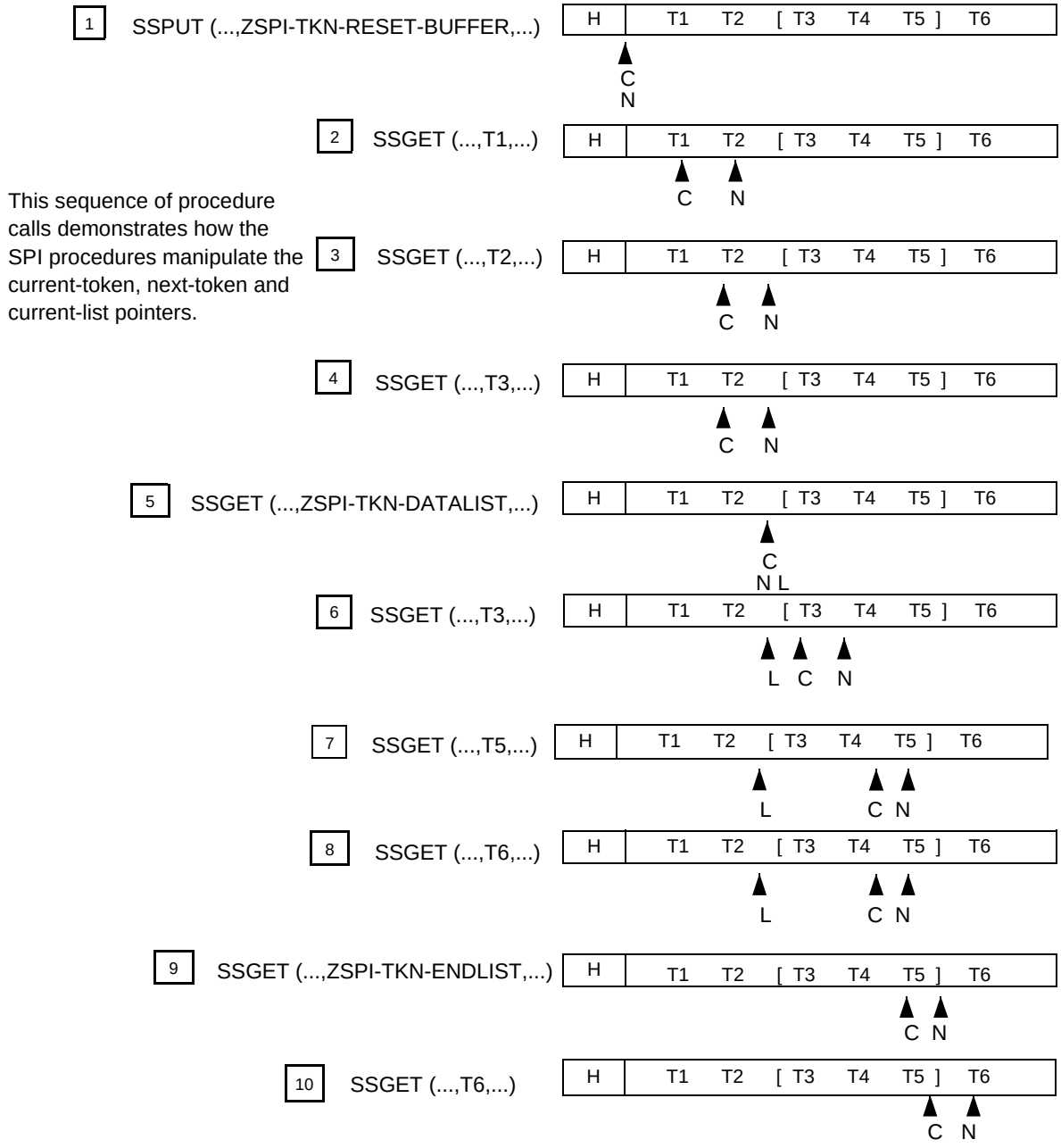
Figure 2-5 on page 2-22 demonstrates how basic procedure calls affect the buffer pointers when working in and around lists. It shows a sequence of procedure calls and the resulting pointer changes. Although the figure shows a data list, the demonstrated behavior applies to all list types:

1. The data portion of the buffer contains eight tokens (T1…T6, ZSPI-TKN-DATALIST, and ZSPI-TKN-ENDLIST) with tokens T3, T4, and T5 in the data list. The SSPUT special operation ZSPI-TKN-RESET-BUFFER resets both the current-token and next-token pointers to the beginning of the buffer. The current-list pointer remains null until a list is selected.
2. The first SSGET call retrieves token T1.
3. The second SSGET call retrieves token T2.
4. The third SSGET call asks for token T3. Because SSGET cannot access tokens within a list until the corresponding list token is selected, the token is not found. ZSPI-ERR-MISTKN is returned and the pointers are unchanged.
5. The fourth SSGET call enters the data list by selecting the list token. The current list pointer is now defined, and the current-token and next-token pointers point to ZSPI-TKN-DATALIST. SSGET can now retrieve tokens from inside the list.
6. Now that the list has been selected, SSGET can retrieve token T3 from in the list. The current-token and next-token pointers are updated in the list.
7. The next SSGET call retrieves token T5 from in the list.
8. The seventh SSGET call asks for token T6, which is in the buffer but not in the current list. SSGET returns ZSPI-ERR-MISTKN, and the pointers are unchanged.
9. The next SSGET call exits from the list by selecting the end list token ZSPI-TKN-ENDLIST.

10. Having exited the list and returned to the top level of the buffer, SSGET retrieves token T6.

The procedure call series in Figure 2-5 are performed by the TAL program in Example E-3 on page E-10 and the C program in Example E-4 on page E-13.

Figure 2-5. Pointer Manipulation and Lists



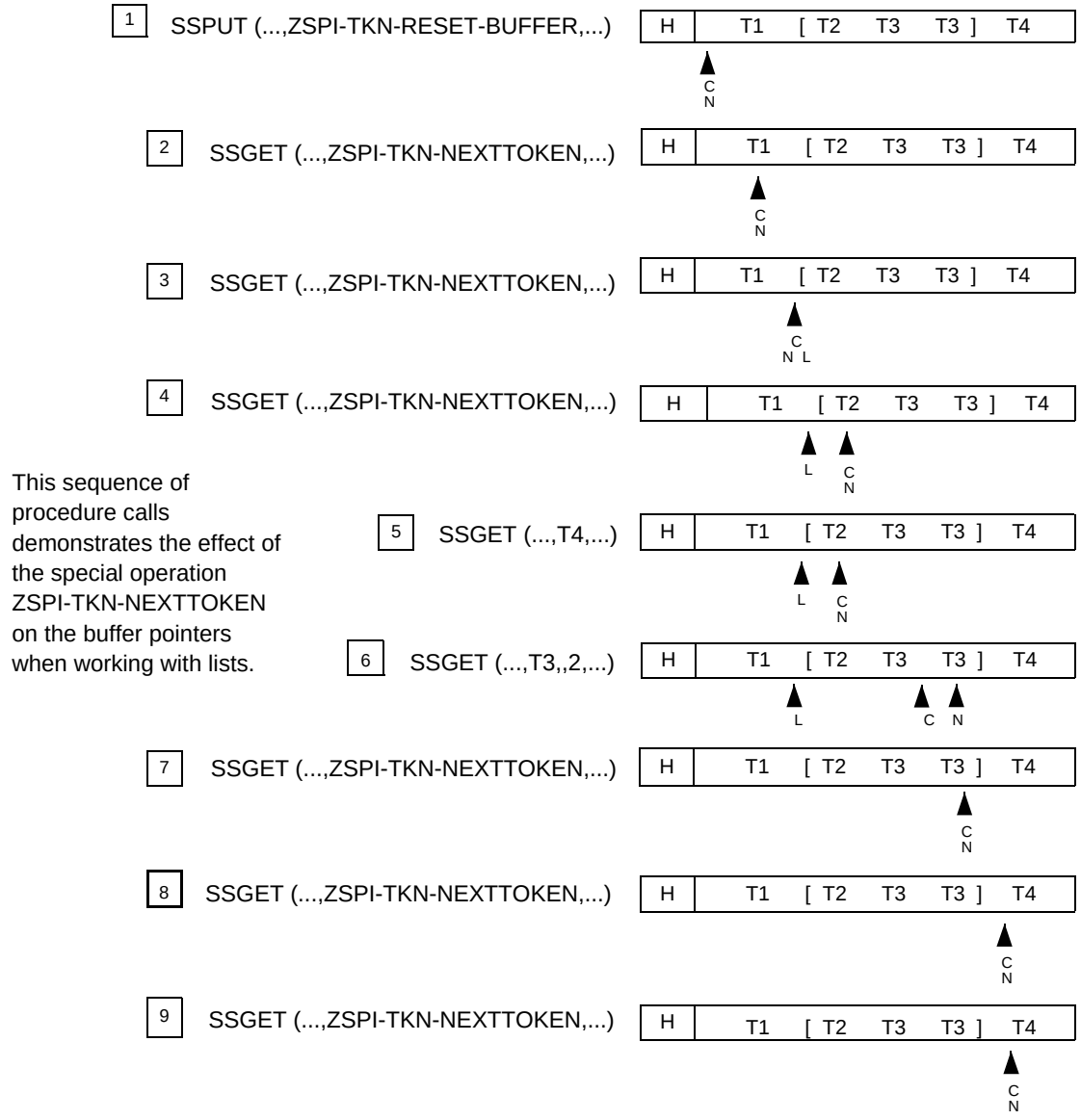
Pointers, Lists, and ZSPI-TKN-NEXTTOKEN

Figure 2-6 on page 2-24 shows the behavior of SSGET special operation ZSPI-TKN-NEXTTOKEN when working in and around lists. It demonstrates a sequence of procedure calls and the resulting pointer changes. Although the figure shows a data list, the demonstrated behavior applies to all list types:

1. The data portion of the buffer contains seven tokens (T1, T2, two occurrences of T3, T4, ZSPI-TKN-DATALIST, and ZSPI-TKN-ENDLIST) with tokens T2 and T3 inside the data list. The SSPUT special operation ZSPI-TKN-RESET-BUFFER resets both the current-token and next-token pointers to the beginning of the buffer. The current-list pointer remains null until a list is selected.
2. The first SSGET call asks for the next token in the buffer, and retrieves token T1.
3. The next SSGET asks for the next token in the buffer, and retrieves the list token ZSPI-TKN-DATALIST. The current-token and next-token pointers are set to this token and, because the list token was selected, the current-list pointer is defined.
4. The next SSGET call retrieves the next token, now from within the list.
5. Because SSGET is working within the list, an attempt to retrieve a token outside the list fails with error ZSPI-ERR-MISTKN, and the pointers are unchanged.
6. An SSGET call uses the index parameter to retrieve the second occurrence of token T3 from within the list.
7. The next token in the buffer is ZSPI-TKN-ENDLIST, and by selecting it SSGET exits the list.
8. Outside the list now, SSGET can retrieve token T4.
9. Token T4 is the last token in the buffer, so a call to SSGET asking for the next token returns error ZSPI-ERR-MISTKN.

The procedure call series in Figure 2-6 is performed by the TAL program in Example E-5 on page E-15 and the C program in Example E-6 on page E-19.

Figure 2-6. Pointers, Lists, and ZSPI-TKN-NEXTTOKEN



Pointers, Lists, and ZSPI-TKN-NEXTCODE

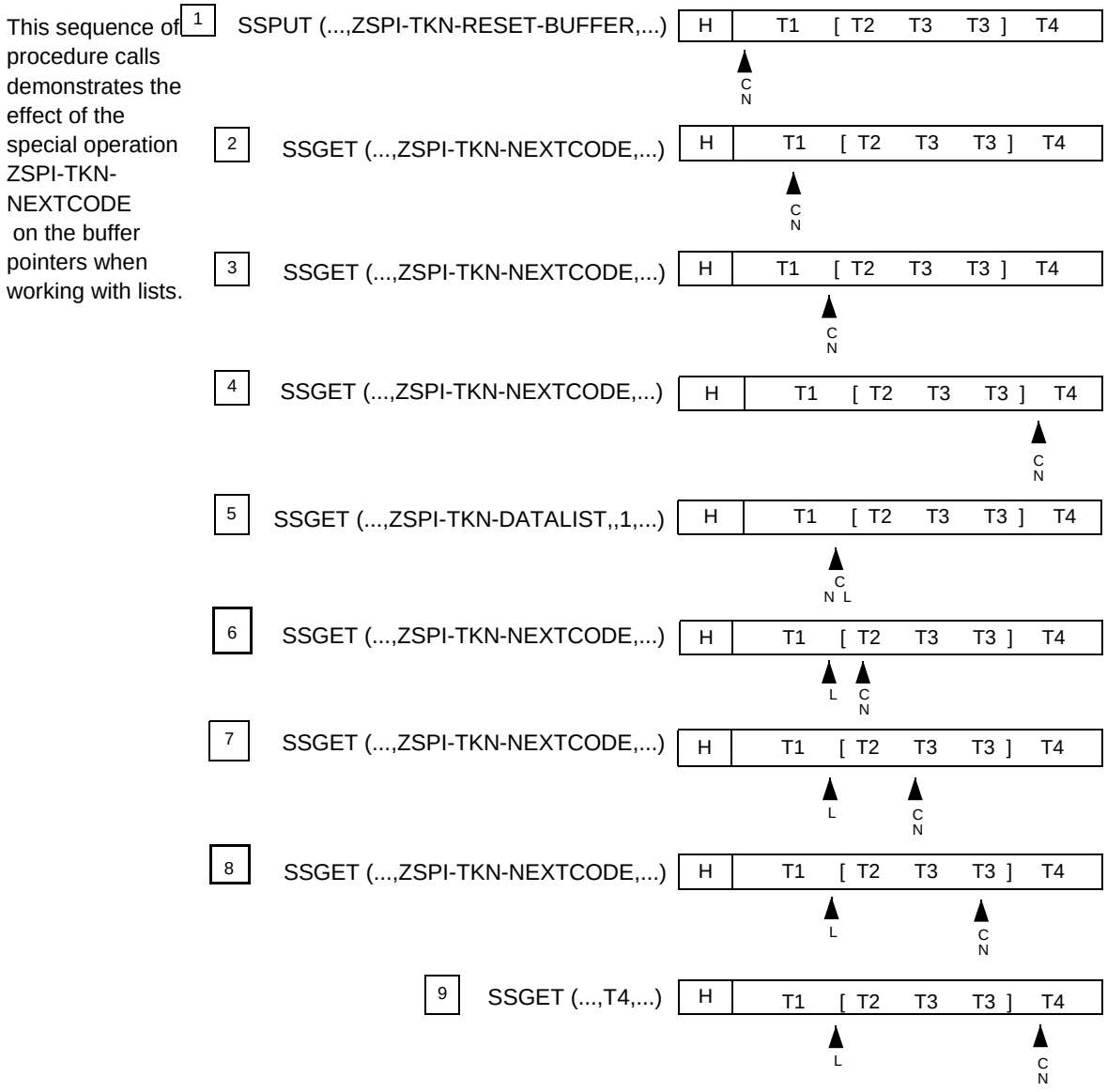
Figure 2-7 on page 2-26 shows the behavior of SSGET special operation ZSPI-TKN-NEXTCODE in the same buffer scanned with ZSPI-TKN-NEXTTOKEN in Figure 2-6 on page 2-24. It demonstrates a sequence of procedure calls and the resulting pointer movements. Although the figure shows a data list, the demonstrated behavior applies to all list types:

1. The data portion of the buffer contains seven tokens (T1, T2, two occurrences of T3, T4, ZSPI-TKN-DATALIST, and ZSPI-TKN-ENDLIST) with tokens T2 and T3 inside the data list. The SSPUT special operation ZSPI-TKN-RESET-BUFFER resets both the current-token and next-token pointers to the beginning of the buffer. The current-list pointer remains null until a list is selected.
2. The first SSGET call asks for the next token code in the buffer, and retrieves token code T1.
3. The next SSGET call asks for the next different token code in the buffer, and retrieves the list token ZSPI-TKN-DATALIST. However, unlike ZSPI-TKN-NEXTTOKEN, the NEXTCODE special operation does not select the list.
4. Because the list was not selected, the next SSGET call ignores the contents of the list and retrieves token T4, the next different token at the top level.
5. SSGET is called to select the list token and enter the list. An index parameter of 1 is specified to ensure that SSGET scans from the beginning of the buffer.
6. Now within the list, SSGET NEXTCODE retrieves the next different token code T2.
7. The same call repeated retrieves token T3.
8. The next call retrieves ZSPI-TKN-ENDLIST.
9. Because SSGET is still in the list, it cannot retrieve token T4, which is outside the list. SSGET returns error ZSPI-ERR-MISTKN, and the pointers are unchanged.

The procedure call series in Figure 2-7 is performed by the TAL program in Example E-7 on page E-22 and the C program in Example E-8 on page E-25.

Figure 2-7. Pointers, Lists, and ZSPI-TKN-NEXTCODE

This sequence of procedure calls demonstrates the effect of the special operation ZSPI-TKN-NEXTCODE on the buffer pointers when working with lists.



Commands

A command directs an SPI server to perform an action on one or more objects. The action can affect the objects in some way or might just retrieve information about the objects.

The command in an SPI request is identified by a command number, which the requester specifies in the SSINIT procedure call that initializes the message. SSINIT stores the command number in the header token ZSPI-TKN-COMMAND. Command numbers have names of the form *subsys*-CMD-*command*.

Basic SPI defines a single command number, ZSPI-CMD-GETVERSION. Additional common commands are defined by the SPI extensions described in the *SPI Common Extensions Manual*. Other commands are defined by individual subsystems and described in the subsystem management programming manuals.

GETVERSION Command

The GETVERSION command returns basic server version information. SPI recognizes two forms of the command: a basic GETVERSION command and an extended command.

The Basic GETVERSION Command

A basic GETVERSION command contains the null object type (specifies ZSPI-VAL-NULL-OBJECT-TYPE for the object type in the SSINIT call). In its response, a server returns its version in the ZSPI-TKN-SERVER-VERSION header token and a server identification string in the response token ZSPI-TKN-SERVER-BANNER. (See the description of ZSPI-TKN-SERVER-BANNER for the format of the server ID string.)

The response can also contain any of:

- Additional SERVER-BANNER tokens declaring the versions of various subsystem components. (The first instance of the token always declares the version of the server specified by the subsystem ID in the command.)
- One or more ZSPI-TKN-IPM-ID tokens to identify interim product modifications that have modified the interface to the subsystem.

A single call to SSINIT, specifying ZSPI-CMD-GETVERSION for the command and ZSPI-VAL-NULL-OBJECT-TYPE for the object type parameter, is sufficient to prepare a basic GETVERSION command—no additional tokens are required.

All SPI servers support this basic GETVERSION command.

Extended GETVERSION Commands

A subsystem can provide additional features in its implementation of the GETVERSION command:

- The subsystem can allow the requester to specify a particular object type other than NULL in order to retrieve the version of some subsystem component.
- The subsystem can allow additional tokens in the command which prompt the subsystem for additional information describing specific features supported by the subsystem.

Responses

Every command sent to an SPI server results in a response. These terms and concepts are central to understanding SPI responses:

- Response—the sum of all information that an SPI server returns to the requester as a result of processing a command. This information can be returned in a single response message, or might require several response messages.
- Response message—a single SPI message sent from an SPI server to the requester to describe the outcome of command processing. A response message can contain a single response record, multiple response records, or, in the case of a segmented response, only part of a response record.
- Response record—all information describing the application of a command to a single object, or, if an error in the command prevented it from being applied to any object, the response record contains general error information.

Types of Responses

Several types of responses result from various combinations of response records and response messages:

Response Type	Consists of...	Page
Simple	One response record returned in a single response message.	2-29
Multirecord	Multiple response records returned in a single response message.	2-30
Continued	One or more response records returned in multiple response messages.	2-34
Segmented	A single response record returned a piece at a time in multiple response messages (special form of continued response).	2-38
Empty	Only the return code token. Also sometimes returned as the last message in a continued response. (Special form of simple response.)	2-41

A requester controls the type of response—within the capabilities of the server—by specifying some combination of these parameters:

- The number of objects to which the command is applied, which can be affected by the use of wild cards in the object name or the use of subsystem-defined command modifiers. A command applied to a single object can result only in a simple or a segmented response.
- The maximum number of response records allowed in a response message, stored in the header token ZSPI-TKN-MAXRESP. If the requester allows only one record per message, a given number of response records requires a continued response consisting of multiple messages, whereas the same number of records might have been returned in a single message if multiple records per message were allowed and the buffer was large enough.
- The size of the message buffer, which can determine if a single response message suffices, or if a continued response is required.
- The type of response records to be returned, as specified in the token ZSPI-TKN-RESPONSE-TYPE. The number of records returned can be greatly reduced by requesting only records that report errors or warnings, if appropriate.
- The specification of ZSPI-VAL-ALLOW-SEGMENTS as a value for ZSPI-TKN-ALLOW, thereby allowing a segmented response.

Simple Responses

A simple response consists of a single response record returned in a single response message.

A server generates a simple response when:

- The server successfully applies a command to a single object, the response record fits in the message buffer, and the requester has asked for normal responses. In this case, the server returns a single response record containing the return code and a server-defined set of tokens. If the requester asks for error or warning response records only, the server returns an empty response. If the response record does not fit in the buffer, the server might be able to generate a segmented response. If the response record is too long but the server does not support segmented responses, the server returns an error.
- The server detects an error in a command directed to one or more objects which prevents the command from being applied to any object. In this case, the server returns a single response record containing a nonzero return code and at least one error list.
- The server cannot apply an otherwise valid command to the object specified in the command. In this case, the server returns a single response record containing a nonzero return code and at least one error list.

Multirecord Responses

When a command is applied to more than one object, the server returns multiple response records (unless the requester suppresses response records; see Suppressing Response Records on page 2-43). In this case, message traffic can be reduced by having the server return more than one response record in each response message. The requester controls the number of response records in each response by assigning a value to the header token ZSPI-TKN-MAXRESP.

The value of ZSPI-TKN-MAXRESP indicates the maximum number of response records the requester will accept in a single response message. The requester can specify these values:

0	Lets the server return one response record per response message, with the record not enclosed in a data list. (Default)	Figure 2-8
n > 0	Lets the server return as many as *n* response records in the response message, with each record enclosed in a data list. This is a limit and not an absolute value—the server might return fewer than *n* records in the message.	Figure 2-9
–1	Lets the server return as many response records as will fit in the buffer, with each record enclosed in a data list.	Figure 2-10

Not all subsystems support multirecord response messages. However, all NonStop Kernel subsystems recognize the MAXRESP parameter to SSINIT and the MAXRESP token.

Each command message carries its own MAXRESP value; the server does not retain this value for use with subsequent commands.

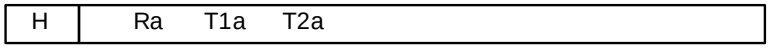
Each response record contains its own return code.

Warning information that pertains to the command itself, rather than to the action of the command on a particular object, is not repeated in every response record if there are multiple response records per response. Any command-related warnings appear only in the first response record in the sequence.

Figure 2-8. ZSPI-TKN-MAXRESP = 0 (Default)

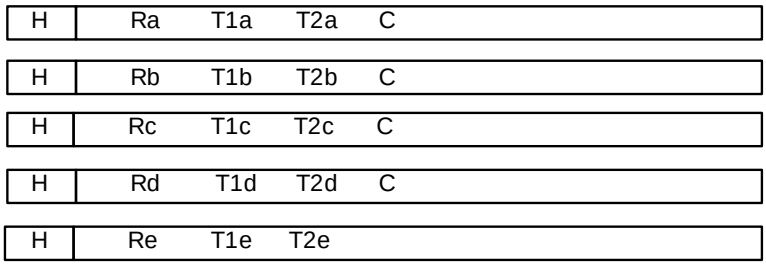
The server returns one response record in each response message. Response records are not enclosed in data lists.

Example: MAXRESP = 0 and one object qualifies for processing:



The server returns one message containing one response record (not in a datalist).

Example: MAXRESP = 0 and five objects (a through e) qualify for processing:



The server returns five messages, each containing one response record (not in a list).

Legend

H Message header

R*x* ZSPI-TKN-RETCODE for object *x*

T*nx* Token *n* describing object *x*

C ZSPI-TKN-CONTEXT

VST012.vsd

Figure 2-8 shows how a requester returns response records when the MAXRESP header token is allowed to default to 0. Only one response record is returned in each response message, and this record in not enclosed in a data list. If the command is applied to more than one object, each response record is returned in a separate response message, each but the last of which includes the context token, following the standard protocol for command continuation.

Figure 2-9. ZSPI-TKN-MAXRESP > 0

The server returns as many as *n* response records in each response message. Each response record is contained in a separate data list.

Example: MAXRESP = -1 and one object qualifies for processing:

H	[Ra T1a T2a]

The server returns one message containing one response record in a datalist.

Example: MAXRESP = -1 and five objects (a through e) qualify for processing:

H	[Ra T1a T2a] [Rb T1b T2b] C
H	[Rc T1c T2c] [Rd T1d T2d] C
H	[Re T1e T2e]

The server returns two response records in the first response message, two in the second and the fifth response record in the third message.

Legend

H	Message header	C	ZSPI-TKN-CONTEXT
T*nx*	Token *n* describing object *x*	[	ZSPI-TKN-DATALIST
R*x*	ZSPI-TKN-RETCODE for object *x*	]	ZSPI-TKN-ENDLIST

VST013.vsd

Figure 2-9 shows how a requester returns response records when the MAXRESP header token is set to a positive integer, *n*. The server returns as many as *n* response records in each response message, with each record enclosed in a data list. If more than one message is needed to return all the response records, the server follows the standard protocol for command continuation until all records are returned.

Figure 2-10. ZSPI-TKN-MAXRESP = –1

The server returns as many response records in each response message as fit in the buffer. Each response record is contained in a separate data list.

Example: MAXRESP = -1 and one object qualifies for processing:

H	[Ra T1a T2a]

The server returns one message containing one response record in a datalist.

Example: MAXRESP = -1 and five objects (a through e) qualify for processing:

H	[Ra T1a T2a] [Rb T1b T2b] [Rc T1c T2c] C

H	[Rd T1d T2d] [Re T1e T2e]

The server returns three response records in the first response message (the fourth does not fit) and two in the second.

Legend

H	Message header	C	ZSPI-TKN-CONTEXT
T*nx*	Token *n* describing object *x*	[	ZSPI-TKN-DATALIST
R*x*	ZSPI-TKN-RETCODE for object *x*	]	ZSPI-TKN-ENDLIST

VST014.vsd

Figure 2-10 shows how a requester returns response records when the MAXRESP header token is set to –1. The server returns as many response records in each response message as fit in the buffer. Each record is enclosed in a data list. If more than one message is needed to return all the response records, the server follows the standard protocol for command continuation until all records are returned.

Continued Responses

Many subsystems let a single command be applied to multiple objects, in which case the server returns a separate response record for each object. If the server cannot return records for all selected objects, either because the buffer is too small or the requester is using ZSPI-TKN-MAXRESP to limit the number of records that can be returned, the server stops processing objects and returns the completed records along with the context token ZSPI-TKN-CONTEXT. That one response message completes the requester-server interaction from the perspective of the interprocess communication mechanism (such as the file system), and the requester must send another command to have the server continue processing the remaining objects.

The requester includes the context token in the followup command to tell the server where to continue processing. The requester copies the context token from the previous response message to a duplicate of the original command and sends the command with context back to the server. The server uses the information that it stored in the context token to resume processing with the next object.

The use of the context token lets subsystems be context-free—that is, independent of any processing that occurred before the current command. A context-free server lets the requester interrupt or abandon the continuation of a series of replies. Most NonStop Kernel subsystems are context-free. A context-sensitive server retains information about previous processing. Context-sensitive servers limit or complicate the requester's ability to interrupt or abandon continuation. When practical, servers should be designed to be context-free.

In the continuation command, the command and parameters must be identical to those sent in the original command; otherwise, the results are unpredictable. Because the values of some tokens in the response message might not be appropriate for the continuation command, your application should not reuse the response message. Instead, the requester should save a copy of the original command message, copy the context token from the response into the duplicate command, and resend.

This independence between requester and server lets your application alter its course of action without retrieving all messages. For most subsystems, the application can issue other commands before issuing the continuation command, or it can abandon the continuation. A program might want to do this, for instance, to recover from an error or to display related information obtained from different commands.

In designing your application, consider that changes can occur in the subsystem environment between response messages. In some cases, it is important to process response messages for a command quickly and with few interruptions, so continuation might not be a good choice. In addition, some subsystems with context-sensitive servers prohibit or restrict continuation. For more information, see the individual subsystem manuals.

For subsystems that support continuation, the absence of a context token in a response message indicates the last message in the response. The last response message can contain a normal response record or an empty response record that indicates that there are no more objects to process. (An empty response record might

indicate, for instance, that there are no more objects to process because the remaining objects were deleted since the server returned the last response.)

The absence of a context token is the only valid indicator of the last response message. Your application should always check for the context token (it can do so with a simple call to SSGET). It should never rely on a particular return-token value or on a particular number of response messages or response records to determine that there are no more replies.

Be aware that the context token can vary in size. It is not safe to store it in a work area—always copy it directly from response to command using the SSMOVE procedure.

Figure 2-11 on page 2-36 illustrates the exchange of messages when the subsystem's response is continued over several response messages.

Note. Although this and other figures illustrating the buffer show the context token last, your application must not depend on its being in any particular position in the buffer. In SSMOVE calls that refer to the context token, specify an index of 1 to ensure position transparency.

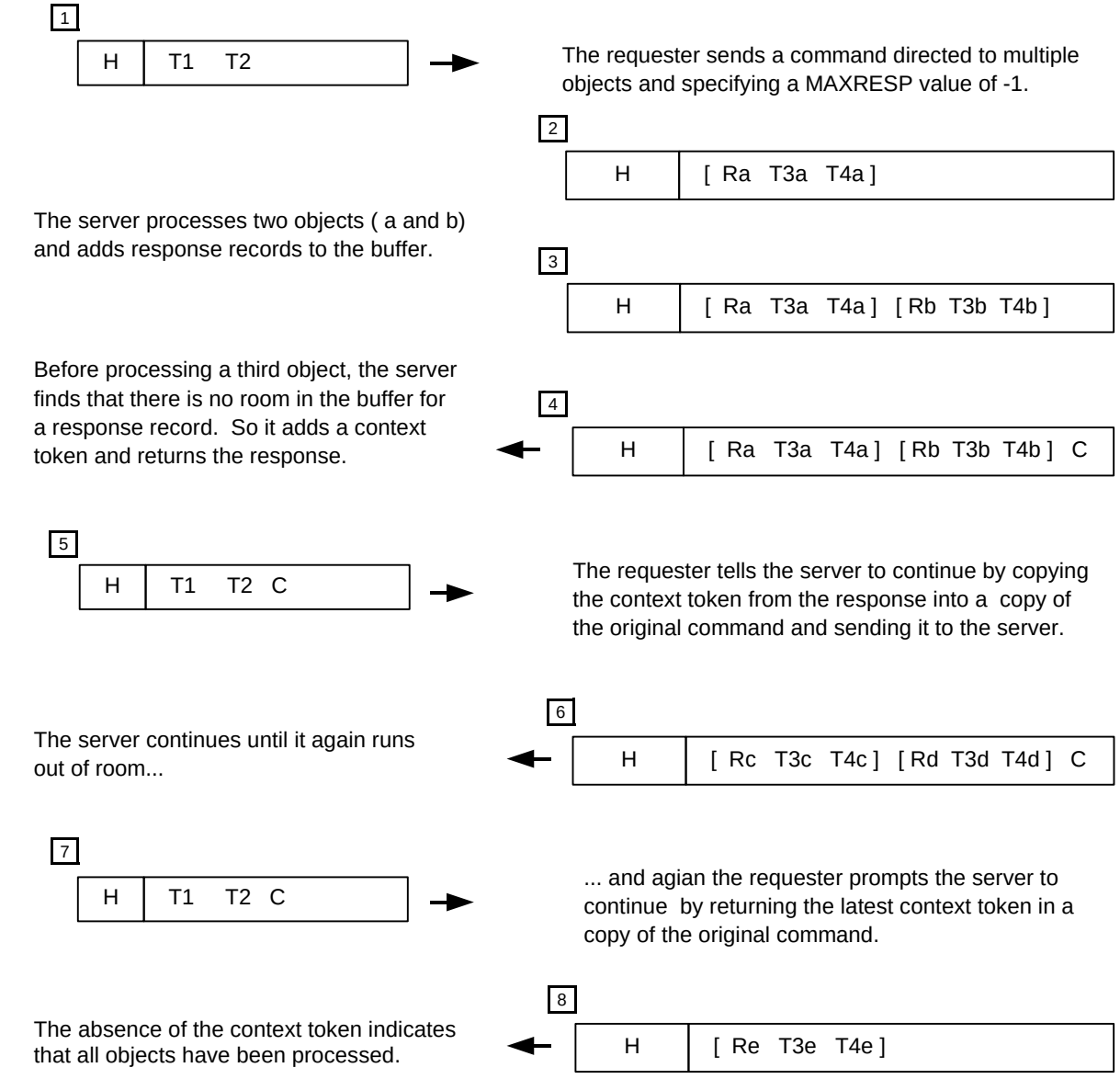
Multiple Response Records per Message in a Continued Response

Even with multiple response records per response message, it still might not be possible to fit the response records for all the objects into a single response message. In such cases, the subsystem forms a continuation response by including the context token, just as it would in the case of a single response record per response.

Figure 2-11 on page 2-36 shows that in the case of multiple response records per response, as in the case of a single response record, the presence of a context token indicates that the response can be continued. In a response message containing multiple response records (or a single response record enclosed in a data list), the context token is the only token that is not enclosed within a data list. The context token is not in a list because it represents an attribute of the entire response message rather than an attribute of a particular response record. As in the single-response-record case, the only valid way to detect that there are no more response messages is by the absence of a context token.

If the requester asks for *n* response records per response, but its buffer is not large enough to hold that many, the server does not consider the situation an error, but returns fewer than *n* response records per response—only the number that fit. The requester can determine the number of response records actually returned by counting the top-level data lists in the response message.

Figure 2-11. Response Continuation



Legend

H	Message Header	T*nx*	Token *n* describing object *x*	[	ZSPI-TKN-DATALIST
R*x*	RETCODE for object *x*	C	ZSPI-TKN-CONTEXT	]	ZSPI-TKN-ENDLIST

VST015.vsd

Response Continuation Protocol

These steps summarize the protocol for continuing a response. Each step corresponds to an item in Figure 2-11 on page 2-36:

1. The requester composes a command that is to be applied to multiple objects. In this example, the requester specifies a value of –1 for ZSPI-TKN-MAXRESP, telling the server to return as many response records as it can fit in each response message. Anticipating numerous response records and the possibility that more than one response message will be needed to return all of the response records, the requester saves a copy of the command buffer before sending the command. (The reason for this is explained in step 5.) The requester then sends the command to the server.
2. When the server finds an object (object *a*) that qualifies for processing based on the criteria in the command, it determines whether there is room in the buffer for both the largest possible response record and the largest possible context token. If there is room, the server applies the command to the object and adds the response record to the buffer. MAXRESP = –1, so the record is enclosed in a data list.
3. The server searches for another qualifying object, and finds object *b*. Again, the server verifies that both the largest possible response record and largest possible context token fit in the remaining buffer, and then applies the command and composes the response record.
4. The server continues to look for qualifying objects, and finds object *c*. This time, however, there is not enough space remaining in the buffer for a response record and a context token, so the server does not apply the command. Instead, it collects the information it will need to resume processing with this object, stores this information in ZSPI-TKN-CONTEXT, and returns this token with the completed response records.
5. Detecting the context token in the response message, the requester knows that not all qualifying objects were processed. To have the server continue processing, the requester copies the context token from the response to a duplicate of the original command (which it saved in step 1) and sends this command to the server. The requester does not examine or modify the contents of the context token.
6. When the server receives the command, it uses the information in the context token along with the information in the original command to resume processing. In this example, the server processes two more objects, *c* and *d*, before it again runs out of buffer space. The server prepares a new context token and returns it in the response message.
7. The requester, seeing that more objects remain to be processed, returns the new context token in a copy of the original command.
8. The server resumes processing, applying the command to one last object (*e*) and returning the response record for that object. The absence of the context token

from this response message tells the requester that all qualifying objects have been processed and the response for this command is complete.

Segmented Responses

A command might generate a response record that contains so many occurrences of a particular token or group of tokens that the record does not fit in even the largest allowable message buffer. This can occur, for example, when a requester sends a LISTOPENS command to a process that is supporting many openers, or when it sends a LISTOBJECTS command to a subsystem with many defined objects. A server can avoid truncating this type of lengthy response by breaking the response record into segments and returning the segments in multiple messages—a method called “response segmentation.”

In a segmented response, each response record is divided into one or more segments. (Figure 2-12 on page 2-39 shows the structure of a segmented response.) Each segment is an SPI data list that starts with ZSPI-TKN-DATALIST and ends with ZSPI-TKN-ENDLIST. Each also contains ZSPI-TKN-RETCODE.

Tokens in the response are divided into a base group, which appears once, and a repeating group, which can appear more than once. A token is included in the base group if it appears in the response record a predefined number of times. A token is included in the repeating group if there is no way to know, before the response is generated, how many times the token will appear.

All base-group tokens appear in the first segment, and are not repeated in later segments of the record.

Each instance of the repeating group is enclosed in a segment list, which starts with ZSPI-TKN-SEGLIST and ends with ZSPI-TKN-ENDLIST. Segment lists follow the base-group tokens. A segmented response always contains at least one segment list (which is empty if the server has no information to return).

If a segment does not complete a response record, it contains ZSPI-TKN-MORE-DATA with a value of TRUE, indicating that more segment lists are available. The message containing this segment also contains ZSPI-TKN-CONTEXT so the requester can follow the standard protocol for continuing the response. (See Continued Responses on page 2-34.)

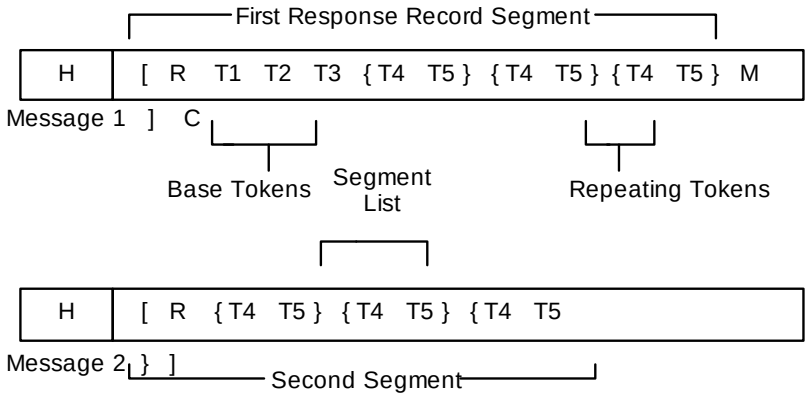
ZSPI-TKN-MORE-DATA does not appear in the last segment (or if it does appear, it has the value FALSE), indicating that this segment completes the response record.

A response message can contain more than one segment. However, within the context of a command and its continuations, all segments describing one object must be returned before any segment describing another object. To ensure this, ZSPI-TKN-MORE-DATA is only allowed in the last segment in a response message.

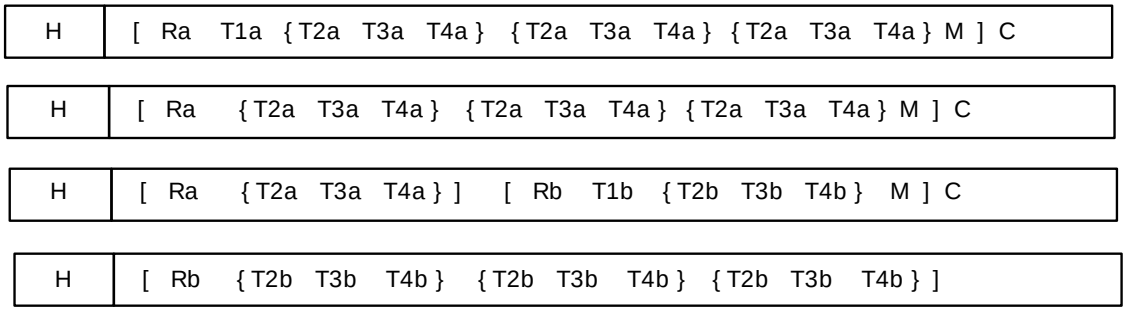
Error lists describing the object to which the command is directed can be included in a segment. Error lists related to information in a particular segment list can be included in that segment list.

Figure 2-12. Segmented Responses

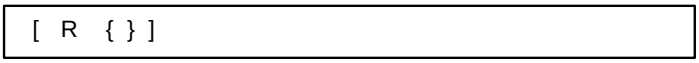
Structure of a Simple Segmented Response:



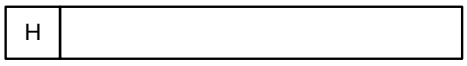
Example: A segmented response for two objects ***a*** and ***b***. The response record for ***a*** requires three segments; the third segment is returned in the same message as the first segment of the response record for object ***b***:



Example: An empty segmented response, which must contain an empty segment list:



Legend



An SPI message

T*n* A token with ID number *n*
T*nx* A token that refers to object *x*
H The SPI message header

R ZSPI-TKN-RETCODE
R*x* Return code for object *x*

[ZSPI-TKN-DATALIST
{ ZSPI-TKN-SEGLIST
] } ZSPI-TKN-ENDLIST
M ZSPI-TKN-MORE-DATA
C ZSPI-TKN-CONTEXT

VST016.vsd

Determining Subsystem Support for Segmented Responses

A subsystem that returns ZSPI-TKN-SEGMENTATION with a value of TRUE in its GETVERSION response can generate segmented responses.

A subsystem that supports segmentation for any command must support segmentation for all commands.

Requesting a Segmented Response

To request a segmented response, a command must include ZSPI-TKN-ALLOW with the value ZSPI-VAL-ALLOW-SEGMENTS. Otherwise, segmentation is disallowed. A requester must allow or disallow segmented responses for each command it issues. The default is to disallow segmented responses.

Table 2-2 summarizes possible subsystem responses to a request to allow or disallow segmentation. Response varies, depending on the content of the request and the subsystem's level of support for segmented responses.

Table 2-2. Subsystem Response to Requests for Segmented Responses

Does the command contain ZSPI-TKN-ALLOW with value ZSPI-VAL-ALLOW-SEGMENTS?	If subsystem supports segmentation…	If subsystem does not support segmentation…	If subsystem requires requester to accept segmented responses…
Yes	The subsystem returns a segmented response.	The subsystem returns an "invalid token" error after finding ZSPI-TKN-ALLOW with value ZSPI-VAL-ALLOW-SEGMENTS.	The subsystem returns a segmented response.
No	The subsystem returns a response record that is truncated if it does not fit in the buffer.	The subsystem returns a response record that is truncated if it does not fit in the buffer.	The subsystem returns ZSPI-ERR-MISTKN after failing to find ZSPI-TKN-ALLOW with value ZSPI-VAL-ALLOW-SEGMENTS.

Empty Responses

An empty response consists of a single response message that contains no response records. An empty response must contain a return code (ZSPI-TKN-RETCODE) with a subsystem-defined error number that declares that the response message contains no response records. In special cases, an “empty response” might also contain error information describing the command. (See point 3.)

An empty response message does not always indicate an empty response. An empty response message is sometimes needed to complete a continued response, in which case only the last message of the response is empty, but not the response itself (see point 2 below).

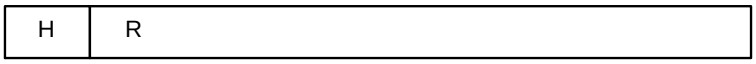
A server returns an empty response message when (see Figure 2-13 on page 2-42):

1. The requester sets ZSPI-TKN-RESPONSE-TYPE to ZSPI-VAL-ERR-AND-WARN, and the server detects no errors or warnings during command processing. In this case, the requester has asked the server to return a response record only if an error or warning is detected while applying the command to an object. Because no errors or warnings are detected, the server returns an empty response. (For information about the RESPONSE-TYPE token, see Suppressing Response Records on page 2-43.)
2. Objects awaiting command continuation are deleted before the server sends the continuation request. This is possible any time the server returns a partial response and has additional objects to process. The context token in the partial response identifies the next object to be processed. If the objects awaiting processing are deleted before the requester sends the continuation command to the server, the server finds no objects to process and returns an empty response message to complete the continued response.
3. The requester sets ZSPI-TKN-RESPONSE-TYPE to ZSPI-VAL-ERR-AND-WARN and the server detects no errors or warnings when applying the command to individual objects, but does want to return warning information describing the command in general. In this case, the server returns the warning information in the “empty” response message. (It is still called an “empty” response message because ZSPI-TKN-RETCODE contains the value indicating an empty response.)

Figure 2-13. Empty Responses

Three situations in which a server returns an empty response:

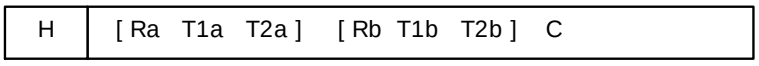
1. The requester sets ZSPI-TKN-RESPONSE-TYPE to ZSPI-VAL-ERR-AND-WARN, and no errors or warnings are encountered during command processing:



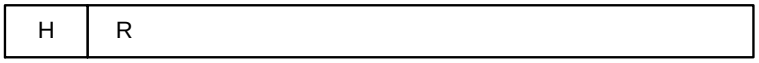
The server returns an empty response.

2. Objects not processed for one response message are deleted before the requester returns the continuation command:

Objects to process: a, b, c object c awaits command continuation

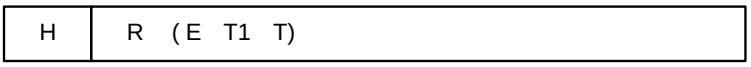


Objects to process: c (object c deleted before command is continued)



The second response message is empty.

3. The requester sets ZSPI-TKN-RESPONSE-TYPE to ZSPI-VAL-ERR-AND-WARN, and warnings are detected that describe the command in general, but not the application of the command to any particular object:



The error information is returned along with a return code indicating an empty response.

Legend

H	Message header	T*nx*	Token *n* describing object *x*	[	ZSPI-TKN-DATALIST
R	ZSPI-TKN-RETCODE	C	ZSPI-TKN-CONTEXT	(	ZSPI-TKN-ERRLIST
R*x*	Return code for object *x*	E	ZSPI-TKN-ERROR	])	ZSPI-TKN-ENDLIST

VST017.vsd

Object Identification in Responses

A response record must contain a subsystem-defined token identifying the object that the response record describes. This object-name token appears even if the response contains an error list that includes the object name. The exceptions to this rule are:

- If the response record corresponds to a command that does not have an object name, the response record does not include an object name.
- If the response record corresponds to a command that operates on an unnamed object and the command includes an object-name token set to some null value (such as all blanks), the response record contains an object name in the same form.
- If the response record reports an error that prevents the command from being attempted at all, the response record does not necessarily include an object name.
- If the response record is the empty response record sometimes needed with continuation, the object-name token does not appear.

Return Code

Every response record contains a return code token (ZSPI-TKN-RETCODE). The value of this token indicates whether the command completed successfully on the object described by the response record. ZSPI-TKN-RETCODE can contain these values:

zero	indicates that the command was successfully applied to the object. Such a response record can contain error lists describing warnings or other conditions of interest to the requester.
nonzero	indicates either an empty response or that the server could not perform the command as expected. The particular nonzero value is a subsystem-defined error number identifying the error, or one of the errors, that the server encountered.

Except in the case of an empty response, every response record that contains a nonzero return code also contains an error list in which ZSPI-TKN-ERROR contains the same nonzero value. This error list might also contain additional information describing the error condition.

Suppressing Response Records

For commands that operate on more than one object, some subsystems support a standard SPI token—the response-type token (ZSPI-TKN-RESPONSE-TYPE)—that lets the requester specify which kinds of response records the server should return. This kind of control is useful if an application is performing a command on a large number of objects and is interested only in response information about objects for which something unusual occurred (an error or a warning).

The two possible values for ZSPI-TKN-RESPONSE-TYPE are:

- ZSPI-VAL-ERR-WARN-AND-NORM directs the server to return a response record for every object in the set of objects specified in the command. This action is the default if the response-type token is not included in the command.
- ZSPI-VAL-ERR-AND-WARN directs the server to return response records only for objects for which something unusual occurred—that is, response records containing at least one error list (regardless of the value of the return token). If ZSPI-VAL-ERR-AND-WARN is specified and the command encounters no errors or warnings on any object, the subsystem returns an empty response record. If ZSPI-VAL-ERR-AND-WARN is specified and there are warnings about the command itself, the server holds these command-related warnings until it generates a response record due to an error or warning on one of the objects. If no warnings or errors are generated on any of the objects, the server places the command warnings in an empty response record.

Subsystems that support the response-type feature do so for all commands that can change the state or configuration of an object and can accept the specification of more than one object in a single command. Subsystems that do not support this feature always return a response record for each object.

Some subsystems support the response-type feature even for informational commands. In most situations, you will not want your application to suppress normal responses for such commands.

Subsystem IDs (SSIDs)

A subsystem ID (SSID) is a structure that uniquely identifies a subsystem. SPI, the Event Management Service (EMS), and the Distributed Name Service (DNS) all use the same SSID format.

In SPI commands and responses, SSIDs identify both the subsystem that is to process a command and the subsystems that put each token in the message. All tokens in an SPI message are associated with a subsystem ID so that it is always possible to determine who put the token in the message. An SSID is a 12-byte structure with the format shown in Figure 2-14 on page 2-46.

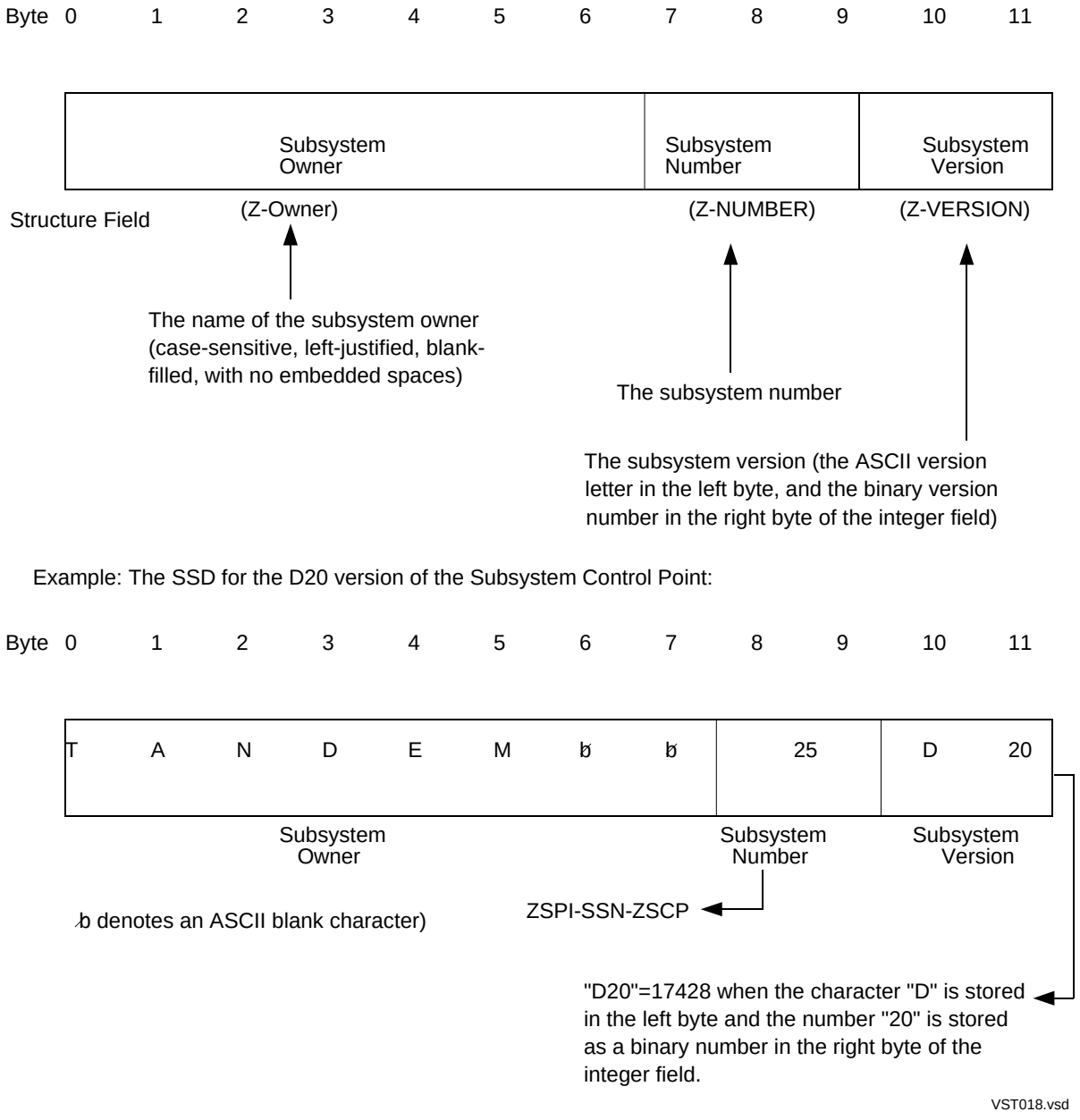
The subsystem owner field (Z-OWNER) contains an eight-character string that identifies the company or organization providing the subsystem. For all NonStop Kernel subsystems, this field contains the string value “TANDEM~~bb~~” (with two trailing blanks).

The subsystem number field (Z-NUMBER) is a 16-bit signed integer value that identifies the subsystem within the set of subsystems provided by the subsystem owner. The SPI standard definition files include subsystem-number declarations for all NonStop Kernel subsystems that have programmatic command interfaces based on SPI, report EMS events, or define errors to be passed through by other subsystems in SPI error lists. (For a list of these subsystems and their symbolic abbreviations, see Appendix D, NonStop Kernel Subsystem Numbers and Abbreviations.)

The subsystem version field (Z-VERSION) is a 16-bit unsigned integer value representing the software release version of the subsystem. For NonStop Kernel subsystems, this value is in the form returned by the TOSVERSION procedure: that is, the left byte contains the letter part of the version as an ASCII uppercase alphabetic character, and the right byte contains the numeric part of the version as an unsigned integer value. For example, for the version “G06” the left byte is the ASCII character *G* and the right byte is 06, so the result is the unsigned integer 18182.

Each NonStop Kernel subsystem defines a subsystem ID structure giving the values of all three fields for that subsystem. This structure has the name *subsys*-VAL-SSID, where *subsys* is the subsystem abbreviation.

Figure 2-14. The Subsystem ID Structure



SSID Scope

Subsystem IDs are stored in the header token ZSPI-TKN-SSID, in some token codes, and in the error list token ZSPI-TKN-ERROR. The SSID that governs a particular token is determined as follows:

- If the token code contains an SSID, that SSID owns the token.
- If the token code does not contain an SSID, the token belongs to the SSID of the list in which the token is enclosed. In the case of nested lists, the SSID of the token is that of the list that immediately encloses the token.

- If the token is not enclosed in a list and its token code does not contain an SSID, it belongs to the SSID specified in the header token ZSPI-TKN-SSID.

Errors and Warnings

An SPI server reports an error when it cannot complete a command. It issues a warning when it completes a command but the results are suspect or it wants to provide the requester with additional advisory information. Like all response information, error information returned in a message is contained in tokens. However, although normal responses tend to be very regular and well-defined, error responses can be highly variable and unpredictable. As a result of an error or warning during command processing, a response message can contain:

- Both response data and error or warning information.
- A single response record containing multiple error lists, particularly in the case of warnings.
- A pass-through error, one that originated in another subsystem that was called by the subsystem to which the command was sent. Errors can be passed through several subsystems, with each subsystem adding its own information. For instance, here is an extreme example of an error that FUP might report in response to a LOAD command: “FUP could not do the LOAD because of a SORT error caused by a NEWPROCESS procedure failure due to a file-system error on the swap file returned by the disk process because there was no disk space available.”

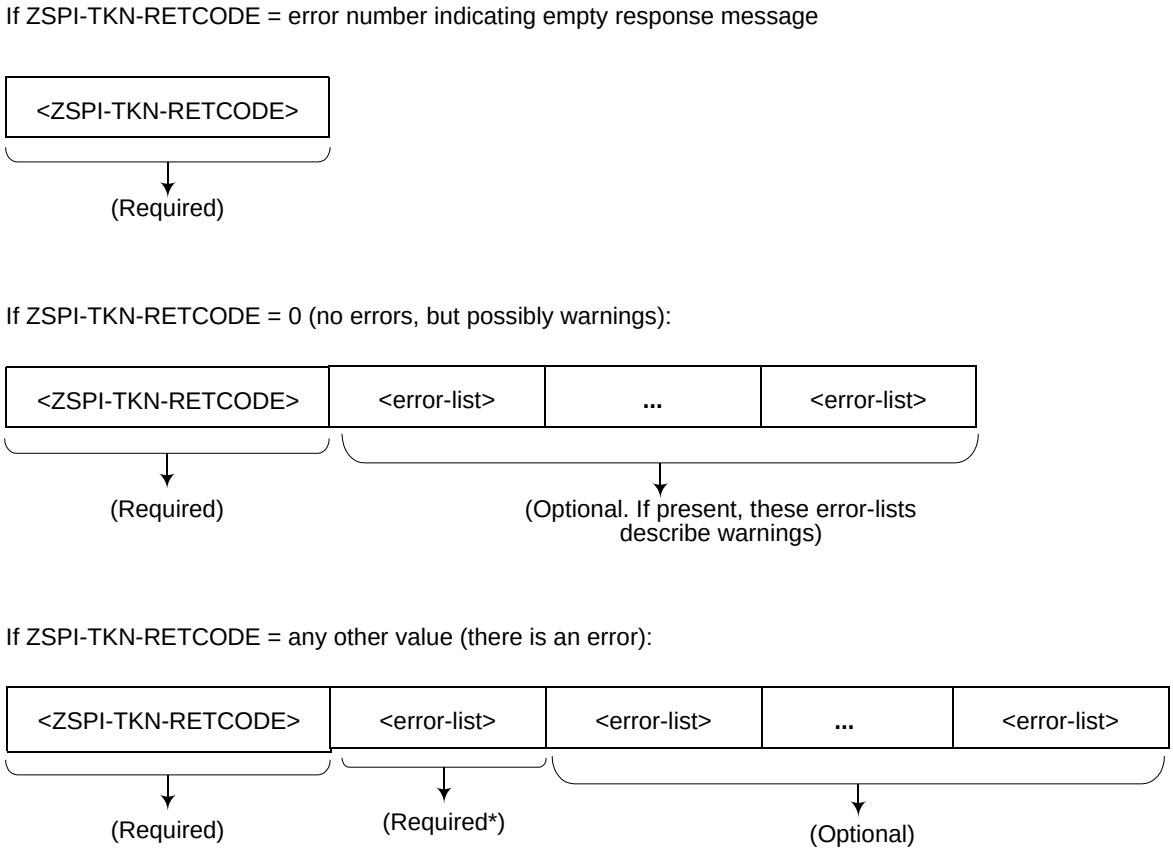
SPI responses report error information in these ways:

- A standard token called the return token (ZSPI-TKN-RETCODE), which is included in every response record. The value of ZSPI-TKN-RETCODE is a single integer error code.
- One or more error lists, which can be nested (as in the case of pass-through errors).

This scheme, summarized in Figure 2-15, separates error information from normal response information and facilitates the retrieval of error information.

Each subsystem defines its own set of error numbers, which are described in the subsystem’s management programming manual.

Figure 2-15. Error Information in a Response Record



Error Lists

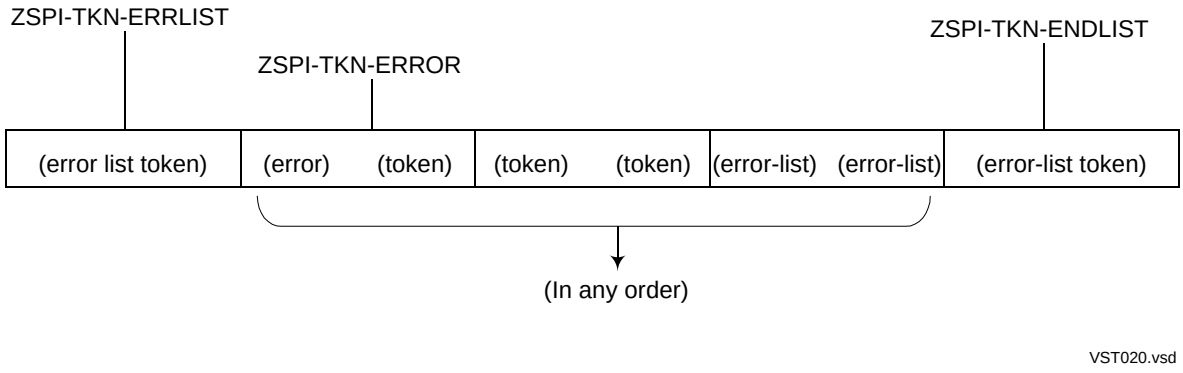
Except in the case of the empty response record used in continuation, if the return token is nonzero, the response record must include at least one error list. This error list must contain an error token (ZSPI-TKN-ERROR) with an error number that matches the value of the return code. The response record can contain other error lists as well.

Error lists can be present in a response record even if the value of the return token is zero. Such lists describe warnings; these are unusual conditions about which the requester might want to know, but which do not prevent the server from completing the command.

Normal response tokens are returned outside of error lists; but if they are necessary to describe the error, these tokens are repeated inside the relevant error lists. Each error list is a complete, self-contained description of an error.

Each error list contains an error token (ZSPI-TKN-ERROR) that identifies the error described by other tokens in the list. ZSPI-TKN-ERROR is a fixed structure consisting of the subsystem ID of the subsystem that reported the error and the error number of an error defined by that subsystem. (The version portion of the subsystem ID in an error token can be either zero or the version of the subsystem reporting the error.) Every error list must contain one error token, and it can also contain additional tokens and embedded error lists giving more information about the error.

Figure 2-16. Error List Contents



Pass-Through Errors

A subsystem can pass on an error list from another subsystem; in general, the error list describing the pass-through error is qualified by a subsystem ID other than that of the subsystem composing the response message. If an error list is qualified by the subsystem ID of another subsystem, all tokens within those lists are implicitly qualified by that subsystem ID.

Some software facilities for the NonStop server, including some operating system procedures and the file system, do not have a programmatic command interface based on SPI, but do define standard error lists. If a NonStop Kernel subsystem that does have an SPI-based command interface calls one of these facilities and receives an error, the subsystem usually tries to recover from the error. If that is impossible, the subsystem reports the error in an SPI error token (ZSPI-TKN-ERROR), embeds it in an error list of the prescribed form, and encloses this error list in its response. This practice ensures that errors from NonStop Kernel subsystems and software components are reported in a consistent way.

The SPI procedures detect certain errors, which are reported in the status parameter of the procedure call. When a NonStop Kernel subsystem calls one of the SPI procedures and receives a nonzero status from which it (the subsystem) cannot recover, the subsystem returns to the application an error list containing information about the error. For descriptions of the SPI errors, see Appendix A, Errors. For descriptions of the error lists returned by NonStop Kernel subsystems when they encounter an SPI procedure error from which they cannot recover, see the *Guardian Procedure Errors and Messages Manual*. To identify which errors can occur on calls to each procedure, see the SPI procedure descriptions in Section 3, The SPI Procedures, and the

corresponding TACL built-in function descriptions in Section 8, SPI Programming in TACL.

Continuing Despite Errors

Some subsystems support another standard SPI token to let the requester specify under what conditions the server should continue processing a set of objects. This is the allow-type token (ZSPI-TKN-ALLOW-TYPE). This token is useful when the application is performing a command on a large number of objects (for instance, terminals) and wants to operate immediately on as many of those objects as it can, even if errors or warnings occur for some objects. In such cases, the application can address the cases with errors or warnings later.

ZSPI-TKN-ALLOW-TYPE has three possible values:

- ZSPI-VAL-NORM-ONLY directs the server to continue processing on the next object in the set only if the command was completely successful, with no warnings, on the previous object—that is, if the response record on that object contained no error list. This action is the default if the allow-type token is not included in the command.
- ZSPI-VAL-WARN-AND-NORM directs the server to continue processing on the next object if the command completed its operation on the previous object, regardless of warnings—that is, if the value of the return token was zero (error lists representing warnings can be present).
- ZSPI-VAL-ERR-WARN-AND-NORM directs the server to continue processing on the next object regardless of any problems encountered on the previous object.

If a condition occurs that, based on the allow-type value, directs the subsystem not to process the next object, the server completes the response record for the object on which the unusual condition occurred (if it has not already been completed), adds a context token to the response if appropriate, and sends the response to the requester. The requester can take appropriate corrective action for the error or warning, and then reissue the command to have the subsystem continue processing with the next object in the set.

The allow-type feature controls only whether the server proceeds to the next object in a set when the action of a command on one object yields an error or warning. This feature does not influence the action on an object after the server has begun work on that object. For example, selecting ZSPI-VAL-NORM-ONLY does not mean that the server should stop the operation on an object immediately when a warning condition is detected. The operation on that object still continues to completion. After completing with that object, the server notices that a warning occurred on the object, and it issues a response at that point.

NonStop Kernel subsystems that support the allow-type feature do so for all commands that can cause some action or modification of an object and can accept the specification of more than one object in a single command. NonStop Kernel

subsystems that do not support this feature do not proceed to the next object in the set if any error or warning occurred on the previous object.

Recovering From an Error on an Object in a Set

The use of the context token for continuation allows for resuming processing after an error, starting with the object following the one on which the error occurred. That makes it easy for your application to continue the command immediately.

If preferred, your application can instead correct the problem that caused the error and have the command continue with the object that got the error. If the server is context free, the requester can issue any number of commands between the receipt of a response with a context token and the sending of the continuation that includes that context. Your application can correct the problem immediately by issuing commands naming the particular object that failed. When everything is resolved for that object (including sending a command duplicating the original command that failed, but directed just to that one object), your application can use the context token to continue the original command with the next object.

Sample Error Responses

In these examples, indenting shows the nesting levels of error lists and the names of fields within fixed structures.

Error List Example 1

A simple error response from subsystem SYSB:

```
<ZSPI-TKN-RETCODE>                      ! n (command not
supported)
<ZSPI-TKN-ERRLIST>
  <ZSPI-TKN-ERROR>
    Z-SSID                              ! Subsystem ID for SYSB
    Z-ERROR                             ! n (command not
supported)
<SYSB.ZSPI-TKN-ENDLIST>                 ! end of error list
```

In this example, subsystem SYSB returns a nonzero return code indicating that it does not support the command specified by the requester in the original request. The Z-ERROR field of the error token in the corresponding error list has the same error number, indicating that it contains information about the error.

Error List Example 2

Single error list (subsystem SYSB):

```
<ZSPI-TKN-RETCODE>                      ! m (value out of range)
<ZSPI-TKN-ERRLIST>
  <ZSPI-TKN-ERROR>
    Z-SSID                              ! Subsystem ID for SYSB
    Z-ERROR                             ! m (value out of range)
  <ZSPI-TKN-PARM-ERR>
    Z-TOKENCODE                         ! token code in error (or
                                        !   first 32 bits of token
                                        !   map)
    Z-INDEX                             ! index of that token code
    Z-OFFSET                            ! offset of field in error
                                        !   if token is a structure
<SYSB.ZSPI-TKN-ENDLIST>                 ! end of error list
```

In this example, the ZSPI-TKN-PARM-ERR token, whose structure is defined in Section 4, ZSPI Data Definitions, is used to identify which command-parameter token contained the out-of-range value. The index value indicates the position of that token in the buffer, and the offset denotes the position of the out-of-range field within that token if the token value is a structure.

Error List Example 3

Nested error lists (subsystems FUP and FILESYS):

```
<ZSPI-TKN-RETCODE>                       ! x (FUP command failed
                                         !   with file-system
error)
<ZSPI-TKN-ERRLIST>
  <ZSPI-TKN-ERROR>
    Z-SSID                               ! Subsystem ID for FUP
    Z-ERROR                              ! x (FUP command failed
                                         !   with file-system
error)
  <ZFUP-MAP-CMD-ERROR>
    Z-COMMAND                            ! LOAD command
    Z-OBJECT                             ! FILE object type
    Z-NAME                               ! Name of file on which
                                         !   error occurred
  <ZSPI-TKN-ERRLIST>                     ! List from file system
    <ZSPI-TKN-ERROR>
      Z-SSID                             ! SSID of file system
      Z-ERROR                            ! y (file-system error)
    <ZSPI-TKN-PROC-ERR>                  ! WRITE procedure
  <ZSPI-TKN-ENDLIST>                     ! End FILESYS error list
<ZSPI-TKN-ENDLIST>                       ! End FUP error list
```

In this example, a FUP LOAD command failed because a file-system error occurred on a WRITE procedure call. Because the file system does not have an SPI interface, FUP constructs an error list, on behalf of the file system, to describe the file-system error. To

indicate that this error list describes an error reported by the file system, FUP gives the error list the subsystem ID of the file system.

Error List Example 4

Nested error lists (subsystems FUP, SORT, and GUARDLIB):

```
<ZSPI-TKN-RETCODE>                  ! y (FUP command failed with
                                    !    SORT error)
<ZSPI-TKN-ERRLIST>
  <ZSPI-TKN-ERROR>
    Z-SSID                          ! Subsystem ID for FUP
    Z-ERROR                         ! y (FUP command failed with
                                    !    SORT error)
  <ZFUP-MAP-CMD-ERROR>
    Z-COMMAND                       ! LOAD command
    Z-OBJECT                        ! FILE object type
    Z-NAME                          ! name of file FUP was unable
                                    !    to load
  <ZSPI-TKN-ERRLIST>                ! List from SORT
    <ZSPI-TKN-ERROR>
      Z-SSID                        ! Subsystem ID for SORT
      Z-ERROR                       ! 64 (cannot allocate segment)

    <ZSPI-TKN-ERRLIST>              ! List from operating system
      <ZSPI-TKN-ERROR>
        Z-SSID                      ! SSID for procedure library
        Z-ERROR                     ! 43 (no space on disk)
      <ZSPI-TKN-PROC-ERR>           ! ALLOCATESEGMENT
    <ZSPI-TKN-ENDLIST>              ! end GUARDLIB error list
  <ZSPI-TKN-ENDLIST>                ! end SORT error list
<ZSPI-TKN-ENDLIST>                  ! end FUP error list
```

In this example, a FUP LOAD command failed because SORT could not allocate an extended segment. Again, because SORT and the system library procedures (including ALLOCATESEGMENT) do not have programmatic command interfaces based on SPI, FUP constructs error lists on their behalf, nesting the error lists to show the pass-through relationships.

3 The SPI Procedures

This section describes these SPI procedures:

This section describes the procedures and their parameters in a general context. The exact syntax of an SPI procedure call depends on the programming language you use. For the syntax necessary in a particular programming language, see the language-specific sections of this manual.

Overview of the SPI Procedures

Both SPI requesters and servers use the SSINIT, SSNULL, SSPUT, SSGET, and SSMOVE procedures to initialize messages and extensible structured tokens, place tokens in messages, retrieve token values from messages, and move tokens from one message buffer to another. The functions of these five basic procedures are:

SSINIT	Initializes an SPI message buffer
SSNULL	Initializes an extensible structured token with null values
SSPUT	Puts a token into a buffer or performs a resetting or repositioning operation on the buffer
SSGET	Extracts a token value or other information from the buffer
SSMOVE	Copies a token or list from one buffer to another

These variants perform the same functions as their namesakes, differing only in the way they refer to token codes in the procedure parameters:

SSPUTTKN	Adds a token to the buffer or performs a resetting or repositioning operation on the buffer
SSGETTKN	Extracts a token value or other information from the buffer
SSMOVETKN	Copies a token or list from one buffer to another

The SSIDTOTEXT and TEXTTOSSID procedures perform these functions:

SSIDTOTEXT	Converts an internal form subsystem ID to its external representation
TEXTTOSSID	Finds an external representation of a subsystem ID and converts it to the internal representation

Special Operations

The SSPUT(TKN) and SSGET(TKN) procedures can perform special operations in addition to their primary functions. Applications request special operations by specifying special token codes in the procedure calls. Special token codes and the operations they request are listed with each procedure description.

Manipulating Header Tokens

The SSPUT(TKN) and SSGET(TKN) procedures can also modify and retrieve the values of some SPI message header tokens, as indicated in the procedure descriptions.

Procedure Status

The procedure calls return a status code that summarizes the outcome of procedure processing:

0	No error
–1	Invalid buffer format
–2	Invalid parameter value
–3	Missing parameter
–4	Invalid parameter address
–5	Buffer full
–6	Invalid checksum
–7	Internal error
–8	Token not found
–9	Invalid token code or map
–10	Invalid subsystem ID
–11	Operation not supported
–12	Insufficient stack space

These codes are described in Appendix A, Errors.

Using the SPI Procedures

The SPI procedures are in the standard operating-system library and can be called through:

- The **tal** interface directive in C (See Section 6, SPI Programming in C.)
- ENTER TAL from COBOL (See Section 7, SPI Programming in COBOL.)
- Corresponding built-in functions from TACL (See Section 8, SPI Programming in TACL.)
- Directly from TAL (See Section 9, SPI Programming in TAL.)

To issue a command to an SPI server, a requester first calls the SSINIT procedure, supplying the buffer, buffer length, subsystem ID, command, and object type if needed. SSINIT initializes the buffer, placing the supplied information in the appropriate fields of the message header.

Before adding an extensible structured token to a message, an application must call the SSNULL procedure to initialize the fields of the structure to null values. A program must always call SSNULL for each extensible structure to ensure version compatibility, even if the program explicitly sets all currently defined fields. Then the program sets the desired fields of each structure.

The application then calls SSPUT to assign values to tokens and add the tokens to the message.

When the message is complete, the application sends it to the server.

When the server receives the message, it calls the SSPUT procedure to reset control information in the buffer, uses SSGET to extract relevant token values, and performs the processing requested by the message. The server uses SSINIT to initialize a response buffer, uses SSPUT to place a return-code token and other response tokens in the buffer, and then returns the response message to the requester.

When it receives the response message, the requester calls SSPUT to reset the buffer, and then uses SSGET to extract token values. The requester can retrieve token values in any order.

In addition to their main functions, SSPUT and SSGET accept special token codes that allow an application to retrieve or modify the values of header tokens, scan the data portion of the buffer token by token, and perform control and positioning operations on the buffer.

Finally, programs can use the SSMOVE procedure to copy any number of tokens from one SPI buffer to another. This is useful when copying context information from a continuation response to a followup command message or when forwarding error lists from other subsystems.

SSINIT Procedure

An application must use the SSINIT procedure to initialize a command buffer. The SSINIT procedure initializes an SPI buffer with an appropriate header and places values in header fields. The previous contents of the buffer are overwritten.

Do not use SSINIT to initialize an event-message buffer. Instead, use EMSINIT, as described in the *EMS Manual*.

General Syntax

```
SSINIT    (     buffer                                  ! o
            ,   buffer-length                           ! i
            ,   ssid                                    ! i
            ,   header-type                             ! i
            , [ command             ]                   ! i
            , [ object-type         ]                   ! i
            , [ max-resp            ]                   ! i
            , [ server-version      ]                   ! i
            , [ checksum            ]                   ! i
            , [ max-field-version ])                    ! i
```

buffer output

INT .EXT:ref:*

is the buffer that the procedure initializes as an SPI message buffer.

buffer-length input

INT:value

is the buffer length in bytes. Use the length recommended by the subsystem to which you are sending the message, if one is defined. For NonStop Kernel subsystems, the recommended buffer length has a name of the form *subsys*-VAL-BUFLEN.

ssid input

INT .EXT:ref:6

is the subsystem ID of the subsystem to which the message is sent. Its structure is described in Section 2, SPI Concepts and Protocol. Requesters use this value to identify the target subsystem, and the version field of the SSID must specify the version of the subsystem definitions that the requester is using. Servers check the SSID to verify that they are the intended recipient of the message.

Use the SSID defined by the subsystem to which you are sending the message. NonStop Kernel subsystems provide an SSID with a name of the form *subsys*-VAL-SSID.

header-type input

INT:value

specifies the type of SPI buffer to initialize. This parameter determines how SSINIT interprets the parameters that follow it. You should always give the value ZSPI-VAL-CMDHDR (0), indicating the standard command header (for a command or response). Other values are used internally by software for the NonStop server.

command input

INT:value

is the command number. If not supplied, it defaults to zero.

object-type input

INT:value

is the object type. If not supplied, it defaults to ZSPI-VAL-NULL-OBJECT-TYPE.

max-resp input

INT:value

is the maximum number of response records to be returned by the subsystem in each reply message. A value of zero (the default) specifies one response record per reply, not enclosed in a list. Any positive value specifies up to that number of response records, each enclosed in a list. A value of -1 specifies as many response records as will fit, each enclosed in a list.

server-version input

INT:value

is normally provided only by subsystems or by other programs that are acting as a server. In those cases, it is a 16-bit unsigned integer value representing the version of the subsystem or server program. SSINIT places this value in the header token ZSPI-TKN-SERVER-VERSION for use in version compatibility checking. If not supplied, this version number defaults to zero.

checksum input

INT:value

is the checksum flag. If this parameter is zero or not supplied, checksum protection of the buffer is disabled; if it is nonzero, checksum protection is enabled.

max-field-version input

INT:value

is an unsigned integer value that initializes the maximum field version field of the buffer header. If it is not supplied, a default value of zero is used.

Note. The procedure can be called from both 32-bit and 64-bit programs.

SSNULL Procedure

The SSNULL procedure initializes an extensible structure with null values. Always use this procedure before setting values within an extensible structured token. For more information about null values, see Section 5, General SPI Programming Guidelines.

General Syntax

```
SSNULL    (    token-map                                  ! i
             , struct                                     ! o
           [ , constants  ]  )                            ! i
```

token-map

INT .EXT:ref:*

is a token map to be used in initializing the fields of the structure.

struct

STRING .EXT:ref:*

is the structure to be initialized with null values.

constants

FIXED .ref:1

is a pointer to the set of constant values returned by the XSTACKTEST procedure. If the constants are specified in the call, SSNULL does not need to call XSTACKTEST, which it must do if the constants are not provided. This parameter is used only by software that HP provides for the NonStop server.

Considerations

Always use SSNULL to initialize an extensible structured token, even if you put values into all of the structure's fields. Doing so ensures that your application continues to run correctly if new fields are later added to the structure.

SSPUT and SSPUTTKN Procedures

The SSPUT and SSPUTTKN procedures insert tokens in an SPI buffer previously initialized by SSINIT. The two procedures are identical except for the type of the *token-id* parameter (SSPUT passes *token-id* by reference and SSPUTTKN passes it by value) and the consequent fact that SSPUTTKN cannot be used with a token map.

General Syntax

```
SSPUT     (     buffer                                  ! i/o
SSPUTTKN    ,   token-id                                ! i
            , [ token-value ]                           ! i
            , [ count ]                                 ! i
            , [ ssid  ]         )                       ! i
```

buffer input, output

INT .EXT:ref:*

is the SPI buffer in which tokens are placed.

token-id input

INT .EXT:ref:* (SSPUT)
INT(32):value (SSPUTTKN)

is a token code or (for SSPUT only) a token map. This parameter either identifies the token being supplied or indicates a special operation. In the latter case, the interpretation of the *token-value* parameter can vary; see Special Operations With SSPUT and SSPUTTKN on page 3-9.

token-value input

STRING .EXT:ref:*

if present, is the value of the token. Its data representation is determined by the token-type field of the *token-id*.

count input

INT:value

is the token count. The *token-value* parameter is an array of *count* elements, each of which is described by the *token-id*. If not supplied, the count defaults to 1.

ssid input

INT .EXT:ref:6

is a subsystem ID that qualifies the token code. If *ssid* is not supplied or is equal to zero (6*[0]), the default applies. If SSPUT is currently adding tokens to a list, *ssid* defaults to the subsystem ID of that list; otherwise, *ssid* defaults to the subsystem ID in the SPI message header (ZSPI-TKN-SSID).

Note. These procedures can be called from both 32-bit and 64-bit programs.

Special Operations With SSPUT and SSPUTTKN

Table 3-1 lists the special tokens your programs can supply to SSPUT and SSPUTTKN to set or change the values of header tokens and perform other special operations. The token type listed in the table indicates the type of the *token-value* parameter, which is always an input parameter.

Each operation is described following the table. For some operations, the procedure parameters differ in type and meaning from those indicated in the main SSPUT(TKN) syntax description. When this is true, the modified syntax and semantics are given or the differences are described.

Table 3-1. SSPUT(TKN) Special Operations

Token Specified in SSPUT(TKN) Call	Type	Effect
ZSPI-TKN-BUFLEN	UINT	Modify buffer length
ZSPI-TKN-CHECKSUM	BOOLEAN	Enable or disable buffer checksum
ZSPI-TKN-CLEARERR	SSCTL	Clear last-error information to zero
ZSPI-TKN-DATA-FLUSH	SSCTL	Flush tokens starting at current position
ZSPI-TKN-DELETE	TOKENCODE	Delete a token from the buffer
ZSPI-TKN-INITIAL-POSITION	BOOLEAN	Reset position to start of buffer or list
ZSPI-TKN-MAX-FIELD-VERSION	UINT	Increase maximum field version
ZSPI-TKN-MAXRESP	INT	Set maximum-responses header token
ZSPI-TKN-POSITION	POSITION	Restore a previously saved position
ZSPI-TKN-RESET-BUFFER	UINT	Reset buffer
ZSPI-TKN-SERVER-VERSION	UINT	Set server-version header token

ZSPI-TKN-BUFLEN: Modify Buffer Length

Use this token code to modify the SPI buffer length. If the specified length is less than the actual number of bytes used in the buffer, as given in the header token ZSPI-TKN-

USEDLEN, the procedure returns ZSPI-ERR-NOSPACE. However, the procedure still resets the maximum buffer length in the SPI message header, and subsequent SPI calls for that buffer fails with ZSPI-ERR-INVBUF.

ZSPI-TKN-CHECKSUM: Set Checksum Flag

With this token code, a nonzero *token-value* enables checksum protection of the buffer; a zero *token-value* disables it.

ZSPI-TKN-CLEARERR: Clear Last SPI Error

Use this token code to clear the last-error information to zero. If supplied, the *token-value* parameter is ignored.

You might use this operation before issuing a series of SSPUT and SSGET calls that are followed by a check of the last error. You need this operation only if you use SSGET to check the header token ZSPI-TKN-LASTERR.

ZSPI-TKN-DATA-FLUSH: Clear Buffer From Current Position

Use this token code to flush all information in the message buffer located at and following the current-token pointer. If supplied, *token-value* is ignored. The ZSPI-TKN-DATA-FLUSH operation does not update the header token ZSPI-TKN-MAX-FIELD-VERSION. As a result, following this operation, that token can indicate a version higher than the version of any field remaining in the buffer.

ZSPI-TKN-DELETE: Delete a Token or List

Use this token code to delete a particular token or a list from the buffer. The call syntax is:

```
SSPUT       (      buffer
SSPUTTKN      ,    ZSPI-TKN-DELETE
              ,    token-code,                              ! i
              ,    index,                                   ! i
              , [  ssid ]             )                     ! i
```

token-code

is the token code of the token to be deleted. If *token-code* is ZSPI-TKN-LIST, -DATALIST, -ERRLIST, or -SEGLIST, the entire list is deleted.

index

INT:value

if greater than zero, specifies an absolute index for *token-id*, starting from the beginning of the buffer or current list. An *index* of one deletes the first occurrence of that token code, an *index* of two deletes the second occurrence, and so on.

if zero or not supplied, deletes the next occurrence of the token code at or following the current-token pointer in the buffer.

if less than zero, returns an error.

ssid

can be included to qualify *token-code*.

The ZSPI-TKN-DELETE operation does not update the header token ZSPI-TKN-MAX-FIELD-VERSION. As a result, following this operation, that token can indicate a version higher than the version of any field remaining in the buffer.

ZSPI-TKN-INITIAL-POSITION: Reset Current -Token and Next-Token Pointers

This special operation resets the current position and next position to either the initial position in the buffer (the position just prior to the first token that is not a header token) or the initial position in the currently selected list (the position just prior to the list token). If *token-value* is ZSPI-VAL-INITIAL-BUFFER (0), the position is reset to the beginning of the buffer. If *token-value* is ZSPI-VAL-INITIAL-LIST (-1), the position is reset to the beginning of the current list.

ZSPI-TKN-MAX-FIELD-VERSION: Increase Maximum Version of Structure Fields

Use this token code to increase the maximum field version of the buffer. If the value specified is greater than the current value, the specified value is used. Otherwise, the current value is retained.

ZSPI-TKN-MAXRESP: Set Maximum Responses

Use this token code to set the header token that specifies the maximum number of response records to return in a single reply message. A *token-value* of zero (the default) specifies one response record per reply, not enclosed in a list. Any positive *token-value* specifies up to that number of response records, each enclosed in a list. A *token-value* of -1 specifies as many response records as fit, each enclosed in a list.

ZSPI-TKN-POSITION: Set Current-Token Pointer

Use this token code to restore a position previously saved using SSGET or SSGETTKN. The *token-value* is a four-word position descriptor. For this operation to be valid, the contents of the buffer prior to the previously saved position must not have been modified by ZSPI-TKN-DELETE, ZSPI-TKN-DATAFLUSH, or SSMOVE operations. Otherwise, this operation can corrupt the buffer and cause later operations to give indeterminate results. If *token-value* is all zeros or not supplied, this

operation sets the current-token pointer to the beginning of the buffer following the header.

ZSPI-TKN-RESET-BUFFER: Reset the Buffer

Use this token code before extracting tokens from an SPI buffer received (in either a command or a response) from another process. The calling sequence is:

```
SSPUT         ( buffer
SSPUTTKN      , ZSPI-TKN-RESET-BUFFER
              , maxlen )                                    ! i
```

This operation:

- Resets the maximum buffer length to *maxlen* bytes
- Clears the last-error information to zero (equivalent to the action of ZSPI-TKN-CLEARERR)
- Resets the current-token pointer to the beginning of the buffer (equivalent to the action of ZSPI-TKN-INITIAL-POSITION with ZSPI-VAL-INITIAL-BUFFER)

If *maxlen* is less than the actual number of bytes used in the buffer, as given in the header token ZSPI-TKN-USEDLEN, the procedure returns ZSPI-ERR-NOSPACE. However, the procedure still resets the maximum buffer length in the SPI message header, and subsequent SPI calls for that buffer fail with ZSPI-ERR-INVBUF.

ZSPI-TKN-SERVER-VERSION: Set Server Version Header Token

Use this token code to set the header token containing the release version of the server. For *token-value*, supply an unsigned integer representing the appropriate release version. For example, if the server is a NonStop Kernel subsystem of version G06, *token-value* should be the unsigned integer with the ASCII character *G* in the left byte and “06” in the right byte, or 18182.

Considerations

- The *token-value* parameter is optional if the token length specified by *token-id* is zero (for instance, if *token-id* is ZSPI-TKN-DATALIST, ZSPI-TKN-ERRLIST, ZSPI-TKN-ENDLIST, or ZSPI-TKN-LIST). Otherwise, the *token-value* parameter is required.
- Specifying a *count* parameter greater than one is equivalent to calling SSPUT or SSPUTTKN *count* number of times in succession without specifying that parameter (but updating *token-value* before each call).
- If *count* is greater than one and the token is of variable length, the values in the *token-value* array must be word-aligned.

- The order in which tokens are added to the buffer is not significant except in the case of: (1) SSPUT and SSPUTTKN calls with token codes for tokens that start and end lists, and (2) a few subsystem-specific exceptions mentioned in the subsystem manuals (for example, the ZEMS-TKN-SUBJECT-MARK token in an event message).
- Adding a token to the buffer with SSPUT or SSPUTTKN does not affect the current-token pointer for subsequent calls to SSGET or SSGETTKN.
- When SSPUT is called with a token map, it uses the null-value and version information in the token map, if necessary, to update the header token ZSPI-TKN-MAX-FIELD-VERSION. The token map is not stored in the buffer; instead, SSPUT creates a token code of type ZSPI-TYP-STRUCT with the token number of the map.
- If an error is returned by the procedure, buffer pointer information is not updated.

SSGET and SSGETTKN Procedures

The SSGET and SSGETTKN procedures extract tokens and related information from an SPI buffer. The two procedures produce the same results, and they are identical except for the type of the *token-id* parameter (SSGET passes *token-id* by reference and SSGETTKN passes it by value) and the consequent fact that SSGETTKN cannot be used with a token map.

A program can retrieve tokens from an SPI buffer in two ways using SSGET. The first way is to extract a particular token by name. This way is usually most desirable for management applications. The second way is to use one of the two special operations ZSPI-TKN-NEXTCODE and ZSPI-TKN-NEXTTOKEN to scan the buffer item by item. Servers often use this method to determine the contents of a message and to check for tokens that should not be present in the requester's message.

Programs should not rely on the relative order of tokens within the buffer, except in the case of multiple tokens with the same token code and subsystem ID. Such multiple occurrences are always kept in the order in which they were placed in the buffer, and can be treated as an array.

General Syntax

```
SSGET          (     buffer                          ! i/o
SSGETTKN        ,   token-id                          ! i
                , [ token-value ]                     ! i/o
                , [ index       ]                     ! i
                , [ count       ]                     ! i/o
                , [ ssid        ]   )                 ! i/o
```

buffer input, output

INT .EXT:ref:*

is the SPI buffer from which information is to be extracted.

token-id input

INT .EXT:ref:* (SSGET)
INT(32):value (SSGETTKN)

is a token code or (for SSGET only) a token map. This parameter normally identifies the token to be retrieved. If the *token-id* is one of the SPI standard token codes indicating a special operation, the interpretation of the *token-value*, *count*, and *index* parameters can vary from the descriptions here.

If *token-id* is a token that marks the beginning of a list (ZSPI-TKN-DATALIST, ZSPI-TKN-ERRLIST, ZSPI-TKN-SEGLIST, or ZSPI-TKN-LIST), the procedure selects the list so that subsequent calls can retrieve tokens in the list.

token-value input, output

STRING .EXT:ref:*

is normally the variable in which the requested token value is to be returned. For control and positioning operations, *token-value* can be an output parameter. Its data representation depends on the token-type field of the *token-id*.

index input

INT:value

if greater than zero, specifies an absolute index for *token-id*, starting from the beginning of the buffer or current list. An *index* of one gets the first occurrence of that token code, an *index* of two gets the second occurrence, and so on.

if zero or not supplied, returns the next occurrence of the token code following the current-token pointer in the buffer. For example, if the token occurs five times, calling SSGET or SSGETTKN once with an *index* of one and four times with *index* 0 would return all five occurrences.

if less than zero, returns an error.

SSGET resets the current-token pointer to the token value returned if no error is reported by the procedure.

To search from the beginning of the buffer or current list, a program must either supply a nonzero *index* or first reset the initial position (using SSPUT or SSPUTTKN with ZSPI-TKN-INITIAL-POSITION or ZSPI-TKN-RESET-BUFFER).

count input, output

INT .EXT:ref:1

is normally used as an input and output count parameter:

- On the call, it specifies the maximum number of token values to return. The *token-value* parameter is an array of *count* elements, each of which is

described by the *token-id*. If not supplied, it defaults to one. If less than zero, it causes an error.

- On return, it specifies the actual number of token values returned.

If a count greater than one is specified, SSGET or SSGETTKN continues searching until it either satisfies the requested count or reaches the end of the buffer or list.

For certain tokens for special operations, SSGET and SSGETTKN use the *count* parameter to return attribute information such as length, byte offset, or number of occurrences.

ssid input, output

INT .EXT:ref:6

is a subsystem ID that qualifies the token code. If not supplied or equal to zero (6*[0]), *ssid* defaults to one of:

- If the current-token pointer is in a list, the subsystem ID of the current list
- If the current-token pointer is not in a list, the subsystem ID in the SPI message header (ZSPI-TKN-SSID)

The version field of this parameter is not used when searching the buffer.

Special Operations With SSGET and SSGETTKN

Table 3-2 lists the special token codes you can supply to SSGET and SSGETTKN to retrieve header token values and perform other special operations.

Each operation is described following the table. For some operations, the procedure parameters differ in type and meaning from those indicated in the main syntax description. Modified syntax and semantics are given, or the differences are described.

Table 3-2. SSGET(TKN) Special Operations (page 1 of 2)

Token Specified in SSGET(TKN) Call	Type	Effect
ZSPI-TKN-ADDR	TOKENCODE	Retrieve the address of a token
ZSPI-TKN-BUFLEN	UINT	Retrieve buffer length
ZSPI-TKN-CHECKSUM	BOOLEAN	Retrieve checksum flag
ZSPI-TKN-COMMAND	ENUM	Retrieve command number
ZSPI-TKN-COUNT	TOKENCODE	Count the occurrences of a token
ZSPI-TKN-DEFAULT-SSID	SSID	Retrieve SSID at current position
ZSPI-TKN-HDRTYPE	UINT	Retrieve message header type

Table 3-2. SSGET(TKN) Special Operations (page 2 of 2)

Token Specified in SSGET(TKN) Call	Type	Effect
ZSPI-TKN-LASTERR	ENUM	Retrieve last procedure call error
ZSPI-TKN-LASTERRCODE	TOKENCODE	Retrieve code of token in last error
ZSPI-TKN-LASTPOSITION	POSITION	Retrieve position of last put token
ZSPI-TKN-LEN	TOKENCODE	Retrieve the length of a token value
ZSPI-TKN-MAX-FIELD-VERSION	UINT	Retrieve maximum field version
ZSPI-TKN-MAXRESP	INT	Retrieve maximum responses value
ZSPI-TKN-NEXTCODE	TOKENCODE	Retrieve the next different token code
ZSPI-TKN-NEXTTOKEN	TOKENCODE	Retrieve the next token code
ZSPI-TKN-OBJECT-TYPE	ENUM	Retrieve object type
ZSPI-TKN-OFFSET	TOKENCODE	Get the byte offset of a token value
ZSPI-TKN-POSITION	POSITION	Retrieve current-token pointer
ZSPI-TKN-SERVER-VERSION	UINT	Retrieve server version
ZSPI-TKN-SSID	SSID	Retrieve SSID of message
ZSPI-TKN-USEDLEN	UINT	Retrieve used buffer length

ZSPI-TKN-ADDR: Retrieve Address of a Token Value

Use this token code to get the extended address of a specific token value:

```
SSGET         (     buffer
SSGETTKN        ,   ZSPI-TKN-ADDR
                , [ token-id      ]                    ! i
                , [ index         ]                    ! i
                ,   token-address                      ! o
                , [ ssid          ]   )                ! i
```

This operation returns, in *token-address*, INT .EXT:ref:2, the 32-bit extended address of the value specified by *token-id* and *index*. For variable-length token values, this is the address of the length word at the start of the token value (see Figure 2-2 on page 2-6). If *token-id* is either omitted or equal to ZSPI-VAL-NULL-TOKENCODE and *index* is either omitted or zero, SSGET or SSGETTKN returns the address of the current token.

ZSPI-TKN-BUFLEN: Retrieve Current Buffer Length

Call SSGET or SSGETTKN with this token code to retrieve the current buffer length from the corresponding header token. If you specify an *index* parameter, it must have a value of zero or one.

ZSPI-TKN-CHECKSUM: Retrieve Value of Checksum Flag

Call SSGET or SSGETTKN with this token code to retrieve the checksum flag from the corresponding header token. If you specify an *index* parameter, it must have a value of zero or one.

ZSPI-TKN-COMMAND: Retrieve Command Number

Call SSGET or SSGETTKN with this token code to retrieve the command number from the corresponding header token. If you specify an *index* parameter, it must have a value of zero or one. If the command does not contain a command number, or if the subsystem could not obtain it, the error response from NonStop Kernel subsystems includes the null command number (ZSPI-VAL-NULL-COMMAND). In a response reporting a malformed command, the command number might not be reliable.

ZSPI-TKN-COUNT: Count the Occurrences of a Token

Use this token code to get the total number of occurrences of a specific token (starting from a specified index):

```
SSGET         (     buffer
SSGETTKN        ,   ZSPI-TKN-COUNT
                , [ token-id  ]                          ! i
                , [ index     ]                          ! i
                ,   count                                ! o
                , [ ssid      ]       )                  ! i
```

This operation returns, in *count*, INT .EXT:ref:1, the total number of occurrences of *token-id*. If *index* is zero or not supplied, counting starts from the current-token pointer. To count all occurrences in the current list, specify an *index* of 1.

If *token-id* is either omitted or equal to ZSPI-VAL-NULL-TOKENCODE and *index* is either omitted or zero, then SSGET or SSGETTKN counts occurrences of the current token beginning with the current occurrence.

ZSPI-TKN-DEFAULT-SSID: Retrieve the Current Default SSID

Use this token code to get the default subsystem ID value of the token at the current-token pointer:

```
SSGET         (     buffer
SSGETTKN        ,   ZSPI-TKN-DEFAULT-SSID
                ,   ssid                    )            ! o
```

Here SSGET or SSGETTKN returns, in *ssid*, INT .EXT:ref:6, the default subsystem ID value at the current-token pointer. SSPUT, SSPUTTKN, SSGET, and SSGETTKN use this value whenever the *ssid* parameter is omitted or null.

If the default subsystem ID comes from a list token, then the version field of *ssid* is set to ZSPI-VAL-NULL-VERSION. Therefore, when comparing subsystem ID values for equality, your program should omit the version field from the test.

ZSPI-TKN-HDRTYPE: Retrieve Header Type

Call SSGET or SSGETTKN with this token code to retrieve the header type from the corresponding header token. If you specify an *index* parameter, it must have a value of 0 or 1.

ZSPI-TKN-LASTERR: Retrieve Last SPI Error Number

Call SSGET or SSGETTKN with this token code to retrieve, from the corresponding header token, the error number returned by the last SPI procedure error. If you specify an *index* parameter, it must have a value of 0 or 1.

ZSPI-TKN-LASTERRCODE: Retrieve Token Code Involved in Last SPI Error

Call SSGET or SSGETTKN with this token code to retrieve, from the corresponding header token, the code of the token involved in the last SPI procedure error. If you specify an *index* parameter, it must have a value of 0 or 1.

ZSPI-TKN-LASTPOSITION: Retrieve Position of Last Token Put in Buffer

Call SSGET or SSGETTKN with this token code to retrieve, from the corresponding header token, the position where the last token was added by SSPUT(TKN). The value returned is a 4-word position descriptor, INT.EXT:ref:4, that you can later use to reset the position with the SSPUT or SSPUTTKN special operation ZSPI-TKN-POSITION. If you specify an *index* parameter, it must have a value of 0 or 1.

Saving the position of the last token is useful when you want to add a group of related tokens for which the buffer might not have sufficient space. After adding the first of the related tokens, call SSGET with ZSPI-TKN-LASTPOSITION to get the position of that token. Then, if the buffer does not have space for the remaining related tokens, you can use the saved position to remove the tokens following it. First, call SSPUT with ZSPI-TKN-POSITION and the value returned by ZSPI-TKN-LASTPOSITION to set the current-token pointer to the first of the related tokens. Second, call SSPUT with ZSPI-TKN-DATA-FLUSH to remove the current token and all the tokens that follow.

ZSPI-TKN-LEN: Get the Length of a Token Value

Use this token code to get the byte length of a specific token value:

```
SSGET          (      buffer
SSGETTKN         ,    ZSPI-TKN-LEN
                 , [  token-id     ]                          ! i
                 , [  index        ]                          ! i
                 ,    byte-length                             ! o
                 , [  ssid         ]   )                      ! i
```

This operation returns in *byte-length*, INT .EXT:ref:1, the size of the buffer needed to contain the specified occurrence of the token value. For variable-length token values, this includes the 2 bytes required for the length word, as shown in Figure 2-2 on page 2-6: the *byte-length* returned is *token-value*[0] + 2.

If *token-id* is either omitted or equal to ZSPI-VAL-NULL-TOKENCODE and *index* is either omitted or zero, then SSGET or SSGETTKN returns the length of the current token.

If *token-id* is a token map, this operation returns the length of the structure corresponding to that token map; the actual value in the buffer can be longer or shorter than this length. To get the actual length of the token value in the buffer, call SSGET with ZSPI-TKN-LEN and a token code made up of ZSPI-TYP-STRUCT and the token number from the token map. This operation returns the length of the structure value, including 2 bytes for the length field. Then subtract 2 from this value to get the length of the value that starts at the address obtained by ZSPI-TKN-ADDR with the token map.

ZSPI-TKN-MAX-FIELD-VERSION: Retrieve Maximum Version of Structure Fields

Call SSGET or SSGETTKN with this token code to retrieve the maximum field version from the corresponding header token. If you specify an *index* parameter, it must have a value of 0 or 1.

ZSPI-TKN-MAXRESP: Retrieve Maximum Responses Setting

Call SSGET or SSGETTKN with this token code to retrieve the setting of the maximum responses parameter from the corresponding header token. If you specify an *index* parameter, it must have a value of 0 or 1.

ZSPI-TKN-NEXTCODE: Get the Next Different Token Code in the Buffer

Use this token code to get the next token code in the buffer that is different from the current token code:

```
SSGET       (     buffer
SSGETTKN      ,   ZSPI-TKN-NEXTCODE
              , [ next-token-code ]                    ! o
              ,
              , [ occurs          ]                    ! o
              , [ ssid            ]  )                 ! o
```

`next-token-code`

contains the next different token code in the buffer.

`occurs`

contains the number of contiguous occurrences of next-token-code.

For this operation, *`ssid`* is an output parameter only; you do not supply any information in this token, but merely provide a variable in which SSGET will return the subsystem ID that qualifies the token code. If you do not supply a variable for *`ssid`*, and the subsystem ID associated with the next token code is not the same as the default subsystem ID, a “missing parameter” error is returned. Therefore, always supply a variable for *`ssid`* when calling SSGET with ZSPI-TKN-NEXTCODE unless, for some special reason, you are certain that all tokens the program could encounter are qualified by the default subsystem ID. The *`ssid`* returned always has a version field of zero (null).

The *`index`* parameter has no effect on this operation. If supplied, it must be equal to zero.

Note. The special operations ZSPI-TKN-NEXTCODE and ZSPI-TKN-NEXTTOKEN return only token codes. In particular, tokens that were added to the buffer by using SSPUT with a token map are carried in the buffer with a token code of type ZSPI-TYP-STRUCT. The NEXTCODE and NEXTTOKEN operations return this token code, not the token map used with SSPUT.

The best way to determine which token has been returned by NEXTCODE or NEXTTOKEN is to extract the token number from the token code and test it. This technique prevents any problem that might occur because of a change in the type of an extensible structured token.

ZSPI-TKN-NEXTTOKEN: Get the Next Token Code in the Buffer

Use this token code to get the token code of the next token in the buffer:

```
SSGET          (      buffer
SSGETTKN         ,    ZSPI-TKN-NEXTTOKEN
                 , [  next-token-code ]                    ! o
                 ,
                 ,
                 , [  ssid            ]    )               ! o
```

This operation differs from ZSPI-TKN-NEXTCODE in that it always returns the token code of the next token, whether or not it is the same as that of the current token, and whether or not the token is within a list. The operation returns multiple occurrences of the same token code in the same order as they were added to the buffer with SSPUT or SSPUTTKN.

For this operation, *ssid* is an output parameter only; you do not supply any information in this token, but merely provide a variable in which SSGET returns the subsystem ID that qualifies the token code. If you do not supply a variable for *ssid*, and the subsystem ID associated with the next token code is not the same as the default subsystem ID, a “missing parameter” error is returned. Therefore, always supply a variable for SSID when calling SSGET with ZSPI-TKN-NEXTTOKEN unless, for some special reason, you are certain that all tokens the program could encounter are qualified by the default subsystem ID. The *ssid* returned always has a version field of zero (null).

The *index* and *count* parameters have no effect on this operation. If supplied, *index* must be equal to zero, and *count* is always returned as 1.

See the NOTE at the end of ZSPI-TKN-NEXTCODE: Get the Next Different Token Code in the Buffer on page 3-20.

ZSPI-TKN-OBJECT-TYPE: Retrieve Object Type

Call SSGET or SSGETTKN with this token code to retrieve the object type from the corresponding header token. If you specify an *index* parameter, it must have a value of 0 or 1. In a response reporting a malformed command, the object-type number might not be reliable.

ZSPI-TKN-OFFSET: Retrieve the Byte Offset of a Token Value

Use this token code to get the byte offset of a specific token value:

```
SSGET     (     bufferSSGETTKN
            ,   ZSPI-TKN-OFFSET
            , [ token-id     ]                          ! i
            , [ index        ]                          ! i
            ,   byte-offset                             ! o
            , [ ssid         ]     )                    ! i
```

This operation returns, in *byte-offset*, INT .EXT:ref:1, the byte offset from the start of the buffer to the value associated with the specified token code and index. (For variable-length values, the token value begins with the length word; the offset given is the offset to that length word.)

If *token-id* is either omitted or equal to ZSPI-VAL-NULL-TOKENCODE and *index* is either omitted or zero, then SSGET or SSGETTKN returns the offset of the current occurrence of the current token.

ZSPI-TKN-POSITION: Retrieve Current-Token Position

Call SSGET or SSGETTKN with this token code to retrieve the pointer to the current token. The value returned is a four-word position descriptor, INT .EXT:ref:4, that you can later use to reset the position with the SSPUT or SSPUTTKN special operation ZSPI-TKN-POSITION. If you specify an *index* parameter, it must have a value of 0 or 1.

ZSPI-TKN-SERVER-VERSION: Retrieve Server Version

Call SSGET or SSGETTKN with this token code to retrieve the server version from the corresponding header token. If you specify an *index* parameter, it must have a value of 0 or 1.

ZSPI-TKN-SSID: Retrieve SSID Used When Buffer Was Initialized

Call SSGET or SSGETTKN with this token code to retrieve the default SSID from the corresponding header token. If you specify an *index* parameter, it must have a value of 0 or 1.

ZSPI-TKN-USEDLEN: Retrieve Used Length of Buffer

Call SSGET or SSGETTKN with this token code to retrieve the length of the used portion of the buffer from the corresponding header token. If you specify an *index* parameter, it must have a value of 0 or 1.

Considerations

- Tokens extracted by SSGET and SSGETTKN are not deleted or removed from the buffer.
- For checkpointing purposes, note that calls to SSGET and SSGETTKN can modify the SPI message header. For instance, the header tokens ZSPI-TKN-LASTERR and ZSPI-TKN-LASTERRCODE change if an SPI error occurs on the call. Positioning information in the header also changes frequently, and future versions of SPI might introduce other kinds of change. Programs should never assume that *any* SSGET or SSGETTKN operation leaves the buffer unchanged.
- When the current-token pointer is within a particular list, all SSGET and SSGETTKN calls pertain only to tokens within that list, but the header tokens listed in Table 3-2 on page 3-15 are always accessible. Your program can exit the list by calling SSGET to get the ZSPI-TKN-ENDLIST token.
- The *index* and *count* parameters have no effect when *token-id* is ZSPI-TKN-ENDLIST. If supplied, *index* must be equal to zero or 1, and *count* is always returned as 1.
- If an error is returned, buffer pointer information is not updated.
- When you use a token map for the *token-id* parameter, the map can specify a structure version that is longer or shorter than the structure contained in the buffer. If the requested version is longer than the version in the buffer, SSGET calls SSNULL to set to null values any new fields that are not obtained from the buffer. If the requested version is shorter than the one in the buffer, SSGET returns only the requested length.
- Some of the special operations listed in Table 3-2 on page 3-15 return certain attributes of tokens in the buffer. These are ZSPI-TKN-COUNT, which gets the total number of occurrences of a specific token; ZSPI-TKN-LEN, which gets the length of a token value; ZSPI-TKN-ADDR, which gets the extended address of a token value; and ZSPI-TKN-OFFSET, which gets the byte offset, from the beginning of the buffer, of a token value. These special operations are useful to programs retrieving token values from the buffer. (ZSPI-TKN-ADDR is not meaningful—and therefore is not available—in TACL.)

  When obtaining an attribute of a token with an index of zero, SSGET follows the same search rules as it does when getting a token value (the search begins with the token indicated by the next-token pointer) with these exceptions:

  - A special calling mode exists to request an attribute of the current token: if the token code and index are both null or not supplied, the current token is used.
  - The current and next pointers are not changed if the requested token is the current token.
  - If the requested token is not the current token, both the current and next pointers are set to the requested token.

This example illustrates the second exception:

```
CALL SSGETTKN (buffer, ZSPI-TKN-LEN, , , len)
```

Before the call, the positioning is:

```
current    next
   |         |
   v         v
<TKN-A>  <TKN-B>  <TKN-C>  <TKN-A>
```

After the call, the positioning is unchanged:

```
current    next
   |         |
   v         v
<TKN-A>  <TKN-B>  <TKN-C>  <TKN-A>
```

This example illustrates the third exception:

```
CALL SSGETTKN (buffer, ZSPI-TKN-LEN, TKN-A, , len)
```

Before the call, the positioning is:

```
current    next
   |         |
   v         v
<TKN-A>  <TKN-B>  <TKN-C>  <TKN-A>
```

However, after the call, the positioning is:

```
                      current + next
                              |
                              v
<TKN-A>  <TKN-B>  <TKN-C>  <TKN-A>
```

A position (ZSPI-TKN-POSITION), a token address (ZSPI-TKN-ADDR), or a token offset (ZSPI-TKN-OFFSET) returned by SSGET remains valid until a token is deleted using the SSPUT operations ZSPI-TKN-DELETE or ZSPI-TKN-DATA-FLUSH or until a call to SSMOVE replaces a token in the buffer. If the contents of the buffer are copied to another buffer with SSMOVE, the position, address, or offset is still valid when used with the original buffer, but is not valid for use with the target buffer of the SSMOVE.

SSMOVE and SSMOVETKN Procedures

The SSMOVE and SSMOVETKN procedures copy tokens from one SPI buffer to another. One call can copy a single token, a sequence of tokens with the same token code, or a list.

SSMOVE performs a sequence of SSGET and SSPUT operations. Likewise, SSMOVETKN performs a sequence of SSGETTKN and SSPUTTKN (or SSGET and SSPUT) operations. The two procedures produce the same results, and they are identical except for the type of the *token-id* parameter (SSMOVE passes *token-id* by reference and SSMOVETKN passes it by value) and the consequent fact that SSMOVETKN cannot be used with a token map.

General Syntax

```
SSMOVE          (      token-id                          ! i
SSMOVETKN         ,    source-buffer                     ! i/o
                  , [ source-index  ]                    ! i
                  ,    dest-buffer                       ! i/o
                  , [ dest-index    ]                    ! i
                  , [ count         ]                    ! i/o
                  , [ ssid          ]   )                ! i
```

token-id input

INT .EXT:ref:* (SSMOVE)
INT(32):value (SSMOVETKN)

is a token code or (for SSMOVE only) a token map that identifies the token to be copied. If *token-id* is a list token, the entire list is copied.

source-buffer input, output

INT .EXT:ref:*

is the SPI buffer containing the token or tokens to be copied.

source-index input

INT:value

if greater than zero, identifies the first occurrence of *token-id* to be copied from the source buffer. (One or more occurrences can be copied, depending on the value of *count*.) A *source-index* value of 1 specifies that the copy is to start with the first occurrence of the token code, a value of 2 specifies the second occurrence, and so on.

if zero or not supplied, directs SSMOVE or SSMOVETKN to start with the next occurrence of the token code after the current-token pointer in the source buffer.

dest-buffer input, output

INT .EXT:ref:*

is the SPI buffer to which the specified token or tokens are to be copied.

dest-index input

INT:value

if greater than zero, identifies the first occurrence of *token-id* to be replaced in the destination buffer. A value of 1 specifies that replacement should start with the first occurrence of the token code, a value of 2 specifies the second occurrence, and so on. If the specified occurrences are not found in the destination buffer, the tokens being copied are added to the end of the buffer.

if zero or not supplied, SSMOVE or SSMOVETKN adds the tokens from the source buffer to the end of the destination buffer.

count input, output

INT .EXT:ref:1

is used as an input and output count parameter:

- On the call, it specifies the maximum number of tokens to copy, unless *token-id* is a list token; in the latter case, it gives the maximum number of lists to move. If not supplied, *count* defaults to 1.
- On return, it specifies the actual number of token values or lists copied.

ssid input

INT .EXT:ref:6

is a subsystem ID, as described in Section 4, ZSPI Data Definitions, that qualifies the token ID. If not supplied or equal to zero (6*[0]), *ssid* defaults to the subsystem ID of the current list, or if the current-token pointer is not in a list, then to the subsystem ID specified in the SPI message header (ZSPI-TKN-SSID). The version field of *ssid* is not used in searching the source buffer.

Considerations

- Any odd-length buffer fields are padded at the end so that the next token added by SSMOVE or SSMOVETKN always starts on a word (even) boundary. The same circumstances also apply to SSPUT or SSPUTTKN operations.
- Tokens copied by SSMOVE or SSMOVETKN are not deleted or removed from the source buffer.
- For checkpointing purposes, note that calls to SSMOVE and SSMOVETKN can modify the SPI message header of the source buffer. Positioning information in the header changes frequently, and future versions of SPI might introduce other kinds

of change. Programs should never assume that *any* SSMOVE or SSMOVETKN operation leaves the source buffer unchanged.

- After a successful SSMOVE or SSMOVETKN operation, the current-token position in the source buffer is changed to the position of the last token copied.
- When SSMOVE copies a token identified by a token map, the value obtained from the source buffer is truncated or padded according to the map specifications, and the ZSPI-TKN-MAX-FIELD-VERSION header token in the destination buffer is appropriately adjusted.
- If an error occurs on SSMOVE or SSMOVETKN, the ZSPI-TKN-LASTERR and ZSPI-TKN-LASTERRCODE header tokens can be set in either the source buffer or the destination buffer, depending on whether the error occurred on the logical SSGET[TKN] or SSPUT[TKN] part of the copy.
- SSMOVE or SSMOVETKN can copy an incomplete list (a list with no corresponding end-list token) if and only if *dest-index* is not supplied or is zero. If a nonzero destination index is specified, meaning that a replacement operation is being requested, an incomplete list causes SSMOVE or SSMOVETKN to return ZSPI-ERR-MISTKN.

Example: Moving Buffer Tokens Using SSMOVETKN

The C source code program in Example 3-1 demonstrates the proper use of SSMOVETKN to move the remaining tokens from one SPI buffer to another buffer.

Example 3-1. Moving Buffer Tokens Using SSMOVETKN (page 1 of 7)

```
/*
 *   Try moving some buffer tokens around using SSMOVETKN.
 */
#pragma symbols
#pragma inspect
#pragma nomap
#pragma nolmap

#define max_bufsize  256      /* in bytes */

#include "secc.h"
#pragma list
#include "seccutlc"
short    ZERO = 0;

#pragma page "MAIN"
main(/* int argc, char *argv[] */)
{
  zspi_ddl_ssid_def       mySsid;

  bufsize = max_bufsize;

  /*
   * Initialize the SPI buffer "b1"
   */
  if (err = SSINIT (b1, bufsize, (short *) &ssid, ZSPI_VAL_CMDHDR))
    display_spi_error (err, ZSPI_VAL_SSINIT, 0L, true);
/*
   * Put four tokens in the SPI buffer "b1"
   */
  if (err = SSPUTTKN (b1, ZSPI_TKN_DATALIST))
    display_spi_error (err, ZSPI_VAL_SSPUTTKN, ZSPI_TKN_DATALIST, true);
  val = 'A';
  if (err = SSPUTTKN (b1, tkn_1, &val))
    display_spi_error (err, ZSPI_VAL_SSPUTTKN, tkn_1, true);
  val = 'B';
  if (err = SSPUTTKN (b1, tkn_2, &val))
    display_spi_error (err, ZSPI_VAL_SSPUTTKN, tkn_2, true);
  val = 'C';
  if (err = SSPUTTKN (b1, tkn_3, &val))
    display_spi_error (err, ZSPI_VAL_SSPUTTKN, tkn_3, true);
  val = 'D';
  if (err = SSPUTTKN (b1, tkn_3, &val))
    display_spi_error (err, ZSPI_VAL_SSPUTTKN, tkn_3, true);
  if (err = SSPUTTKN (b1, ZSPI_TKN_ENDLIST))
    display_spi_error (err, ZSPI_VAL_SSPUTTKN, ZSPI_TKN_ENDLIST, true);
```

Example 3-1. Moving Buffer Tokens Using SSMOVETKN (page 2 of 7)

```
  /*
   * Initialize the SPI buffer "b2"
   */
  if (err = SSINIT (b2, bufsize, (short *) &ssid, ZSPI_VAL_CMDHDR))
    display_spi_error (err, ZSPI_VAL_SSINIT, 0L, true);

  /*
   * Reset the SPI buffer "b1"
   */
  if (err = SSPUTTKN (b1, ZSPI_TKN_RESET_BUFFER, (char *) &bufsize))
    display_spi_error (err, ZSPI_VAL_SSPUTTKN,
                            ZSPI_TKN_RESET_BUFFER, true);

  /*
   * Get the tokens in the SPI buffer "b1"
   */

  get_count = 1;
/*
   * Note that the following code doesn't work as expected. This is
   * because SSMOVETKN will move an entire list if positioned on a
   * list and GET(NEXTTOKEN) will enter the list. So
   * B1 = ["A","B","C","D"]
   * B2 = ["A","B","C","D"],"A","B","C","D" and you will get an
   * error -8 (Missing Tkn) when you try to move the last ENDLIST from
   * B1 to B2. This is returned because there is no matching DATALIST,etc.
   */
  while (!err)
  {
     if (err = SSGETTKN (b1, ZSPI_TKN_NEXTTOKEN, (char *) &tkn_code,
                         0, &get_count, (short *) &mySsid))
       continue;
     else
       err = SSMOVETKN (tkn_code, b1, ZERO,
                                  b2, ZERO,, &mySsid);
  }
  if (tkn_code != ZSPI_TKN_ENDLIST)
  {
     printf ("At end of loop: TKN_CODE should = ENDLIST, \n");
     printf ("B2: \n");
     dump_buf (b2);
  }
  if (err != ZSPI_ERR_MISTKN)
  {
     printf ("ERROR should = -8, was %d \n", err);
     printf ("B2: \n");
     dump_buf (b2);
  }
  /*
   * Reset the SPI buffer "b2"
   */
  if (err = SSPUTTKN (b2, ZSPI_TKN_RESET_BUFFER, (char *) &bufsize))
  {
    display_spi_error (err, ZSPI_VAL_SSPUTTKN,
                            ZSPI_TKN_RESET_BUFFER, true);
  }
```

Example 3-1. Moving Buffer Tokens Using SSMOVETKN (page 3 of 7)

```
/*
 * Get the tokens in the SPI buffer "b2"
 */

get_count = 1;
if (err = SSGETTKN (b2, ZSPI_TKN_NEXTTOKEN, (char *) &tkn_code,
                    0, &get_count, (short *) &mySsid))
{
  display_spi_error (err, ZSPI_VAL_SSGETTKN,
                          ZSPI_TKN_NEXTTOKEN, true);
}
if (tkn_code != ZSPI_TKN_DATALIST)
{
   printf ("B2 check 1:TKN_CODE should = DATALIST, \n");
   printf ("B2: \n");
   dump_buf (b2);
}
if (err = SSGETTKN (b2, ZSPI_TKN_NEXTTOKEN, (char *) &tkn_code,
                    0, &get_count, (short *) &mySsid))
{
  display_spi_error (err, ZSPI_VAL_SSGETTKN,
                          ZSPI_TKN_NEXTTOKEN, true);
}
if (tkn_code != tkn_1)
{
   printf ("B2 check 1:TKN_CODE should = TKN_1, \n");
   printf ("B2: \n");
   dump_buf (b2);
}
if (err = SSGETTKN (b2, ZSPI_TKN_NEXTTOKEN, (char *) &tkn_code,
                    0, &get_count, (short *) &mySsid))
{
  display_spi_error (err, ZSPI_VAL_SSGETTKN,
                          ZSPI_TKN_NEXTTOKEN, true);
}
if (tkn_code != tkn_2)
{
   printf ("B2 check 1:TKN_CODE should = TKN_2, \n");
   printf ("B2: \n");
   dump_buf (b2);
}
if (err = SSGETTKN (b2, ZSPI_TKN_NEXTTOKEN, (char *) &tkn_code,
                    0, &get_count, (short *) &mySsid))
{
  display_spi_error (err, ZSPI_VAL_SSGETTKN,
                          ZSPI_TKN_NEXTTOKEN, true);
}
if (tkn_code != tkn_3)
{
   printf ("B2 check 1:TKN_CODE should = TKN_3, \n");
   printf ("B2: \n");
   dump_buf (b2);
}
```

Example 3-1. Moving Buffer Tokens Using SSMOVETKN (page 4 of 7)

```
 if (err = SSGETTKN (b2, ZSPI_TKN_NEXTTOKEN, (char *) &tkn_code,
                     0, &get_count, (short *) &mySsid))
{
  display_spi_error (err, ZSPI_VAL_SSGETTKN,
                          ZSPI_TKN_NEXTTOKEN, true);
}
if (tkn_code != tkn_3)
{
   printf ("B2 check 1:TKN_CODE should = TKN_3 (2nd), \n");
   printf ("B2: \n");
   dump_buf (b2);
}
if (err = SSGETTKN (b2, ZSPI_TKN_NEXTTOKEN, (char *) &tkn_code,
                     0, &get_count, (short *) &mySsid))
{
  display_spi_error (err, ZSPI_VAL_SSGETTKN,
                          ZSPI_TKN_NEXTTOKEN, true);
}
if (tkn_code != ZSPI_TKN_ENDLIST)
{
   printf ("B2 check 1:TKN_CODE should = ENDLIST, \n");
   printf ("B2: \n");
   dump_buf (b2);
}
if (err = SSGETTKN (b2, ZSPI_TKN_NEXTTOKEN, (char *) &tkn_code,
                     0, &get_count, (short *) &mySsid))
{
  display_spi_error (err, ZSPI_VAL_SSGETTKN,
                          ZSPI_TKN_NEXTTOKEN, true);
}
if (tkn_code != tkn_1)
{
   printf ("B2 check 1:TKN_CODE should = TKN_1 (2nd), \n");
   printf ("B2: \n");
   dump_buf (b2);
}
if (err = SSGETTKN (b2, ZSPI_TKN_NEXTTOKEN, (char *) &tkn_code,
                     0, &get_count, (short *) &mySsid))
{
   display_spi_error (err, ZSPI_VAL_SSGETTKN,
                          ZSPI_TKN_NEXTTOKEN, true);
}
if (tkn_code != tkn_2)
{
   printf ("B2 check 1:TKN_CODE should = TKN_2 (2nd), \n");
   printf ("B2: \n");
   dump_buf (b2);
}
 if (err = SSGETTKN (b2, ZSPI_TKN_NEXTTOKEN, (char *) &tkn_code,
                     0, &get_count, (short *) &mySsid))
{
  display_spi_error (err, ZSPI_VAL_SSGETTKN,
                          ZSPI_TKN_NEXTTOKEN, true);
}
```

Example 3-1. Moving Buffer Tokens Using SSMOVETKN (page 5 of 7)

```
if (tkn_code != tkn_3)
{
   printf ("B2 check 1:TKN_CODE should = TKN_3 (3rd), \n");
   printf ("B2: \n");
   dump_buf (b2);
}
if (err = SSGETTKN (b2, ZSPI_TKN_NEXTTOKEN, (char *) &tkn_code,
                    0, &get_count, (short *) &mySsid))
{
  display_spi_error (err, ZSPI_VAL_SSGETTKN,
                          ZSPI_TKN_NEXTTOKEN, true);
}
if (tkn_code != tkn_3)
{
   printf ("B2 check 1:TKN_CODE should = TKN_3 (4th), \n");
   printf ("B2: \n");
   dump_buf (b2);
}
err = SSGETTKN (b2, ZSPI_TKN_NEXTTOKEN, (char *) &tkn_code,
                    0, &get_count, (short *) &mySsid);
if (err != ZSPI_ERR_MISTKN)
{
   printf ("B2 check 1:Error should = -8, is %d \n", err);
   printf ("B2: \n");
   dump_buf (b2);
}
/*
 * Now do it the correct way
 */
/*
 * Initialize the SPI buffer "b2"
 */
if (err = SSINIT (b2, bufsize, (short *) &ssid, ZSPI_VAL_CMDHDR))
{
  display_spi_error (err, ZSPI_VAL_SSINIT, 0L, true);
}
/*
 * Reset the SPI buffer "b1"
 */
if (err = SSPUTTKN (b1, ZSPI_TKN_RESET_BUFFER, (char *) &bufsize))
{
  display_spi_error (err, ZSPI_VAL_SSPUTTKN,
                          ZSPI_TKN_RESET_BUFFER, true);
}
```

Example 3-1. Moving Buffer Tokens Using SSMOVETKN (page 6 of 7)

```
/*
 * Get the tokens in the SPI buffer "b1"
 */

get_count = 1;
/*
 * Note that the following code moves all the nonheader tokens from
 * B1 to B2.
 * B1 = ["A","B","C","D"]
 * B2 = ["A","B","C","D"]
 */
while (!err)
{
   if (err = SSGETTKN (b1, ZSPI_TKN_NEXTTOKEN, (char *) &tkn_code,
                       0, &get_count, (short *) &mySsid))
     continue;
   else
     err = SSMOVETKN (tkn_code, b1, ZERO,
                                b2, ZERO,, &mySsid);
   if (tkn_code == ZSPI_TKN_DATALIST
   ||  tkn_code == ZSPI_TKN_ERRLIST
   ||  tkn_code == ZSPI_TKN_SEGLIST)
   {
      /* The SSMOVETKN call moved the entire list. Go to the ENDLIST
         and then continue on with the GET (NEXTTOKEN) loop */
      err = SSGETTKN (b1, ZSPI_TKN_ENDLIST, (char *) &tkn_code,
                       0, &get_count, (short *) &mySsid);
   }
}
if (tkn_code != ZSPI_TKN_DATALIST)
{
   printf ("At end of loop2: TKN_CODE should = DATALIST, \n");
   printf ("B2: \n");
   dump_buf (b2);

if (err != ZSPI_ERR_MISTKN)
{
   printf ("At end of loop2: ERROR should = -8, was %d \n", err);
   printf ("B2: \n");
   dump_buf (b2);
}
/*
 * Reset the SPI buffer "b2"
 */
if (err = SSPUTTKN (b2, ZSPI_TKN_RESET_BUFFER, (char *) &bufsize))
{
  display_spi_error (err, ZSPI_VAL_SSPUTTKN,
                          ZSPI_TKN_RESET_BUFFER, true);
}
/*
 * Get the tokens in the SPI buffer "b2"
 */

get_count = 1;
```

Example 3-1. Moving Buffer Tokens Using SSMOVETKN (page 7 of 7)

```
  if (err = SSGETTKN (b2, ZSPI_TKN_NEXTTOKEN, (char *) &tkn_code,
                      0, &get_count, (short *) &mySsid))
  {
    display_spi_error (err, ZSPI_VAL_SSGETTKN,
                            ZSPI_TKN_NEXTTOKEN, true);
 }  }
  if (tkn_code != ZSPI_TKN_DATALIST)
  {
     printf ("B2 check 2:TKN_CODE should = DATALIST, \n");
     printf ("B2: \n");
     dump_buf (b2);
  }
  if (err = SSGETTKN (b2, ZSPI_TKN_NEXTTOKEN, (char *) &tkn_code,
                      0, &get_count, (short *) &mySsid))
  {
    display_spi_error (err, ZSPI_VAL_SSGETTKN,
                            ZSPI_TKN_NEXTTOKEN, true);
  }
  if (tkn_code != tkn_1)
  {
     printf ("B2 check 2:TKN_CODE should = TKN_1, \n");
     printf ("B2: \n");
     dump_buf (b2);
  }
  if (err = SSGETTKN (b2, ZSPI_TKN_NEXTTOKEN, (char *) &tkn_code,
                      0, &get_count, (short *) &mySsid))
  {
    display_spi_error (err, ZSPI_VAL_SSGETTKN,
                            ZSPI_TKN_NEXTTOKEN, true);

  }
  if (tkn_code != tkn_3)
  {
     printf ("B2 check 2:TKN_CODE should = TKN_3 (2nd), \n");
     printf ("B2: \n");
     dump_buf (b2);
  }
  if (err = SSGETTKN (b2, ZSPI_TKN_NEXTTOKEN, (char *) &tkn_code,
                      0, &get_count, (short *) &mySsid))
  {
    display_spi_error (err, ZSPI_VAL_SSGETTKN,
                            ZSPI_TKN_NEXTTOKEN, true);
  }
  if (tkn_code != ZSPI_TKN_ENDLIST)
  {
     printf ("B2 check 2:TKN_CODE should = ENDLIST, \n");
     printf ("B2: \n");
     dump_buf (b2);
  }
  err = SSGETTKN (b2, ZSPI_TKN_NEXTTOKEN, (char *) &tkn_code,
                      0, &get_count, (short *) &mySsid);
  if (err != ZSPI_ERR_MISTKN)
  {
     printf ("B2 check 2:Error should = -8, is %d \n", err);
     printf ("B2: \n");
     dump_buf (b2);
  }
  printf ("Program finished.\n");
}
```

SSIDTOTEXT Procedure

The SSIDTOTEXT procedure converts an internal form subsystem ID (SSID) to its external representation.

General Syntax

```
len := SSIDTOTEXT ( ssid                                    !i
                  , chars                                   !o
                 [, status ] );                             !o
```

`len` returned value

INT

returns the number of characters placed into *chars*. Zero is returned if one of the errors (0, 29), (4,0), or (8,0) occurs. Some other errors may prevent the subsystem name from being obtained. In those cases, a text representation of the subsystem ID is still produced, but it contains the subsystem number rather than the subsystem name.

ssid input

INT .EXT:ref:6

contains the subsystem ID to be converted into displayable form.

chars output

STRING .EXT:ref:*

is the string into which the displayable representation of ssid is placed. The number of characters placed into *chars* is returned as *len*. For a description of the external form of the subsystem ID, see Considerations on page 3-36.

The caller is responsible for supplying enough space in *chars* to hold the result. No more than 23 characters can be placed into *chars*.

status output

INT(32) .EXT:ref:1

is a status code that indicates any problems encountered. These number pairs describe the two halves of the INT(32) value:

Error numbers	Meaning
(0,0)	No error
(0,*x*)	Something was wrong with the calling sequence. The *x* half of INT(32) can have these values and meanings: *x*= 29 a required parameter is missing. *x*=632 not enough stack space is available.
(1,*x*)	An error occurred allocating the private segment. The value of *x* is the error code for the ALLOCATESEGEMENT.
(2,*x*)	A problem occurred opening the nonresident template file. The *x* half of INT(32) can have these values and meanings: *x*=>0 file-system error code *x*=-1 file code not 844 *x*=-2 file not disk file *x*=-3 file not key sequenced *x*=-4 file has wrong record size *x*=-5 file has wrong primary key definition
(3,*x*)	An error occurred reading the nonresident template file. The *x* half of INT(32) is the file-system error.
(4,0)	Invalid value in the internal form of the subsystem ID.
(7,*x*)	An error occurred accessing the private segment. The *x* half of the INT(32) value is the error code returned from MOVEX.
(8,0)	Internal error

Considerations

The external form of the subsystem ID is *owner.ss.version* or 0.0.0, where

owner

is 1 to 8 letters, digits, or hyphens.

ss

is either the subsystem number or the subsystem name. A subsystem number is a string of digits that can be preceded by a minus sign. The value of the number is between -32767 and 32767. A subsystem name is 1 to 8 letters, digits, or hyphens.

version

is either a string of digits that represents a TOSVERSION-format (A*nn*) or a value from 0 to 65535.

The 0.0.0 form is used to represent the null subsystem ID. Its internal representation is binary zero. The number of zeros in each field can vary. For example, 000.0.000 is equivalent to 0.0.0

Examples

These are examples of the external form of the subsystem ID:

```
TANDEM.SAFEGUARD.D40

TANDEM.94.0

0.0.0
```

TEXTTOSSID Procedure

TEXTTOSSID scans a character string, expecting to find the external representation of a subsystem ID starting in the first byte (no leading spaces are accepted). It returns the internal representation of the subsystem ID that it finds.

General Syntax

```
len := TEXTTOSSID ( chars                                    !i
                  , ssid                                     !o
                 [, status ] );                              !o
```

len returned value

INT

returns the number of characters placed from *chars*. Zero is returned if an error occurs.

chars input

STRING .EXT:ref:*

is the string containing the external representation of a subsystem ID. The number of characters scanned is returned as *len*. For a description of the external form of the subsystem ID, see Considerations on page 3-38.

ssid output

INT .EXT:ref:6

receives the internal form of the subsystem ID contained in *chars*.

status output

INT(32) .EXT:ref:1

is a status code that indicates any problems encountered. These number pairs describe the two halves of the INT(32) value:

Error numbers	Meaning
(0,0)	No error
(0,*x*)	Something was wrong with the calling sequence. The *x* half of INT(32) can have these values and meanings: *x*= 29 A required parameter is missing. *x*=632 Not enough stack space is available.
(1,*x*)	An error occurred allocating the private segment. The value of *x* is the error code for the ALLOCATESEGEMENT.
(2,*x*)	A problem occurred opening the nonresident template file. The *x* half of INT(32) can have these values and meanings: *x*=>0 file-system error code *x*=-1 filecode not 844 *x*=-2 file not disk file *x*=-3 file not key sequenced *x*=-4 file has wrong record size *x*=-5 file has wrong primary key definition
(3,*x*)	An error occurred reading the nonresident template file. The *x* half of INT(32) is the file-system error.
(4,0)	Invalid value in the internal form of the subsystem ID.
(7,*x*)	An error occurred accessing the private segment. The *x* half of the INT(32) value is the error code returned from MOVEX.
(8,0)	Internal error

Considerations

The external form of the subsystem ID is *owner.ss.version* or 0.0.0, where

owner

is 1 to 8 letters, digits, or hyphens, the first of which must be a letter; letters are not upshifted so the end user must enter *owner* in the proper case.

ss

is either the subsystem number or the subsystem name. A subsystem number is a string of digits that can be preceded by a minus sign. The value of the number is between -32767 and 32767. A subsystem name is 1 to 8 letters, digits, or hyphens.

version

is either a string of digits that represents a TOSVERSION-format (A*nn*) or a value from 0 to 65535.

The 0.0.0 form is used to represent the null subsystem ID. Its internal representation is binary zero. The number of zeros in each field can vary. For example, 000.0.000 is equivalent to 0.0.0

Examples

These are examples of the external form of the subsystem ID:

```
TANDEM.SAFEGUARD.D40

TANDEM.94.0

0.0.0
```

4 ZSPI Data Definitions

The data types, tokens, values, and other declarations on which SPI is based are referred to as the SPI standard definitions or the ZSPI definitions.

This section describes these ZSPI definition types:

HP distributes these definitions in the ZSPI definition files normally located in $*software-release-volume*.ZSPIDEF.* (although they can be placed elsewhere). For more information about these files, see Data Definitions on page 2-10.

Fundamental Data Structures

This subsection describes the fundamental data structures with which SPI tokens and other elements are built. A token type (TYP) associates a defined token data type (TDT) with a corresponding data structure (DDL). The token type can then be used in the TOKEN TYPE IS clause of a token code definition. Names of the fundamental structure definitions are of the form ZSPI-DDL-....

In the DDL definitions, the SPI-NULL clauses give the null values used by the SSNULL procedure when it initializes fields of extensible structured tokens. The TACL clauses name special TACL data types in which the associated token values or fields are represented in TACL. For more information about the DDL definition statements, see Appendix B, Summary of DDL for SPI.

ZSPI-DDL-BOOLEAN

```
def ZSPI-DDL-BOOLEAN       type logical         spi-null " ".
```

ZSPI-DDL-BOOLEAN defines a Boolean value.

ZSPI-DDL-BYTE

```
def ZSPI-DDL-BYTE      type binary 8 unsigned     spi-null 0.
```

ZSPI-DDL-BYTE defines a single byte.

ZSPI-DDL-BYTE-PAIR

```
def ZSPI-DDL-BYTE-PAIR.
   02 Z-BYTE           type ZSPI-DDL-BYTE     occurs 2 times.
end
```

ZSPI-DDL-BYTE-PAIR defines a pair of bytes.

ZSPI-DDL-CHAR

```
def ZSPI-DDL-CHAR            pic x              spi-null " ".
```

ZSPI-DDL-CHAR defines a single ASCII character.

ZSPI-DDL-CHAR-PAIR

```
def ZSPI-DDL-CHAR-PAIR.
   02 Z-C                  pic x(2)           spi-null " ".
   02 Z-S redefines Z-C.
      03  Z-I              type binary 16.
   02 Z-B redefines Z-C    pic x              occurs 2 times.
end
```

ZSPI-DDL-CHAR-PAIR defines a string of two ASCII characters, also addressable as a single integer or two individual characters.

ZSPI-DDL-CHAR3

```
def ZSPI-DDL-CHAR3.
   02 Z-C                pic x(3)            spi-null " ".
   02 Z-S redefines Z-C.
      03  Z-I            type binary 16.
      03  filler         pic x.
   02 Z-B redefines Z-C  pic x               occurs 3 times.
end
```

ZSPI-DDL-CHAR3 defines a string of three ASCII characters, also addressable as a single integer or three individual characters.

ZSPI-DDL-CHAR4

```
def ZSPI-DDL-CHAR4.
   02 Z-C                 pic x(4)          spi-null " ".
   02 Z-S redefines Z-C.
      03  Z-I             type binary 16    occurs 2 times.
   02 Z-B redefines Z-C   pic x             occurs 4 times.
end
```

ZSPI-DDL-CHAR4 defines a string of four ASCII characters, also addressable as two integers or four individual characters.

ZSPI-DDL-CHAR5

```
def ZSPI-DDL-CHAR5.
   02 Z-C                 pic x(5)          spi-null " ".
   02 Z-S redefines Z-C.
      03  Z-I             type binary 16    occurs 2 times.
      03  filler          pic x.
   02 Z-B redefines Z-C   pic x             occurs 5 times.
end
```

ZSPI-DDL-CHAR5 defines a string of five ASCII characters, also addressable as two integers or five individual characters.

ZSPI-DDL-CHAR6

```
def ZSPI-DDL-CHAR6.
   02 Z-C                 pic x(6)          spi-null " ".
   02 Z-S redefines Z-C.
      03  Z-I             type binary 16    occurs 3 times.
   02 Z-B redefines Z-C   pic x             occurs 6 times.
end
```

ZSPI-DDL-CHAR6 defines a string of six ASCII characters, also addressable as three integers or six individual characters.

ZSPI-DDL-CHAR7

```
def ZSPI-DDL-CHAR7.
   02 Z-C                 pic x(7)          spi-null " ".
   02 Z-S redefines Z-C.
      03  Z-I             type binary 16    occurs 3 times.
      03  filler          pic x.
   02 Z-B redefines Z-C   pic x             occurs 7 times.
end
```

ZSPI-DDL-CHAR7 defines a string of seven ASCII characters, also addressable as three integers or seven individual characters.

ZSPI-DDL-CHAR8

```
def ZSPI-DDL-CHAR8.
   02 Z-C                 pic x(8)          spi-null " ".
   02 Z-S redefines Z-C.
      03  Z-I             type binary 16    occurs 4 times.
   02 Z-B redefines Z-C   pic x             occurs 8 times.
end
```

ZSPI-DDL-CHAR8 defines a string of eight ASCII characters, also addressable as four integers or eight individual characters.

ZSPI-DDL-CHAR16

```
def ZSPI-DDL-CHAR16.
   02 Z-C                 pic x(16)         spi-null " ".
   02 Z-S redefines Z-C.
      03  Z-I             type binary 16    occurs 8 times.
   02 Z-B redefines Z-C   pic x             occurs 16 times.
end
```

ZSPI-DDL-CHAR16 defines a string of 16 ASCII characters, also addressable as 8 integers or 16 individual characters.

ZSPI-DDL-CHAR24

```
def ZSPI-DDL-CHAR24.
   02 Z-C                 pic x(24)         spi-null " ".
   02 Z-S redefines Z-C.
      03  Z-I             type binary 16    occurs 12 times.
   02 Z-B redefines Z-C   pic x             occurs 24 times.
end
```

ZSPI-DDL-CHAR24 defines a string of 24 ASCII characters, also addressable as 12 integers or 24 individual characters.

ZSPI-DDL-CHAR40

```
def ZSPI-DDL-CHAR40.
   02 Z-C                 pic x(40)         spi-null " ".
   02 Z-S redefines Z-C.
      03  Z-I             type binary 16    occurs 20 times.
   02 Z-B redefines Z-C   pic x             occurs 40 times.
end
```

ZSPI-DDL-CHAR40 defines a string of 40 ASCII characters, also addressable as 20 integers or 40 individual characters.

ZSPI-DDL-CHAR50

```
def ZSPI-DDL-CHAR50.
   02 Z-C                pic x(50)         spi-null " ".
   02 Z-S redefines Z-C.
      03  Z-I            type binary 16    occurs 25 times.
   02 Z-B redefines Z-C  pic x             occurs 50 times.
end
```

ZSPI-DDL-CHAR50 defines a string of 50 ASCII characters, also addressable as 25 integers or 50 individual characters.

ZSPI-DDL-CHAR64

```
def ZSPI-DDL-CHAR64.
   02 Z-C                pic x(64)         spi-null " ".
   02 Z-S redefines Z-C.
      03  Z-I            type binary 16    occurs 32 times.
   02 Z-B redefines Z-C  pic x             occurs 64 times.
end
```

ZSPI-DDL-CHAR64 defines a string of 64 ASCII characters, also addressable as 32 integers or 64 individual characters.

ZSPI-DDL-CHAR128

```
def ZSPI-DDL-CHAR128.
   02 Z-C                pic x(128)         spi-null " ".
   02 Z-S redefines Z-C.
      03  Z-I            type binary 16    occurs  64 times.
   02 Z-B redefines Z-C  pic x             occurs 128 times.
end
```

ZSPI-DDL-CHAR128 defines a string of 128 ASCII characters, also addressable as 64 integers or 128 individual characters.

ZSPI-DDL-CRTPID

```
def ZSPI-DDL-CRTPID      tacl crtpid.
   02 Z-PROCNAME    type ZSPI-DDL-CHAR6.
   02 Z-CRT         redefines Z-PROCNAME  type ZSPI-DDL-INT
                    occurs 3 times.
   02 Z-PID.
      03  Z-CPU     type ZSPI-DDL-BYTE    spi-null " ".
      03  Z-PIN     type ZSPI-DDL-BYTE    spi-null " ".
   02 Z-CPUPIN      redefines Z-PID
                    type ZSPI-DDL-UINT.
end
```

ZSPI-DDL-CRTPID defines an 8-byte internal-format process ID for C-series RVUs.

ZSPI-DDL-DEVICE

```
def ZSPI-DDL-DEVICE      type ZSPI-DDL-CHAR8  tacl device.
```

ZSPI-DDL-DEVICE defines an 8-byte device name.

ZSPI-DDL-DEVNAME

```
def ZSPI-DDL-DEVNAME     tacl fname.
   02 Z-DEVNAME     type ZSPI-DDL-DEVICE.
   02 Z-SUBDEVNAME  type ZSPI-DDL-CHAR8.
   02 Z-FILLER      type character 8       spi-null " ".
end
```

ZSPI-DDL-DEVNAME defines an internal-format device name.

ZSPI-DDL-DISCNAME

```
def ZSPI-DDL-DISCNAME    tacl fname.
   02 Z-VOLUME          type ZSPI-DDL-CHAR8.
   02 Z-SUBVOLUME       type ZSPI-DDL-CHAR8.
   02 Z-FILENAME        type ZSPI-DDL-CHAR8.
end
```

ZSPI-DDL-DISCNAME defines an internal-format disk name.

ZSPI-DDL-ENUM

```
def ZSPI-DDL-ENUM        pic s9(4) comp    spi-null 255
                         tacl enum.
```

SPI-DDL-ENUM defines a 16-bit enumerated value.

ZSPI-DDL-ERROR

```
def ZSPI-DDL-ERROR.
   02 Z-SSID              type ZSPI-DDL-SSID.
   02 Z-ERROR             type ZSPI-DDL-ENUM  spi null 0.
end
```

ZSPI-DDL-ERROR defines an SSID-qualified SPI error value.

ZSPI-DDL-EXIOADDR

```
def ZSPI-DDL-EXIOADDR.
   02 Z-PATYPE            type ZSPI-DDL-ENUM  spi-null 255.
   02 Z-CHNL              type ZSPI-DDL-UINT  spi-null 255.
   02 Z-CTLR              type ZSPI-DDL-UINT  spi-null 255.
   02 Z-UNIT              type ZSPI-DDL-UINT  spi-null 255.
   02 Z-CPU               type ZSPI-DDL-UINT  spi-null 255.
   02 Z-FILLER            type ZSPI-DDL-INT2  spi-null 255.
end
```

ZSPI-DDL-EXIOADDR defines an extended I/O address.

ZSPI-DDL-FLT

```
def ZSPI-DDL-FLT         type float 32     spi-null 0.
```

ZSPI-DDL-FLT defines a 32-bit floating-point number.

ZSPI-DDL-FLT2

```
def ZSPI-DDL-FLT2        type float 64     spi-null 0.
```

ZSPI-DDL-FLT2 defines a 64-bit floating-point number.

ZSPI-DDL-FNAME

```
def ZSPI-DDL-FNAME  tacl fname.
   02 Z-DISC      type ZSPI-DDL-DISCNAME.
   02 Z-PROCESS   type ZSPI-DDL-PROCNAME  redefines Z-DISC.
   02 Z-DEVICE    type ZSPI-DDL-DEVNAME   redefines Z-DISC.
end
```

ZSPI-DDL-FNAME defines a 24-byte internal-format file name.

ZSPI-DDL-FNAME32

```
def ZSPI-DDL-FNAME32      tacl fname32.
   02 Z-SYSNAME           type ZSPI-DDL-CHAR8.
   02 Z-LOCALNAME         type ZSPI-DDL-FNAME.
end
```

ZSPI-DDL-FNAME32 defines a 32-byte internal file name.

ZSPI-DDL-HEADER

```
def ZSPI-DDL-HEADER.
   02 Z-MSGCODE       type ZSPI-DDL-INT.
   02 Z-BUFLEN        type ZSPI-DDL-UINT.
   02 Z-OCCURS        type ZSPI-DDL-UINT.
   02 Z-FILLER        type ZSPI-DDL-BYTE
                        occurs 0 to 94 times
                        depending on Z-OCCURS.
end
```

ZSPI-DDL-HEADER defines an SPI message header.

ZSPI-DDL-INT

```
def ZSPI-DDL-INT          type binary 16    spi-null 0.
```

ZSPI-DDL-INT defines a 16-bit signed integer.

ZSPI-DDL-INT-PAIR

```
def ZSPI-DDL-INT-PAIR.
   02 Z-INT          type ZSPI-DDL-INT      occurs 2 times.
end
```

ZSPI-DDL-INT-PAIR defines a pair of 16-bit integers.

ZSPI-DDL-INT2

```
def ZSPI-DDL-INT2         type binary 32    spi-null 0.
```

ZSPI-DDL-INT2 defines a 32-bit signed integer.

ZSPI-DDL-INT2-PAIR

```
def ZSPI-DDL-INT2-PAIR.
   02 Z-INT2        type ZSPI-DDL-INT2     occurs 2 times.
end
```

ZSPI-DDL-INT2-PAIR defines a pair of 32-bit integers.

ZSPI-DDL-INT4

```
def ZSPI-DDL-INT4       type binary 64     spi-null 0.
```

ZSPI-DDL-INT4 defines a 64-bit fixed-point number.

ZSPI-DDL-PARM-ERR

```
def ZSPI-DDL-PARM-ERR.
   02 Z-TOKENCODE         type ZSPI-DDL-TOKENCODE.
   02 Z-INDEX             type ZSPI-DDL-UINT.
   02 Z-OFFSET            type ZSPI-DDL-UINT.
end
```

ZSPI-DDL-PARM-ERR defines a structure returned in error lists when an SPI server reports a parameter error in a command or procedure call made by the server.

Z-TOKENCODE

is the token code, or the first 32 bits of the token map, of the token involved in the error. Usually this corresponds to the first 32 bits of the *token-id* parameter passed to one of the SPI procedures.

Z-INDEX

Z-INDEX identifies which occurrence of the token was involved in the error when multiple occurrences of the token are in the buffer. Usually this corresponds to the *index* or *source-index* parameter passed to one of the SPI procedures. If an *index* or *source-index* parameter was not given, Z-INDEX is zero.

Z-OFFSET

Z-OFFSET gives the offset, in bytes from the beginning of the token value, of the parameter in error. For a simple token, Z-OFFSET is zero. The value of this field corresponds to the offset given by DDL in a DEFLIST listing.

ZSPI-DDL-PHANDLE

```
def ZSPI-DDL-PHANDLE.
  02 Z-BYTE  type ZSPI-DDL-BYTE occurs 20 times  spi-null
255.
end
```

ZSPI-DDL-PHANDLE defines a process handle for D-series RVUs.

ZSPI-DDL-PROCNAME

```
def ZSPI-DDL-PROCNAME    tacl fname.
   02 Z-CRTPID           type ZSPI-DDL-CRTPID.
   02 Z-QUAL1            type ZSPI-DDL-CHAR8.
   02 Z-QUAL2            type ZSPI-DDL-CHAR8.
end
```

ZSPI-DDL-PROCNAME defines an internal-format process file name.

ZSPI-DDL-SSID

```
def ZSPI-DDL-SSID   tacl ssid.
   02 Z-OWNER        type ZSPI-DDL-CHAR8    spi-null 0.
   02 Z-NUMBER       type ZSPI-DDL-INT.
   02 Z-VERSION      type ZSPI-DDL-UINT.
end
```

ZSPI-DDL-SSID defines a subsystem ID.

ZSPI-DDL-SUBVOL

```
def ZSPI-DDL-SUBVOL      tacl subvol.
   02 Z-VOLUME      type ZSPI-DDL-CHAR8.
   02 Z-DEVNAME     redefines Z-VOLUME
                                  type ZSPI-DDL-CHAR8.
   02 Z-SUBVOLUME   type ZSPI-DDL-CHAR8.
   02 Z-SUBDEVNAME  redefines Z-SUBVOLUME
                                  type ZSPI-DDL-CHAR8.
end
```

ZSPI-DDL-SUBVOL defines a 16-byte subvolume name for the NonStop server.

ZSPI-DDL-TIMESTAMP

```
def ZSPI-DDL-TIMESTAMP   type ZSPI-DDL-INT4   spi-null 255
                         tacl tstamp.
```

ZSPI-DDL-TIMESTAMP defines a 64-bit Julian timestamp.

ZSPI-DDL-TOKENCODE

```
def ZSPI-DDL-TOKENCODE.
   02 Z-TKN.
      03 Z-DATATYPE      type ZSPI-DDL-BYTE.
      03 Z-BYTELEN       type ZSPI-DDL-BYTE.
      03 Z-NUMBER        type ZSPI-DDL-INT.
   02 Z-TKNTYPE          redefines Z-TKN
                         type ZSPI-DDL-INT occurs 2 times.
   02 Z-TKNCODE          redefines Z-TKN
                         type ZSPI-DDL-INT2.
end
```

ZSPI-DDL-TOKENCODE defines a token code for an SSGET or SSPUT special operation for which the token value (in the *token-value* parameter) is itself a token code.

ZSPI-DDL-TRANSID

```
def ZSPI-DDL-TRANSID tacl transid.
   02 Z-TRANSID          type binary 64.
end
```

ZSPI-DDL-TRANSID defines a 64-bit internal-format transaction ID for the HP NonStop Transaction Management Facility (TMF).

ZSPI-DDL-UINT

```
def ZSPI-DDL-UINT   type binary 16 unsigned   spi-null 0.
```

ZSPI-DDL-UINT defines a 16-bit unsigned integer.

ZSPI-DDL-USERID

```
def ZSPI-DDL-USERID      type ZSPI-DDL-BYTE-PAIR.
```

ZSPI-DDL-USERID defines a 2-byte user ID for the NonStop server.

ZSPI-DDL-USERNAME

```
def ZSPI-DDL-USERNAME    tacl username.
   02 Z-GROUPNAME        type ZSPI-DDL-CHAR8.
   02 Z-USERNAME         type ZSPI-DDL-CHAR8.
end
```

ZSPI-DDL-USERNAME defines a 16-byte internal-format user name.

ZSPI-DDL-VERSION

```
def ZSPI-DDL-VERSION    type binary 16 unsigned    spi-null
0.
```

ZSPI-DDL-VERSION is used with a labeled dump version formatting procedure. It is equivalent to ZSPI-DDL-UINT.

Token Data Types

A token data type is part of every token code and identifies the fundamental data type of the token's value. For the data definition of a particular type, see the corresponding token type (ZSPI-TYP-…) in Token Types on page 4-18.

Table 4-1. SPI-Defined Token Data Types (ZSPI-TDT-…) (page 1 of 2)

Token Data Type	Type of Data Item
ZSPI-TDT-BOOLEAN	Boolean value
ZSPI-TDT-BYTE	8-bit unsigned integer
ZSPI-TDT-CHAR	8-bit ASCII character
ZSPI-TDT-CRTPID	8-byte internal-format process ID for C-series RVUs
ZSPI-TDT-DEVICE	8-byte internal-format device name
ZSPI-TDT-ENUM	16-bit signed item with an enumerated set of values
ZSPI-TDT-ERROR	Fully qualified error token
ZSPI-TDT-FLT	32-bit floating-point number
ZSPI-TDT-FLT2	64-bit floating-point number
ZSPI-TDT-FNAME	24-byte internal-format file name
ZSPI-TDT-FNAME32	8-byte node name and 24-byte local file name
ZSPI-TDT-INT	16-bit signed integer
ZSPI-TDT-INT2	32-bit signed integer
ZSPI-TDT-INT4	64-bit fixed-point number
ZSPI-TDT-LIST	Start-of-list token
ZSPI-TDT-MAP	Token map
ZSPI-TDT-MARK	Marker token for marking a buffer position
ZSPI-TDT-PHANDLE	10-word D-series process handle
ZSPI-TDT-SSCTL	SPI control token
ZSPI-TDT-SSID	6-word SPI subsystem identifier
ZSPI-TDT-STRUCT	Subsystem-defined data structure
ZSPI-TDT-SUBVOL	First 16 bytes of an internal-format file name
ZSPI-TDT-TIMESTAMP	64-bit, microsecond-resolution timestamp or elapsed time

Table 4-1. SPI-Defined Token Data Types (ZSPI-TDT-…) (page 2 of 2)

Token Data Type	Type of Data Item
ZSPI-TDT-TOKENCODE	32-bit token code
ZSPI-TDT-TRANSID	64-bit TMF internal-format transaction ID
ZSPI-TDT-UINT	16-bit unsigned integer
ZSPI-TDT-UNDEF	Unknown or undefined data type
ZSPI-TDT-USERNAME	16-byte internal-format user name

ZSPI-TDT-BOOLEAN

```
constant          ZSPI-TDT-BOOLEAN                  value is 10.
```

The BOOLEAN data type identifies a 16-bit signed Boolean item containing one of the values ZSPI-VAL-TRUE or ZSPI-VAL-FALSE.

ZSPI-TDT-BYTE

```
constant          ZSPI-TDT-BYTE                     value is 12.
```

The BYTE data type identifies an 8-bit unsigned integer in the range 0 to 255. This data type is not supported by COBOL.

ZSPI-TDT-CHAR

```
constant          ZSPI-TDT-CHAR                      value is 1.
```

The CHAR data type identifies an 8-bit ASCII character.

ZSPI-TDT-CRTPID

```
constant          ZSPI-TDT-CRTPID                   value is 22.
```

The CRTPID data type identifies an 8-byte internal-format process ID for C-series RVUs.

ZSPI-TDT-DEVICE

```
constant          ZSPI-TDT-DEVICE                   value is 21.
```

The DEVICE data type identifies an 8-byte internal-format device name. Its fields can be addressed as either string or integer values (except in TACL).

ZSPI-TDT-ENUM

```
constant          ZSPI-TDT-ENUM                    value is 11.
```

The ENUM data type identifies a 16-bit signed item for which the range of acceptable values is enumerated. The maximum range of numeric values for this type is -32768 to +32767. Its format is the same as ZSPI-TDT-INT.

ZSPI-TDT-ERROR

```
constant          ZSPI-TDT-ERROR                   value is 28.
```

The ERROR data type identifies an error token in fully qualified form (including the subsystem ID).

ZSPI-TDT-FLT

```
constant          ZSPI-TDT-FLT                      value is 5.
```

The FLT data type identifies a 32-bit floating-point (real) number. Its value range is ± 8.63617 * (10**-78) to ± 1.15792 * (10**77). This data type is not supported by COBOL or TACL.

ZSPI-TDT-FLT2

```
constant          ZSPI-TDT-FLT2                     value is 6.
```

The FLT2 data type identifies a 64-bit floating-point (real) number. Its value range is ± 8.6361685550944446 * (10**-78) to ± 1.15792089237316189 * (10**77). This data type is not supported by COBOL or TACL.

ZSPI-TDT-FNAME

```
constant          ZSPI-TDT-FNAME                   value is 20.
```

The FNAME data type identifies a 24-byte internal-format file name for a disk file, process, or device, as might be generated by the FNAMEEXPAND procedure.

ZSPI-TDT-FNAME32

```
constant          ZSPI-TDT-FNAME32                 value is 25.
```

The FNAME32 data type identifies a 32-byte internal file name of the form used by the Distributed Name Service, Pathway, data-communications subsystems, and some

other subsystems in event messages and error lists. This form consists of an 8-byte internal-format node name followed by a 24-byte internal-format local file name.

ZSPI-TDT-INT

```
constant          ZSPI-TDT-INT                          value is 2.
```

The INT data type identifies a 16-bit signed integer. Its value range is -32768 to +32767.

ZSPI-TDT-INT2

```
constant          ZSPI-TDT-INT2                         value is 3.
```

The INT2 data type identifies a 32-bit signed integer. Its value range is -2,147,483,648 to +2,147,483,647.

ZSPI-TDT-INT4

```
constant          ZSPI-TDT-INT4                         value is 4.
```

The INT4 data type identifies a 64-bit fixed-point number. Its value range is –9,223,372,036,854,775,808 to +9,223,372,036,854,775,807.

ZSPI-TDT-LIST

```
constant          ZSPI-TDT-LIST                        value is 37.
```

The LIST data type identifies a list (seen as a unit) or a token indicating the start of a list. A token of this data type always has a token length of zero.

ZSPI-TDT-MAP

```
constant          ZSPI-TDT-MAP                          value is 8.
```

The MAP data type identifies a token map.

ZSPI-TDT-MARK

```
constant     ZSPI-TDT-MARK                             value is 31.
```

The MARK data type identifies a special kind of token used to tag the token that follows it in the buffer. Such tokens are used occasionally by NonStop Kernel subsystems for special purposes; for instance, a token with this token data type

appears in every event message to mark this token as the subject token. A token with this token data type has no token value; that is, it has a zero token length.

ZSPI-TDT-PHANDLE

```
constant          ZSPI-TDT-PHANDLE                    value is 32.
```

The PHANDLE data type identifies a process handle for D-series RVUs.

ZSPI-TDT-SSCTL

```
constant          ZSPI-TDT-SSCTL                      value is 39.
```

The SSCTL data type identifies a special token code that directs SPI procedures to perform a control operation on the buffer, such as clearing error information, flushing data from the buffer, or ending a list. This token data type is reserved for use by SPI only; servers you write should not use it.

ZSPI-TDT-SSID

```
constant          ZSPI-TDT-SSID                       value is 24.
```

The SSID data type identifies a 6-word SPI subsystem identifier.

ZSPI-TDT-STRUCT

```
constant          ZSPI-TDT-STRUCT                      value is 7.
```

The STRUCT data type identifies a structured data item whose internal structure is subsystem defined. This token data type identifies an extensible structured token inside an SPI buffer. It is also used for certain fixed-length structures, such as ZSPI-TKN-PARM-ERR. It can also be used for structures whose last component varies in size, resulting in a variable-length structure.

ZSPI-TDT-SUBVOL

```
constant          ZSPI-TDT-SUBVOL                     value is 26.
```

The SUBVOL data type identifies the first two parts (16 bytes) of an internal-format file name. Subsystems normally use this token data type for a disk volume name and subvolume name, but they sometimes also use it for a device name and subdevice name or for a process name and its first qualifier name. (For subdevice names, subsystems often use ZSPI-TDT-FNAME instead.)

ZSPI-TDT-TIMESTAMP

```
constant          ZSPI-TDT-TIMESTAMP                value is 23.
```

The TIMESTAMP data type identifies a 64-bit, microsecond-resolution Julian timestamp (in Greenwich Mean Time) or an elapsed-time value in microseconds.

ZSPI-TDT-TOKENCODE

```
constant          ZSPI-TDT-TOKENCODE                value is 29.
```

The TOKENCODE data type is used for special SSGET, SSPUT, and EMSGET operations for which the *token-value* parameter is itself a token code. These operations include getting the address, length, offset, or number of occurrences of a token in the buffer; deleting a specified token from the buffer; getting the next token or the next nonmatching token; and getting the subject token from an event message. For some of these operations, the token code in *token-value* is an input parameter; in others, it is an output parameter. The desired result for some operations is returned in another parameter.

ZSPI-TDT-TRANSID

```
constant          ZSPI-TDT-TRANSID                  value is 30.
```

The TRANSID data type identifies a 64-bit TMF internal-format transaction ID.

ZSPI-TDT-UINT

```
constant          ZSPI-TDT-UINT                      value is 9.
```

The UINT data type identifies a 16-bit unsigned integer whose value can range from 0 through 65535.

ZSPI-TDT-UNDEF

```
constant          ZSPI-TDT-UNDEF                     value is 0.
```

The UNDEF data type identifies an unknown or undefined data type. This data type is reserved for use by software for the NonStop system only.

ZSPI-TDT-USERNAME

```
constant          ZSPI-TDT-USERNAME                  value is 27.
```

The USERNAME data type identifies a 16-byte internal-format user name, such as the USERIDTOUSERNAME procedure might generate.

Token Types

Every token has a token type. A token type (TYP) associates a defined token data type (TDT) with a corresponding data definition. The token type can then be used in the TOKEN TYPE IS clause of a token code definition. Using the standard token data types, SPI defines a number of token types, which are described here. Most of these token types are based on structure definitions.

ZSPI-TYP-BOOLEAN

```
token type  ZSPI-TYP-BOOLEAN         value is ZSPI-TDT-BOOLEAN
                                      def is ZSPI-DDL-BOOLEAN.
```

ZSPI-TYP-BOOLEAN specifies that a token's value is Boolean. A token of this type has a BOOLEAN token data type and is based on the BOOLEAN data structure.

ZSPI-TYP-BYTE

```
token type  ZSPI-TYP-BYTE            value is ZSPI-TDT-BYTE
                                       def is ZSPI-DDL-BYTE.
```

ZSPI-TYP-BYTE specifies that a token's value is a single byte. A token of this type has a BYTE token data type and is based on the BYTE structure.

ZSPI-TYP-BYTE-PAIR

```
token type  ZSPI-TYP-BYTE-PAIR      value is ZSPI-TDT-BYTE
                                    def is ZSPI-DDL-BYTE-PAIR.
```

ZSPI-TYP-BYTE-PAIR specifies that a token has a 2-byte value. A token of this type has a BYTE token data type and is based on the BYTE-PAIR structure.

ZSPI-TYP-BYTESTRING

```
token type  ZSPI-TYP-BYTESTRING      value is ZSPI-TDT-BYTE
                                       occurs varying.
```

ZSPI-TYP-BYTESTRING specifies that a token's value is a string of bytes. A token of this type has a BYTE token data type and can vary in length.

ZSPI-TYP-CHAR

```
token type  ZSPI-TYP-CHAR           value is ZSPI-TDT-CHAR
                                      def is ZSPI-DDL-CHAR.
```

ZSPI-TYP-CHAR specifies that a token's value consists of a single ASCII character. A token of this type has a CHAR token data type and is based on the CHAR structure.

ZSPI-TYP-CHAR-PAIR

```
token type  ZSPI-TYP-CHAR-PAIR     value is ZSPI-TDT-CHAR
                                     def is ZSPI-DDL-CHAR-
PAIR.
```

ZSPI-TYP-CHAR-PAIR specifies that a token's value consists of two ASCII characters. A token of this type has a CHAR token data type and is based on the CHAR-PAIR structure.

ZSPI-TYP-CHAR3

```
token type  ZSPI-TYP-CHAR3          value is ZSPI-TDT-CHAR
                                      def is ZSPI-DDL-CHAR3.
```

ZSPI-TYP-CHAR3 specifies that a token's value consists of three ASCII characters. A token of this type has a CHAR token data type and is based on the CHAR3 structure.

ZSPI-TYP-CHAR4

```
token type  ZSPI-TYP-CHAR4          value is ZSPI-TDT-CHAR
                                      def is ZSPI-DDL-CHAR4.
```

ZSPI-TYP-CHAR4 specifies that a token's value consists of four ASCII characters. A token of this type has a CHAR token data type and is based on the CHAR4 structure.

ZSPI-TYP-CHAR5

```
token type  ZSPI-TYP-CHAR5          value is ZSPI-TDT-CHAR
                                      def is ZSPI-DDL-CHAR5.
```

ZSPI-TYP-CHAR5 specifies that a token's value consists of five ASCII characters. A token of this type has a CHAR token data type and is based on the CHAR5 structure.

ZSPI-TYP-CHAR6

```
token type  ZSPI-TYP-CHAR6          value is ZSPI-TDT-CHAR
                                       def is ZSPI-DDL-CHAR6.
```

ZSPI-TYP-CHAR6 specifies that a token's value consists of six ASCII characters. A token of this type has a CHAR token data type and is based on the CHAR6 structure.

ZSPI-TYP-CHAR7

```
token type  ZSPI-TYP-CHAR7          value is ZSPI-TDT-CHAR
                                       def is ZSPI-DDL-CHAR7.
```

ZSPI-TYP-CHAR7 specifies that a token's value consists of seven ASCII characters. A token of this type has a CHAR token data type and is based on the CHAR7 structure.

ZSPI-TYP-CHAR8

```
token type  ZSPI-TYP-CHAR8          value is ZSPI-TDT-CHAR
                                       def is ZSPI-DDL-CHAR8.
```

ZSPI-TYP-CHAR8 specifies that a token's value consists of eight ASCII characters. A token of this type has a CHAR token data type and is based on the CHAR8 structure.

ZSPI-TYP-CHAR16

```
token type  ZSPI-TYP-CHAR16         value is ZSPI-TDT-CHAR
                                       def is ZSPI-DDL-CHAR16.
```

ZSPI-TYP-CHAR16 specifies that a token's value consists of 16 ASCII characters. A token of this type has a CHAR token data type and is based on the CHAR16 structure.

ZSPI-TYP-CHAR24

```
token type  ZSPI-TYP-CHAR24         value is ZSPI-TDT-CHAR
                                       def is ZSPI-DDL-CHAR24.
```

ZSPI-TYP-CHAR24 specifies that a token's value consists of 24 ASCII characters. A token of this type has a CHAR token data type and is based on the CHAR24 structure.

ZSPI-TYP-CHAR40

```
token type  ZSPI-TYP-CHAR40        value is ZSPI-TDT-CHAR
                                     def is ZSPI-DDL-CHAR40.
```

ZSPI-TYP-CHAR40 specifies that a token's value consists of 40 ASCII characters. A token of this type has a CHAR token data type and is based on the CHAR40 structure.

ZSPI-TYP-CHAR50

```
token type  ZSPI-TYP-CHAR50        value is ZSPI-TDT-CHAR
                                     def is ZSPI-DDL-CHAR50.
```

ZSPI-TYP-CHAR50 specifies that a token's value consists of 50 ASCII characters. A token of this type has a CHAR token data type and is based on the CHAR50 structure.

ZSPI-TYP-CHAR64

```
token type  ZSPI-TYP-CHAR64        value is ZSPI-TDT-CHAR
                                     def is ZSPI-DDL-CHAR64.
```

ZSPI-TYP-CHAR64 specifies that a token's value consists of 64 ASCII characters. A token of this type has a CHAR token data type and is based on the CHAR64 structure.

ZSPI-TYP-CHAR128

```
token type  ZSPI-TYP-CHAR128       value is ZSPI-TDT-CHAR
                                     def is ZSPI-DDL-CHAR128.
```

ZSPI-TYP-CHAR128 specifies that a token's value consists of 128 ASCII characters. A token of this type has a CHAR token data type and is based on the CHAR128 structure.

ZSPI-TYP-CRTPID

```
token type  ZSPI-TYP-CRTPID        value is ZSPI-TDT-CRTPID
                                     def is ZSPI-DDL-CRTPID.
```

ZSPI-TYP-CRTPID is the token type of an 8-byte internal-format process ID.

ZSPI-TYP-DEVICE

```
token type  ZSPI-TYP-DEVICE          value is ZSPI-TDT-DEVICE
                                       def is ZSPI-DDL-DEVICE.
```

ZSPI-TYP-DEVICE specifies that a token's value is an 8-byte internal-format device name. A token of this type has a DEVICE token data type and is based on the DEVICE structure.

ZSPI-TYP-ENUM

```
token type  ZSPI-TYP-ENUM            value is ZSPI-TDT-ENUM
                                       def is ZSPI-DDL-ENUM.
```

SPI-TYP-ENUM specifies that a token's value is a 16-bit enumerated value. A token of this type has an ENUM token data type and is based on the ENUM structure.

ZSPI-TYP-ERROR

```
token type  ZSPI-TYP-ERROR           value is ZSPI-TDT-ERROR
                                       def is ZSPI-DDL-ERROR.
```

ZSPI-TYP-ERROR specifies that a token's value is an SSID-qualified error number. A token of this type has an ERROR token data type and is based on the ERROR structure.

ZSPI-TYP-FLT

```
token type  ZSPI-TYP-FLT             value is ZSPI-TDT-FLT
                                       def is ZSPI-DDL-FLT.
```

ZSPI-TYP-FLT specifies that a token's value is a 32-bit floating-point number. A token of this type has an FLT token data type and is based on the FLT structure.

ZSPI-TYP-FLT2

```
token type  ZSPI-TYP-FLT2            value is ZSPI-TDT-FLT2
                                       def is ZSPI-DDL-FLT2.
```

ZSPI-TYP-FLT2 specifies that a token's value is a 64-bit floating-point number. A token of this type has an FLT2 token data type and is based on the FLT2 structure.

ZSPI-TYP-FNAME

```
token type  ZSPI-TYP-FNAME          value is ZSPI-TDT-FNAME
                                      def is ZSPI-DDL-FNAME.
```

ZSPI-TYP-FNAME specifies that a token's value is a 24-byte internal-format file name. A token of this type has an FNAME token data type and is based on the FNAME structure.

ZSPI-TYP-FNAME32

```
token type  ZSPI-TYP-FNAME32        value is ZSPI-TDT-FNAME32
                                      def is ZSPI-DDL-FNAME32.
```

ZSPI-TYP-FNAME32 specifies that a token's value is a 32-byte internal file name. A token of this type has an FNAME32 token data type and is based on the FNAME32 structure.

ZSPI-TYP-INT

```
token type  ZSPI-TYP-INT            value is ZSPI-TDT-INT
                                      def is ZSPI-DDL-INT.
```

ZSPI-TYP-INT specifies that a token's value is a 16-bit signed integer. A token of this type has an INT token data type and is based on the INT structure.

ZSPI-TYP-INT-PAIR

```
token type  ZSPI-TYP-INT-PAIR       value is ZSPI-TDT-INT
                                      def is ZSPI-DDL-INT-PAIR.
```

ZSPI-TYP-INT-PAIR specifies that a token's value is a pair of 16-bit integers. A token of this type has an INT token data type and is based on the INT-PAIR structure.

ZSPI-TYP-INT2

```
token type  ZSPI-TYP-INT2           value is ZSPI-TDT-INT2
                                      def is ZSPI-DDL-INT2.
```

ZSPI-TYP-INT2 specifies that a token's value is a 32-bit signed integer. A token of this type has an INT2 token data type and is based on the INT2 structure.

ZSPI-TYP-INT2-PAIR

```
token type  ZSPI-TYP-INT2-PAIR     value is ZSPI-TDT-INT2
                                   def is ZSPI-DDL-INT2-PAIR.
```

ZSPI-TYP-INT2-PAIR specifies that a token's value is a pair of 32-bit integers. A token of this type has an INT2 token data type and is based on the INT2-PAIR structure.

ZSPI-TYP-INT4

```
token type  ZSPI-TYP-INT4          value is ZSPI-TDT-INT4
                                     def is ZSPI-DDL-INT4.
```

ZSPI-TYP-INT4 specifies that a token's value is a 64-bit fixed-point number. A token of this type has an INT4 token data type and is based on the INT4 structure.

ZSPI-TYP-LASTERR

```
token type  ZSPI-TYP-LASTERR      value is ZSPI-TDT-ENUM
                                  def is ZSPI-DDL-ERR-ENUM.
```

ZSPI-TYP-LASTERR is used to convey SPI error numbers.

ZSPI-TYP-LIST

```
token type  ZSPI-TYP-LIST          value is ZSPI-TDT-LIST
                                     occurs 0 times.
```

ZSPI-TYP-LIST specifies that the token marks the start of a list. A token of this type has a LIST token data type and no value.

ZSPI-TYP-MAP

```
token type  ZSPI-TYP-MAP           value is ZSPI-TDT-MAP
                                     occurs varying.
```

ZSPI-TYP-MAP specifies that a token's value is a token map. A token of this type has a MAP and a variable length.

ZSPI-TYP-MARK

```
token type  ZSPI-TYP-MARK          value is ZSPI-TDT-MARK
                                     occurs 0 times.
```

ZSPI-TYP-MARK specifies that the token is used as a mark or tag for the token that follows it in the buffer. For instance, a token code with this token type appears in every

event message to mark this token as the subject token. A token of this type has a MARK token data type and no value.

ZSPI-TYP-PARM-ERR

```
token type  ZSPI-TYP-PARM-ERR     value is ZSPI-TDT-STRUCT
                                     def is ZSPI-DDL-PARM-ERR.
```

ZSPI-TYP-PARM-ERR specifies that a token's value is a structure returned by a subsystem in an error list when a parameter error occurs in a command to the subsystem or in an SPI procedure call made by the subsystem. Its value includes information to identify the parameter in error. A token of this type has a STRUCT token data type and is based on the PARM-ERR structure.

ZSPI-TYP-PHANDLE

```
token type  ZSPI-TYP-PHANDLE       value is ZSPI-TDT-PHANDLE
                                     def is ZSPI-DDL-PHANDLE.
```

ZSPI-TYP-PHANDLE specifies that a token's value is a process handle for D-series RVUs. A token of this type is based on the PHANDLE structure.

ZSPI-TYP-POSITION

```
token type  ZSPI-TYP-POSITION      value is ZSPI-TDT-SSCTL
                                     def is ZSPI-DDL-INT4.
```

ZSPI-TYP-POSITION specifies that a token's value is a 64-bit position descriptor representing a position in a buffer. A token of this type has an SSCTL token data type and is based on the INT4 structure.

ZSPI-TYP-RESPONSE-TYPE

```
token type  ZSPI-TYP-RESPONSE-TYPE     value is ZSPI-TDT-ENUM
                          def is ZSPI-DDL-RESPONSE-TYPE-ENM.
```

ZSPI-TYP-RESPONSE-TYPE specifies that a token's value is based on the RESPONSE-TYPE-ENUM enumeration structure. Subsystems must not use this token type.

ZSPI-TYP-SSCTL

```
token type  ZSPI-TYP-SSCTL          value is ZSPI-TDT-SSCTL
                                      occurs 0 times.
```

ZSPI-TYP-SSCTL specifies that a token is a special token code used to direct one of the SPI procedures to perform a control operation on the buffer, such as clearing error information, flushing data from the buffer, or ending a list. A token of this type has an SSCTL token data type and no value. Subsystems should not use this token type.

ZSPI-TYP-SSID

```
token type  ZSPI-TYP-SSID           value is ZSPI-TDT-SSID
                                      def is ZSPI-DDL-SSID.
```

ZSPI-TYP-SSID specifies that a token's value is a subsystem ID. A token of this type has an SSID token data type and is based on the SSID structure.

ZSPI-TYP-STRING

```
token type  ZSPI-TYP-STRING         value is ZSPI-TDT-CHAR
                                      occurs varying.
```

ZSPI-TYP-STRING specifies that a token's value is a variable-length ASCII character string. A token of this type has a CHAR token data type and a variable length.

ZSPI-TYP-STRUCT

```
token type  ZSPI-TYP-STRUCT         value is ZSPI-TDT-STRUCT
                                      occurs varying.
```

ZSPI-TYP-STRUCT specifies that a token's value is a subsystem-defined structure. A token of this type has a STRUCT token data type and a variable length.

ZSPI-TYP-SUBVOL

```
token type  ZSPI-TYP-SUBVOL         value is ZSPI-TDT-SUBVOL
                                      def is ZSPI-DDL-SUBVOL.
```

ZSPI-TYP-SUBVOL specifies that a token's value is a 16-byte subvolume name for the NonStop server. A token of this type has a SUBVOL token data type and is based on the SUBVOL structure.

ZSPI-TYP-TIMESTAMP

```
token type  ZSPI-TYP-TIMESTAMP    value is ZSPI-TDT-TIMESTAMP
                                    def is ZSPI-DDL-TIMESTAMP.
```

ZSPI-TYP-TIMESTAMP specifies that a token's value is a 64-bit Julian timestamp. A token of this type has a TIMESTAMP token data type and is based on the TIMESTAMP structure.

ZSPI-TYP-TOKENCODE

```
token type  ZSPI-TYP-TOKENCODE    value is ZSPI-TDT-TOKENCODE
                                    def is ZSPI-DDL-TOKENCODE.
```

ZSPI-TYP-TOKENCODE specifies that a token's value is itself a token code. A token of this type has a TOKENCODE token data type and is based on the TOKENCODE structure.

ZSPI-TYP-TRANSID

```
token type  ZSPI-TYP-TRANSID       value is ZSPI-TDT-TRANSID
                                     def is ZSPI-DDL-TRANSID.
```

ZSPI-TYP-TRANSID specifies that a token's value is a 64-bit internal-format transaction ID for TMF. A token of this type has a TRANSID token data type and is based on the TRANSID structure.

ZSPI-TYP-UINT

```
token type  ZSPI-TYP-UINT          value is ZSPI-TDT-UINT
                                     def is ZSPI-DDL-UINT.
```

ZSPI-TYP-UINT specifies that a token's value is a 16-bit unsigned integer. A token of this type has a UINT token data type and is based on the UINT structure.

ZSPI-TYP-USERID

```
token type  ZSPI-TYP-USERID        value is ZSPI-TDT-BYTE
                                   def is ZSPI-DDL-BYTE-PAIR.
```

ZSPI-TYP-USERID specifies that a token's value is a 2-byte user ID for the NonStop server. A token of this type has a BYTE token data type and is based on the BYTE-PAIR structure.

ZSPI-TYP-USERNAME

```
token type  ZSPI-TYP-USERNAME      value is ZSPI-TDT-USERNAME
                                    def is ZSPI-DDL-USERNAME.
```

ZSPI-TYP-USERNAME specifies that a token's value is a 16-byte internal-format user name. A token of this type has a USERNAME token data type and is based on the USERNAME structure.

ZSPI-TYP-VERSION

```
token type  ZSPI-TYP-VERSION       value is ZSPI-TDT-UINT
                                     def is ZSPI-DDL-VERSION.
```

ZSPI-TYP-VERSION specifies that a token's value is a software release version. A token of this type has a UINT token data type and is based on the VERSION structure.

Token Numbers

Every token has a token number. Token numbers have symbolic names of the form ZSPI-TNM-*name*, where *name* matches the name portion of the corresponding token code definition, ZSPI-TKN-*name*.

Table 4-2 lists the symbolic names and the numeric values of the token numbers defined by SPI. (Numeric values are provided for debugging purposes only—always use the symbolic name in programs.)

Table 4-2. SPI Token Numbers (page 1 of 2)

ZSPI Token Number	Value	ZSPI Token Number	Value
ZSPI-TNM-ADDR	–443	ZSPI-TNM-LIST	–255
ZSPI-TNM-ALLOW	–239	ZSPI-TNM-MANAGER	–243
ZSPI-TNM-ALLOW-TYPE	–249	ZSPI-TNM-MAX-FIELD-VERSION	–503
ZSPI-TNM-BUFLEN	–500	ZSPI-TNM-MAXRESP	–502
ZSPI-TNM-CHECKSUM	–512	ZSPI-TNM-MORE-DATA	–238
ZSPI-TNM-CLEARERR	–508	ZSPI-TNM-NEXTCODE	–448
ZSPI-TNM-COMMAND	–510	ZSPI-TNM-NEXTTOKEN	–447
ZSPI-TNM-COMMENT	–247	ZSPI-TNM-OBJECT-TYPE	–509
ZSPI-TNM-CONTEXT	–256	ZSPI-TNM-OFFSET	–444
ZSPI-TNM-COUNT	–446	ZSPI-TNM-PARM-ERR	–250
ZSPI-TNM-DATA-FLUSH	–441	ZSPI-TNM-POSITION	–442
ZSPI-TNM-DATALIST	–253	ZSPI-TNM-PROC-ERR	–244
ZSPI-TNM-DEFAULT-SSID	–437	ZSPI-TNM-PROG-FNAME	–241

Table 4-2. SPI Token Numbers (page 2 of 2)

ZSPI Token Number	Value	ZSPI Token Number	Value
ZSPI-TNM-DELETE	–440	ZSPI-TNM-RESET-BUFFER	–436
ZSPI-TNM-ENDLIST	–254	ZSPI-TNM-RESPONSE-TYPE	–248
ZSPI-TNM-ERRLIST	–252	ZSPI-TNM-RETCODE	0
ZSPI-TNM-ERROR	–251	ZSPI-TNM-SEGLIST	–240
ZSPI-TNM-HDRTYPE	–511	ZSPI-TNM-SEGMENTATION	–237
ZSPI-TNM-INITIAL-POSITION	–438	ZSPI-TNM-SERVER-BANNER	–246
ZSPI-TNM-IPM-ID	–242	ZSPI-TNM-SERVER-VERSION	–501
ZSPI-TNM-LASTERR	–507	ZSPI-TNM-SSID	–505
ZSPI-TNM-LASTERRCODE	–506	ZSPI-TNM-SSID-ERR	–245
ZSPI-TNM-LASTPOSITION	–439	ZSPI-TNM-USEDLEN	–504
ZSPI-TNM-LEN	–445		

Token Number Definition Syntax

This syntax box lists the token numbers by numeric value. (Numeric values are provided for debugging purposes only—always use the symbolic name in programs.)

```
constant   ZSPI-TNM-CHECKSUM              VALUE IS -512.
constant   ZSPI-TNM-HDRTYPE               VALUE IS -511.
constant   ZSPI-TNM-COMMAND               VALUE IS -510.
constant   ZSPI-TNM-OBJECT-TYPE           VALUE IS -509.
constant   ZSPI-TNM-CLEARERR              VALUE IS -508.
constant   ZSPI-TNM-LASTERR               VALUE IS -507.
constant   ZSPI-TNM-LASTERRCODE           VALUE IS -506.
constant   ZSPI-TNM-SSID                  VALUE IS -505.
constant   ZSPI-TNM-USEDLEN               VALUE IS -504.
constant   ZSPI-TNM-MAX-FIELD-VERSION     VALUE IS -503.
constant   ZSPI-TNM-MAXRESP               VALUE IS -502.
constant   ZSPI-TNM-SERVER-VERSION        VALUE IS -501.
constant   ZSPI-TNM-BUFLEN                VALUE IS -500.
constant   ZSPI-TNM-NEXTCODE              VALUE IS -448.
constant   ZSPI-TNM-NEXTTOKEN             VALUE IS -447.
constant   ZSPI-TNM-COUNT                 VALUE IS -446.
constant   ZSPI-TNM-LEN                   VALUE IS -445.
constant   ZSPI-TNM-OFFSET                VALUE IS -444.
constant   ZSPI-TNM-ADDR                  VALUE IS -443.
constant   ZSPI-TNM-POSITION              VALUE IS -442.
constant   ZSPI-TNM-DATA-FLUSH            VALUE IS -441.
constant   ZSPI-TNM-DELETE                VALUE IS -440.
constant   ZSPI-TNM-LASTPOSITION          VALUE IS -439.
constant   ZSPI-TNM-INITIAL-POSITION      VALUE IS -438.
constant   ZSPI-TNM-DEFAULT-SSID          VALUE IS -437.
constant   ZSPI-TNM-RESET-BUFFER          VALUE IS -436.
constant   ZSPI-TNM-CONTEXT               VALUE IS -256.
constant   ZSPI-TNM-LIST                  VALUE IS -255.
constant   ZSPI-TNM-ENDLIST               VALUE IS -254.
constant   ZSPI-TNM-DATALIST              VALUE IS -253.
constant   ZSPI-TNM-ERRLIST               VALUE IS -252.
constant   ZSPI-TNM-ERROR                 VALUE IS -251.
constant   ZSPI-TNM-PARM-ERR              VALUE IS -250.
constant   ZSPI-TNM-ALLOW-TYPE            VALUE IS -249.
constant   ZSPI-TNM-RESPONSE-TYPE         VALUE IS -248.
constant   ZSPI-TNM-COMMENT               VALUE IS -247.

constant   ZSPI-TNM-SERVER-BANNER         VALUE IS -246.
constant   ZSPI-TNM-SSID-ERR              VALUE IS -245.
constant   ZSPI-TNM-PROC-ERR              VALUE IS -244.
constant   ZSPI-TNM-MANAGER               VALUE IS -243.
constant   ZSPI-TNM-IPM-ID                VALUE IS -242.
constant   ZSPI-TNM-PROG-FNAME            VALUE IS -241.
constant   ZSPI-TNM-SEGLIST               VALUE IS -240.
constant   ZSPI-TNM-ALLOW                 VALUE IS -239.
constant   ZSPI-TNM-MORE-DATA             VALUE IS -238.
constant   ZSPI-TNM-SEGMENTATION          VALUE IS -237.
constant   ZSPI-TNM-RETCODE               VALUE IS    0.
```

Token Codes

This subsection describes the token codes defined by SPI. A token code associates a token number with a token type, an SSID, and a heading for formatted display of SPI messages.

ZSPI-TKN-ADDR

```
token-code        ZSPI-TKN-ADDR
  value           ZSPI-TNM-ADDR
  token-type      ZSPI-TYP-TOKENCODE
  ssid            ZSPI-VAL-NULL-EXT-SSID
  heading         "Token_address".
```

ZSPI-TKN-ADDR is a special operation token that directs SSGET to return the address of a specified token value.

ZSPI-TKN-ALLOW

```
token-code        ZSPI-TKN-ALLOW
  value           ZSPI-TNM-ALLOW
  token-type      ZSPI-TYP-ENUM
  ssid            ZSPI-VAL-NULL-EXT-SSID
  heading         "Function_allowed".
```

A requester uses this token to ask for a segmented response by assigning the value ZSPI-VAL-ALLOW-SEGMENTS and including the token in the command.

ZSPI-TKN-ALLOW-TYPE

```
token-code        ZSPI-TKN-ALLOW-TYPE
  value           ZSPI-TNM-ALLOW-TYPE
  token-type      ZSPI-TYP-ALLOW-TYPE
  ssid            ZSPI-VAL-NULL-EXT-SSID
  heading         "Allow_type".
```

Requesters include ZSPI-TKN-ALLOW-TYPE in a command to tell a server how to proceed when it encounters an error or warning on an object.

Valid values for this token are:

ZSPI-VAL-NORM-ONLY	The server continues command processing with the next object in the set only if the command was completely successful on the previous object (no error list). This is the default if ZSPI-TKN-ALLOW-TYPE is not included in the command.
ZSPI-VAL-WARN-AND-NORM	The server processes the next object in the set even if a warning is encountered on the previous object. (A warning means that ZSPI-TKN-RETCODE contains ZSPI-ERR-OK, but the response contains an error list.)
ZSPI-VAL-ERR-WARN-AND-NORM	The server processes the next object in the set regardless of any problems encountered on the previous object.

ZSPI-TKN-BUFLEN

```
token-code        ZSPI-TKN-BUFLEN
  value           ZSPI-TNM-BUFLEN
  token-type      ZSPI-TYP-UINT
  ssid            ZSPI-VAL-NULL-EXT-SSID
  heading         "Buffer_byte_length".
```

ZSPI-TKN-BUFLEN contains the SPI buffer length. This value is set by SSINIT, can be retrieved using SSGET, and can be modified using SSPUT.

ZSPI-TKN-CHECKSUM

```
token-code        ZSPI-TKN-CHECKSUM
  value           ZSPI-TNM-CHECKSUM
  token-type      ZSPI-TYP-BOOLEAN
  ssid            ZSPI-VAL-NULL-EXT-SSID
  heading         "Checksum".
```

ZSPI-TKN-CHECKSUM contains the buffer checksum flag. If set to zero, the SPI procedures do not maintain a buffer checksum. If set to any nonzero value, SPI maintains a checksum to validate buffer integrity. This value is set by SSINIT, can be retrieved using SSGET, and can be modified using SSPUT.

ZSPI-TKN-CLEARERR

```
token-code        ZSPI-TKN-CLEARERR
  value           ZSPI-TNM-CLEARERR
  token-type      ZSPI-TYP-SSCTL.
```

ZSPI-TKN-CLEARERR, passed to SSPUT, clears (resets to zero) the last-error information in the SPI message header. Its token type is ZSPI-TYP-SSCTL.

ZSPI-TKN-COMMAND

```
token-code        ZSPI-TKN-COMMAND
  value           ZSPI-TNM-COMMAND
  token-type      ZSPI-TYP-ENUM
  ssid            ZSPI-VAL-NULL-EXT-SSID
  heading         "Command".
```

ZSPI-TKN-COMMAND contains the command number. This value is set by SSINIT and can be retrieved using SSGET. This value cannot be modified using SSPUT.

ZSPI-TKN-COMMENT

```
token-code        ZSPI-TKN-COMMENT
  value           ZSPI-TNM-COMMENT
  token-type      ZSPI-TYP-STRING
  ssid            ZSPI-VAL-NULL-EXT-SSID
  heading         "Comment".
```

ZSPI-TKN-COMMENT is a token that a requester can use to include arbitrary information of its own in a command. NonStop Kernel subsystems ignore this token and do not return it in responses. Its token type is ZSPI-TYP-STRING. Its value is a variable-length character string. The buffer sizes recommended by NonStop Kernel subsystems allow for one 80-byte comment token in every command.

ZSPI-TKN-CONTEXT

```
token-code        ZSPI-TKN-CONTEXT
  value           ZSPI-TNM-CONTEXT
  token-type      ZSPI-TYP-BYTESTRING
  ssid            ZSPI-VAL-NULL-EXT-SSID
  heading         "Context".
```

ZSPI-TKN-CONTEXT is a special token that indicates (by its presence or absence) whether or not there are more objects to process. If this token is present in a response, the response can be continued in another response message; if it is absent, this is the last response message. The token value provides information needed by the server to determine where to resume processing. The requester should ignore the token value, but must send the token back to the server in a copy of the original command message. For more information, see Section 2, SPI Concepts and Protocol.

The token type of this token is ZSPI-TYP-BYTESTRING.

ZSPI-TKN-COUNT

```
token-code        ZSPI-TKN-COUNT
  value           ZSPI-TNM-COUNT
  token-type      ZSPI-TYP-TOKENCODE
  ssid            ZSPI-VAL-NULL-EXT-SSID
  heading         "Token_count".
```

ZSPI-TKN-COUNT directs SSGET to obtain the total number of occurrences of a specific token in the buffer. Its token type is ZSPI-TYP-TOKENCODE.

ZSPI-TKN-DATA-FLUSH

```
token-code        ZSPI-TKN-DATA-FLUSH
  value           ZSPI-TNM-DATA-FLUSH
  token-type      ZSPI-TYP-SSCTL.
```

ZSPI-TKN-DATA-FLUSH directs SSPUT to flush all information in the buffer starting at, and including, the token at the current position. Its token type is ZSPI-TYP-SSCTL.

ZSPI-TKN-DATALIST

```
token-code        ZSPI-TKN-DATALIST
  value           ZSPI-TNM-DATALIST
  token-type      ZSPI-TYP-LIST
  ssid            ZSPI-VAL-NULL-EXT-SSID
  heading         "Data_list".
```

ZSPI-TKN-DATALIST begins a list that encloses a single response record when multiple response records are contained in a single buffer, when the buffer indicates a nonzero value for the header token ZSPI-TKN-MAXRESP, or when a segmented response is generated. Its token type is ZSPI-TYP-LIST. Calling SSGET with ZSPI-TKN-DATALIST selects the next list that begins with that token so that tokens within it can be read; calling SSPUT with this token places it in the buffer and starts a new list.

ZSPI-TKN-DEFAULT-SSID

```
token-code        ZSPI-TKN-DEFAULT-SSID
  value           ZSPI-TNM-DEFAULT-SSID
  token-type      ZSPI-TYP-SSID.
```

ZSPI-TKN-DEFAULT-SSID directs SSGET to get the default subsystem ID of the token at the current position. Its token type is ZSPI-TYP-SSID.

ZSPI-TKN-DELETE

```
token-code        ZSPI-TKN-DELETE
  value           ZSPI-TNM-DELETE
  token-type      ZSPI-TYP-TOKENCODE.
```

ZSPI-TKN-DELETE directs SSPUT to delete a specified token from the buffer. Its token type is ZSPI-TYP-TOKENCODE.

ZSPI-TKN-ENDLIST

```
token-code        ZSPI-TKN-ENDLIST
  value           ZSPI-TNM-ENDLIST
  token-type      ZSPI-TYP-SSCTL
  ssid            ZSPI-VAL-NULL-EXT-SSID
  heading         "End_of_list".
```

ZSPI-TKN-ENDLIST ends the current list (of any type). Its token type is ZSPI-TYP-SSCTL.

Calling SSGET with ZSPI-TKN-ENDLIST pops out of the list. Calling SSPUT with ZSPI-TKN-ENDLIST places that token in the buffer, thus ending the list, and pops out of the list.

ZSPI-TKN-ERRLIST

```
token-code        ZSPI-TKN-ERRLIST
  value           ZSPI-TNM-ERRLIST
  token-type      ZSPI-TYP-LIST
  ssid            ZSPI-VAL-NULL-EXT-SSID
  heading         "Error_list".
```

ZSPI-TKN-ERRLIST begins a list that encloses information about an error. Its token type is ZSPI-TYP-LIST. Calling SSGET with ZSPI-TKN-ERRLIST selects the next list that begins with that token, so that tokens within it can be read; calling SSPUT with this token places it in the buffer, thus starting a new error list.

ZSPI-TKN-ERROR

```
token-code        ZSPI-TKN-ERROR
  value           ZSPI-TNM-ERROR
  token-type      ZSPI-TYP-ERROR
  ssid            ZSPI-VAL-NULL-EXT-SSID
  heading         "Error".
```

ZSPI-TKN-ERROR is a token, returned within an error list in a response, that identifies an error detected by a subsystem. Its token type is ZSPI-TYP-ERROR. Its value consists of the subsystem ID of the subsystem that found the error, followed by the

error number (defined by that subsystem). For more information, see Error Numbers on page 4-46. For information about error lists, see Section 2, SPI Concepts and Protocol.

ZSPI-TKN-HDRTYPE

```
token-code        ZSPI-TKN-HDRTYPE
  value           ZSPI-TNM-HDRTYPE
  token-type      ZSPI-TYP-UINT
  ssid            ZSPI-VAL-NULL-EXT-SSID
  heading         "Header_type".
```

ZSPI-TKN-HDRTYPE identifies the type of header that the message contains. (The message buffer used by the EMS—like the command or response buffer used in control and inquiry—is an SPI buffer, but some of the tokens in the header of an event message differ from those in the header of a command or response message.) This value is set by SSINIT, and can be retrieved using SSGET. This value cannot be modified using SSPUT.

ZSPI-TKN-INITIAL-POSITION

```
token-code        ZSPI-TKN-INITIAL-POSITION
  value           ZSPI-TNM-INITIAL-POSITION
  token-type      ZSPI-TYP-BOOLEAN.
```

ZSPI-TKN-INITIAL-POSITION, passed to SSPUT, resets the current position as indicated by the supplied token value. Its token type is ZSPI-TYP-BOOLEAN. A token value of ZSPI-VAL-INITIAL-BUFFER (0) resets the position to the beginning of the buffer. A token value of ZSPI-VAL-INITIAL-LIST (-1) resets the position to the beginning of the current list.

ZSPI-TKN-IPM-ID

```
token-code        ZSPI-TKN-IPM-ID
  value           ZSPI-TNM-IPM-ID
  token-type      ZSPI-TYP-ENUM
  ssid            ZSPI-VAL-NULL-EXT-SSID
  heading         "IPM_ID".
```

ZSPI-TKN-IPM-ID indicates that the subsystem version returned in the GETVERSION command has had a software product revision (SPR) since the version was released. ZSPI-TKN-IPM-ID can occur multiple times if more than one change to the SPI interface was made in the SPR. The subsystem might also return an IPM-ID token for a change in the SPR that did not affect the SPI interface.

ZSPI-TKN-LASTERR

```
token-code        ZSPI-TKN-LASTERR
  value           ZSPI-TNM-LASTERR
  token-type      ZSPI-TYP-LASTERR
  ssid            ZSPI-VAL-NULL-EXT-SSID
  heading         "Last_error".
```

ZSPI-TKN-LASTERR records the last nonzero status code returned by an SPI procedure while processing this buffer. This value is set by the procedure that detected the error and can be retrieved using SSGET. This value can be cleared using the special SSPUT operation ZSPI-TKN-CLEARERR.

ZSPI-TKN-LASTERRCODE

```
token-code        ZSPI-TKN-LASTERRCODE
  value           ZSPI-TNM-LASTERRCODE
  token-type      ZSPI-TYP-TOKENCODE
  ssid            ZSPI-VAL-NULL-EXT-SSID
  heading         "Last_error_tkncode".
```

ZSPI-TKN-LASTERRCODE records the token code of the token involved in the last nonzero status code returned by an SPI procedure while processing this buffer. This value is set by the procedure that detected the error and can be retrieved using SSGET. This value can be cleared using the special SSPUT operation ZSPI-TKN-CLEARERR.

ZSPI-TKN-LASTPOSITION

```
token-code        ZSPI-TKN-LASTPOSITION
  value           ZSPI-TNM-LASTPOSITION
  token-type      ZSPI-TYP-POSITION
  ssid            ZSPI-VAL-NULL-EXT-SSID
  heading         "Last_token_position".
```

ZSPI-TKN-LASTPOSITION contains the position of the last token that was added to the buffer by SSPUT. You can retrieve this position using SSGET.

ZSPI-TKN-LEN

```
token-code        ZSPI-TKN-LEN
  value           ZSPI-TNM-LEN
  token-type      ZSPI-TYP-TOKENCODE
  ssid            ZSPI-VAL-NULL-EXT-SSID
  heading         "Token_length".
```

ZSPI-TKN-LEN directs SSGET to get the length, in bytes, of a specific token value. Its token type is ZSPI-TYP-TOKENCODE.

ZSPI-TKN-LIST

```
token-code        ZSPI-TKN-LIST
  value           ZSPI-TNM-LIST
  token-type      ZSPI-TYP-LIST
  ssid            ZSPI-VAL-NULL-EXT-SSID
  heading         "List".
```

ZSPI-TKN-LIST begins a generic list. This token is provided for the convenience of subsystems you write. NonStop Kernel subsystems provided by HP do not use it. Its token type is ZSPI-TYP-LIST. Calling SSGET with ZSPI-TKN-LIST selects the next list that begins with that token so that tokens within it can be read; calling SSPUT with this token places it in the buffer, thus starting a new list.

ZSPI-TKN-MANAGER

```
token-code        ZSPI-TKN-MANAGER
  value           ZSPI-TNM-MANAGER
  token-type      ZSPI-TYP-FNAME32
  ssid            ZSPI-VAL-NULL-EXT-SSID
  heading         "Manager".
```

ZSPI-TKN-MANAGER is the process name of a particular subsystem process. This token is used in commands to the Subsystem Control Point (SCP) to identify the target subsystem, in error lists returned by Pathway, and in some event messages to qualify the subject of the event. In the future, it might have other uses. Its token type is ZSPI-TYP-FNAME32.

ZSPI-TKN-MAX-FIELD-VERSION

```
token-code        ZSPI-TKN-MAX-FIELD-VERSION
  value           ZSPI-TNM-MAX-FIELD-VERSION
  token-type      ZSPI-TYP-VERSION
  ssid            ZSPI-VAL-NULL-EXT-SSID
  heading         "Max-field-version".
```

ZSPI-TKN-MAX-FIELD-VERSION contains the highest version of any non-null field in any extensible structured token added to the buffer. The maximum field version is kept for version compatibility checking by subsystems. This value is checked by SSPUT, and reset if necessary each time it puts an extensible structure into the buffer. The value can be retrieved using SSGET.

ZSPI-TKN-MAXRESP

```
token-code        ZSPI-TKN-MAXRESP
  value           ZSPI-TNM-MAXRESP
  token-type      ZSPI-TYP-INT
  ssid            ZSPI-VAL-NULL-EXT-SSID
  heading         "Max_responses".
```

ZSPI-TKN-MAXRESP contains a value indicating the number of response records the requester accepts in a response message. This value can be set using SSINIT or SSPUT, and can be retrieved using SSGET. For more information, see Multirecord Responses on page 2-30.

ZSPI-TKN-MORE-DATA

```
token-code        ZSPI-TKN-MORE-DATA
  value           ZSPI-TNM-MORE-DATA
  token-type      ZSPI-TYP-BOOLEAN
  ssid            ZSPI-VAL-NULL-EXT-SSID
  heading         "More_data_available".
```

A server uses this token to tell a requester whether or not the current segment completes the response record. If this token has the value TRUE, the record is incomplete and more segment lists are available. If this token has the value FALSE, or if the token is not in the segment, the segment completes the response record.

ZSPI-TKN-NEXTCODE

```
token-code        ZSPI-TKN-NEXTCODE
  value           ZSPI-TNM-NEXTCODE
  token-type      ZSPI-TYP-TOKENCODE
  ssid            ZSPI-VAL-NULL-EXT-SSID
  heading         "Nextcode".
```

ZSPI-TKN-NEXTCODE directs SSGET to get the next token code in the buffer that is different from the current token code. Its token type is ZSPI-TYP-TOKENCODE.

ZSPI-TKN-NEXTTOKEN

```
token-code        ZSPI-TKN-NEXTTOKEN
  value           ZSPI-TNM-NEXTTOKEN
  token-type      ZSPI-TYP-TOKENCODE
  ssid            ZSPI-VAL-NULL-EXT-SSID
  heading         "Nexttoken".
```

ZSPI-TKN-NEXTTOKEN directs SSGET to get the very next token code in the buffer, whether or not it is different from the current token code. Its token type is ZSPI-TYP-TOKENCODE.

ZSPI-TKN-OBJECT-TYPE

```
token-code        ZSPI-TKN-OBJECT-TYPE
  value           ZSPI-TNM-OBJECT-TYPE
  token-type      ZSPI-TYP-ENUM
  ssid            ZSPI-VAL-NULL-EXT-SSID
  heading         "Object_type".
```

ZSPI-TKN-OBJECT-TYPE contains the object type to which the command is to be applied. This value is set by SSINIT and can be retrieved using SSGET. This value cannot be modified using SSPUT.

ZSPI-TKN-OFFSET

```
token-code        ZSPI-TKN-OFFSET
  value           ZSPI-TNM-OFFSET
  token-type      ZSPI-TYP-TOKENCODE
  ssid            ZSPI-VAL-NULL-EXT-SSID
  heading         "Token_offset".
```

ZSPI-TKN-OFFSET directs SSGET to get the byte offset of a specific token value from the start of the buffer. The token type of this token is ZSPI-TYP-TOKENCODE.

ZSPI-TKN-PARM-ERR

```
token-code        ZSPI-TKN-PARM-ERR
  value           ZSPI-TNM-PARM-ERR
  token-type      ZSPI-TYP-PARM-ERR
  ssid            ZSPI-VAL-NULL-EXT-SSID
  heading         "Parameter_error".
```

ZSPI-TKN-PARM-ERR is returned by a subsystem in an error list when a parameter error occurs in a command to the subsystem or in an SPI procedure call made by the subsystem. Its value is a structure giving information to identify the parameter in error, as shown in the description of ZSPI-TYP-PARM-ERR. Its token type is ZSPI-TYP-PARM-ERR.

ZSPI-TKN-POSITION

```
token-code        ZSPI-TKN-POSITION
  value           ZSPI-TNM-POSITION
  token-type      ZSPI-TYP-POSITION
  ssid            ZSPI-VAL-NULL-EXT-SSID
  heading         "Position".
```

ZSPI-TKN-POSITION contains the current-token pointer. You can retrieve this value using SSGET and restore that position using SSPUT. When passed to SSPUT with a position value obtained from a previous SSGET with ZSPI-TKN-POSITION or ZSPI-

TKN-LASTPOSITION, it sets the current position according to the passed position value. Position values are based on the byte offsets of tokens in the buffer; position values are rendered invalid if an SSPUT with ZSPI-TKN-DELETE or ZSPI-TKN-DATA-FLUSH is used to modify the buffer after the position value is obtained.

ZSPI-TKN-PROC-ERR

```
token-code        ZSPI-TKN-PROC-ERR
  value           ZSPI-TNM-PROC-ERR
  token-type      ZSPI-TYP-ENUM
  ssid            ZSPI-VAL-NULL-EXT-SSID
  heading         "Procedure_in_error".
```

ZSPI-TKN-PROC-ERR is returned by a subsystem in an error list when an unexpected error occurs on a call to one of the SPI procedures. Its value indicates which SPI procedure failed. Its token type is ZSPI-TYP-ENUM. Allowed values for error lists defined by SPI (for SPI errors encountered by a subsystem and returned to the application) are ZSPI-VAL-SSGET, ZSPI-VAL-SSGETTKN, ZSPI-VAL-SSINIT, ZSPI-VAL-SSMOVE, ZSPI-VAL-SSMOVETKN, ZSPI-VAL-SSNULL, ZSPI-VAL-SSPUT, and ZSPI-VAL-SSPUTTKN.

ZSPI-TKN-PROG-FNAME

```
token-code        ZSPI-TKN-PROG-FNAME
  value           ZSPI-TNM-PROG-FNAME
  token-type      ZSPI-TYP-STRING
  ssid            ZSPI-VAL-NULL-EXT-SSID
  heading         "Program_filename".
```

ZSPI-TKN-PROG-FNAME conveys a program file name.

ZSPI-TKN-RESET-BUFFER

```
token-code        ZSPI-TKN-RESET-BUFFER
  value           ZSPI-TNM-RESET-BUFFER
  token-type      ZSPI-TYP-UINT.
```

ZSPI-TKN-RESET-BUFFER, passed to SSPUT, performs resetting operations on an SPI buffer received from another process to prepare it for scanning by the receiving process. This operation clears the last-error information, resets the maximum buffer length, and resets the current position to the beginning of the buffer. The token type of this token is ZSPI-TYP-UINT.

ZSPI-TKN-RESPONSE-TYPE

```
token-code        ZSPI-TKN-RESPONSE-TYPE
  value           ZSPI-TNM-RESPONSE-TYPE
  token-type      ZSPI-TYP-RESPONSE-TYPE
  ssid            ZSPI-VAL-NULL-EXT-SSID
  heading         "Response_type".
```

ZSPI-TKN-RESPONSE-TYPE indicates what kinds of response records the subsystem should return. Its token type is ZSPI-TYP-ENUM.

Its allowed values and their meanings are:

ZSPI-VAL-ERR-AND-WARN	Return only responses about objects for which an error or warning occurred (responses that include at least one error list, regardless of the value of the return token).
ZSPI-VAL-ERR-WARN-AND-NORM	Return a response about every object.

For more information, see Suppressing Response Records on page 2-43.

ZSPI-TKN-RETCODE

```
token-code        ZSPI-TKN-RETCODE
  value           ZSPI-TNM-RETCODE
  token-type      ZSPI-TYP-ENUM
  ssid            ZSPI-VAL-NULL-EXT-SSID
  heading         "Retcode".
```

ZSPI-TKN-RETCODE is the return-code token, returned in a response to indicate whether a command was successful and why it failed if it did. This token is provided in every response record returned by a NonStop Kernel subsystem. Its token type is ZSPI-TYP-ENUM. Its value is one of the set of error numbers defined for the subsystem to which the command was sent.

ZSPI-TKN-SEGLIST

```
token-code        ZSPI-TKN-SEGLIST
  value           ZSPI-TNM-SEGLIST
  token-type      ZSPI-TYP-LIST
  ssid            ZSPI-VAL-NULL-EXT-SSID
  heading         "Segment_list".
```

A server uses this token to mark the beginning of a segment list in an SPI response message. The end of the list is marked with ZSPI-TKN-ENDLIST.

ZSPI-TKN-SEGMENTATION

```
token-code        ZSPI-TKN-SEGMENTATION
  value           ZSPI-TNM-SEGMENTATION
  token-type      ZSPI-TYP-BOOLEAN
  ssid            ZSPI-VAL-NULL-EXT-SSID
  heading         "Segmented_response_support".
```

A server uses this token to announce that it can generate segmented responses—by returning the token with a value of TRUE in its GETVERSION response. A value of FALSE (or the absence of the token from the GETVERSION response) indicates that the server does not support segmented responses.

ZSPI-TKN-SERVER-BANNER

```
token-code        ZSPI-TKN-SERVER-BANNER
  value           ZSPI-TNM-SERVER-BANNER
  token-type      ZSPI-TYP-CHAR50
  ssid            ZSPI-VAL-NULL-EXT-SSID
  heading         "Server_banner".
```

ZSPI-TKN-SERVER-BANNER is an ASCII character string, returned in the GETVERSION response, that identifies the subsystem product name, product number, release date, and sometimes additional information, in displayable form. The value of ZSPI-TKN-SERVER-BANNER is intended only for display, not for programs to examine. Its format might change in new releases, so programs should not extract information from this token value.

ZSPI-TKN-SERVER-VERSION

```
token-code        ZSPI-TKN-SERVER-VERSION
  value           ZSPI-TNM-SERVER-VERSION
  token-type      ZSPI-TYP-VERSION
  ssid            ZSPI-VAL-NULL-EXT-SSID
  heading         "Server_version".
```

ZSPI-TKN-SERVER-VERSION contains the version of the server, as set by the subsystem when it prepares its response to a command. This value is set by the server using SSPUT and can be retrieved using SSGET. For NonStop Kernel subsystems, the representation of this version number is the same as that of the Z-VERSION field of the subsystem ID.

ZSPI-TKN-SSID

```
token-code        ZSPI-TKN-SSID
  value           ZSPI-TNM-SSID
  token-type      ZSPI-TYP-SSID
  ssid            ZSPI-VAL-NULL-EXT-SSID
  heading         "SSID".
```

ZSPI-TKN-SSID contains the subsystem ID of the server that is to process the command. This value is set by SSINIT and can be retrieved using SSGET. This value cannot be modified using SSPUT.

ZSPI-TKN-SSID-ERR

```
token-code        ZSPI-TKN-SSID-ERR
  value           ZSPI-TNM-SSID-ERR
  token-type      ZSPI-TYP-SSID
  ssid            ZSPI-VAL-NULL-EXT-SSID
  heading         "SSID_error".
```

ZSPI-TKN-SSID-ERR is returned by a subsystem in an error list when an unexpected error occurs on a call to one of the SPI procedures, or to indicate the qualifying subsystem ID of an unrecognized token if that subsystem ID is different from the default. Its token type is ZSPI-TYP-SSID. Its value is the subsystem ID of the subsystem or system component returning the error.

ZSPI-TKN-USEDLEN

```
token-code        ZSPI-TKN-USEDLEN
  value           ZSPI-TNM-USEDLEN
  token-type      ZSPI-TYP-UINT
  ssid            ZSPI-VAL-NULL-EXT-SSID
  heading         "Used_byte_length".
```

ZSPI-TKN-USEDLEN contains the length of the used portion of the buffer. This value is updated by SSPUT and SSMOVE each time they add information to the buffer, and can be retrieved using SSGET.

Token Length

The second part of the token code is the token length. For fixed-length token values up to 254 bytes long, this field gives the length of the token value in bytes. When the value of this field is 255, the token value is either variable length or a fixed length greater than 254 bytes. When the fixed length is greater than 254, the actual length is supplied in a succeeding word. The special operator ZSPI-TKN-LEN returns the token length. For a variable-length token, the actual length is in the first 16 bits of the token value. To protect against overwriting data, check the length of a variable-length token before calling SSGET to retrieve it.

ZSPI-TLN-VARIABLE

```
constant        ZSPI-TLN-VARIABLE                   value is 255.
```

ZSPI-TLN-VARIABLE is the token-length value 255 that indicates a variable-length token or a token longer than 254 bytes.

Command Numbers

Basic SPI defines a single command number, ZSPI-CMD-GETVERSION, for the GETVERSION command. Any additional commands must be defined by a subsystem or drawn from some other common source, such as the SPI common extensions ZCOM definitions described in the *SPI Common Extensions Manual*.

ZSPI-CMD-GETVERSION

```
constant        ZSPI-CMD-GETVERSION                 value is 0.
```

ZSPI-CMD-GETVERSION is the command number for the GETVERSION command. Its value is zero. Subsystems use this value to define the command number for their GETVERSION commands.

Object-Type Numbers

Basic SPI does not define any object-type numbers. However, SPI does require that subsystems use the number 0 to represent the null object type, which is used in commands when it is either unnecessary or inappropriate to specify a particular object type. (SPI defines ZSPI-VAL-NULL-OBJECT-TYPE for this purpose. See Miscellaneous Values on page 4-47.) All other object types must be defined by a subsystem or drawn from some other common source, like the SPI common extensions ZCOM definitions described in the *SPI Common Extensions Manual.*

Error Numbers

Error numbers are 16-bit signed integers used to identify errors that SPI servers can return in response messages. Error number definition syntax is listed below. The errors are described in Appendix A, Errors.

Error Number Definition Syntax

This syntax box lists the error numbers by numeric value. (Numeric values are provided for debugging purposes only—always use the symbolic name in programs.)

```
constant   ZSPI-ERR-OK                  VALUE IS 0.
constant   ZSPI-ERR-INVBUF              VALUE IS -1.
constant   ZSPI-ERR-ILLPARM             VALUE IS -2.
constant   ZSPI-ERR-MISPARM             VALUE IS -3.
constant   ZSPI-ERR-BADADDR             VALUE IS -4.
constant   ZSPI-ERR-NOSPACE             VALUE IS -5.
constant   ZSPI-ERR-XSUMERR             VALUE IS -6.
constant   ZSPI-ERR-INTERR              VALUE IS -7.
constant   ZSPI-ERR-MISTKN              VALUE IS -8.
constant   ZSPI-ERR-ILLTKN              VALUE IS -9.
constant   ZSPI-ERR-BADSSID             VALUE IS -10.
constant   ZSPI-ERR-NOTIMP              VALUE IS -11.
constant   ZSPI-ERR-NOSTACK             VALUE IS -12.
constant   ZSPI-ERR-ZFIL-ERR            VALUE IS -13.
constant   ZSPI-ERR-ZGRD-ERR            VALUE IS -14.
constant   ZSPI-ERR-INV-FILE            VALUE IS -15.
constant   ZSPI-ERR-CONTINUE            VALUE IS -16.
constant   ZSPI-ERR-NEW-LINE            VALUE IS -17.
constant   ZSPI-ERR-NO-MORE             VALUE IS -18.
constant   ZSPI-ERR-MISS-NAME           VALUE IS -19.
constant   ZSPI-ERR-DUP-NAME            VALUE IS -20.
constant   ZSPI-ERR-MISS-ENUM           VALUE IS -21.
constant   ZSPI-ERR-MISS-STRUCT         VALUE IS -22.
constant   ZSPI-ERR-MISS-OFFSET         VALUE IS -23.
constant   ZSPI-ERR-TOO-LONG            VALUE IS -24.
constant   ZSPI-ERR-MISS-FIELD          VALUE IS -25.
constant   ZSPI-ERR-NO-SCANID           VALUE IS -26.
constant   ZSPI-ERR-NO-FORMATID         VALUE IS -27.
constant   ZSPI-ERR-OCCURS-DEPTH        VALUE IS -28.
constant   ZSPI-ERR-MISS-LABEL          VALUE IS -29.
constant   ZSPI-ERR-BUF-TOO-LARGE       VALUE IS -30.
constant   ZSPI-ERR-OBJFORM             VALUE IS -31.
constant   ZSPI-ERR-OBJCLASS            VALUE IS -32.
constant   ZSPI-ERR-BADNAME             VALUE IS -33.
constant   ZSPI-ERR-TEMPLATE            VALUE IS -34.
constant   ZSPI-ERR-ILL-CHAR            VALUE IS -35.
constant   ZSPI-ERR-TKNDEFID            VALUE IS -36.
constant   ZSPI-ERR-INCOMP-RESP         VALUE IS -37.
```

Subsystem Numbers

The SPI definitions include a subsystem number for every NonStop Kernel subsystem that has a programmatic command interface based on SPI, reports events through EMS, or defines error information in SPI error lists. Your program should use the symbolic names for these subsystem numbers. The names are all of the form ZSPI-SSN-*subsys*, where *subsys* is the appropriate four-character subsystem abbreviation. See Appendix D, NonStop Kernel Subsystem Numbers and Abbreviations.

Miscellaneous Values

The SPI standard definition files include these standard value names for tokens and SPI procedure-call parameters. Use these names in your programs whenever appropriate.

ZSPI-VAL-ALLOW-SEGMENTS

```
constant        ZSPI-VAL-ALLOW-SEGMENTS              value is -1.
```

A requester assigns this value to ZSPI-TKN-ALLOW and includes that token in a command message to ask a server for a segmented response.

ZSPI-VAL-CMDHDR

```
constant        ZSPI-VAL-CMDHDR                      value is 0.
```

ZSPI-VAL-CMDHDR is the value of the header-type field of the SPI message header for a command or response. Use this value for the SSINIT parameter *header-type* when initializing a command or response buffer.

ZSPI-VAL-EVTHDR

```
constant        ZSPI-VAL-EVTHDR                      value is 1.
```

ZSPI-VAL-EVTHDR is the value of the header-type field of the SPI message header for an event message.

ZSPI-VAL-FALSE

```
constant        ZSPI-VAL-FALSE                       value is 0.
```

ZSPI-VAL-FALSE is the Boolean value FALSE, represented internally by the integer 0. This named value is not supported by COBOL. To interpret Boolean values in COBOL, see Section 7, SPI Programming in COBOL.

ZSPI-VAL-HDRSIZE

```
constant       ZSPI-VAL-HDRSIZE                    value is 256.
```

ZSPI-VAL-HDRSIZE is the recommended number of bytes to reserve for the standard command header. It is provided primarily for use by subsystems in determining their recommended buffer sizes.

ZSPI-VAL-MSGCODE

```
constant       ZSPI-VAL-MSGCODE                    value is -28.
```

ZSPI-VAL-MSGCODE is the signed integer value found in the first word of every SPI message. (SPI messages include SPI commands, responses to SPI commands, and event messages.) This value identifies the message as an SPI message.

ZSPI-VAL-NULL-COMMAND

```
constant       ZSPI-VAL-NULL-COMMAND               value is -1.
```

ZSPI-VAL-NULL-COMMAND is the null command number. Some NonStop Kernel subsystems return this value in an error list when an SPI procedure encounters an error that causes the procedure to fail.

ZSPI-VAL-NULL-EXT-SSID

```
constant       ZSPI-VAL-NULL-EXT-SSID       value is "0.0.0".
```

ZSPI-VAL-NULL-EXT-SSID is the null subsystem ID.

ZSPI-VAL-NULL-OBJECT-TYPE

```
constant       ZSPI-VAL-NULL-OBJECT-TYPE           value is 0.
```

ZSPI-VAL-NULL-OBJECT-TYPE is the null object type. Use this value for the SSINIT *object-type* parameter when initializing a buffer for a command for which no object type is required. Some NonStop Kernel subsystems return this value in an error list when an SPI procedure encounters an error that causes the procedure to fail.

ZSPI-VAL-NULL-TOKENCODE

```
constant  ZSPI-VAL-NULL-TOKENCODE  value is 0 type binary 32.
```

ZSPI-VAL-NULL-TOKENCODE is the null token-code value. Use this value for the *token-code* parameter when requesting the SSGET special operations ZSPI-TKN-

ADDR, ZSPI-TKN-LEN, ZSPI-TKN-OFFSET, and ZSPI-TKN-COUNT. Using the null token code has the same effect as omitting *token-code*: the operation returns the address, length, or offset of the currently selected token, or counts occurrences of the current token beginning with the current occurrence.

ZSPI-VAL-NULL-VERSION

```
constant       ZSPI-VAL-NULL-VERSION                value is 0.
```

ZSPI-VAL-NULL-VERSION is the null version value. The subsystem ID returned by SSGET with the special token code ZSPI-TKN-DEFAULT-SSID has this value in the version field. When a subsystem reports a pass-through error from another subsystem that does not itself support a programmatic command interface based on SPI, this value appears for the version in the subsystem ID part of the error token.

ZSPI-VAL-SSID

```
def ZSPI-VAL-SSID   tacl SSID.
  02 Z-FILLER  type character 8     value is ZSPI-VAL-TANDEM.
  02 Z-OWNER   type ZSPI-DDL-CHAR8  redefines Z-FILLER.
  02 Z-NUMBER  type ZSPI-DDL-INT    value is ZSPI-SSN-ZSPI.
  02 Z-VERSION type ZSPI-DDL-UINT   value is ZSPI-VAL-
VERSION.
end
```

ZSPI-VAL-SSID is the subsystem ID value for SPI. Assign these values to the structure fields: ZSPI-VAL-TANDEM to Z-OWNER, ZSPI-SSN-ZSPI to Z-NUMBER, and ZSPI-VAL-VERSION to Z-VERSION.

ZSPI-VAL-TANDEM

```
constant       ZSPI-VAL-TANDEM           value is "TANDEM  ".
```

ZSPI-VAL-TANDEM is the string value “TANDEM~~bb~~” (with two trailing blanks). Your program must assign this value to the owner field of the subsystem ID before sending a command to any NonStop Kernel subsystem.

ZSPI-VAL-TRUE

```
constant       ZSPI-VAL-TRUE                        value is -1.
```

ZSPI-VAL-TRUE is the Boolean value TRUE, represented internally by the integer -1. This named value is not supported by COBOL. To interpret Boolean values in COBOL, see Section 7, SPI Programming in COBOL.

ZSPI-VAL-VERSION

```
constant       ZSPI-VAL-VERSION       value is version "D20".
```

ZSPI-VAL-VERSION is the one-word version value of this version of the SPI standard definitions. The left byte contains the letter part of the version as an ASCII uppercase alphabetic character, and the right byte contains the numeric part of the version as an unsigned integer value. For example, for the version “G06,” the left byte is the ASCII character *G*, and the right byte is “06”. The result is the unsigned integer 18182.

5

General SPI Programming Guidelines

This section offers three categories of programming guidelines:

General Guidelines for All SPI Programs

The guidelines in this section apply to all SPI programs—requesters and servers.

Retrieving Tokens by Name

The most common way to retrieve tokens from the buffer is by passing their names to SSGET. An application can ask for tokens in any convenient order.

A call to SSGET optionally specifies a value for an index parameter. This index lets SSGET select a particular occurrence of a particular token in the buffer; as mentioned in Section 2, SPI Concepts and Protocol, there can be several occurrences (that is, several instances of tokens with the same token code). Two cases are possible:

- If the index value is *n* > 0 than zero, SSGET starts searching at the beginning of the buffer and returns the *n*th value of that token.
- If the index value is zero (the default) or not supplied, SSGET retrieves the next occurrence of the specified token.

An index that is zero or not supplied tells SSGET to start searching with the token indicated by the next-token pointer. If the specified token is found, the current-token position is set to the token that was returned, and the next-token pointer is updated to the token following it. For example:

```
CALL SSGET (buffer, TKN-A, value)
```

Before the call, the positioning is:

```
current     next
   |          |
   v          v
<TKN-A>   <TKN-B>   <TKN-C>   <TKN-A>   <TKN-D>
```

After the call, the positioning is:

```
                              current     next
                                 |         |
                                 v         v
<TKN-A>   <TKN-B>   <TKN-C>   <TKN-A>   <TKN-D>
```

The SSGET procedure also includes a count parameter. This feature lets a program extract up to the specified number of occurrences of a token in one call, whether they are contiguous or not, and receive their values as an array.

Scanning a Buffer Sequentially

A program can scan the buffer sequentially and retrieve each token in order using one of the special SSGET operations ZSPI-TKN-NEXTCODE and ZSPI-TKN-NEXTTOKEN. This method of token retrieval is useful mainly to servers because they must examine every token in the buffer and reject messages that contain unrecognized or invalid tokens.

A call to SSGET with ZSPI-TKN-NEXTTOKEN gets the next token code in the buffer, regardless of whether that token has the same token code as the current token, and regardless of whether that token is within a list.

For example:

```
CALL SSGETTKN (buffer, ZSPI-TKN-NEXTTOKEN, value)
```

Before the call, the positioning is:

```
current     next
   |         |
   v         v
<TKN-A>   <TKN-B>   <TKN-C>   <TKN-A>   <TKN-D>
```

After the call, the positioning is:

```
    current + next
              |
              v
<TKN-A>   <TKN-B>   <TKN-C>   <TKN-A>   <TKN-D>
```

A call to SSGET with ZSPI-TKN-NEXTCODE gets the next token different from the token code at the current position. Unlike ZSPI-TKN-NEXTTOKEN, this operation does not see tokens within lists until the beginning list token is selected, and it skips over any tokens that have the same token code as the current token.

Repeated calls using the ZSPI-TKN-NEXTCODE operation return non-contiguous occurrences of the same token code in the same order as they were supplied by SSPUT. The result gives the number of contiguous occurrences, *not* the total number of occurrences in the buffer. For example, suppose the buffer contains:

```
<TKN A> <TKN A> <TKN A> <TKN B> <TKN A> <TKN A> <TKN C>
```

Successive calls with ZSPI-TKN-NEXTCODE:

```
             token code    occurrences

1st call         A             3
2nd call         B             1
3rd call         A             2
4th call         C             1
5th call         ZSPI-ERR-MISTKN
```

The NEXTCODE operation sets both the current position and the next position to the token code it returns. This means that a subsequent call to SSGET with the returned token code can be used to get the value associated with that token code.

The ZSPI-TKN-NEXTCODE operation does not retrieve token codes that are inside a list unless the list has been explicitly selected by a call to SSGET. After you are positioned inside a list, however, you can use ZSPI-TKN-NEXTCODE to exit the list by requesting the next token code after the ZSPI-TKN-ENDLIST token.

For both ZSPI-TKN-NEXTCODE and ZSPI-TKN-NEXTTOKEN, unlike a regular SSGET call to get a particular token, the subsystem ID is an output parameter only; you merely provide a variable in which SSGET will return the subsystem ID that qualifies the token code. If you do not supply the subsystem ID parameter, and the subsystem ID associated with the next token code is not the same as the default subsystem ID, a “missing parameter” error is returned. Therefore, you should always supply the subsystem ID parameter when calling SSGET with ZSPI-TKN-NEXTCODE and ZSPI-TKN-NEXTTOKEN unless, for some special reason, you are certain that all tokens the program could encounter are qualified by the default subsystem ID. The subsystem ID returned always has a version field of zero (null).

The identifying-code part of each token is always stored in the SPI buffer as a token code, even for extensible structured tokens. The token code for an extensible structured token consists of a token type that identifies the token value as a variable-length structure (ZSPI-TYP-STRUCT) and the token number from the corresponding token map. When a program scans the buffer using SSGET with ZSPI-TKN-NEXTCODE or ZSPI-TKN-NEXTTOKEN, it retrieves this token code, not the token map. After the call with ZSPI-TKN-NEXTCODE or ZSPI-TKN-NEXTTOKEN, your application can extract the value of the structure in either of two ways:

- If you are writing a requester using a structure declaration supplied by the subsystem (which can differ from the version of that structure that you are using), use the token number from the retrieved token code in a CASE statement or similar construct. The CASE statement should select a section of code that calls SSGET specifying the token map that corresponds to the token number.
- If you are coding an interpreter, debugger, text formatter, or other program that does not use subsystem-supplied structure declarations and thus does not have token maps available, obtain the exact token value from the buffer by passing the retrieved token code to SSGET. The token value returned contains the length word from the token map, followed by all the structure fields that were actually submitted with SSPUT.

Positioning the Buffer Pointers

Whenever SSGET retrieves a token from the data portion of the buffer, it starts at a given position and scans the buffer until it finds the appropriate token. The SSGET procedure maintains pointers in the SPI message header to keep track of positioning in the buffer. As your program scans the buffer with SSGET, these pointers are automatically updated. Special token codes can be passed to SSGET and SSPUT to get or change the values of the pointers. SSGET uses four pointers: the current-token pointer, the next-token pointer, the last-put-token pointer, and the current list pointer.

The current-token pointer contains the location of the last token selected with SSGET. SSINIT sets the current -token pointer to the beginning of the buffer (immediately following the header). Thereafter, the current-token pointer is affected implicitly by calls to SSGET and SSMOVE and explicitly by calls to SSPUT with the special token codes ZSPI-TKN-POSITION, ZSPI-TKN-INITIAL-POSITION, and ZSPI-TKN-RESET-BUFFER. Other calls to SSPUT do not affect the current-token pointer.

The next-token pointer normally contains the position of the token immediately following the current position. It points to the token that will be extracted next when SSGET scans the buffer using the special operations ZSPI-TKN-NEXTCODE or ZSPI-TKN-NEXTTOKEN.

The current-list pointer always points to the currently selected list, if any; if no list is currently selected, this pointer is set to null. SPI uses the current and next pointers to determine the starting point of a scan when SSGET is called with an index value of zero. Positioning with lists is described later in this section.

Selecting a header token or an attribute of the current token does not change the current-token pointer or the next-token pointer.

The special SSGET operation ZSPI-TKN-POSITION is available to retrieve the value of the current position. A corresponding SSPUT special operation, also called ZSPI-TKN-POSITION, restores a position previously saved using SSGET.

A position (ZSPI-TKN-POSITION) returned by SSGET remains valid until the buffer involved is modified with the SSPUT operation ZSPI-TKN-DELETE or ZSPI-TKN-DATA-FLUSH (described later in this section) or until a call to SSMOVE replaces a token in the buffer. If the contents of the buffer are moved to another buffer with SSMOVE, the position, address, or offset is still valid when used with the original buffer, but is not valid for use with the target buffer of the SSMOVE.

When an SPI buffer is reset with SSPUT, there is no current list, and both the current and next pointers indicate a point just prior to the first token that is not a header token:

```
current + next
  |
  v
  *  <token>  <token>  <list>  <list>  <token>
```

A similar position exists at the beginning of every list.

Working With Lists

SPI defines four types of lists, as described in Lists on page 2-19. Every list starts with a list token corresponding to the type of list (ZSPI-TKN-LIST, ZSPI-TKN-DATALIST, ZSPI-TKN-SEGLIST, or ZSPI-TKN-ERRLIST) and ends with ZSPI-TKN-ENDLIST. You can think of the list token and the end-list token as analogous to left and right parentheses. With the exception of the ZSPI-TKN-NEXTTOKEN special operation, SSGET only accesses tokens in a list if SSGET is first called to retrieve the list token that marks the beginning of the list. SSGET retrieves tokens from lists as follows:

- To extract tokens from a list by name, a program must first select the list by calling SSGET to get the list token (ZSPI-TKN-DATALIST, ZSPI-TKN-ERRLIST, ZSPI-TKN-SEGLIST, or ZSPI-TKN-LIST) that begins the list. This call sets the current-token pointer and the next-token pointer to the start of the selected list and lets subsequent calls to SSGET get tokens within the list.
- After a list has been selected, SSGET cannot find tokens that are outside the list (except with the special operation ZSPI-TKN-NEXTTOKEN, described later in this section). Your program can exit the list by calling SSGET to get the end-list token, ZSPI-TKN-ENDLIST. This call exits the list and sets the current position to the list token that begins the list.
- Within a list, the default subsystem ID is the subsystem ID that qualifies the list token.
- The SSMOVE procedure can copy an entire list in a single call. To copy a list, your program simply specifies the token code that marks the beginning of the list (the list token) in the SSMOVE call.

Only lists supported by NonStop Kernel subsystems (data lists, segment lists, and error lists) are used in responses. However, subsystems you write can return lists in responses and might define a special-purpose list (using ZSPI-TKN-LIST) used in requests.

This example shows how a program retrieves tokens from an SPI buffer that contains nested lists. The sample buffer is a response buffer containing two data lists and an error list; the error list is within the second data list. All tokens in the example are simple tokens.

Assume a buffer containing:

```
<ZSPI-TKN-DATALIST>
  <TKN-A>
  <TKN-B>
  <ZSPI-TKN-RETCODE>
<ZSPI-TKN-ENDLIST>
<ZSPI-TKN-DATALIST>
  <ZSPI-TKN-ERRLIST>
    <ZSPI-TKN-ERROR>
    <TKN-INFO>
  <ZSPI-TKN-ENDLIST>
  <TKN-A>
```

```
  <TKN-B>
  <ZSPI-TKN-RETCODE>
<ZSPI-TKN-ENDLIST>
<ZSPI-TKN-CONTEXT>
```

When SSGET is first called to scan this buffer (with any token code other than ZSPI-TKN-NEXTTOKEN), it sees only three tokens:

```
<ZSPI-TKN-DATALIST> <ZSPI-TKN-DATALIST> <ZSPI-TKN-CONTEXT>
```

The tokens inside the lists are not available until the list is selected. Calling SSGET with a list token selects that list. After the first ZSPI-TKN-DATALIST is selected, the buffer appears to contain:

```
<TKN-A> <TKN-B> <ZSPI-TKN-RETCODE>
```

Because the current context is a list, SSGET stops searching when it reaches the end-list token. Selecting the end-list token with SSGET has the effect of returning to the previous context (in this case, the topmost level) with the current-token pointer set to the list ended by the end-list token.

If the program selects the next ZSPI-TKN-DATALIST, the buffer then appears to contain:

```
<ZSPI-TKN-ERRLIST> <TKN-A> <TKN-B> <ZSPI-TKN-RETCODE>
```

The error list describes a warning that occurred on this response, so it is nested within the data list for this response. If a program calls SSGET to select the error list, the buffer then appears to contain:

```
<ZSPI-TKN-ERROR> <TKN-INFO>
```

At this point, the program can retrieve either ZSPI-TKN-ERROR or TKN-INFO, can return to the data-list context by selecting ZSPI-TKN-ENDLIST, or can return to the topmost context by calling ZSPI-TKN-INITIAL-POSITION with the token value ZSPI-VAL-INITIAL-BUFFER.

Lists can be nested to any depth, but they are limited by the size of the buffer. A segmented list is the only case in which a list is continued across more than one message, but even in this case no list really spans messages, but is instead broken into smaller lists.

Checking for Null Values

Your program might need to test a field of an extensible structured token (for instance, in a response from a subsystem) to determine whether it has a null value. To perform this test, declare a copy of the structure, initialize the copy with SSNULL, and then compare the field in question with the corresponding field in the copy.

Deleting Tokens From a Buffer

A program can delete a specific token or a list from the buffer by using SSPUT with the special operation ZSPI-TKN-DELETE. It can delete all tokens in the buffer located at

and after the current position, including any tokens within lists, by using SSPUT with the operation ZSPI-TKN-DATA-FLUSH.

These two operations are useful mostly to server programs in preparing response messages. For instance, a subsystem preparing a response buffer with multiple response records can ensure that there is space for the context token by putting a dummy context token in the buffer, then deleting the dummy token when the buffer is filled and substituting the actual context token if the response requires continuation. Such a subsystem typically places response records in the buffer one token at a time until it runs out of buffer space. When this occurs, the subsystem can call SSPUT with ZSPI-TKN-DATA-FLUSH to flush all data from the buffer beginning at the current-token position.

Resetting the Buffer

Before calling SSGET to retrieve any tokens from the data portion of an SPI buffer received from another process, it is recommended that your program call SSPUT to reset the buffer using the special operation ZSPI-TKN-RESET-BUFFER. This operation performs three actions:

- It resets the maximum buffer length to the value specified in the call.
- It clears the last-error information to null values (equivalent to the action of ZSPI-TKN-CLEARERR).
- It resets the current-token pointer to the beginning of the buffer (equivalent to the action of ZSPI-TKN-INITIAL-POSITION with ZSPI-VAL-INITIAL-BUFFER).

If the maximum buffer length specified in the call is less than the actual number of bytes used in the buffer as given in the header token ZSPI-TKN-USEDLEN, the procedure returns ZSPI-ERR-NOSPACE. SPI still resets the maximum buffer length in the SPI message header, causing subsequent SPI calls for that buffer to fail with error ZSPI-ERR-INVBUF. This action enables a program to check for situations where data is lost because its buffer is not large enough to hold the entire message.

Working With SSIDs

These two system procedures are available for use with the SPI procedures:

- TEXTTOSSID converts a character string that represents a subsystem ID to the internal format used by SPI.
- SSIDTOTEXT converts the internal SPI format of a subsystem ID to a character string representing the subsystem ID.

Examples of character strings that represent subsystem IDs are “TANDEM.PATHWAY.C20” and “TANDEM.52.0.” The TEXTTOSSID and SSIDTOTEXT procedures are described in the *Guardian Procedure Calls Reference Manual*.

Writing High-Level Procedures

The SPI procedures provide basic functionality for building and decoding SPI buffers. If your application performs some commands frequently, you might want to simplify your programming task by writing high-level procedures that themselves call on the SPI procedures.

High-level procedures you write can provide some of the parameters needed by the SPI procedures, do error checking and recovery, format SPI buffers for commonly needed commands in one procedure call, or even both format and send commonly needed commands in a single call.

In deciding whether to write your own high-level procedures, and in determining what functions they perform if you decide to write them, consider the tradeoff between ease of programming and flexibility. High-level procedures are most likely to be useful if your application performs the same, or very similar, complex SPI buffer operations repeatedly.

As part of EMS, HP provides two sets of high-level procedures for use in formatting and decoding event-message buffers. For more information about these procedures, see the *EMS Manual*.

Guidelines for SPI Requesters

This overview shows what your application must do to send commands to a NonStop Kernel subsystem and process the responses:

1. Start the appropriate management process for the subsystem, if this process is not already running.
2. Open the subsystem manager process and do whatever else is needed to establish communication with it, such as sending the startup message if you are programming in TAL.
3. Initialize and build the command buffer using the SPI procedures.
4. Send the command to the subsystem.
5. Receive the response from the subsystem.
6. Extract response information from the response buffer using the SPI procedures.
7. Take any appropriate action. If necessary, go back to step 3 to build another command buffer and repeat the cycle.
8. Close the management process.
9. Stop the management process, if necessary.

Starting the Management Process

Some subsystem server processes can be started through SYSGEN or by an operator before the application is run; some can be explicitly started by the application using the NEWPROCESS or PROCESS_CREATE_ procedure; and some can be started either way. How the server can be started and in what form it is run depend on the subsystem and, in some cases, on decisions made at your installation. Some subsystems have processes that should be run as permanent servers under specific process names, to be shared by more than one process. Running the management process as a permanent server saves each requester the overhead (significant in TAL) of starting the process each time it needs to use it. Using a permanent server can also be advantageous if you are using the Safeguard software to implement security restrictions on access to the subsystem.

For some subsystems, the requester can either start a copy of the server for itself or use a shared copy. Some servers that permit this choice let you tell the server whether it should stop when its last requester closes it. For NonStop Kernel subsystem servers that provide such control, you supply this information to the server by giving the appropriate value to the AUTOSTOP parameter when you start the server process.

You include this parameter in the *param-string* part of the RUN command to start the server, or in the corresponding parameter-string part of the startup message your application sends to the server process immediately after it has started the new process by calling the NEWPROCESS or PROCESS_CREATE_ procedure. (For information about the startup message, see the *Guardian Programmer's Guide*.) The AUTOSTOP parameter is an ASCII string of this form:

```
                [ HRS  ]
AUTOSTOP time   [ MINS ]
                [ SECS ]
```

where HRS, MINS, or SECS is the unit of time used, and *time* is an integer whose range is -1 to 9999 for MINS or SECS or -1 to 999 for HRS. If no time unit is specified, SECS is implied.

Specifying AUTOSTOP with a *time* greater than 0 directs the server to wait the interval specified by *time* after its last requester has closed it before stopping itself. The server accepts new opens during this interval, and if a requester opens it before the interval expires, the timing starts fresh the next time its last requester closes it.

Specifying AUTOSTOP with a *time* of -1 directs the server to wait indefinitely (not stop itself). You should generally specify this value for a permanent server.

Specifying AUTOSTOP with a *time* of 0, or not specifying AUTOSTOP at all for a subsystem server that implements it, directs the server to stop immediately when its last requester closes it.

For more information about starting the management process for a particular subsystem, see the appropriate management programming manual.

Opening the Management Process

Your application has complete responsibility for establishing contact with the subsystem, sending and receiving the messages, and terminating contact. How your application performs these tasks depends on the subsystem and the programming language you are using. Usually, communication with the subsystem involves the file system—either directly by calling the file-system procedures from TAL, or indirectly by using COBOL verbs, TACL built-ins, or C interface declarations. C programs also can use the alternate-model I/O routines. The discussions in this section assume the use of the file system.

Process-Name Qualifier for SPI

Your application must establish communication by opening the server with a process name of the form:

`$`*`server-process-name`*`.#ZSPI`

The qualifier #ZSPI tells the server that the requester will be sending and receiving messages in SPI format.

Certain rules for establishing communication depend on the particular management process. These rules include which process to open, how many opens are allowed per process and per opener, and considerations for remote use and security. For these rules, see the appropriate subsystem management programming manual. The programming language can also impose special considerations; for these considerations, see the language-specific section of this manual and to the specific programming-language manual.

Checking for File-System Errors

Your application should check for the usual file-system errors that might occur on attempts to open a server process. These paragraphs provide causes and corrective actions that apply specifically to programs using SPI.

For general information about other file-system errors, see the *Guardian Procedure Errors and Messages Manual*.

Error 11 or 14: File or Device Does Not Exist

The management process rejected an open attempt because it did not recognize the process-name qualifier used. Either the qualifier name given was not #ZSPI, or the management process or subsystem does not support a programmatic command interface based on SPI.

If you used a qualifier other than #ZSPI, use #ZSPI. If you used #ZSPI and still got this error, specify the correct management process. If the subsystem does not support SPI, use whatever other interface is available for this subsystem.

Error 16: File Number Has Not Been Opened

The management process rejected an open attempt because it had not yet completed its own initialization when it received the open request. Try the open again later.

Error 17: Error on Backup Open

The process rejected a backup open attempt because there was no matching primary open; because the parameters for the backup open did not match those of the primary open; or because the primary process was not running.

Open the process by name rather than by *cpu*,*pin*. If opening by name is not feasible, open the primary process first, use matching parameters for the primary and backup opens, and check that the primary process is running.

Error 28: Sync Depth or Nowait Depth Limit Exceeded

The process rejected an open attempt because it could not accommodate the sync depth or nowait depth requested by the opener. Check for program logic errors. If there are no logic errors, reduce the sync depth or nowait depth.

Error 48: Security Violation

The process rejected an open attempt because the requester did not have the proper security to communicate with the process. One reason for this error could be that the application attempted to open an HP data-communications subsystem process for SPI communication. In this case, open an SCP process instead, as described in the *SPI Common Extensions Manual*.

Preparing the Command Buffer

Your application must allocate a buffer for communicating with the subsystem. The application uses the SPI procedures to initialize this buffer, put information into it, and extract information from it when it is returned as a response, as described in Section 3, The SPI Procedures.

Each NonStop Kernel subsystem specifies a recommended buffer size that is large enough to satisfy command and response requirements for all the subsystem's commands. Some subsystems specify different recommended buffer sizes for different commands. You should declare your buffer to be at least as large as the buffer size recommended by the subsystem for the command you are sending.

Your application can allocate the buffer dynamically. At any time between calls to the SPI procedures, your application can make copies of the buffer or move it to a different data location in either the data area or an extended segment. An application should keep a copy of each command buffer it sends in case retries are needed; it also must keep a copy to resend if the subsystem can return its response information in more than one response message.

After you have allocated the buffer, use the SPI procedures to initialize it and add tokens containing the information for your command, as described in Section 3, The SPI Procedures.

Sending the Command

After it has established communication with the management process, your application can use any standard mechanism for sending and receiving messages. A TAL program generally uses the file-system procedure WRITEREAD, and can use AWAITIO (if desired) to perform timed or nowait I/O. A COBOL program normally uses the standard verb READ WITH PROMPT. A TACL program generally uses the #APPENDV and #EXTRACTV built-in functions. A C program uses the **tal** interface declaration or alternate-model I/O. The language-specific sections of this manual give more information about using these languages to send and receive messages encoded as SPI buffers.

The language features just mentioned are recommended for most applications. In some specialized applications, however, it might instead be desirable to use a disk-based buffering scheme or other means. This manual does not go into detail about such alternative ways to send and receive messages, which apply only to communication with subsystems not written by HP.

When sending an SPI message to another process, you need to send only the used portion of the buffer. Your application can obtain the length of the used portion using the ZSPI-TKN-USEDLEN special SSGET operation. For details, see Section 3, The SPI Procedures.

Receiving the Response

When your application receives a response message, it must first check for file errors and reset the buffer. These actions ensure that your command was accepted and that a valid response was returned. Otherwise, your application might attempt to extract invalid information.

The first step is to check for file-system errors. The second step is to reset the buffer; among other things, this operation ensures that your buffer was long enough to contain the response. The third step is to check for errors in the response buffer.

Checking for File-System Errors

Your application should first check for the usual file-system errors that might occur on any WRITEREAD call from a requester to a server, plus a few file-system errors that have special meanings for SPI. These paragraphs provide causes and corrective actions that apply specifically to programs using SPI.

For information about other file-system errors, see the *Guardian Procedure Errors and Messages Manual*.

Error 2: Request Not an SPI Buffer

The process rejected a message because the requester had it open for SPI commands, but the command was not a valid SPI buffer. For instance, the first word of the message did not contain -28, or the subsystem received an event message when it expected a command. Check that the command is a properly formatted SPI buffer.

Error 60: Process Does Not Have Server Open

The management process rejected an SPI command because the requester process did not have the management process open. The target process might have been restarted since the requester opened it. Close the management process and reopen it.

Resetting the Buffer

After checking for file-system errors, your application should reset the response buffer. To do this, use the special SSPUT operation ZSPI-TKN-RESET-BUFFER described in Section 3, The SPI Procedures. This operation resets the positioning pointers used to extract tokens from the buffer, clears the last SPI error reported in operating on the buffer, and changes the buffer-length field in the buffer to the length declared by your application. The buffer length declared by your application can be, and usually is, different from the buffer length declared by the server.

If your application's buffer length specified to the reset-buffer operation is less than the length of the response (the used length of the buffer), your own data following the buffer is protected, but the excess information at the end of the response is lost. This situation is sometimes called a "short read." In this case, SPI returns a "buffer full" error (SPI error -5), and subsequent calls to SPI procedures for that buffer returns an "invalid buffer" error (SPI error -1). If error -5 results from the reset-buffer operation, declare a buffer at least as large as the size recommended by the subsystem.

Resetting the buffer also protects against problems that can occur if your application's buffer is shorter than the server's buffer. If you do not reset the buffer, your own data following the buffer might be overwritten as you extract tokens from the buffer, even if the reply's used length is short enough and no response data is lost. (SSGET calls are not guaranteed not to change the buffer or increase its size.)

Checking the Response Buffer for Errors and Warnings

NonStop Kernel subsystems report most command errors and command warnings in the form of tokens in the response buffer, rather than as file-system errors. An error is a condition that causes the command to fail. A warning is a less serious condition that can be significant to the application, but does not cause the command to fail. Errors and warnings can be present in the response even if the file-system error value is zero. After checking for file-system errors and resetting the buffer, your application should check the return token and (optionally) check any error lists. For further information, see Errors and Warnings on page 2-47.

Your application might be interested only in whether the command succeeded or failed, and perhaps in a limited amount of associated information. In this case, simply check the value of the return token and ignore any error lists. Some applications, however, might want full information about multiple errors and warnings and pass-through errors. The structure of error lists lets these applications retrieve more complete error information in an organized way, focusing on one error at a time.

For each response record, your application should always:

1. Test the value of the return code token ZSPI-TKN-RETCODE.
2. Scan the response message for error lists.

If the return code is zero, the operation completed successfully, although warnings might have occurred. If the return code is nonzero, the operation failed, and the value of the return code identifies the error (or one of the errors, if there were more than one) that prevented completion of the command. When this happens, additional information about the error is contained in an error list that contains the same error number (in ZSPI-TKN-ERROR).

Taking Action Based on the Response

After performing the checks just described, your application can read the other tokens in the response and take whatever actions are appropriate. Such actions often include sending other commands.

SPI requesters are expected to be tolerant of extraneous tokens in responses. Future versions of NonStop Kernel subsystems might return newly defined tokens containing additional information, and subsystems you write can do the same.

Canceling Commands

Requesters can use the file-system CANCEL or CANCELREQ procedure to cancel requests that they have issued to a server. The file system does CANCEL operations automatically in some cases. For instance, any operations outstanding on a file when it is closed are canceled. In addition, if an AWAITIO operation is issued for a specific file with a nonzero time limit and the time limit expires without a completion, the oldest operation outstanding on that file is canceled.

If a message sent to a subsystem is canceled before the subsystem receives it, the subsystem does not receive the message. If the subsystem has already received the message, the subsystem might perform the operation (whether it does so depends on the subsystem), but the file system discards any response.

Closing the Management Process

When your application is finished communicating with the subsystem, it should close the management process in whatever way is appropriate to the method used to open the process.

Stopping the Management Process

Some subsystems have a special command to stop the management process explicitly. Others stop this process automatically. Still others determine how they will stop by reading the AUTOSTOP parameter sent as part of the startup message (see Starting the Management Process on page 5-9). For details, see the individual subsystem management programming manual.

Maintaining Compatibility

For compatibility with future versions of HP software, your requester should:

- Declare SPI buffers at least as large as the subsystems' recommended sizes.
- Tolerate unrecognized tokens in responses. (The simplest way to do that is to get tokens only by name and avoid using NEXTCODE and NEXTTOKEN.)
- Avoid beginning any of your own declared names with Z.
- Call the SSNULL procedure to initialize the values of all extensible structured tokens.
- After calling SSNULL for each extensible structured token, set only token fields that you use.
- If your requester must communicate with several different versions of a subsystem, including versions that are older than your requester, avoid sending any command, token, or extensible-structured-token field that a subsystem does not support. If necessary, use the GETVERSION command to check the server version and tailor your command to it.

  Responses to the GETVERSION command issued by a subsystem that has an SPR must contain one instance of ZSPI-TKN-IPM-ID for each change to the SPI interface for the subsystem. If more than one change affects the same token, the ZSPI-TKN-IPM-ID for only the most recent change is returned. A subsystem might also return a ZSPI-TKN-IPM-ID for a change that does not affect the SPI interface.

NonStop Kernel subsystems report errors if they receive commands, tokens, or extensible-structured-token fields that they do not recognize. These errors are described in the management programming manuals for the individual subsystems.

Summary of Requester Role

An SPI requester must:

- Use subsystem-supplied definition files.
- Declare a buffer at least as large as that recommended by the subsystem.
- Use SSNULL to initialize all extensible structured tokens.

- Initialize every request with the SSID provided by the subsystem. Do not reuse a response message without initializing it—some header values set by the server are not appropriate for the request.
- Avoid defining data items with names beginning with *Z*.
- Tolerate unrecognized tokens in a response.
- Ignore responses that contain the “empty response” return code.
- Use only the absence of a context token as an indicator that a response message completes a response.
- Be aware that the context token can vary in size—copy it directly from the response to the new request.
- Supply the same read count in a WRITEREAD call as was used for the SSINIT buffer length.
- Always call SSPUT with the special operation code ZSPI-TKN-RESET-BUFFER after receiving a response message.

Guidelines for SPI Servers

Review these guidelines if you are going to write an SPI interface to a subsystem.

Recommending a Buffer Size

SPI servers should declare a recommended buffer size. The value selected should be large enough to guarantee that a requester that allocates a buffer of that size can accommodate all server-supported commands and associated responses. The value should be large enough to satisfy command and response requirements for at least as long as you intend to support the current version of the server. Make this value available to requesters by including a CONSTANT declaration in the server DDL. All NonStop Kernel subsystems define the recommended buffer size with a name of the form *subsys*-VAL-BUFLEN.

Note. TACL has an absolute maximum I/O buffer size of 4096 bytes.

Calculating a Recommended Buffer Size

You can approximate the minimum required buffer size:

```
approx size =  hdr + (4 * tokens)
                   + (12 * ssid-qual-tokens)
                   + sum-value-size
                   + (2 * lists)
                   + cushion
```

where

hdr	is the size of the SPI message header, defined by SPI as ZSPI-VAL-HDRSIZE.
tokens	is the total number of tokens in the longest command or response message defined by the server. This value includes list tokens (ZSPI-TKN-LIST, -DATALIST, -SEGLIST, -ENDLIST). A token code is 4 bytes long.
ssid-qual-tokens	is the total number of tokens that are qualified by an SSID. The SSID adds 12 bytes to the length of a token code.
sum-value-size	is the sum of the sizes of all of the token values in the message, including the values of comment (ZSPI-TKN-COMMENT) and context (ZSPI-TKN-CONTEXT) tokens.
lists	is the number of lists in the message. Each pair of list and end-list tokens requires 2 additional bytes beyond the 8 bytes required for the 2 token codes.
cushion	is a generous safety margin to cover future enhancements.

If the resulting recommended buffer size is large because of long commands or responses that are not likely to be issued or received by some requesters, you can recommend additional buffer sizes. Each requester can then choose the value that is appropriate for its needs.

For each recommended buffer size you define, provide a corresponding declaration for a buffer of that size:

```
def subsys-DDL-MSG-BUFFER.
  02 Z-MSGCODE              type ZSPI-DDL-INT.
  02 Z-BUFLEN               type ZSPI-DDL-UINT.
  02 Z-OCCURS               type ZSPI-DDL-UINT.
  02 Z-FILLER               type ZSPI-DDL-BYTE
                              occurs 0 to buffer-size times
                              depending on Z-OCCURS.
  end
```

subsys

is the abbreviation for your server.

buffer-size

is the number of bytes needed to make the buffer the recommended length. You can use the constant name you define for the corresponding recommended length (for instance, *subsys*-VAL-BUFLEN).

Defining Simple Tokens

Tokens that are fixed structures should be used primarily for items that are almost as primitive as simple data types—such as the internal file name or SPI subsystem ID,

each of which SPI defines as a fixed structure. It is recommended that most structured tokens defined for a subsystem be extensible structured tokens.

SPI uses a token length of zero in some tokens—such as the tokens that begin and end lists—to indicate that these tokens have no token values. However, it is recommended that tokens you define always have a value. If there are only two choices, the value can be of a Boolean or enumerated type.

Each token you define should have a unique token number. To avoid conflicts with token numbers defined by software for the NonStop server, always use token numbers in the range 1 through 9998.

Your subsystem can define its own private token types, using names of the form *subsys*-TYP-*name* and values built from the SPI standard token data types and appropriate token lengths. For example, an ATM-management subsystem might define an ATM name as a simple character-string token of a given size, and an ATM location as a fixed structure containing three character strings identifying the area, city, and branch. The TOKEN-TYPE statements for these types might resemble:

```
TOKEN-TYPE ATMX-TYP-ATMNAME       VALUE IS ZSPI-TDT-CHAR
                                  DEF IS ATMX-DDL-ATMNAME.
TOKEN-TYPE ATMX-TYP-ATMLOC        VALUE IS ZSPI-TDT-STRUCT
                                  DEF IS ATMX-DDL-ATMLOC.
```

Preceding these TOKEN-TYPE definitions, the subsystem would need to include DDL DEF statements for ATMX-DDL-ATMNAME and ATMX-DDL-ATMLOC. These could be similar to:

```
DEF ATMX-DDL-ATMNAME.
   02 Z-C                 PIC X(8)          SPI-NULL " ".
   02 Z-S REDEFINES Z-C.
      03  Z-I             TYPE BINARY 16    OCCURS 4 TIMES.
   02 Z-B REDEFINES Z-C   PIC X             OCCURS 8 TIMES.
END

DEF ATMX-DDL-ATMLOC.
   02 X-AREA              TYPE ZSPI-DDL-CHAR16.
   02 X-CITY              TYPE ZSPI-DDL-CHAR16.
   02 X-BRANCH            TYPE ZSPI-DDL-CHAR16.
END
```

Defining Extensible Structured Tokens

Some restrictions on what can be done with extensible structured tokens are:

- Any item that is inherently of variable length cannot be included as a field of an extensible structured token. Although an extensible structured token can grow from release to release, in any given release it must have a fixed size. You can define such an item as a fixed-length field large enough for the maximum size plus a separate field for the length, or you can define it as a simple token by itself.
- An extensible structured token can be extended only by adding fields to the end of the structure. Therefore, the grouping of the fields within an extensible structured token must not be important to the operation of the interface. If the interface defines any partitioning of the fields within an extensible structured token, no new fields can be added unless they fall into the category of the group of fields that was at the end of the original structure.
- REDEFINES is permitted within the structure definition (DEF) for an extensible structured token, but redefined fields have the same null value as the fields they overlay. (See Null Values on page 5-20.)

Subsystems should use extensible structured tokens to gather related parameters of a command or response to reduce the number of procedure calls needed to construct a message or interpret it. Generally, an extensible structured token should be used with just a single command. Even if all the fields of such a structure are used by several commands, that might not continue to be true as extensions are added in later versions of the interface. Consider this when deciding whether to share a structure among commands.

If an extensible structured token is used with a number of commands, when an extension is made to the corresponding structure, look at each of the commands that accept that token and decide how each command should treat the new fields. If a command does not have a use for a new field, about the only options that maintain compatibility with earlier requesters are ignoring the new field or insisting that it be null. Insisting that a new field be null ensures that the command can be extended later to use the value in the field without changing the behavior of a program that was implemented since the time the new field was added.

Generally, if a command takes or returns attributes of an object, define all those attributes as a single extensible structure, although there might be circumstances for which it is reasonable to organize them differently. For instance, an extensible structure must be a fixed size in any given release, so it might be better to put an attribute of variable length in a token of its own rather than reserving a fixed field of maximum size plus a length field for it. Similar advice applies to a repeating attribute or a group of attributes that are not made into a subsidiary object type.

Beware of dividing attributes of a single object into a number of structures based on some characteristic that might change in the future. For instance, it is not a good idea to make one structure for alterable attributes and another for the remaining attributes with the idea of using the first as the parameter to ALTER and using both as the

parameters to ADD. If a change is made that makes a formerly unalterable attribute alterable, problems arise. You can not move the field from the second structure to the first, because removing a field is an incompatible change. If you leave the field where it is, your partitioning of fields is no longer consistent.

Do not place other command parameters or results in the structures that contain object attributes. If a group of parameters are related to each other, they can be placed together in an extensible structured token. Unrelated parameters should be kept separate. No firm rule explains what constitutes enough of a relationship to justify placing parameters in the same structure. The fact that the parameters go with a given command/object-type pair is reasonable grounds for grouping them, especially if there is no desire to attempt grouping according to other criteria. If possible, take into account future extensions of the interface and choose an approach to grouping that is less likely to cause difficulties.

You can define single parameters either as simple tokens or as extensible structured tokens that have only one field. The latter choice is recommended if there is some chance that command extensions will add parameters that logically should be grouped with the initial solitary parameter. Otherwise, we recommend using simple tokens.

Null Values

A requester might not always have a definite value to supply for each field of a command parameter (or set of parameters) that is an extensible structured token. To indicate the absence of a value for a field, the server must define a null value for the field. SPI requires a null value to be defined for every field of an extensible structured token.

An SPI null value in DDL is a single-byte value, defined either as a character or as an integer from 0 to 255. The SPI procedures form the null value of each field by concatenating the given byte value with itself as many times as necessary to fill the data field. Because of the repetition to fill the field, the null value of the field is, in general, not the value specified to DDL. For example, an SPI null value of 1 for a 16-bit field gives a null value of 257 (1 << 8 + 1). Because the null values are generated by repetition of a given byte value, only some possible values of the field can be null values. For example, a 16-bit integer field cannot have 1 as its null value, because the value 1 is not formed by repeating the same bit pattern in both bytes of the field.

If a field is one whose presence or absence must be detectable, the null value for the field must be a value that is not in the legitimate range of values for the field. Fields conveying attribute values for an ALTER command are a common example of this.

To choose appropriate null values for the fields of your structures:

1. Examine what valid, meaningful values each field can have or might have in the future.
2. Choose as your null value some value that consists of the same byte value repeated in every byte of the field and (if the presence or absence of the field must be detectable) is not a valid value for the field. (If this field is never expected to

have a negative value, 255 is a good choice; a numeric field of any length, with this value in every byte, is always -1.)

In choosing a null value for a field, consider future possible valid values for the field and avoid making one of those values the null value; otherwise, changes to your subsystem might make it incompatible with existing applications.

Fields for which it is not necessary to distinguish between being absent and having a default value can use the default value as the null value. Be cautious about doing this because if it later becomes necessary to distinguish the cases, changing the null value of a field introduces incompatibilities.

The data-type definitions provided by SPI for use in declaring the types of fields in token definitions contain assumed null-value definitions. If desired, your subsystem can explicitly override these assumptions.

The token map for an extensible structured token defines the null value for each field. The SPI SSNULL procedure places null values in the fields of a structure, using the token map for reference. When a program assigns a value to a field, that value replaces the null value.

Is-Present Fields

Sometimes every possible value of a field is a legitimate one, leaving no value available to serve as the null value. If this is true, and your subsystem needs to distinguish whether the field was assigned a value, there are two possible solutions:

- One solution is to choose a larger size for the field than otherwise necessary. This makes many new values possible, one of which can serve as the null value. For example, if a field is a 16-bit integer and all 16-bit values are valid for that field, the field could be defined as a 32-bit integer.
- The other solution is to include in the structure an extra field, called an is-present field, whose only purpose is to indicate whether the field in question should be considered present. This extra field must be of a Boolean type. For consistency and clarity, HP recommends that the name of the extra field include the words IS-PRESENT. For example:

```
.
.
02 ZSIZE-IS-PRESENT  type ZSPI-DDL-BOOLEAN.
02 ZSIZE             type ZSPI-DDL-INT.
.
.
```

With this approach, the program building an SPI buffer must store the value TRUE in the is-present field whenever it stores a value into the field associated with it.

The significance of an is-present field is not known to DDL, so DDL still insists that there be a null value defined for the field associated with the is-present field. Any value will do. When the version information for the structure is described in the token-map declaration, the token-map field that is associated with the is-present

field must be described as NOVERSION to let SPI's automatic determination of maximum field version work properly.

The first approach has the advantage that no extra field is involved. The second has the advantage that the data type of the field is what it naturally should be.

Reset Values

Your subsystem might also want to define a set of special values that, when specified in structured-token fields representing the attributes for an ALTER command, direct the subsystem to reset those attributes to their default values as defined for the object. The Pathway subsystem uses this feature.

If you implement this feature, choose reset values that are different from the null values but are, likewise, not meaningful values for the corresponding fields. However, if your reset values are not legitimate values for the corresponding STRUCT fields in TACL, applications coded in TACL must use the #SETBYTES built-in to move values into those fields, as described in Section 8, SPI Programming in TACL.

As discussed in Section 2, SPI Concepts and Protocol, token maps let the SPI procedures maintain compatibility between different versions of an extensible structured token. To define an extensible structured token and its token map, you specify the structure in a DEF statement, and then write a TOKEN-MAP statement that references the DEF.

When writing the DDL DEF statements that specify the structures for your extensible structured tokens, you specify the null value for each field in an SPI-NULL clause. For fields with no corresponding SPI-NULL clause, DDL assigns the value 255 (a field with all bits set).

The SSNULL procedure uses token maps to set a structure to its null values. SSPUT uses token maps to set the maximum field version (ZSPI-TKN-MAX-FIELD-VERSION) in the SPI message header for use in version compatibility checking. Finally, SSGET uses token maps to truncate or pad the value in the buffer to match the value expected by the caller.

The maximum field version in a token map is the most recent release version associated with any of the fields defined in the map. When an application adds an extensible structured token to a buffer, the SPI procedure SSPUT updates the maximum field version in the message header, as necessary, so it reflects the most recent version of any non-null field of any extensible structured token in the entire buffer. (When the special SSPUT operation ZSPI-TKN-DATA-FLUSH or ZSPI-TKN-DELETE has been performed on the buffer, the maximum field version can actually be greater—that is, more recent—than the most recent field version still represented in the buffer.)

The format of a token map is given in Appendix C, SPI Internal Structures. You should only need this information for debugging.

Token Map

For extensible structured tokens, programs pass to the SPI procedures an extended identifying code called a token map. A token map is a variable-length integer array. The first two words of a token map have the same structure as a token code. All token maps have a token data type of ZSPI-TDT-MAP, a token length of 255, and a token number specified by the subsystem.

Most subsystems define a set of token maps. Names of token maps are of the form *subsys*-MAP-*name*, where *subsys* is the four-character subsystem abbreviation and *name* identifies the token.

Coding Subsystem Definitions

To produce your subsystem definitions:

1. Write them in DDL.
2. Run the DDL compiler to translate them into definitions in the programming language you are using.
3. Include the appropriate language versions of the definitions in your subsystem programs. Application programs that communicate with your subsystem must also include the corresponding definition files. Because all the language versions of each definition file are derived from the same DDL source, the programming languages these applications use can be different from yours.

For consistency, the names in your DDL definitions should follow the same naming conventions used by NonStop Kernel subsystems (see Naming Conventions on page 2-11). Do not begin your own names (for definitions and for component fields of structures) with uppercase or lowercase *Z*. Any names beginning with *Z* might be used now or later by software for the NonStop server.

This discussion gives special considerations for using DDL to code definition files for a subsystem. For syntax and general programming information for DDL, see the *Data Definition Language (DDL) Reference Manual*.

Your subsystem should provide definitions for:

- One or more recommended buffer-size values
- Buffers of the recommended sizes
- The subsystem ID
- Command numbers for all commands
- Object-type numbers for all object types
- Token numbers for all tokens
- Error numbers for all errors
- Event numbers for all event messages

- Possible values for any enumerated-type tokens or fields
- Structure definitions for all extensible structured tokens
- Token codes for all simple tokens
- Token maps for all extensible structured tokens

You can also include any other definitions that might be of use to requester programs sending commands to your subsystem.

Using the SPI Standard DDL Definitions

When beginning to code your definition files, print a copy of the ZSPIDEF.ZSPIDDL file from the disk volume chosen by your site. The DDL definitions in this file provide a starting point for your subsystem-specific definitions. Use these definitions or build on them whenever possible. Assume that these definitions are known; do not repeat them in your own definition files.

This file is also a useful guide in coding the DDL for your own definitions. For further guidance, print the DDL definition files for one or more NonStop Kernel subsystems (which have names of the form `ZSPIDEF.`*`subsys`*`DDL`). The SPI definition file includes some types of declarations, such as token data types, that subsystems do not need to provide. In addition, the subsystem definition files contain some other declarations—such as command numbers, object-type numbers, and token maps— not included in the SPI definition file.

The ZSPIDEF.ZSPIDDL file begins with structure declarations for some simple tokens. For sample definitions for extensible structured tokens, see the DDL definition file for one of the NonStop Kernel subsystems.

Token codes are declared using TOKEN-CODE statements. Token maps are declared with TOKEN-MAP statements. The subsystem ID, the buffer, and structure definitions are declared using DEF statements. All other items are simple constants, suited to CONSTANT declarations. For the syntax of these DDL statements, see the *Data Definition Language (DDL) Reference Manual*.

Suggestions on Data Representation

Each of the four programming languages that support SPI (TAL, C, COBOL, and TACL) lacks support for certain data types. These guidelines minimize the problems that might arise when applications written in these languages send commands to your subsystem. These guidelines apply both to the data types of fields in structures and to the data types of individual tokens:

- Signed 16-bit and 32-bit integers are valid in all four languages. However, signed 64-bit integers are valid only in TAL, COBOL, and TACL. C for NonStop systems does not support signed 64-bit integers.
- Unless there is a good reason to do otherwise, pass unsigned 16-bit integers in 32-bit signed integer fields, and pass 8-bit integers in signed 16-bit integer fields. This

ensures that applications written in COBOL can accommodate the full range of values of these integers.

- Do not use 32-bit floating point or 64-bit floating point values unless absolutely necessary. These data types are not supported by COBOL or TACL.
- Do not use collections of bit flags, short integers, or short enumerated fields packed into a word unless there is a good reason to do so. If at all possible, pass this information in as many individual fields as necessary.
- Regarding the previous three guidelines, if the data in question is already used in existing external interfaces to software for NonStop servers (such as system procedures), and it seems likely that the application would find it advantageous to be able to use the data in conjunction with those interfaces, that is a good reason to keep the data in its original format. One example of such data is the PID.
- Do not use scale factors unless absolutely necessary.
- Represent all numeric values as binary values, not as strings of ASCII digits.
- Represent Boolean values as 16-bit signed integers. Use -1 for TRUE and 0 for FALSE. Except for the null value and possibly a reset value, all other values should be invalid.
- Represent enumerated values as 16-bit signed integers.
- Declare fields that are to contain ASCII strings like (for the example of an 8-character field):

```
def subsys-DDL-CHARFIELD.
   02 Z-C                pic x(8)          null " ".
   02 Z-S redefines Z-C.
      03  Z-I             type binary 16    occurs  4 times.
   02 Z-B redefines Z-C   pic x             occurs  8 times.
end
```

  Some system procedures need word addresses for ASCII data. This declaration ensures that such addresses are available. In addition, Enable ignores REDEFINES clauses, taking only the first description of a field; Enform and TACL also take the first description by default, but do permit use of REDEFINES if you explicitly name the field. This form of declaration can thus be used correctly by TAL, COBOL, TACL, Enform, and Enable.

- If you must declare an ASCII field that does not comply with the previous guideline, this can cause trouble in TAL when the field is located above the 32K boundary.
- The SPI DDL definitions contain structures for a number of character-field sizes as well as for all of the standard SPI data types. These definitions are suitable for use in defining your own tokens or structures, and you are encouraged to use them. For further descriptions, see Section 4, ZSPI Data Definitions.

  If you have to define a character field with a number of subfields, the SPI definitions might not be suitable. The SPI definitions force word alignment, so if

any of your fields are of odd length, filler bytes might appear where you don't want them.

- In the TAL declarations, make arrays have a lower bound of zero.
- Represent time intervals in microsecond precision and stored in fields of the SPI type ZSPI-DDL-TIMESTAMP.

Dividing Your Definition File Into Sections

Organize your definitions such that someone who wants to build a DDL dictionary containing your subsystem's definitions can simply ?SOURCE in your entire *subsys*DDL file, and can compile your definitions into a dictionary already containing the definitions for another subsystem.

One way to accomplish this is to set up a subvolume containing these two files:

```
--------------------  file subsysDDL  -------------------
!
!  subsystem name  (release date)
!
!   Depends on ZSPIDDL
!
?SECTION  BUILDING-BLOCKS
building-block-only DEFS, that is, DEFs whose output a requester
program using the interface would have no need
of using
?SECTION  REQUESTER-VISIBLE-1
DEFs the user of the interface needs to have
?SECTION  REQUESTER-VISIBLE-2
constants, token codes, token maps, subsystem ID
?SECTION  REQUESTER-VISIBLE-3
DEF declaring the buffer structure
------------------  end file subsysDDL  -----------------

-------------------  file subsysROOT  -------------------
?DICT !
?SOURCE $SYSTEM.ZSPIDEF.ZSPIDDL
?SOURCE subsysDDL (BUILDING-BLOCKS)
?TAL   subsysTAL!
?COBOL subsysCOB!
?C subsysC!
?TACL  subsysTACL!
?SOURCE subsysDDL (REQUESTER-VISIBLE-1)
?SETSECTION CONSTANTS
?SOURCE subsysDDL (REQUESTER-VISIBLE-2)
?SETSECTION
?SOURCE subsysDDL (REQUESTER-VISIBLE-3)
-----------------  end file subsysROOT  -----------------
```

To generate the definition files, enter a command line such as:

```
DDL /IN subsysROOT, OUT $S/
```

The names *subsys*ROOT, BUILDING-BLOCKS, REQUESTER-VISIBLE-1, REQUESTER-VISIBLE-2, and REQUESTER-VISIBLE-3 are just illustrative—you can choose other names, because those names are not of interest to someone who wants to use the *subsys*DDL file just to compile the definitions into a DDL dictionary. That person just ?SOURCEs in the whole file without naming the sections individually.

Version Compatibility

It is useful to design your subsystem (server) so that when you make future enhancements to it, older requesters that communicate with it continue to run:

1. Do not change the meaning of token codes, command numbers, object-type numbers, error numbers, token values, and structure field values.
2. Token values and structure fields should not change size or data type.
3. The offset of fields from the beginning of a structure should not change.
4. Command numbers, object-type numbers, token codes, valid token values, and valid structure field values should not be deleted.
5. New features can be defined by defining new command numbers, new object-type numbers, new tokens, new values for existing command tokens, new values for existing structure fields in command tokens, new fields at the end of extensible structured tokens, or new fields replacing FILLER in extensible structured tokens.
6. New values of existing response tokens or of existing structure fields in response tokens can be defined if the new values are returned only as the result of explicit choice on the part of the requester to make use of optional new capabilities.
7. The size of a command message should not grow larger than the smallest size accepted by any currently supported version of the server.

   Estimate how much growth in command size your server will have to accommodate during its supported life and make the server accept messages at least that size.
8. Do not change the names of tokens, literals, structures, and other items declared in the definition files.
9. To avoid naming conflicts with HP declarations in future RVUs, avoid using names beginning with *Z*.
10. Unrecognized command numbers, object types, tokens, fields of structured tokens, values of tokens, or values of fields of structured tokens should cause rejection of the command.
11. Recognized, but unexpected, tokens (including duplicate tokens if there are more than expected) should also cause rejection of the command.
12. If the field ZSPI-TKN-MAX-FIELD-VERSION contains a version greater than the version of the server, your server should reject the command.

13. The size of a response message should not grow larger than the smallest size ever given as the recommended buffer size.

When an interface is first designed, the recommended buffer size should factor in an estimate of how much the response is likely to grow in the future. Allow a generous amount, but keep the buffer size within reason. The recommended buffer size can be changed in later RVUs, but such a change does not increase the size of the buffer in programs that were compiled before the change; it is desirable that those programs continue to run.

A server can always return its current version of the response as long as that response (in single response mode) fits in the smallest buffer size that ever was recommended for the interface. If a response grows too large to fit into an earlier recommended buffer size, then, at least for requesters using an earlier buffer size, the server should check the version in the subsystem ID supplied in the command and return a version of the response that fits in and is compatible with the buffer. The response does not need to be exactly what the earlier version of the server returned—the response need only fit in the requester's buffer and contain at least all the items that were defined in the earlier version. The server need do this only for single-response mode or if the first response does not fit in the buffer in multiple-response mode.

Suppose the requester's version is 11, the server's version is 12, and the recommended buffer size at version 11 is too small to hold some of the version 12 responses. If the version 12 server has to send one of those longer responses to a version 11 requester and the requester's buffer happens to be long enough (it declared a buffer longer than the recommended size), the server can return the version 12 response. If the buffer happens not to be long enough, the version 12 server should return a response that is cut back to fit in the size available. The response does not need to be exactly what version 11 of the server returned—the response version 12 of the server returns to a requester of version 11 need only fit in the requester's buffer and contain at least all the items that were defined in version 11 of the server.

If desired, your server can do more in the way of adjusting responses to the version of the requester.

Defining Objects

To determine the kinds of n objects to be defined for the programmatic interface, start with the set of object types it would make sense to present to a user of a human command interface for the subsystem. For each of these object types, list the attributes needed to define an object of that type.

If an object type has attributes that can be repeated indefinitely, that can cause trouble. For one thing, SPI commands must have some definite upper limit on their size. Two simple ways to avoid problems in this case are:

- Impose an upper limit on the number of repetitions of the attributes. This approach has its place, but often is not suitable.

- Remove the attributes that can be repeated from the original object type and put them in a new object type that is a subsidiary of the original object type, making each repetition a separate instance of the subsidiary type. The name of an instance of the subsidiary object type is generally one of the attributes of the subsidiary object type concatenated with the name of the instance of the original object type.

For example, the human command interface to the Pathway subsystem has an object type known as a server class. The attributes of a server class include attributes of assign messages that are sent to each process of a server class as the process is started. There can be an indefinite number of assign messages for a given server class. The Pathway programmatic interface has an object type called a server class, which has all the nonrepeating attributes of the human-interface server class. It also has a server-assign object type. The attributes of the server-assign object type describe one assign message. The name of an object of the server-assign type is formed by taking the name of the server-class object whose assign message is being described and concatenating it with the logical name of the assign message.

If in the human interface a server-class object name SVR1 has two assign messages named MAT-FILE and ITM-FILE, in the programmatic interface there would be a server-class object named SVR1 and two server-assign objects named (SVR1, MAT-FILE) and (SVR1, ITM-FILE).

When subsidiary object types exist, the server must recognize the relationship between them and the main object type. It must prevent adding a subsidiary object for which there is no matching main object, and deletion of a main object must automatically delete all subsidiary objects related to it before deleting the main object itself.

Similar instances of repeating groups of information can arise when designing STATUS or STATS commands. The same two approaches for resolving the problem apply to those cases, too. This can result in subsidiary object types that are only used with the STATUS or STATS command.

Subsystem ID

All HP software for DSM uses the standard subsystem ID format defined in Subsystem IDs (SSIDs) on page 2-44. The owner ID field in this format lets you define a subsystem ID that is unlikely to conflict with those of NonStop Kernel subsystems provided by HP or subsystems provided by other vendors. For this field, choose an eight-character string to identify your company or organization.

This name must start with an alphabetic character and must contain only alphabetic, numeric, and hyphen characters, padded on the right with blanks; no embedded blanks are allowed except at the end. The case of alphabetic characters is significant; for example, “COMPANY” and “Company” are recognized as different names. To avoid confusion, HP recommends that you define your owner ID with all alphabetic characters in uppercase, as in the HP owner ID (“TANDEM~~bb~~”).

Choose a value for the subsystem-number field. If your company is writing more than one subsystem, this number should be unique for each subsystem.

Note. When subsystems from a number of different suppliers are installed on a given system, there is a potential for confusion and improper operation if two subsystems share the same owner and subsystem number. Organizations developing subsystems should carefully select a Z-OWNER value that is unlikely to be duplicated by other organizations developing subsystems. The company name is often a good choice, but it might not always be suitable. Because HP does not maintain a registry of these names, HP cannot guarantee that conflicts will not occur.

Your version numbers need not have the letter-number format used by software for NonStop servers. You can use any version numbering scheme representable with 16-bit unsigned numbers.

Requester-Server Communication

These considerations apply to communication between your subsystem server and the application requesters that send commands to it:

Server Startup

Servers that are started dynamically using NEWPROCESS or PROCESS_CREATE_ follow the standard startup message protocol. The server should follow the standard practice of verifying that the startup message is from the process that created it. (You can use the INITIALIZER procedure to do this verification automatically.)

For servers that can be started directly by the requester, keep in mind that a requester written in TAL or C might not be passing along assign and param messages (because this does not happen automatically in TAL or C). Carefully consider any plans to have the server use assign or param information.

Identifying SPI Messages From Applications

Requesters that send SPI command messages to your subsystem do so by opening your server with a process-name qualifier of #ZSPI, which then appears as the first qualifier name in the interprocess OPEN message. Your server should expect all SPI messages it receives to come through opens with this qualifier.

Receiving Commands and Sending Responses

In writing a subsystem, you are programming from a point of view different from that of a management application. The application sends commands to the subsystem as SPI buffers and processes the response buffers resulting from those commands. The subsystem receives and processes command buffers and sends the appropriate replies.

A server reads each SPI command message from its $RECEIVE file. To read the message and reply to it, a TAL program ordinarily uses the file-system procedures READUPDATE and REPLY directly; a C program uses the same procedures through

the **tal** interface declaration. A COBOL program normally uses READ and WRITE. A TACL routine generally uses a loop with #INPUTV and #REPLYV.

After the server has received an SPI message, it can use the SPI procedures to get command, object-type, and parameter information from the message buffer and put context, status, and other response information into a response buffer.

Checking the Command Message for Validity

When your subsystem receives a command message, it should first check that the message is a valid SPI command buffer:

1. Check that the byte count read from $RECEIVE is at least 6 bytes.
2. Check that the value of the first word of the buffer is -28.
3. Check that the buffer-length field, Z-BUFLEN, is greater than or equal to the used length, ZSPI-TKN-USEDLEN.
4. Check that the header type, ZSPI-TKN-HDRTYPE, is equal to ZSPI-VAL-CMDHDR, indicating that the buffer is a command buffer.

If any of the preceding tests fails, your subsystem should not attempt to send an SPI response buffer but should return file-system error 2 to the requester.

The read count used by the requester should not affect your server’s response, because detecting a short read (fewer bytes read than the number requested) is the responsibility of the requester.

Checking Whether Your Subsystem Can Process the Command

If the command message passes all the validity checks, next ensure that your subsystem can process the command:

1. Check that the subsystem ID in the command, ZSPI-TKN-SSID, matches the subsystem ID for your subsystem, to ensure that the command is intended for your subsystem.
2. Check that the maximum field version, ZSPI-TKN-MAX-FIELD-VERSION, is less than or equal to your subsystem’s version.
3. Reset the buffer by calling SSPUT with ZSPI-TKN-RESET-BUFFER, as requesters do when they receive a response buffer; set the buffer length to the length declared by your subsystem. If you receive an SPI error -5 (buffer full) when resetting the buffer, the command is too long for your subsystem to process.

If one of these tests fails, your subsystem should return an SPI error response containing at least a return token (ZSPI-TKN-RETCODE), and, if possible, other information about the error. The value of the return token should be an error number defined by your subsystem to identify the particular error.

Likewise, your subsystem should report any subsequent errors as SPI error responses, each identifying the error with a subsystem-defined error number in the return token.

Checking Tokens in the Command

Your subsystem should report missing required tokens, and should detect and report any extraneous tokens (including too many occurrences of valid tokens).

This pseudocode example illustrates one way to approach these tasks. It shows the code to check a buffer for a specific command. It is assumed that there is a section of code like this for each command:

```
int    count[min^tnm:max^tnm]
struct token(ZSPI^DDL^TOKENCODE^DEF)
struct ssid(ZSPI^DDL^SSID^DEF)
int    byte^len

for each token number do
   count [token number] := 0
   end do

! Scan the buffer, checking that we recognize each token,
! counting the occurrences, and retrieving the values.

SSPUT(buf,ZSPI^TKN^INITIAL^POSITION,0)   ! position to
                                         ! start of buffer
do until end of buffer
   SSGET(buf,ZSPI^TKN^NEXTTOKEN,token,,,ssid)
   if SSGET error then
      begin
      ssgeterr(ZSPI^TKN^NEXTTOKEN,0,SSGET status)
      go to send^reply
      end
   if ssid <> subsys^VAL^SSID then
      begin
      reject(token,1,subsys^ERR^IMPROPER^TOKEN,ssid)
      go to send^reply
      end

   case token.z^tkn.z^number of

      subsys^TNM^TOKEN1 ->          ! simple token

          if token.z^tkncode <> subsys^TKN^TOKEN1 then
             begin
             reject(token,1,subsys^ERR^IMPROPER^TOKEN)
             go to send^reply
             end
          count[subsys^TNM^TOKEN1] :=
                                count[subsys^TNM^TOKEN1] + 1
          if count[subsys^TNM^TOKEN1] > limit^for^token1 then
             begin
             reject(token,count[subsys^TNM^TOKEN1],
                    subsys^ERR^TOO^MANY^OCCURENCES)
             go to send^reply
             end

          ! For simple, variable-length tokens, a check of
```

```
         ! the length is needed.  Omit this for fixed-
         ! length tokens.  (Omitting the token code that
         ! is the third argument to SSGET means to use
         ! the token at the current position.)

         SSGET(buf,ZSPI^TKN^LEN,,,byte^len)
         if byte^len > size of token1^value then
            begin
            reject(token,count[subsys^TNM^TOKEN1],
                subsys^ERR^VALUE^TOO^LONG)
           go to send^reply
           end

        SSGET(buf,subsys^TKN^TOKEN1,
              token1^value[count[subsys^TNM^TOKEN1]-1])
        if SSGET error then
           begin
           ssgeterr(subsys^TKN^TOKEN1,0,SSGET status)
           go to send^reply
           end

    subsys^TNM^TOKEN2 ->    ! extensible structured token

        if token.z^tkntype <> ZSPI^TYP^STRUCT then
           begin
           reject(token,1,subsys^ERR^IMPROPER^TOKEN)
           go to send^reply
           end
        count[subsys^TNM^TOKEN2] :=
                               count[subsys^TNM^TOKEN2] + 1
        if count[subsys^TNM^TOKEN2] > limit^for^token2
                                                      then
           begin
           reject(token,count[subsys^TNM^TOKEN2],
                  subsys^ERR^TOO^MANY^OCCURENCES)
           go to send^reply
           end
        SSGET(buf,subsys^MAP^TOKEN2^V,
              token2^value[count[subsys^TNM^TOKEN2]-1])
        if SSGET error then
           begin
           ssgeterr(subsys^MAP^TOKEN2^V,0,SSGET status)
           go to send^reply
           end

    (Repeat for each token allowed in this command.)



    otherwise ->
          reject(token,1,subsys^ERR^IMPROPER^TOKEN)
          go to send^reply
    end case
end do
```

```
! Check that all the required tokens were supplied.

if count[subsys^TNM^REQUIRED1] = 0 then
   reject(subsys^TKN^REQUIRED1,0,
                                subsys^ERR^REQUIRED^MISSING)
   go to send^reply
   end

if count[subsys^TNM^REQUIRED2] = 0 then
   reject(subsys^MAP^REQUIRED2^V,0,
                                subsys^ERR^REQUIRED^MISSING)
   go to send^reply
   end

(Repeat for each required token.)

!  do inter-field value and presence checks, if any

if required1^value = x and count[subsys^TNM^TOKEN2] = 0 then
   begin
   reject(subsys^MAP^TOKEN2^V,0,
                                subsys^ERR^OPTIONAL^MISSING)
   go to send^reply
   end

if count[subsys^TNM^TOKEN1] <> 0 and
                          count[subsys^TNM^TOKEN3] <> 0 then
   begin
   reject(subsys^TKN^TOKEN3,1,
                               subsys^ERR^CONFLICTING^TOKENS)
   go to send^reply
   end

if token1^value[3] > max^token1^value then
   begin
   reject(subsys^TKN^TOKEN1,3,subsys^ERR^BAD^VALUE)
   go to send^reply
   end

(Include as many tests of whatever form as are needed.)

(Execute the command using the values obtained above.)



REJECT:  Inserts RETCODE and error list into response buffer
         to report missing or incorrect token in command

proc reject(tokencode,index,errcode,ssid) variable
   int(32) .tokencode
   int      index
   int      errcode
   struct  .ssid(ZSPI^DDL^SSID^DEF)
   begin
   struct error(ZSPI^DDL^ERROR^DEF)
```

```
   struct parm^err(ZSPI^DDL^PARM^ERR^DEF)

   SSPUT(reply,ZSPI^TKN^RETCODE,errcode)
   SSPUT(reply,ZSPI^TKN^ERRLIST)
   error.z^ssid := subsys^VAL^SSID
   error.z^error := errcode
   SSPUT(reply,ZSPI^TKN^ERROR,error)
   parm^err.z^tokencode.z^tkncode := tokencode
   parm^err.z^index := index
   parm^err.z^offset := 0
   SSPUT(reply,ZSPI^TKN^PARM^ERR,parm^err)
   if $param(ssid) then
      begin
      SSPUT(reply,ZSPI^TKN^SSID^ERR,ssid)
      end;
   SSPUT(reply,ZSPI^TKN^ENDLIST)
   end reject

SSGETERR:  Inserts RETCODE and error list into response buffer
           to report SSGET error

proc ssgeterr(tokencode,index,errcode)
   int(32) .tokencode
   int      index
   int      errcode
   begin
   struct error(ZSPI^DDL^ERROR^DEF)
   struct parm^err(ZSPI^DDL^PARM^ERR^DEF)

   SSPUT(reply,ZSPI^TKN^RETCODE,subsys^ERR^SSPROC^ERROR)
   SSPUT(reply,ZSPI^TKN^ERRLIST)
   error.z^ssid := subsys^VAL^SSID
   error.z^error := subsys^ERR^SSPROC^ERROR
   SSPUT(reply,ZSPI^TKN^ERROR,error)
   SSPUT(reply,ZSPI^TKN^ERRLIST,,,ZSPI^VAL^SSID)
   error.z^ssid := ZSPI^VAL^SSID
   error.z^error := errcode
   SSPUT(reply,ZSPI^TKN^ERROR,error)
   SSPUT(reply,ZSPI^TKN^PROC^ERR,ZSPI^VAL^SSGET)
   parm^err.z^tokencode := tokencode
   parm^err.z^index := index
   parm^err.z^offset := 0
   SSPUT(reply,ZSPI^TKN^PARM^ERR,parm^err)
   SSPUT(reply,ZSPI^TKN^ENDLIST)
   SSPUT(reply,ZSPI^TKN^ENDLIST)
   end ssgeterr
```

This example does not take into account that one cannot pass a literal as a reference parameter in TAL; your subsystem must include code to pass the parameters properly.

For token maps, the example assumes that the *subsys*^MAP^xxxx definitions are used to declare and initialize variables whose names are formed by adding ^V to the end: *subsys*^MAP^xxxx^V. There are, of course, other ways to make the token maps available in the program.

Because this example focuses on scanning for tokens in the command, it does not include error handling for errors from the SSPUT calls that build the response. Do not overlook those errors when writing your subsystem.

The example assumes that the tokens are permitted to appear multiple times. For those tokens that are allowed to appear only once, there is no need to subscript the *subsys*^value variable with the occurrence count.

The example uses an array indexed by token number to keep track of what tokens have appeared. If you wish, you can use individual variables rather than an array. Using individual variables might save substantially on the amount of storage used, if the array would contain many unused entries for tokens that are not permitted in commands.

Here are a few more points about the example:

- The program checks the subsystem ID returned by SSGET with NEXTTOKEN to be sure that all the tokens are qualified by the subsystem ID for this subsystem—a good practice. Your subsystem should not assume that is true, but should check.
- Before calling SSGET to retrieve the value of the token, the program checks the token code to make sure its token type matches the server's definition (the token type includes both the token data type and the length). For extensible structured tokens, it checks just the token type field of the token code. The test is against ZSPI^TYP^STRUCT, because that is the token type used in the buffer for extensible structured tokens. For simple tokens, it checks the entire token code against the server's literal for the token code; doing so is somewhat easier than determining what the value of the token-type field by itself should be.

  Because the token number is supposed to identify the token uniquely, it might appear that those extra checks would not be required. However, the server needs to determine whether the requester's code accidentally modifies the value of a token code or used another subsystem's token code.
- To preserve version compatibility, your subsystem should always use a token map when retrieving the value of a structured token (rather than using the simple token code of type ZSPI-TYP-STRUCT that is stored in the buffer).
- When an error occurs with an extensible structured token, sometimes the token-code field in ZSPI-TKN-PARM-ERR is of type ZSPI-TYP-STRUCT, and other times it is of type ZSPI-TYP-MAP.
- If there are no validity checks involving more than one token, you do not need the inter-field checks part of the program—you can do validation of a token value in that token's branch of the CASE statement. Checks that involve more than one token must be done after all the tokens have been retrieved, because your program cannot count on tokens appearing in a particular order.

The method described here is not the only valid approach, but it does illustrate the functions that should be performed.

Checking for Command Cancellation

As described in Canceling Commands on page 5-14, an application can use the file-system CANCEL or CANCELREQ procedure to cancel command requests issued to your subsystem.

Servers do not have to take any special action. A server can continue processing a command that has been canceled and issue a response for it. The file system automatically discards the response if the corresponding command has been canceled. The only negative consequence is that the server might be slower than it could be in responding to new commands if it is busy working on ones that have been canceled.

Servers can learn whether commands they have accepted have been canceled by using the MESSAGESTATUS procedure or SETMODE 80. If your server wishes to check on a single command, it can call MESSAGESTATUS and pass it the *tag* of the command; MESSAGESTATUS returns a value indicating whether that command has been canceled. If your server is handling many different commands at once (often true of a multithreaded server), it can call SETMODE 80 and ask the file system to send it all interprocess CANCEL messages on commands issued to the server. For more detailed information about these features, see the *Guardian Procedure Calls Reference Manual*.

It is strongly recommended that servers implementing commands that can incur significant processing delays, such as waiting for asynchronous events, doing I/O, and so on, include checks at appropriate points during the execution of the command (and during periods of waiting for I/O completions, if practical) to see whether the command has been canceled. If the server discovers that its command has been canceled, it should stop processing the command and clean up after it with as little further processing as possible, consistent with maintaining the integrity of the server and the objects it controls.

Only check for cancellation when it is reasonable and possible. Do not check for cancellation when:

- The command is typically of short duration and can be delayed only by factors that would also prevent processing of the cancellation (such as lack of CPU or memory resources).
- The command, after it is started, must proceed to completion and cannot be safely interrupted (such as PUP SPARE of an online disk).
- The only possible delay is due to an operation that must be done in a waited manner (such as invoking the SORT library or performing I/O from COBOL). If a command involves several such operations, a check for cancellation between them would be appropriate unless condition 2 applies.

Servers that do not otherwise need to support command cancellation need not be aware of command cancellation at all.

If a subsystem needs to provide a way for a requester to cancel a command and learn something about how much the command did before it was canceled, that subsystem must implement an additional form of cancellation.

Using SSPUT to Place Lists in the Buffer

To place a list in the buffer:

1. A program calls SSPUT to put the beginning token—ZSPI-TKN-DATALIST, ZSPI-TKN-ERRLIST, ZSPI-TKN-SEGLIST, or ZSPI-TKN-LIST—in the buffer.
2. SSPUT adds that list token to the buffer and selects the list, so that subsequent SSPUT calls place the specified tokens inside the list.
3. To end the list, the program calls SSPUT with ZSPI-TKN-ENDLIST.
4. SSPUT adds the ZSPI-TKN-ENDLIST token to the buffer, pops out of the list so that the list is no longer selected, and sets the current-token pointer to the list token that begins the list.

Your program can initialize the current position to the beginning of the currently selected list by using ZSPI-TKN-INITIAL-POSITION with a token value of ZSPI-VAL-INITIAL-LIST (-1). If no list is currently selected, SSPUT sets the position to the beginning of the buffer.

This pseudocode example shows how a sequence of SSPUT operations can be used to build a response buffer containing lists. It contains two data lists and one error list, the latter enclosed within the second data list.

The same response tokens (TKN-A and TKN-B) appear in both response records, but that they have different values (A[1] and B[1], A[2] and B[2]), representing results for different objects.

```
SSINIT (BUF, LEN, SSID, ZSPI^VAL^CMD^HDR, CMD);  !INITIALIZE
                                                 ! BUFFER

SSPUT  (BUF, ZSPI-TKN-DATALIST)      !FIRST RESPONSE RECORD
SSPUT  (BUF, TKN-A, A[1])
SSPUT  (BUF, TKN-B, B[1])
SSPUT  (BUF, ZSPI-TKN-RETCODE, STATUS)
SSPUT  (BUF, ZSPI-TKN-ENDLIST)          !END OF FIRST
                                        ! RESPONSE RECORD

SSPUT  (BUF, ZSPI-TKN-DATALIST)         !SECOND RESPONSE
                                        ! RECORD
SSPUT  (BUF, ZSPI-TKN-ERRLIST)          !WARNING
SSPUT  (BUF, ZSPI-TKN-ERROR, ERROR)     !ERROR TOKEN
SSPUT  (BUF, TKN-INFO, INFO)            !ERROR INFORMATION
SSPUT  (BUF, ZSPI-TKN-ENDLIST)          !END OF WARNING
SSPUT  (BUF, TKN-A, A[2])
SSPUT  (BUF, TKN-B, B[2])
SSPUT  (BUF, ZSPI-TKN-RETCODE, STATUS)
SSPUT  (BUF, ZSPI-TKN-ENDLIST)          !END OF SECOND
                                        ! RESPONSE RECORD
```

```
SSPUT  (BUF, ZSPI-TKN-CONTEXT, CONTEXT) ! CONTEXT INFORMATION
```

Defining Commands

After you have determined what objects your subsystem will manage, you must decide what management services the subsystem needs to have, and then design programmatic commands to implement those services. Most SPI interfaces need to provide the same basic operations on their objects: creating them; altering them; deleting them; starting, stopping, and otherwise controlling their execution; and reporting their configuration and state. If applications will use your subsystem in conjunction with NonStop Kernel subsystems provided by HP, HP recommends that your programmatic commands be as consistent as possible with the corresponding programmatic commands defined by those subsystems. Preserving consistency makes application programming easier.

A single command can operate on a single object or, if appropriate, on multiple objects.

You must define your commands and parameters so that a single command always fits into a buffer of the recommended size you define. (SPI does not support multiple commands in one SPI buffer or commands continued across multiple buffers.) Likewise, you must define your commands and recommended buffer size so that a single response record and its associated context information always fits into a buffer of the recommended size. SPI does not support continuation of a single response over multiple buffers.

Ensuring that a single response record fits into a single buffer can require redefining the command. For example, consider the conversational FUP INFO command. It can return several different types of information: partition information, alternate-key information, alternate-file information, extent information, statistics information, and so on. For programmatic use, this function needs to be redefined as separate commands, each of which returns a single type of information: INFO EXTENTS, INFO ALTKEYS, INFO ALTFILES, INFO STATISTICS, and so on. For each of these separate commands, a single response record (information about a single file) easily fits into a single buffer of reasonable size.

In the example above, it would not be sufficient to simply define EXTENTS, ALTKEYS, ALTFILES, and so on as command parameters. If an application were to specify all the parameters, the resulting response record might overflow a single buffer, violating the rule that a single response record must always fit into a buffer of the recommended size.

GETVERSION Command

All NonStop Kernel subsystems include a GETVERSION command, which returns the version number of the server in both internal form (in the header token ZSPI-TKN-SERVER-VERSION) and displayable form (in the nonheader token ZSPI-TKN-SERVER-BANNER). Your subsystem should include a similar command. Optionally, your GETVERSION command can return additional information.

Every command sent to a NonStop Kernel subsystem returns the internal form of the server version in ZSPI-TKN-SERVER-VERSION. However, GETVERSION is useful because it is a safe command—one that simply returns information and makes no changes to the state of the system or network.

Single and Multiple Response Records per Response

All NonStop Kernel subsystems support multiple response records per response buffer, at least to the extent of recognizing the ZSPI-TKN-MAXRESP token and using it to decide whether to enclose each response record in a data list. For consistency, subsystems you write should do the same. However, unless an application specifically requests multiple response records, the subsystem should return a single response record per buffer and should not enclose it in a list.

When returning a single response record per response buffer, the server does not need to check the size of the requester's buffer; it can assume that the requester has a buffer of the recommended size. However, when returning multiple response records, the server must check the size of the requester's response buffer, in order to determine how many response records it can return per response message. Your server can check the requester's buffer size either by calling the file-system procedure RECEIVEINFO. Checking the buffer-length field (Z-BUFLEN) in the command is not a reliable method for determining the requester's buffer length because the value stored in that field is the buffer size that the requester specified in its SSINIT call for the command, which may be smaller than the actual available buffer space.

If the requester has a larger buffer than the server has space in which to build a response, the server should just use the space it has.

If the requester has asked for up to *n* response records, but its buffer is not large enough to hold that many, the server should simply return fewer than *n* response records—however many will fit in the space available. This action, rather than an error response, is recommended so that a later revision of your subsystem with an increased response-record size does not cause errors in existing applications.

If the requester has supplied a buffer smaller than needed for even one response record, the server should ignore the size of the requester's buffer and use the recommended buffer size.

Defining the Context Token

If your subsystem supports response continuation, you must provide a context token in each response message that is not the end of the response for the command. For context-free subsystems, this context token must contain all the information the server needs to continue processing where it left off. A response message should never contain more than one context token.

As described in Section 2, SPI Concepts and Protocol, the requester should return this context token in the next command message along with the original command.

Each server that receives a command with a context token should check for:

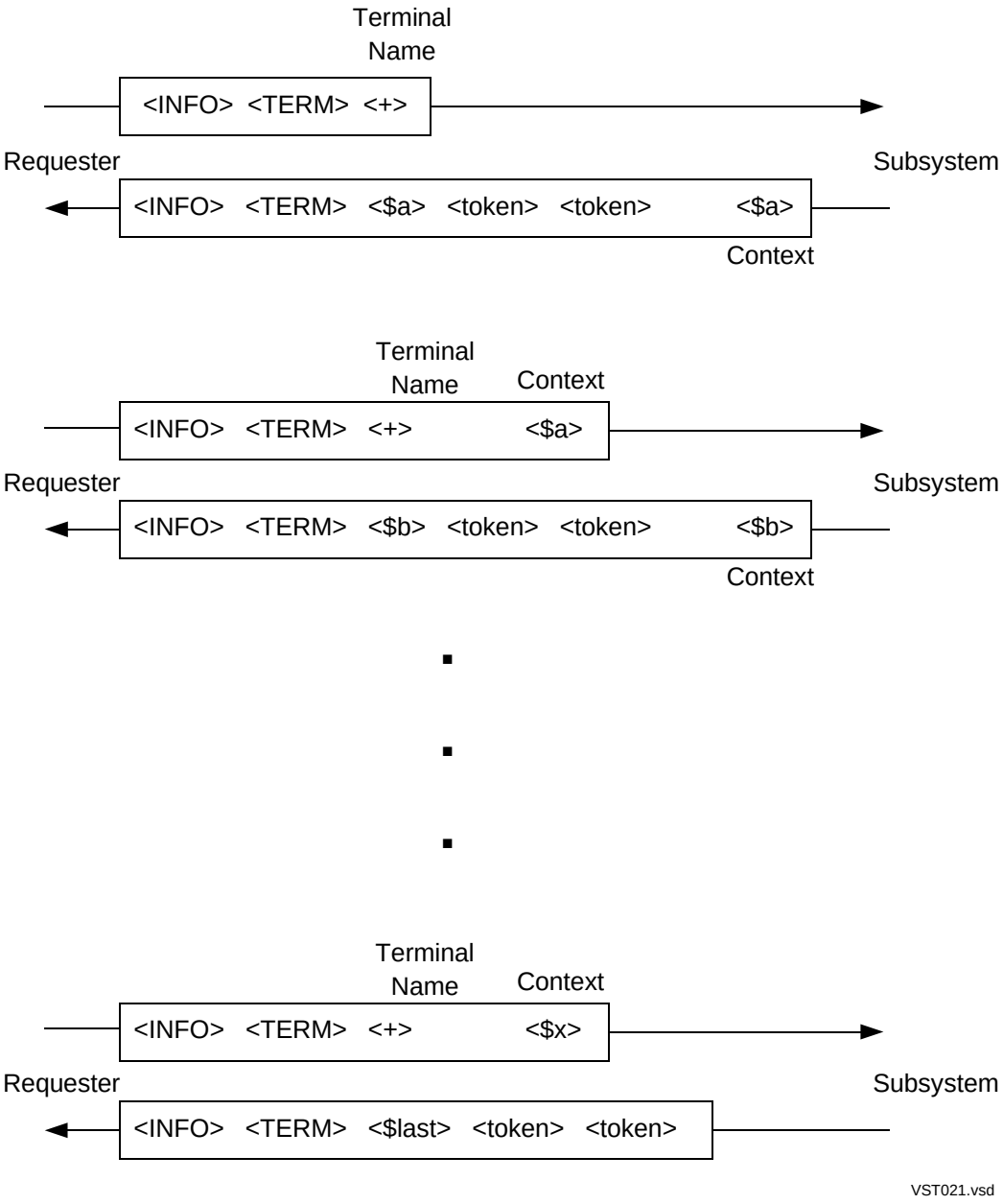
- The command and parameters are consistent with the context supplied.
- The requester has not bypassed security or integrity constraints by supplying a forged or modified context token.

As an example of the latter case, consider an application that sends a command to start a number of terminals, some of which it is not permitted to start. The context token in this case needs to include a name or identifying number for the next terminal on which the command is to be performed. When the server attempts to start a terminal that the application is not permitted to start—say, TERM3—it might return an error list indicating a security error and a context token indicating that the next terminal to start is TERM4. The application might then attempt to circumvent this restriction by changing the terminal name in the context token from TERM4 to TERM3. To prevent such a breach of security, the server should apply security checks on each new command.

Another example of the latter case is an ALTER LIKE command (to alter the characteristics of an object to be the same as those of another object named in the command) implemented so that the set of values being altered is carried in the context token. In this situation, the server should protect the integrity of the context by checking these values for validity each time the context token is returned in a new command.

Figure 5-1 illustrates the exchange of messages for a command requesting information about all terminals in an application (INFO TERM *):

Figure 5-1. Response Continuation for a Typical Information Command



Because the terminal name is a required part of the context, it is included in the context token even though it is already present in the TERMNAME token. This way, the requester does not need special-case knowledge about which tokens must be provided to continue the command. The requester always extracts the context token and supplies it with the original command parameters.

If the server detects something wrong with the context or a mismatch between the context and the new command, it should return a response record containing an error number that identifies the problem, at least to the extent that the problem concerns the

context. If desired, the server can also provide more detailed information about the error.

A server is not required to accept a context token from a different version of the server. Requesters cannot expect to save a context token from some session with a server and use it with another session sometime later. The server is not required to detect or reject such long-term saving of context tokens.

When the server receives a command with context, it might find that there are no more objects to process. (Perhaps the last one was deleted between the time the last response was composed and the time the continuation arrived.) To cover this case, the server must be able to build a response record that cannot be mistaken for a response record about a particular object. The server should return a response record whose return token contains the error number for an empty response record.

Context Sensitivity

Subsystems you write usually should be context-free servers. However, under some circumstances you might need to write your subsystem as a context-sensitive server.

A context-sensitive server should use the context token for response continuation just as a context-free server does, so that programs can use the same continuation mechanism with all subsystems. In this case, the context token can contain a dummy value—only its presence or absence is important.

If a requester just stops sending commands, even though the last response had a context token, a context-sensitive server might have resources tied up that are no longer needed. Such a server might provide a null command that the application could send to let the server know that it no longer is interested in continuing the command in progress. Alternatively, the server could use the receipt of any command not containing a context token as the signal that the application is no longer interested in continuing the command in progress. The latter design would prevent the application from issuing commands to recover from errors or to implement higher-level constructs, so you should consider carefully whether such restrictions are necessary.

A context-sensitive server might have to make some other departures from the guidelines given here. However, you should find ways to keep the differences as few as possible, to maximize the consistency between subsystems.

Determining How Many Response Records Fit in a Buffer

The guidelines for continuation say that for a command that causes any change to an object, the server should carry out the command only on objects for which responses can be returned in the response message immediately following the action on the objects.

Inquiry commands do not change an object, so they are not affected by this rule. If your server runs out of buffer space while building an inquiry response, it can flush the partial response from the buffer, construct a context token as if it never started on the object whose response overflowed the buffer, and send that response to the requester.

Assume that the server builds the response record in the buffer during the course of executing the command on each object, and assume that the server knows a definite limit on the size of every possible response record. Before starting execution of the command on the next object, the server must determine whether the space left in the buffer is sufficient for the largest possible response record plus the largest possible context token. If there is not enough space, the server should construct a context token, put it in the buffer, and send the response.

The situation is more complex if the server cannot put a definite limit on the size of a response record. One way the server could proceed is:

- Before starting the operation on the next object, perform a test as in the previous case, using the expected length of the response record. If it there appears to be enough space, begin working on the next object.
- While performing the command, before adding each error list or regular response token to the response, record the current position in the buffer.
- After adding the token or error list, determine whether the length remaining in the buffer is enough to hold a minimal error list, a return token, an end-list token, and a context token.
  - If there is enough space, continue processing.
  - If there is not enough space, reposition to the last saved position; flush the data; insert the minimal error list; insert the return token, the end-list token, and the context token; and send the response.

The minimal error list should contain a subsystem-defined error number indicating that the response was truncated because it was larger than estimated, and the object name whose response overflowed. This approach sacrifices some information if it runs out of space, but always lets the requester know approximately what happened.

Here is a pseudocode example to show how this approach might be implemented. The lines marked with "*" are the ones needed to handle the case of an unexpectedly long response. The other lines would be needed in any case.

```
cmd-xxxxx:     ! beginning of code for a particular command
   response-need := estimated max size of response
                    + size of this command's context;
   overflow-need := size of overflow error list           *
                    + size of ZSPI-TKN-RETCODE            *
                    + size of ZSPI-TKN-ENDLIST            *
                    + size of this command's context;     *
   num-responses := 0;
   multiple-responses := MAXRESP <> 0;
   if MAXRESP = 0 then MAXRESP := 1;
   and other setup needed;
next-obj-loop:
   current-retcode := 0;
   find next object;
   if no next object then go to done;
   if ZSPI-TKN-BUFLEN - ZSPI-TKN-USEDLEN < response-need
```

```
                              then go to continue;
   if MAXRESP > -1 then
        if num-responses >= MAXRESP then go to continue;
   if multiple-responses then
            SSPUT(ZSPI-TKN-DATALIST);  ! start response
SSPUT(subsys-TKN-xxxxx,object name);
processing-loop:
   do some work;
   SSGET(ZSPI-TKN-LASTPOSITION,remember-place);           *
   insert next token or error list into response;
   if buffer is full then go to backout;                  *
   if ZSPI-TKN-BUFLEN - ZSPI-TKN-USEDLEN < overflow-need  *
                              then go to backout;         *
   if more to do then go to processing-loop;
   go to response-done;
backout:                                                  *
   SSPUT(ZSPI-TKN-POSITION,remember-place);               *
   SSGET(ZSPI-TKN-NEXTTOKEN,dummy);                       *
   SSPUT(ZSPI-TKN-DATAFLUSH);                             *
   SSPUT(ZSPI-TKN-ERRLIST);                               *
   SSPUT(ZSPI-TKN-ERROR, overflow error number);          *
   SSPUT(subsys-TKN-xxxxx, object-name);                  *
   SSPUT(ZSPI-TKN-ENDLIST);                               *
response-done:
   SSPUT(ZSPI-TKN-RETCODE,current-retcode);
   if multiple-responses then
          SSPUT(ZSPI-TKN-ENDLIST);
   num-responses := num-responses + 1;
   go to next-obj-loop;
continue:
   SSPUT(ZSPI-TKN-CONTEXT,context);
   done:
      send reply;
```

If the server needs to decide the number of response records that fit in a buffer before starting to fill the buffer, the code must be a bit different, but the same idea applies. The server would divide the buffer size by the maximum expected size for a response record to get the number of response records that should fit. Then the test in the processing loop would check to see whether the response record has run past its allowed ending point in the buffer, by comparing ZSPI-TKN-USEDLEN to the response number times the size allowed for each response.

Consistency Between Response Records in Different Replies

When information about various parts of an object is returned in different responses, there is a potential for presenting an inconsistent picture of the object to the requester because the object might change between commands. If you encounter a case where this situation might be a problem (not all cases present a problem), you can:

- Implement some form of locking. A problem with locking is knowing when to release the lock if the requester fails to do so.

- Make the server context-sensitive and hold a snapshot of the object's current state in the server's memory.
- Take a snapshot of the object, store it in a file (called a "snapshot file"), and use the context token to keep track of where the snapshot is stored. This solution can be implemented in a context-free server, but the server needs to open and close the file on each command, because there is no guarantee each command would be sent to the same server process if there are several server processes.

  A variation on this solution is to have the requester create a snapshot file into which the server writes the information in an externally defined format. Then the requester can read the information from the file.

In many cases, the consequences of inconsistency are not serious enough to warrant making the extra effort involved in any of the approaches just described. But for those cases in which consistency is important, one of these approaches probably will work.

Checking the Context Token

Your server should check the context token for consistency with the command. The nature and extent of the tests should depend on how your server is organized and what problems it could encounter if the context is invalid in some way. For instance, it might be helpful to put the command number and object-type number in the context and determine whether they match those of the new command. If the context contains table indexes, be sure to check them against the table sizes before using them.

Take into account that a requester might send a context returned by a different release of the server. It is not imperative that your server detect this case and return an error, provided it does not compromise the server to accept the context from a different version of the server. However, it is recommended that your server detect and reject this case.

To protect against a forged context token being used to bypass security, the server must treat the contents of the context token with as much caution as it treats command parameters. The server must not use the context token to carry any kind of user identification or access rights that it determined while processing the previous part of the command, unless it can verify that the requester has not tampered with that information. A simple approach is to have the server do any security tests from scratch on each command. If the tests are time-consuming, you might prefer to find a way to use the context to reduce the work on subsequent commands.

Reporting Errors

SPI servers should report errors as described in Section 2, SPI Concepts and Protocol. Remember to define the tokens you return in responses and in error lists. If desired, your subsystem can also define special lists using the generic-list token type, ZSPI-TYP-LIST, with a subsystem-supplied token number.

In addition, if your subsystem encounters errors in calls to software for the NonStop system facilities that do not have a programmatic command interface based on SPI but

do define standard error lists (for instance, some system procedures such as NEWPROCESS), and if your subsystem encounters critical errors in calls to the SPI procedures, HP recommends that you return error lists to applications in the standard form, and use the appropriate HP definition files for the tokens in these error lists rather than defining your own.

SPI servers should report these error conditions:

- Empty response
- Command too long
- Bad context
- Response longer than expected
- Wrong subsystem ID in command
- Command requires newer version of server
- Extraneous token in a command
- Error from another subsystem or from SPI (if desired, your subsystem can define different error numbers for different sources of the error)
- Required token is missing (you can define either a single error number for all cases or many individual error numbers for the different tokens)
- Command parameter has an invalid value (you can define either a single error number for all cases or many individual error numbers for the different parameters)

Control of Types of Response Records

To let requesters ask for either all response records or only those with errors or warnings, include support for the ZSPI-TKN-RESPONSE-TYPE token, as some NonStop Kernel subsystems do.

Continuing Despite Errors

To let requesters specify under what conditions your server should continue processing on a set of objects, you can support the ZSPI-TKN-ALLOW-TYPE token as some NonStop Kernel subsystems do. For guidelines governing the use of this token, see ZSPI-TKN-ALLOW-TYPE on page 4-31.

Reporting Errors From the SPI Procedures

These recommendations for all NonStop Kernel subsystems describe how a server should report errors from the SPI procedures.

Failure of SSGET on a Header Token

If SSGET fails when attempting to get any header token from a command, your subsystem should return an error response to report the problem. The return token of this response should be the one your subsystem defines to indicate that an SPI error occurred. The associated error list should contain the same error number and a nested error list constructed according to the guidelines given in the *Guardian Procedure Errors and Messages Manual*. The command number and object type can be null or can be the values from the command.

Some suggestions:

- Initialize a new buffer for a response (do not use ZSPI-TKN-DATAFLUSH on the command, because the command buffer might be the source of the problem). If the command number or object type has been retrieved from the command, your server can use them in the SSINIT call. Alternatively, for simplicity, your server can use the null values (ZSPI-VAL-NULL-COMMAND and ZSPI-VAL-NULL-OBJECT-TYPE) whether or not they have been retrieved from the command.
- Do not attempt to determine the value of ZSPI-TKN-MAXRESP; in this situation, your server probably cannot count on this value to be accurate. Return a single response record per response.
- Check that the buffer used for this error response is large enough. Then, if any errors occur from the procedures called in creating the response, handle them as described in Failure of SSINIT When Initializing a Response Buffer or Failure of SSPUT When Building a Response on page 5-49.

Failure of SSGET on a Nonheader Token

If SSGET fails when getting a nonheader token from a command, the action depends on the error:

- If the error is “token not found” and the token is optional for the command, take the default action for that token.
- If the error is “token not found” and the token is required for the command, the response should contain the return-token value your subsystem defines for a missing required token, and an error list that identifies the token in question.
- If the error is any other error, the error response should follow the same guidelines as for Failure of SSGET on a Header Token on page 5-48.

Some suggestions:

- Initialize and build a response buffer as described in Failure of SSGET on a Header Token on page 5-48.
- In this situation, it is not imperative that you ignore ZSPI-TKN-MAXRESP. However, for simplicity, HP recommends that you return a single response record per response.

- As for the failure of SSGET on a header token, check that the buffer for this error response is large enough, and handle any errors from the procedures involved in creating the response.

Failure of SSINIT When Initializing a Response Buffer

If SSINIT fails for any reason except “buffer full,” a serious error has occurred. If your buffers are all of adequate size, you should consider a “buffer full” error to be a serious error, too.

If your server process is one that should stop on internal errors to limit data corruption, have the process stop if it gets an error from SSINIT.

If your process is one that does not need to stop to limit data corruption, your server can either stop or attempt to report the SSINIT failure.

If you report a failure of SSINIT, your server should return an error response to report the problem (see the guidelines for Failure of SSGET on a Header Token on page 5-48.

Some suggestions:

- HP recommends that your subsystem stop immediately instead of attempting to continue.
- If you do attempt to continue and report the failure, initialize and build a buffer for a response (see Failure of SSGET on a Header Token on page 5-48).
- It is not imperative that you ignore ZSPI-TKN-MAXRESP. However, for simplicity, it is recommended that you return a single response record per response.
- If errors occur while attempting to build the response, your server can ignore them and return whatever gets built as the response. This might result in a garbage response, but in this situation it is unlikely that any other method would provide more information.

Failure of SSPUT When Building a Response

If your server builds a response containing multiple response records and relies on getting the “buffer full” error to determine when it is at the end of the buffer, it should perform whatever corrective action is necessary when the buffer is full.

In all other cases, an SSPUT error is a serious failure. If your process is one that should stop on internal errors to limit data corruption, have the process stop if it gets an error from SSPUT.

If your process is one that does not need to stop to limit data corruption, your server can either stop or attempt to report the SSPUT failure.

If you do report a failure of SSPUT, your server should return an error response to report the problem, (see the guidelines for Failure of SSGET on a Header Token on page 5-48).

Some suggestions:

- Your subsystem should stop immediately instead of attempting to continue.
- If you do attempt to continue and report the failure, initialize and build a buffer for a response (see Failure of SSGET on a Header Token on page 5-48).
- It is not imperative that you ignore ZSPI-TKN-MAXRESP. However, for simplicity, it is recommended that you return a single response record per response.
- If errors occur while attempting to build the response, your server can ignore them and return whatever gets built as the response. This might result in a garbage response, but in this situation it is unlikely that any other method would provide more information.

It is assumed that when formatting a response in single-response-per-buffer form, your subsystem initializes a buffer at least as large as the subsystem's recommended size. A "buffer full" error in that situation should be considered a serious error. If your subsystem attempts to use the buffer length specified in the command message and encounters a "buffer full" error, the corrective action is up to you to define.

The technique of building the response in the command buffer by using SSPUT with ZSPI-TKN-DATAFLUSH to empty it is probably not a good idea. A problem in the command buffer might cause a later SSPUT to fail, and certain garbage requests could cause the server to stop. It is probably best to initialize a new buffer.

Failure of SSNULL

An error from SSNULL should be considered as serious as the errors from SSINIT or SSPUT; it probably indicates the data stack has been corrupted, so it should cause the server to stop.

If you do attempt to generate a response, proceed as in the case of SSINIT or SSPUT.

Error on SSMOVE

When using SSMOVE on a server-initialized buffer, you need to clear the last-error information in at least the server-initialized buffer just before the operation (use SSPUT with ZSPI-TKN-CLEARERR).

If an error occurs on SSMOVE, you can use SSGET with ZSPI-TKN-LASTERR on the server-initialized buffer to determine whether the error was on the server-initialized buffer. If it was, the error indicates data stack corruption of the server, and the server should stop.

If you do attempt to generate a response, proceed as in the case of SSINIT or SSPUT.

Failure of SSGET With ZSPI-TKN-USEDLEN

If after completing a response message, the server calls SSGET with ZSPI-TKN-USEDLEN to determine the count to use when sending the response, and SSGET fails, the server's data stack is corrupt and the server should stop.

If you do attempt to generate a response, proceed as in the case of SSINIT or SSPUT.

Pass-Through Error Lists

These guidelines govern the contents and construction of pass-through error lists from NonStop Kernel subsystems provided by HP. Your own subsystems should also follow these guidelines. For examples of pass-through error lists, see Sample Error Responses on page 2-51.

Assume that a requester A issued a command to a server B and that B is creating an error list in its response to A. Assume that B uses another server or library procedure called C. These guidelines govern the design of error lists:

1. An error list begins with the token ZSPI-TKN-ERRLIST and ends with the token ZSPI-TKN-ENDLIST.
2. Except for nested error lists, all the tokens in an error list are considered to be qualified by B (including the ERRLIST token at the beginning and the ENDLIST token at the end).
3. An error list must contain a ZSPI^TKN^ERROR token with the value being one of the errors defined by B, not an error number returned from C.
4. An error list must contain a token defined by B whose value is the name of the object to which the command to B was directed. If the command references no objects, the error list need not contain an object name.

   If the command is directed to an object that has no name and the command includes an object name of some null value such as blanks, the error list must contain that null object name. If the command is directed to an object that has no name and the command does not include any form of object name, the error list need not contain an object name.
5. An error list must contain any additional items necessary to describe what action B was asked to perform. If the error number is not sufficient to determine the command number and object type number in A's command to B, they must be included in the error list.

   There should not be a large amount of additional information—be guided by what is appropriate to be included in an error message. Except as directed in guideline 8, do not include information to describe the command or response from C—that is the job of C's error list.
6. If the error detected by B indicates some inappropriate condition (regardless of whether some of the information used by the logic came from C), the error list must

contain those items needed to tell what inappropriate condition was detected and any additional values needed to understand the error.

For example, if a command asks to start a subdevice when the line is not started and the name of the line is inherent in the name of the subdevice, no additional information need be included. But if the command asks to start an object and fails because an attribute of the configuration of the object conflicts with some information obtained from the device, the item of information obtained from the device itself would be valuable in understanding the problem and should be included in the error list.

7. If a description of an error from C is relevant in reporting the error, and if C is accessed through an SPI interface, the error list B creates must contain a copy of all the error lists in the error response returned by C.

8. If a description of an error from C is relevant in reporting the error, and if C is not accessed through an SPI interface, B should follow the directions C provides for constructing an error list to report errors from C.

   Although some details might vary, C defines a token for each item of information that is important for understanding what C was asked to do and what went wrong. B forms an error list on behalf of C and nests this error list within the error list B creates. The ZSPI-TKN-ERROR token in this error list is generally the error status returned from C. The values of the other tokens are generally the values of the arguments passed to C if C is a procedure or are similar parts of the command for other styles of interface. If C defines a token for a parameter B did not use, usually B omits that token when forming the error list. In some cases, the actual argument passed to C might not be meaningful to another process (such as a file number), and in such cases, the directions supplied by C explain what B must do to obtain an appropriate representation.

   When forming an error list on behalf of C, B should make the version portion of C's subsystem ID in the ZSPI-TKN-ERROR token be zero. This is to indicate that because the error was not constructed by C itself, the actual version of C cannot be determined from the error list.

   In all cases, the error number in B's level of the error list is one defined by B. The error number might mean simply "I got an error from another subsystem." It might be specific to a subsystem: "I got an error from subsystem X." Or it might have a meaning in B's terms and the error list from C is just additional information relevant to that error. The error number in B's level never describes C's error directly.

9. If the error is a problem in the formation of the command to B, such as a missing required token, an extra or unrecognized token, a too-high server version in a command, an unrecognized command number, an unrecognized object type number, or an invalid value in a field of the command, then guidelines 4 through 8 do not apply.

   The error list must contain B's error code for the problem detected and identification of the token and field causing the problem (except for server version too high). Include the ZSPI-TKN-PARM-ERR token to describe the token and field.

Fill in only the Z-TOKENCODE field to describe a missing required token. If more than one field is involved in the error (as in the case of a field that conflicts with the value of another field), repeat the ZSPI-TKN-PARM-ERR token for each field involved. In addition, if the subsystem ID qualifying the token is not the default subsystem ID, include the ZSPI-TKN-SSID-ERR token to identify that SSID.

These guidelines apply to error lists at every level. That is, when C creates an error list to pass back to B, the point of view shifts—because C is creating an error list, it takes the role of B in the above guidelines and the guidelines all apply to C. The principle applies similarly for any depth of subordinate servers.

Errors from SPI procedures (assuming the program does not recover from them) are reported according to guideline 8. For guidelines for constructing error lists to report such errors, see the *Guardian Procedure Errors and Messages Manual*.

When relaying pass-through error information from multiple subsystems, you can use the SSMOVE procedure to move an entire error list from one SPI buffer to another.

Guideline 8 refers to token definitions and directions for their use that are provided by procedures and other services that do not have SPI interfaces. If you wish to provide such tokens and directions, you must decide what items of information are useful to report in a description of an error from your service and define how each is to be included in the error list. The names of the tokens and other definitions must follow the same guidelines as the definitions for a complete SPI interface, but only those few declarations needed to build the error lists need be supplied. The declarations are packaged the same way as for a complete SPI interface.

When defining guidelines for handling error lists to report errors from servers with non-SPI interfaces, consider:

- Certainly the error number is of interest; your subsystem should place it in the ZSPI-TKN-ERROR token unless there is good reason to put some other value there.
- For procedures, your server should examine each argument to the procedure. You should define tokens for those arguments that would be useful in an error message or for other analysis of the error. If an argument is a structure, it might be appropriate to define tokens for just some of the fields of the structure, or it might be appropriate to include the whole structure in a token. Decide that on a case-by-case basis.
- In some cases, there might be important information that is not among the parameters. Define tokens for such information and explain how to obtain the information to put in those tokens if it is not obvious.
- In some cases, the primary information might not be in a form usable by another process. A file number in a file-system WRITE call is a good example. In these cases, give directions telling how to transform what is available into what is useful and define a token for the result of the transformation, not for the starting values.
- Be sure to tell what to do for a token corresponding to an optional argument to the procedure in the case that argument is not used in the call that encountered the

error. Omitting the token from the error list seems like a reasonable approach in that case, but if that is not suitable, include explicit directions for what to do.

- If a set of procedures are closely related (such as the HP file-system procedures), such procedures can share a subsystem ID. In these cases, you should use the token ZSPI-TKN-PROC-ERR token to indicate which of the procedures is involved. Define a token value to denote each procedure in the group.
- Instead of defining separate tokens for each item of interest, you can choose to group some or all of the items into one or more extensible structured tokens.

Summary of Server Role

An SPI server must:

- Verify, upon receipt of an SPI message, that the message is a valid SPI message and that the message fits in the message buffer. (See Checking the Command Message for Validity on page 5-31.)
- Reject any request in which ZSPI-TKN-MAX-FIELD-VERSION contains a version greater than the server's version.
- Reject any request that contains an unrecognized or invalid subsystem ID, command number, object type, token code, or token value.
- Set ZSPI-TKN-SERVER-VERSION to its own (the server's) version.
- Continue to recognize and process old-version requests and tokens for as long as possible.
- Accept but otherwise ignore any number of comment tokens (ZSPI-TKN-COMMENT).
- Check continuation requests to verify that the requester is not bypassing security or forging a context token.
- Support opens from a backup requester.
- If a request is not safely repeatable, save sync IDs, replies, and any other necessary information.
- Always use a token map to retrieve the contents of an extensible structured token.

6 SPI Programming in C

This section provides language-specific information for the programmer who is using C to write an SPI requester or server:

Definition Names in C

Symbolic names in this section are in the C form, using underscore (_) symbols rather than hyphens. For example, the DDL token code ZSPI-TKN-RETCODE is expressed as ZSPI_TKN_RETCODE in C.

C Definition Files

Each C module of your application that uses the SPI must begin with **#include** preprocessor directives to include the C versions of the SPI standard definitions and the definitions for all subsystems with which your program communicates. The C version of the SPI standard definitions is in the file named ZSPIDEF.ZSPIC on the disk volume chosen by your site. For NonStop Kernel subsystems, the C versions of the subsystem definitions have file names of the form ZSPIDEF.*subsys*C, where *subsys* is the 4-character subsystem abbreviation given in Appendix D, NonStop Kernel Subsystem Numbers and Abbreviations.

You must write **#include** directives to read these definition files:

```
ZSPIDEF.ZSPIC       SPI standard definitions

ZSPIDEF.ZEMSC       EMS standard definitions

ZSPIDEF.ZCOMC       Extended SPI standard definitions
ZSPIDEF.ZCMKC
ZSPIDEF.ZCDGC

ZSPIDEF.subsysC     Standard definitions for each subsystem
  .                 your program communicates with
  .
  .
ZSPIDEF.subsysC
```

In these definition files, the structures are in lowercase (for example, zspi-ddl-ssid-def) and the defines are in uppercase (for example, ZSPI-TKN-SSID).

The **#include** directive for ZSPIDEF.ZSPIC must appear first. Any **#include** directives for files that contain your own declarations must come after the **#include** directives for the standard files.

The DDL compiler combines the items of a DDL REDEFINES clause into a union. For more information, see the *Data Definition Language (DDL) Reference Manual*.

Declarations Needed in C Programs

In addition to the declarations provided in the definition files, you must add these declarations to your C programs.

SPI Buffer

The ZSPIDEF.*subsys*C definition file for each NonStop Kernel subsystem includes a buffer declaration named *subsys*_**ddl_msg_buffer_def**. Use this declaration to allocate a buffer variable of the recommended size (*subsys*_**val_buflen**):

```
zpwy_ddl_msg_buffer_def  spibuf;
```

This declaration is for the Pathway subsystem. It lets you refer to the **z_msgcode** (-28), **z_buflen**, and **z_occurs** fields of this structure.

Some subsystems provide additional buffer declarations allocating different recommended buffer-size values for different commands. For details, see the individual subsystem management programming manual.

If you wish to define a buffer larger than the recommended size to handle a large number of response records in a reply, you can write your own buffer declaration, following the pattern of *subsys*_**ddl_msg_buffer_def**.

Subsystem ID

You must initialize the subsystem ID before the first time you call SSINIT. For NonStop Kernel subsystems, the name of the subsystem ID structure in the C definition file is *subsys*_**val_ssid_def**. To initialize it, use these values:

```
ZSPI_VAL_TANDEM for the z_owner field
ZSPI_SSN_subsys for the z_number field
ZSPI_VAL_VERSION for the z_version field
```

C provides two ways to initialize this structure.

The first way (for the OSI/AS subsystem, for example):

```
zosi_val_ssid_def  zosi_val_ssid ={ ZSPI_VAL_TANDEM,
                                    ZSPI_SSN_ZOSI,
                                    ZOSI_VAL_VERSION };
```

Performing the initialization this way causes compiler Warning 74: initializer data truncated. You can ignore the warning; the initialization is performed correctly.

The second way to initialize the structure:

```
#include stringh

strcpy ( zosi_val_ssid.z_owner, ZSPI_VAL_TANDEM );
strcpy ( zosi_val_ssid.z_number, ZSPI_SSN_ZOSI );
strcpy ( zosi_val_ssid.z_version, ZOSI_VAL_VERSION );
```

The compiler does not issue a warning if you use this initialization method.

You must enter the names of the values you place in the structure in uppercase characters.

Passing Tokens by Value

C applications can pass tokens as parameters to the SPI procedures by value only; passing parameters by reference is not allowed. To allow access to a token map, which represents an extensible structured token, declare and pass a pointer to the structure.

Use SSGET, SSMOVE, and SSPUT to pass pointers to token maps. Use SSGETTKN, SSMOVETKN, and SSPUTTKN to pass token codes.

C Types

The C compiler does not support 64-bit integer values. For information about how to handle these values in C programs, see the *C/C++ Programmer's Guide*.

Interprocess Communication

Requesters written in C can use the **tal** interface declaration to call the WRITEREAD file-system procedure. C programs can also use the alternate-model I/O routines that are described in the *C/C++ Programmer's Guide*.

Writing a Server in C

If you are writing a server, you must include **#include** preprocessor directives for all the definition files you use, as you would when writing a requester. You must also declare the SPI buffer and initialize the subsystem ID.

Your subsystem must open $RECEIVE for I/O to receive SPI messages. Use the file-system procedures READUPDATE and REPLY to read and answer the messages.

SPI Procedure Syntax in C

To call SPI procedures from a C program, you use the **tal** interface declaration, just as you would when calling other operating system procedures. For descriptions of the SPI procedures and their parameter, see Section 3, The SPI Procedures.

SSINIT

```
#include <cextdecs(SSINIT)>

short SSINIT (   short *buffer
               ,  short buffer-length
               ,  short *ssid
               ,  short header-type
               ,  short command
               ,[ short object-type       ]
               ,[ short max-resp          ]
               ,[ short server-version    ]
               ,[ short checksum          ]
               ,[ short max-field-version ]  );
```

SSNULL

```
#include <cextdecs(SSNULL)>

short SSNULL (   short *token-map
               ,  char *struct
     ,[ long long *constants  ]   );
```

SSPUT and SSPUTTKN

```
#include <cextdecs(SSPUT)>

short SSPUT (   short *buffer
             ,  short *token-id
             ,[ char *token-value  ]
             ,[ short count        ]
             ,[ short *ssid        ]  );


#include <cextdecs(SSPUTTKN)>

short SSPUTTKN (   short *buffer
                ,  long token-id
                ,[ char *token-value  ]
                ,[ short count        ]
                ,[ short *ssid        ]  );
```

SSGET and SSGETTKN

```
#include <cextdecs(SSGET)>

short SSGET (   short *buffer
             ,  short *token-id
             ,[ char *token-value  ]
             ,[ short index        ]
             ,[ short *count       ]
             ,[ short *ssid        ]  );


#include <cextdecs(SSGETTKN)>

short SSGETTKN (   short *buffer
                ,  long token-id
                ,[ char *token-value  ]
                ,[ short index        ]
                ,[ short *count       ]
                ,[ short *ssid        ]  );
```

SSMOVE and SSMOVETKN

```
#include <cextdecs(SSMOVE)>

short SSMOVE (   short *token-id
              ,  short *source-buffer
              ,[ short source-index     ]
              ,  short *dest-buffer
              ,[ short dest-index       ]
              ,[ short *count           ]
              ,[ short *ssid            ]  );


#include <cextdecs(SSMOVETKN)>

short SSMOVETKN (   long  token-id
                 ,  short *source-buffer
                 ,[ short source-index     ]
                 ,  short *dest-buffer
                 ,[ short dest-index       ]
                 ,[ short *count           ]
                 ,[ short *ssid            ]  );
```

Examples

For example source code programs written in C, see Appendix E, SPI Programming Examples.

7 SPI Programming in COBOL

This section provides language-specific information for the programmer who is using COBOL to write an SPI requester or server. (COBOL74 does not support SPI.)

Definition Names in COBOL

Symbolic names in this section are in the COBOL form using hyphens. For example, the DDL token code ZSPI-TKN-RETCODE is expressed the same way in COBOL.

COBOL Definition Files

COBOL applications include COPY statements to copy in each section of the COBOL definition files (copy libraries) that they use, including the ZSPI definition file and the definition files for all subsystems with which the programs communicate.

The COBOL version of the SPI standard definitions is in a file named ZSPIDEF.ZSPICOB on the disk volume chosen by your site. For NonStop Kernel subsystems, the COBOL versions of the subsystem definitions have file names of the form ZSPIDEF.*subsys*COB, where *subsys* is the 4-character subsystem abbreviation. For the individual section names, see the listings of these files. You can include these COPY statements in any order.

For example, if your application sends the CONTROL PM command to Pathway, it should include COPY statements similar to:

```
EXTENDED-STORAGE SECTION.
   copy CONSTANTS                of $SYSTEM.ZSPIDEF.ZSPICOB.
   copy ZPWY-DDL-PAR-CONTROL-PM of $SYSTEM.ZSPIDEF.ZPWYCOB.
   copy CONSTANTS                of $SYSTEM.ZSPIDEF.ZPWYCOB.
```

Declarations Needed in COBOL Programs

In addition to the declarations provided in the definition files, you must add these declarations to your COBOL programs.

SPI Buffer

In the Data Division of your program, you must set up the file for requester-server communication. Use a file description (FD) entry similar to:

```
FD  SERVER-FILE            LABEL RECORDS ARE OMITTED.
```

Immediately following this statement, include a COPY statement for the SPI buffer declaration. The ZSPIDEF.*subsys*COB definition file for each NonStop Kernel subsystem includes a buffer declaration named *subsys*-DDL-MSG-BUFFER, which has the structure described in SPI Message Buffer on page 2-13. To allocate a buffer of the recommended size, your COPY statement should be similar to (this example is for sending commands to Pathway):

```
copy ZPWY-DDL-MSG-BUFFER     of $SYSTEM.ZSPIDEF.ZSPICOB.
```

Then you can easily refer to the Z-MSGCODE (-28), Z-BUFLEN, and Z-OCCURS fields of this structure as needed.

If you wish to define a buffer larger than the recommended size in order to handle a large number of response records per response message, you can write your own buffer declaration, following the pattern of *subsys*-DDL-MSG-BUFFER.

Interpreting Boolean Values

Tokens and token fields of token type ZSPI-TYP-BOOLEAN (or other token types based on ZSPI-DDL-BOOLEAN) become type PIC XX in COBOL. Therefore, COBOL does not support the named values ZSPI-VAL-TRUE and ZSPI-VAL-FALSE.

To interpret the values of Boolean tokens and fields, it is recommended that your programs define PIC XX variables or fields in working storage, use VALUE clauses to initialize them to HIGH-VALUES (true) or LOW-VALUES (false), and then reference the PIC XX fields when the true or false values are needed in operations with fields of type ZSPI-TYP-BOOLEAN.

For example, consider the ZTMF field of ZPWY-DDL-DEF-PROG. A COBOL program could set or test this field as follows:

```
    .
    .
WORKING-STORAGE.
01  COBOL-VAL-TRUE     PIC XX VALUE HIGH-VALUES.
01  COBOL-VAL-FALSE    PIC XX VALUE LOW-VALUES.
    .
    .
PROCEDURE DIVISION
```

```
.
.
MOVE COBOL-VAL-TRUE TO ZTMF OF ZPWY-DDL-DEF-PROG.

    or

IF ZTMF OF ZPWY-DDL-DEF-PROG = COBOL-VAL-FALSE . . .

    or

IF ZTMF OF ZPWY-DDL-DEF-PROG NOT = COBOL-VAL-FALSE . . .
.
.
```

It is recommended that you use the comparison "NOT = *false-value*" rather than "= *true-value*," in case a subsystem uses a value other than -1 for TRUE.

It is a good idea to put the declarations of COBOL-VALUE-TRUE and COBOL-VALUE-FALSE into a COPY library and copy it into each program that needs the definitions, in case future versions of HP software change the data type used in COBOL for Boolean fields.

Interprocess Communication

These considerations apply to communication with subsystem servers using SPI messages.

Selecting the External File

In the Environment Division of your program, you must use a SELECT clause to identify the external file to which the server file is connected. If your server has a fixed process name, use a SELECT clause similar to:

```
SELECT SERVER-FILE ASSIGN TO "$TRPM".
```

If your server does not have a fixed name, include a SELECT clause such as:

```
SELECT SERVER-FILE ASSIGN TO "#DYNAMIC".
```

Then (for a non-fixed-name server) use the COBOLASSIGN utility routine to assign the process name at run time.

Starting the Server

To start any servers that your application starts dynamically, use the CREATEPROCESS utility routine.

If a server that you start dynamically honors assign and param messages, and you want to allow the user of your application to provide these messages to the server, you should compile your application with the ?SAVEALL directive. Doing this ensures that assign and param messages are saved and passed on.

Communicating With the Server

Except in the case of the EMS collector process ($0), use the OPEN verb to open the server, and use the READ WITH PROMPT verb to send commands to it.

To open the EMS collector process to send commands to it or to report events, you must use the utility routine COBOL^SPECIAL^OPEN. To open the collector to send commands to it, specify “$0.#ZSPI” in the SELECT clause. To open the EMS collector to send event messages to it, specify “$0” in the SELECT clause. For more information about COBOL^SPECIAL^OPEN, see the *COBOL85 Reference Manual*.

After you have opened $0.#ZSPI using COBOL^SPECIAL^OPEN, you send commands to it using READ WITH PROMPT, just as for any other server process. However, after opening $0 using COBOL^SPECIAL^OPEN, you send event messages to it using the WRITE verb.

Writing a Server in COBOL

If you are writing a subsystem server, you should include COPY statements for the sections of the SPI definition files (and any other HP definitions) you use, as you would in writing a requester. You must also set up the file (FD) and copy the *subsys*-DDL-MSG-BUFFER definition.

A subsystem should open $RECEIVE for I-O to receive SPI messages and then use the READ verb to read the messages and WRITE to send the replies.

SPI Procedure Syntax in COBOL

To call the SPI procedures from a COBOL program, use the COBOL ENTER TAL feature just as you would to call other operating system procedures. For complete descriptions of the SPI procedures and their parameters, see Section 3, The SPI Procedures.

Note. Always call the procedures SSGET, SSMOVE, and SSPUT for both token codes and token maps in COBOL. SSPUTTKN, SSGETTKN, and SSMOVETKN are not needed or recommended in COBOL programs.

SSINIT

```
ENTER TAL "SSINIT"
   USING   buffer
           buffer-length
           ssid
           header-type
           command
         [ object-type         ]
         [ max-resp            ]
         [ server-version      ]
         [ checksum            ]
         [ max-field-version   ]
   GIVING  status.
```

SSNULL

```
ENTER TAL "SSNULL"
   USING   token-map
           struct
   GIVING  status.
```

SSPUT

```
ENTER TAL "SSPUT"
   USING   buffer
           token-id
         [ token-value  ]
         [ count        ]
         [ ssid         ]
   GIVING  status.
```

SSPUTTKN

Use SSPUT rather than SSPUTTKN in COBOL programs.

SSGET

```
ENTER TAL "SSGET"
   USING   buffer
           token-id
         [ token-value  ]
         [ index        ]
         [ count        ]
         [ ssid         ]
   GIVING  status.
```

SSGETTKN

Use SSGET rather than SSGETTKN in COBOL programs.

SSMOVE

```
ENTER TAL "SSMOVE"
   USING   token-id
           source-buffer
         [ source-index  ]
           dest-buffer
         [ dest-index    ]
         [ count         ]
         [ ssid          ]
   GIVING  status.
```

SSMOVETKN

Use SSMOVE rather than SSMOVETKN in COBOL programs.

Examples

For sample programs using SPI in COBOL, see the *Distributed Name Service (DNS) Management Programming Manual* and the *Pathway/iTS Management Programming Manual.*

8 SPI Programming in TACL

This section provides language-specific information for the programmer who is using the TACL to write an SPI requester or server:

Definition Names in TACL

Symbolic names in this section are in the TACL form, using circumflex (^) symbols rather than hyphens. For example, the DDL token code ZSPI-TKN-RETCODE is expressed as ZSPI^TKN^RETCODE in TACL.

Limitations of TACL for SPI Programming

TACL is an interpreted language, so TACL macros and routines are quick to code and test but usually run more slowly than compiled TAL, COBOL, or C programs. For these reasons, TACL is most useful in prototype applications and in applications for which performance is not an issue. In addition, you should be aware of several other restrictions when deciding whether to implement your application in TACL.

TACL does not let routines run as process pairs. If you require a process to run as a process pair, you must code it in TAL, COBOL, or C.

TACL does not support the programmatic interfaces other than SPI to subsystems such as the spooler, Measure, Sort/FastSort, and Enform. If your application is in TACL and it must communicate with one of these subsystems, it must use the conversational text interface to the subsystem. For instance, it must communicate with the spooler through the text interface to SPOOLCOM or Peruse.

TACL has an absolute maximum I/O buffer size of 4096 bytes, and STRUCTs are also limited to 4096 bytes. In addition, TACL requesters using the #REQUESTER feature cannot do nowait I/O for buffers larger than 239 bytes.

If you are writing a subsystem, TACL does not support the EMS high-level procedures for preparing event messages (EMSINIT, EMSADDSUBJECT, EMSADDTOKENS, and EMSADDBUFFER). Therefore, if your subsystem reports events, you must code the module that prepares the event messages in TAL, C, or COBOL.

TACL Definition Files

TACL macros and routines must load the SPI standard definition file named ZSPIDEF.ZSPITACL on the disk volume chosen by your site, and must also load the definition files in TACL for all subsystems with which your program communicates. For NonStop Kernel subsystems, the names of the definition files in TACL are of the form ZSPIDEF.*subsys*TACL, where *subsys* is the four-character subsystem abbreviation.

To avoid text buffer overflows during loading, load each definition file:

```
PUSH X
#LOAD / LOADED X / $volume.ZSPIDEF.subsysTACL
POP X
```

The LOADED option directs TACL to put the loaded variables into the variable named in the option (here, X) instead of returning them in the expansion to #LOAD; #LOAD then expands to nothing.

Declarations and Data Representations in TACL

TACL processes data in both external format (displayable strings of ASCII characters) and STRUCT format (binary data defined as a STRUCT). TACL provides a set of built-in functions, described later in this section, that correspond to the SPI procedures. Two different built-ins are available for SSGET functionality and two for SSPUT functionality, so that you can pass and return token values in either format.

When you pass parameters to and receive results from these built-ins:

- The SPI buffer must be in STRUCT format.
- The subsystem ID must be in external format when it is passed as the *ssid* parameter to a built-in. When it is passed or returned as a token value, its representation depends on the built-in, as is true for other token values.
- Token codes can be in either external or STRUCT format.
- Token maps (and token codes of token data type ZSPI^TDT^STRUCT) must be in STRUCT format.
- Token values passed to the #SSPUT built-in must be in external format, and token values returned from the #SSGET built-in are always in external format (returned in the expansion of the built-in).
- Token values passed to the #SSPUTV and #SSMOVE built-ins must be in STRUCT format, and token values returned from the #SSGETV and #SSMOVE built-ins are always in STRUCT format.
- Most other parameters—including the count, index, command, object type, maximum responses, server version, and checksum parameters—must be in external format.

- For parameters that require external-format numbers, a numeric variable (a TEXT variable whose contents represent a number) is acceptable.

SPI Buffer

An SPI buffer in TACL is a writable variable level of type STRUCT. The definition of the STRUCT is irrelevant to TACL, except that TACL passes its length to the SSINIT procedure.

The ZSPIDEF.*subsys*TACL definition file for each NonStop Kernel subsystem includes a buffer declaration whose name is *subsys*^DDL^MSG^BUFFER. Use this declaration to allocate a buffer variable of the recommended size (*subsys*^VAL^BUFLEN). Some subsystems can provide additional buffer declarations allocating different recommended buffer-size values for different commands. For details, see the individual subsystem management programming manual.

To define a buffer larger than the recommended size in order to handle a large number of response records per response message, you can write your own buffer declaration. For example, you could declare an SPI buffer:

```
#DEF buf STRUCT BEGIN BYTE b(0:3999); END;
```

Subsystem ID

The external representation of a subsystem ID is an 8-character subsystem owner, a period separator, a subsystem number, a period separator, and a version number. The subsystem owner can contain hyphens but the first character must be alphabetic. All three fields—the subsystem owner, the subsystem number, and the version number—are required.

The subsystem owner must be entered exactly as specified by the subsystem (except that you can omit trailing blanks); TACL does not case-shift it. For NonStop Kernel subsystems, the subsystem owner is “TANDEM~~bb~~”.

For NonStop Kernel subsystems, you can supply the appropriate mnemonic from Appendix D, NonStop Kernel Subsystem Numbers and Abbreviations, in place of the subsystem number. These mnemonics are case-sensitive; all alphabetic characters are in uppercase, and you must enter them in uppercase.

For NonStop Kernel subsystems provided by HP, and for your own subsystems if you use the same version-number format HP uses, you can use the external form for the version—a letter followed by two decimal digits. If you enter the letter in lowercase, TACL upshifts it.

Examples of valid subsystem IDs:

```
TANDEM.EMS.G06
TANDEM.PATHWAY.G06
MYORG.1.2
MYORG.1.A00
```

In the first two examples, you could substitute 18182 for G06; however, G06 is recommended for clarity. The last two examples are possible subsystem IDs for a subsystem you might write. The third example assumes that you define your versions as simple integers. The fourth example assumes that you define them in the same format HP uses.

The null value of the subsystem ID in TACL is 0.0.0.

Your TACL macro or routine must initialize a STRUCT for each subsystem with which your application communicates. For NonStop Kernel subsystems, the name of this STRUCT in the TACL definition file is *subsys*^VAL^SSID. To initialize it, use ZSPI^VAL^TANDEM for the Z^OWNER field, ZSPI^SSN^*subsys* for the Z^NUMBER field, and *subsys*^VAL^VERSION for the subsystem Z^VERSION field. For example, if your application sends commands to TMF, your macro or routine loads the definitions supplied by TMF (in the file ZSPIDEF.ZTMFTACL) and might then include this #SET call:

```
#SET ZTMF^VAL^SSID &
[ZSPI^VAL^TANDEM].[ZSPI^SSN^ZTMF].[ZTMF^VAL^VERSION]
```

If you are sending a command to a subsystem provided by a company other than HP, you must make the appropriate, different entries for the Z^OWNER, Z^NUMBER, and Z^VERSION fields.

Token Codes

The built-in functions for SPI accept token codes either as external-format integers or numeric values, or as 32-bit STRUCTs.

Token-code STRUCT definitions, as generated by DDL, are included in the TACL versions of the SPI and subsystem definition files. Your TACL macros and routines can refer to the individual fields of a token code by copying the token code into ZSPI^DDL^TOKENCODE:Z^TOKENCODE.

Token codes cannot be composed or decomposed by simple arithmetic, because the token number is signed and simple arithmetic would extend the sign.

Token Maps

In TACL, a token map is a STRUCT whose data contains a valid SPI token map. The definition of the STRUCT is irrelevant.

Token-map definitions, as generated by DDL, are included in the TACL definition files for subsystems that define extensible structured tokens.

When using a token map, TACL verifies that the contents of the token map are consistent with the size of the STRUCT in which it is stored.

Token Values

These pages give the TACL types, value ranges, external representations, and special considerations for token values of the various token data types.

TACL Types and Value Ranges

In the definition files for SPI and the various NonStop Kernel subsystems, most tokens and token fields translate into appropriate TACL variable and field types, including high-level TACL types such as CRTPID, FNAME, and TIMESTAMP.

Table 8-1 lists the SPI token data types that TACL recognizes and gives, for each, the corresponding TACL type used in the ZSPIDEF.*subsys* TACL definition files and the acceptable range of values for an item of the basic length of that token data type.

TACL performs type checking on token values and reports any incompatibilities between the values and the corresponding TACL types. To perform comparisons and assignments involving type-incompatible null values or reset values, see Identifying Null Values on page 8-7 and Setting Reset Values on page 8-8.

TACL does not fill in default values for missing fields in values of token data types ZSPI^TDT^FNAME, ZSPI^TDT^FNAME32, or ZSPI^TDT^SUBVOL.

TACL interprets token data types it does not recognize (including ZSPI^TDT^LIST, ZSPI^TDT^MARK, ZSPI^TDT^FLT, and ZSPI^TDT^FLT2) as if they were ZSPI^TDT^INT. This action of TACL causes no problems for the first two token data types (because they are of token length 0 and have no corresponding token values), but it amounts to nonsupport for the floating-point token data types.

Table 8-1. TACL Data Types for SPI (page 1 of 2)

Token Data Type	TACL Type	Value Range
ZSPI^TDT^BOOLEAN	BOOL	-32768 through 32767
ZSPI^TDT^BYTE	BYTE	0 through 255
ZSPI^TDT^CHAR	CHAR	Binary 0 through 127
ZSPI^TDT^CRTPID	CRTPID	Any valid internal-format process ID
ZSPI^TDT^DEVICE	DEVICE	Any valid internal-format device name
ZSPI^TDT^ENUM	ENUM	–32768 through 32767
ZSPI^TDT^ERROR	SSID plus INT	Any valid subsystem ID followed by a number from -32768 through 32767
ZSPI^TDT^FNAME	FNAME	Any valid 24-byte internal-format file name
ZSPI^TDT^FNAME32	FNAME32	Any valid 8-byte internal-format system name followed by any valid 24-byte internal-format local file name
ZSPI^TDT^INT	INT	-32768 through 32767
ZSPI^TDT^INT2	INT2	-2147483648 through 2147483647

Table 8-1. TACL Data Types for SPI (page 2 of 2)

Token Data Type	TACL Type	Value Range
ZSPI^TDT^INT4	INT4	-(2**63) through (2**63)-1
ZSPI^TDT^MAP	STRUCT	Depends on STRUCT definition
ZSPI^TDT^SSCTL	BYTE	0 through 255
ZSPI^TDT^SSID	SSID	Any valid subsystem ID
ZSPI^TDT^STRUCT	STRUCT	Depends on STRUCT definition
ZSPI^TDT^SUBVOL	SUBVOL	Any valid volume and subvolume name
ZSPI^TDT^TIMESTAMP	TSTAMP	-(2**63) through (2**63)-1
ZSPI^TDT^TOKENCODE	INT2	-2147483648 through 2147483647 or, on input, a 32-bit STRUCT
ZSPI^TDT^TRANSID	TRANSID	Any 64-bit TMF internal-format transaction ID
ZSPI^TDT^UINT	UINT	0 through 65535
ZSPI^TDT^USERNAME	USERNAME	Any valid user name (not user number)

External Representations

TACL represents values of these token data types as numeric character strings representing numbers within the acceptable ranges as given in Table 8-1:

ZSPI^TDT^BYTE
ZSPI^TDT^BOOLEAN
ZSPI^TDT^ENUM
ZSPI^TDT^INT
ZSPI^TDT^INT2
ZSPI^TDT^INT4
ZSPI^TDT^UINT
ZSPI^TDT^TIMESTAMP

TACL represents values of token data types ZSPI^TDT^CHAR and ZSPI^TDT^USERNAME as character strings of the same form accepted by other languages. For ZSPI^TDT^CHAR values, that form is a number of contiguous characters equal to the actual byte length of the token. (The actual byte length is the value of the token-length field if that value is less than 255, or the length of the token value if the token-length field is 255.)

TACL accepts and displays the usual external representations for values of token data types ZSPI^TDT^CRTPID, ZSPI^TDT^DEVICE, ZSPI^TDT^FNAME, ZSPI^TDT^FNAME32, and ZSPI^TDT^SUBVOL. For ZSPI^TDT^CRTPID, ZSPI^TDT^DEVICE, ZSPI^TDT^FNAME, and ZSPI^TDT^SUBVOL values, systems whose numbers cannot be found are given system number 255 on input, and shown as \?? on output. For values of token data type ZSPI^TDT^CRTPID, programs can access the creation time of an unnamed process or the *cpu*,*pin* of any process only by copying the value into a STRUCT whose fields have been redefined accordingly.

Values of token data type ZSPI^TDT^SSID have the external format described in Subsystem ID on page 8-3. Values of token data type ZSPI^TDT^ERROR consist of a subsystem ID in that format followed by a period and a numeric character string representing a number in the range -32768 to +32767.

Values of token data type ZSPI^TDT^TRANSID have this external format:

```
\system-name(crash-count).cpu.sequence
```

If the internal-format TRANSID contains a crash count of zero, the TRANSID is formatted as:

```
\system-name.cpu.sequence
```

In any of the cases above, if the system is not named the TRANSID is formatted as:

```
cpu.sequence
```

On output, if TACL cannot find the name of a system, TACL replaces *system-name* with *system-number*.

Values of token data types ZSPI^TDT^MAP and ZSPI^TDT^STRUCT must be placed in a STRUCT or retrieved from a STRUCT, and your macro or routine must handle them with #SSGETV and #SSPUTV. The definition of the STRUCT provides the rules for conversion between internal and external formats.

If the token-length field of a token is less than 255 and the token data type is any other than ZSPI^TDT^CHAR, ZSPI^TDT^MAP, or ZSPI^TDT^STRUCT, the token value is represented by a space-separated list of *m*/*n* items, where *m* is the true length of the token and *n* is the basic length of the token data type. If *m* is not evenly divisible by *n*, the last bytes cannot be set or seen. Each item is in its usual external representation.

If the token-length field of a token is 255, the token value is of variable length. TACL represents such a token value externally as a numeric character string representing the one-word byte length of the token, followed by a space, followed by the token value or values. For example, a variable-length character string containing the characters “abcd” is represented as

```
4 abcd
```

Likewise, a variable-length array containing the integers 1, 2, 3, and 4 is represented as

```
8 1 2 3 4
```

Identifying Null Values

Null values for fields of an extensible structured token often are not acceptable values in TACL. You can determine whether a field of an extensible structured token (for instance, in a response from a subsystem) has a null value by comparing the field in question with the corresponding field of an extra copy of the structure, initialized with SSNULL. To perform this comparison without causing a TACL type-checking error, use the #COMPAREV built-in function.

Setting Reset Values

Some subsystems, such as Pathway, define special values (similar to null values) that, when assigned to structured-token fields that represent command parameters, direct the subsystem to reset those parameters to their default values. However, these values might not always be legitimate values for the field as defined in TACL.

To circumvent this problem, use the TACL built-in #SETBYTES to set these fields. To use #SETBYTES, you first define a STRUCT, then put the reset value into the STRUCT; then you call #SETBYTES, referring to the field you are resetting and the name of the STRUCT.

For example, suppose that you are using Pathway and want to set the subfield ZINSPECTFILE of the field ZINSPECTINFO of the structure ZPWY^DDL^DEF^TERM. This field is of type FNAME32, and the corresponding reset constant, ZPWY^VAL^RESETALPHABYTE, is not compatible with FNAME32 in TACL. To set the field to its reset value, you can use this TACL code:

```
[#DEF reset^alpha^byte STRUCT
   BEGIN BYTE reset^alpha
      VALUE [zpwy^val^resetalphabyte]; END;]

[#SETBYTES zpwy^ddl^def^term:zinspectinfo:zinspectfile
   reset^alpha^byte]
```

For comparisons involving such values, use the #COMPAREV built-in.

Syntax of the TACL Built-Ins

These built-in functions are the TACL counterparts of the SPI procedures described in Section 3, The SPI Procedures. (They call the SPI procedures internally.)

#SSINIT

Use #SSINIT to initialize a STRUCT as an SPI buffer, preparing it for use with the other #SS*xxx* built-in functions. This operation gives the buffer an appropriate header and, optionally, adds parameter information.

You can use #SSINIT only to initialize a buffer for a command or a response; it cannot be used to initialize an event-message buffer.

```
#SSINIT [ / TYPE 0 / ] buffer-var ssid
   command [ / type-0-option [ , type-0-option ]... / ]
```

`TYPE 0`

indicates the header type of the SPI message buffer being initialized. Type 0, a command or response header, is the default. It is also the only header type currently supported by TACL.

`buffer-var`

is the name of the variable to be initialized as an SPI buffer. This variable must be a writable STRUCT. #SSINIT automatically passes the data length of the STRUCT to SSINIT. The current contents of the STRUCT are lost.

`ssid`

is the subsystem ID of the subsystem. For requester functions, this ID identifies the target subsystem, and the version field must identify the version of the subsystem definitions that your requester program is using. For server functions (subsystems), this ID must identify your server program, including its version.

`command`

is the command number.

`type-0-option`

can be any of:

`CHECKSUM` *`checksum`*

gives the checksum flag. If *`checksum`* is zero or if you do not supply this option, checksum protection of the buffer is disabled; if *`checksum`* is nonzero, checksum protection is enabled.

`MAXFIELDVERSION` *`max-field-version`*

is an unsigned integer value that initializes the maximum field version field of the buffer header. The default value is zero.

`MAXRESPONSES` *`max-resp`*

gives the maximum number of response records to be returned by the subsystem in each response message. A *`max-resp`* value of zero (the default) specifies one response record per response, not enclosed in a list. Any positive value specifies up to that number of response records, each enclosed in a list. A value of -1 specifies as many response records as will fit, each enclosed in a list.

OBJECT *object-type*

gives the object type. If you do not supply this option, the object type defaults to ZSPI^VAL^NULL^OBJECT^TYPE (zero).

SERVERVERSION *server-version*

is normally provided only by subsystems or by other programs that are acting as a server. In those cases, it is a 16-bit unsigned integer value representing the version of the subsystem or server program. SSINIT places this value in the header token ZSPI^TKN^SERVER^VERSION for use in version compatibility checking. If you do not supply this option, the server version defaults to zero.

Expansion

#SSINIT expands to a numeric status code indicating the outcome of the operation. The status code has one of these values:

```
  0    No error
 -2    Illegal parameter value
 -3    Missing parameter
 -4    Illegal parameter address
 -5    Buffer full
 -7    Internal error
-10    Invalid subsystem ID
-12    Insufficient stack space
```

For more information about nonzero status codes, see Appendix A, Errors.

Consideration

- For the *ssid* parameter, if you are initializing a command buffer to send to a NonStop Kernel subsystem, you can expand a name of the form *subsys*^VAL^SSID. For example, this #SSINIT call initializes a buffer to send the GETVERSION command to EMS:

```
#SSINIT buf [ZEMS^VAL^SSID] [ZSPI^VAL^GETVERSION]
```

#SSNULL

The #SSNULL built-in function initializes a structure with null values. Use this procedure before setting values within a structure for an extensible structured token.

Note. Your macro or routine must **always** use #SSNULL before placing values in the fields of a structure, even when all currently defined fields are set explicitly. This practice allows the program to continue to work with future software releases.

```
#SSNULL token-map struct
```

token-map

is a token map to be used in initializing the fields of the structure.

struct

is the structure to be initialized with null values.

Expansion

#SSNULL expands to a numeric status code indicating the outcome of the operation. The status code has one of these values:

```
 0    No error
-3    Missing parameter
-4    Illegal parameter address
-7    Internal error
-9    Illegal token code or map
```

For more information about nonzero status codes, see Appendix A, Errors.

#SSPUT

Use #SSPUT to convert token values from external representation to binary form and insert them into an SPI buffer previously initialized by #SSINIT. #SSPUT cannot insert values of extensible structured tokens using a token map or using a token code of type ZSPI^TDT^STRUCT. For this purpose, use #SSPUTV.

```
#SSPUT [ / option [ , option ] / ] buffer-var
       token-code [ token-value [ token-value ]... ]
```

option

is either of:

COUNT *count*

gives the token count. If *count* is greater than 1, *token-value* is assumed to be a space-separated list of *count* elements, each of which is described by the *token-code*. *count* must be an integer in the range 1 to 65535, inclusive, and you must provide that many token values. (You must represent a variable-length token value in two parts—the byte length followed by the actual value—separated by a space.) The default count is 1.

If *token-code* is a special operation that does not allow a token value, you must omit COUNT *count*. For the special token code ZSPI^TKN^DELETE, you must supply COUNT *count*, and #SSPUT interprets *count* as the index value of the token code to be deleted.

SSID *ssid*

is a subsystem ID that qualifies the token code. If *ssid* is zero (0.0.0) or if you do not supply this option, the default applies. If #SSPUT is currently adding tokens to a list, *ssid* defaults to the subsystem ID of that list; otherwise, *ssid* defaults to the subsystem ID in the SPI message header (ZSPI^TKN^SSID).

buffer-var

is the name of the SPI message-buffer variable into which tokens are to be placed.

token-code

either identifies the token being supplied or denotes a special operation. The special operations are described further in Special Operations for #SSPUT and #SSPUTV on page 8-13. (To specify a token map, or a token code of type ZSPI^TYP^STRUCT, use #SSPUTV.)

token-value

if present, is the value of the token. Its data representation is determined by the token-type field of the *token-code*.

Expansion

#SSPUT expands to a numeric status code indicating the outcome of the operation. The status code has one of these values:

```
  0    No error
 -1    Invalid buffer format
 -2    Illegal parameter value
 -3    Missing parameter
 -4    Illegal parameter address
 -5    Buffer full
 -6    Invalid checksum
 -7    Internal error
 -8    List token not found
 -9    Illegal token code or map
-10    Invalid subsystem ID
-11    Operation not supported
-12    Insufficient stack space
```

For more information about nonzero status codes, see Appendix A, Errors.

Considerations

- The *token-value* parameter is optional if the token length specified by *token-code* is zero (for instance, if *token-code* is ZSPI^TKN^DATALIST, ZSPI^TKN^ERRLIST, ZSPI^TKN^LIST, ZSPI^TKN^ENDLIST, or ZEMS^TKN^SUBJECT^MARK). Otherwise, the *token-value* parameter is required.

- Specifying a *count* parameter greater than one for #SSPUT is equivalent to calling #SSPUT *count* number of times in succession with a *count* of 1 (but supplying a new *token-value* before each call).
- If *count* is greater than 1 and the token is of variable length, the length of each token value must be an even number of bytes to ensure word alignment. For example, the object-name token used by extended-SPI subsystems, ZCOM^TKN^OBJNAME, is a variable-length character string. This #SSPUT call places in the buffer two object-name tokens whose values are “abc” and “defgh”:

  ```
  #SSPUT /COUNT 1/ buf ZCOM^TKN^OBJNAME 3 abc 5 defgh
  ```

- The order in which tokens are added to the buffer is not significant except in the case of (1) #SSPUT calls with token codes for tokens that start and end lists (ZSPI^TKN^DATALIST, ZSPI^TKN^ERRLIST, ZSPI^TKN^LIST, ZSPI^TKN^SEGLIST, and ZSPI^TKN^ENDLIST), and (2) a few subsystem-specific exceptions mentioned in the subsystem manuals (for example, the ZEMS^TKN^SUBJECT^MARK token in an event message).
- Adding a token to the buffer with #SSPUT does not affect the current position for subsequent calls to #SSGET or #SSGETV.

Special Operations for #SSPUT and #SSPUTV

Table 8-2 lists the header tokens your programs can supply to #SSPUT and #SSPUTV to set or change the corresponding values, and the tokens your programs can pass to #SSPUT and #SSPUTV to perform special operations.

Header tokens for #SSPUT and #SSPUTV include tokens to enable or disable checksum protection, specify the maximum-response value, restore the current position to an earlier value, and specify the server version. These tokens are present in the buffer but differ from other tokens in certain ways, as described in Section 2, SPI Concepts and Protocol. (One header token, ZSPI^TKN^POSITION, can also be considered a special operation for SSPUT. It is classified here as a header token because it is a header token for #SSGET and #SSGETV, and because it is present in the buffer and has the header-token characteristics given in Section 2, SPI Concepts and Protocol.)

Special operations for #SSPUT and #SSPUTV include clearing the last-error information, flushing data from the buffer, deleting tokens, and initializing the current position. These tokens are not present in the buffer but simply serve as parameters to #SSPUT and #SSPUTV.

Table 8-2. #SSPUT(V) Special Operations (page 1 of 2)

Token Specified in #SSPUT(V) call	Type	Effect
ZSPI^TKN^BUFLEN	UINT	Modify buffer length
ZSPI^TKN^CHECKSUM	BOOLEAN	Enable or disable buffer checksum
ZSPI^TKN^CLEARERR		Clear last-error information to zero

Table 8-2. #SSPUT(V) Special Operations (page 2 of 2)

Token Specified in #SSPUT(V) call	Type	Effect
ZSPI^TKN^DATA^FLUSH		Flush tokens starting at current position
ZSPI^TKN^DELETE	INT2	Delete a token from the buffer
ZSPI^TKN^INITIAL^POSITION	INT	Reset position to start of buffer or list
ZSPI^TKN^MAX^FIELD^VERSION	UINT	Maximum field version
ZSPI^TKN^MAXRESP	INT	Set maximum-responses header token
ZSPI^TKN^POSITION	BYTE:8	Restore a previously saved position
ZSPI^TKN^RESET^BUFFER	UINT	Reset buffer
ZSPI^TKN^SERVER^VERSION	UINT	Set server-version header token

The actions performed when these tokens are passed in the *token-code* parameter to #SSPUT or the *token-id* parameter to #SSPUTV are:

ZSPI^TKN^BUFLEN: Modify Buffer Length

Use this token code to modify the SPI buffer length. If the buffer length value used with SSPUT is less than the number of bytes indicated by ZSPI-TKN-USEDLEN, the buffer length is modified and SSPUT returns ZSPI-ERR-NOSPACE.

ZSPI^TKN^CHECKSUM: Set Checksum Flag

With this token code, a nonzero *token-value* enables checksum protection of the buffer; a zero *token-value* disables it.

ZSPI^TKN^CLEARERR: Clear Last SPI Error

Use this token code to clear the last-error information to zero. You must omit *token-value* and COUNT *count*.

You might use this operation before issuing a series of #SSPUT[V] and #SSGET[V] calls that are followed by a check of the last error. You need this operation only if you use #SSPUT or #SSPUTV to check the header token ZSPI^TKN^LASTERR.

ZSPI^TKN^DATA^FLUSH: Clear Buffer From Current Position

Use this token code to flush all information in the message buffer located at or after the current position. You must omit *token-value* and COUNT *count*.

The ZSPI^TKN^DATA^FLUSH operation does not cause the header token ZSPI^TKN^MAX^FIELD^VERSION to be updated. As a result, following this operation, that field can indicate a version higher than that contained in the buffer.

ZSPI^TKN^DELETE: Delete a Token or List

Use this token code to delete a token code from the buffer. For *token-value*, specify the token code to be deleted.

You must supply COUNT *count*. For ZSPI^TKN^DELETE, #SSPUT and #SSPUTV interpret the *count* value as the index value of the token code to be deleted.

You can supply SSID *ssid*, if needed, to qualify the *token-code*.

If *token-code* is a token that begins a list, the operation deletes the entire list.

The ZSPI^TKN^DELETE operation does not cause the header token ZSPI^TKN^MAX^FIELD^VERSION to be updated. As a result, following this operation, that field can indicate a version higher than that contained in the buffer.

ZSPI^TKN^INITIAL^POSITION: Reset Current and Next Token Pointers

Use this token code to reset the current position as specified by the value of the *token-value* parameter. If *token-value* is ZSPI^VAL^INITIAL^BUFFER (0), the position is reset to the beginning of the buffer. If *token-value* is ZSPI^VAL^INITIAL^LIST (-1), the position is reset to the start of the current list.

ZSPI^TKN^MAX^FIELD^VERSION: Increase Maximum Version of Structure Fields

Use this token code to increase the maximum field version of the buffer. If the value specified is greater than the current value, then the specified value is used.

ZSPI^TKN^MAXRESP: Set Maximum Responses

Use this token code to set the header token that specifies the maximum number of responses to return in a single response message. A *token-value* of zero (the default) specifies one response record per response, not enclosed in a list. Any positive *token-value* specifies up to that number of response records, each enclosed in a list. A *token-value* of -1 specifies as many response records as will fit, each enclosed in a list.

You must omit COUNT *count*.

ZSPI^TKN^POSITION: Set Current Buffer Position Pointer

Use this token code to restore a position previously saved using #SSGET or #SSGETV. The *token-value* is a position descriptor, represented as eight separate

byte values for #SSPUT or as an 8-byte STRUCT for #SSPUTV. For this operation to be valid, the contents of the buffer prior to the previously saved position must not have been modified by ZSPI^TKN^DELETE, ZSPI^TKN^DATAFLUSH, or #SSMOVE operations. Otherwise, this operation can corrupt the buffer and cause later operations to give indeterminate results. If *token-value* is zero or not supplied, this operation sets the current position to the beginning of the buffer.

ZSPI^TKN^RESET^BUFFER: Reset the Buffer

Use this token code before extracting tokens from an SPI buffer received (in either a command or a response) from another process. This operation performs three actions:

- It resets the maximum buffer length to the value given in *token-value*.
- It clears the last-error information to null values (equivalent to the action of ZSPI^TKN^CLEARERR).
- It resets the current position to the beginning of the buffer (equivalent to the action of ZSPI^TKN^INITIAL^POSITION with ZSPI^VAL^INITIAL^BUFFER).

TACL checks to be sure the value given in *token-value* does not exceed the size of the structure passed as the buffer. If *token-value* exceeds the buffer size, the SPI error -5 (buffer full) is returned. SPI still resets the maximum buffer length in the SPI message header, causing subsequent SPI calls for that buffer to fail with error -1 (invalid buffer format).

ZSPI^TKN^SERVER^VERSION: Set Server Version Header Token

Use this token code to set the header token containing the release version of the server. For *token-value*, supply an unsigned integer representing the appropriate release version. For example, if the server is a NonStop Kernel subsystem of version G06, *token-value* should be the unsigned integer with the ASCII character *G* in the left byte and “06” in the right byte, or 18182.

#SSPUTV

Use #SSPUTV to take binary token values from a variable and insert them into an SPI buffer. You can use #SSPUTV with any type of token. With tokens of type ZSPI^TYP^STRUCT and extensible structured tokens, you must use #SSPUTV.

```
#SSPUTV [ / option [ , option ]... / ]
        buffer-var token-id source-var
```

option

is either of:

COUNT *count*

gives the token count. If *count* is greater than 1, *source-var* is assumed to contain an array of *count* elements, each of which is described by the *token-id*. If you do not supply this option, TACL assumes a *count* of 1.

If *token-id* is one of the special SPI token codes whose semantics do not allow a token value, you must omit COUNT *count*. For the special token code ZSPI^TKN^DELETE, you must supply COUNT *count*, and *count* is interpreted as the index value of the token code to be deleted.

SSID *ssid*

is a subsystem ID, as described in Section 4, that qualifies the token code. If *ssid* is not supplied or is equal to zero (0.0.0), the default applies. If SSPUT is currently adding tokens to a list, *ssid* defaults to the subsystem ID of that list; otherwise, *ssid* defaults to the subsystem ID in the SPI message header (ZSPI^TKN^SSID).

buffer-var

is the name of the SPI message-buffer variable into which tokens are to be placed.

token-id

is a token code or a token map. This parameter either identifies the token being supplied or denotes a special operation. The special operations are described further in Special Operations for #SSPUT and #SSPUTV on page 8-13.

source-var

is the name of the STRUCT from which #SSPUTV is to obtain the binary token values. The contents of the STRUCT are not altered.

Expansion

#SSPUTV expands to a numeric status code indicating the outcome of the operation. The status code has one of these values:

```
  0    No error
 -1    Invalid buffer format
 -2    Illegal parameter value
 -3    Missing parameter
 -4    Illegal parameter address
 -5    Buffer full
 -6    Invalid checksum
 -7    Internal error
 -8    List token not found
 -9    Illegal token code or map
-10    Invalid subsystem ID
-11    Operation not supported
-12    Insufficient stack space
```

For more information about nonzero status codes, see Appendix A, Errors.

Considerations

- If the token length specified by *token-id* is zero (for instance, if *token-id* is ZSPI^TKN^DATALIST, ZSPI^TKN^ERRLIST, ZSPI^TKN^LIST, ZSPI^TKN^ENDLIST, or ZEMS^TKN^SUBJECT^MARK), you must still supply a variable for *source-var*, but its contents do not matter.
- Specifying a *count* parameter greater than 1 for #SSPUTV is equivalent to calling #SSPUTV *count* number of times in succession with a *count* of 1 (but supplying a new *token-value* before each call).
- The order in which tokens are added to the buffer is not significant except in the case of (1) #SSPUTV calls with token codes for tokens that start and end lists (ZSPI^TKN^DATALIST, ZSPI^TKN^ERRLIST, ZSPI^TKN^LIST, and ZSPI^TKN^ENDLIST), and (2) a few subsystem-specific exceptions mentioned in the subsystem management programming manuals (for example, the ZEMS^TKN^SUBJECT^MARK token in an event message).
- If the data in *source-var* is longer than the data area expected by #SSPUTV, the excess bytes are ignored without any indication. If the data in *source-var* is shorter than the data area expected by #SSPUTV, the remainder of the token value is set to unspecified values.
- If the COUNT *count* option is specified, the value of *source-var* is expected to be an array of *count* values of the type of *token-value*. Variable-length token values must be word-aligned. For example, the object-name token for extended SPI subsystems, ZCOM^TKN^OBJNAME, is a variable-length character string. If a program needs to call #SSPUTV to place in the buffer two object-name tokens whose values are “abc” and “defgh”, this #DEF defines the correct STRUCT for *source-var*:

```
[#DEF objnames STRUCT
   BEGIN
      UINT len1      VALUE 3;
      CHAR n1(0:3)   VALUE abc;
      UINT len2      VALUE 5;
      CHAR n2(0:5)   VALUE defgh;
   END;
]
```

- Adding a token to the buffer with #SSPUTV does not affect the current position for subsequent calls to #SSGET or #SSGETV.
- When #SSPUTV is called with a token map for *token-id*, it uses the null-value and version information in the token map, if necessary, to update the header token ZSPI^TKN^MAX^FIELD^VERSION. The token map is not stored in the buffer; instead, #SSPUTV creates a token code consisting of token type ZSPI^TYP^STRUCT and the token number from the map.

- SPI defines a number of token codes for use with #SSPUT and #SSPUTV to set the values of header tokens and perform special operations. See Special Operations for #SSPUT and #SSPUTV on page 8-13.

#SSGET

Use #SSGET to retrieve binary token values from an SPI buffer, convert them to external representation, and make that external representation accessible in the function's expansion.

You cannot use #SSGET to extract values of extensible structured tokens using a token map or using a token code of type ZSPI^TDT^STRUCT. For this purpose, use #SSGETV.

```
#SSGET [ / option [ , option ]... / ] buffer-var get-op
```

option

is any of:

COUNT *count*

gives the maximum number of token values to return. This specifies that the token value returned in the expansion is an array of *count* elements, each of which is described by *token-code*. If not supplied, it defaults to 1.

If a *count* greater than 1 is specified, #SSGET continues searching until it either satisfies the requested count or reaches the end of the buffer or list.

A *count* less than 1 causes an error.

INDEX *index*

if *index* is greater than zero, specifies an absolute index for *token-code*, starting from the beginning of the buffer or list. An *index* of 1 gets the first occurrence of that token, an *index* of 2 gets the second occurrence, and so on.

if *index* is zero or not specified, #SSGET returns the next occurrence of the token following the current token, and resets the current position to the token returned. For example, if a token occurs five times, calling #SSGET once with *index* = 1 and four times with *index* = 0 returns all occurrences.

An *index* value less than zero causes an error.

Your program must *always* either supply a nonzero index or first reset the initial position (#SSPUT or #SSPUTV with ZSPI^TKN^INITIAL^POSITION or ZSPI^TKN^RESET^BUFFER) if the search is to start from the beginning of the buffer (or from the beginning of the current list).

SSID *ssid*

gives a subsystem ID (of type SSID) that qualifies the token code. If *ssid* is omitted or equal to zero (0.0.0), it defaults to the subsystem ID of the current list, or if the current position is not in a list, then to the subsystem ID specified in the SPI message header. The version field of this parameter is not used when searching the buffer.

buffer-var

is the name of the SPI message-buffer variable from which information is to be extracted.

get-op

is one of:

token-code

directs #SSGET to return the token value or values associated with the specified token code. (For a token map, or a token code of type ZSPI^TYP^STRUCT, use #SSGETV.)

If *token-code* is a token that marks the beginning of a list (ZSPI^TKN^DATALIST, ZSPI^TKN^ERRLIST, or ZSPI^TKN^LIST), #SSGET selects the list so that subsequent calls can retrieve tokens within the list. If *token-code* is ZSPI^TKN^ENDLIST, #SSGET deselects (pops out of) the list.

For *token-code*, you can supply one of the header tokens in Table 8-2 on page 8-13. You can also supply the token ZSPI^TKN^DEFAULT^SSID to obtain the default subsystem ID at the current position (see Special Operations for #SSPUT and #SSPUTV on page 8-13).

ZSPI^TKN^COUNT *c-token-id*

directs #SSGET to return the total number of occurrences of the token specified by the token code or token map *c-token-id*, starting at *index*. If *index* is not supplied, counting starts from the current position. To count all occurrences in the current list, specify an *index* of 1.

If *c-token-id* is either omitted or equal to ZSPI^VAL^NULL^TOKENCODE and *index* is either omitted or zero, then #SSGET counts occurrences of the current token beginning with the current occurrence.

ZSPI^TKN^LEN *l-token-id*

directs #SSGET to return the byte length of the token specified by the token code or token map *l-token-id*. The value returned is the size of the buffer needed to contain the specified occurrence of the token value. For variable-length token values, this includes the two bytes required for the length word: the byte length returned is *token-value*[0] + 2.

If *l-token-id* is either omitted or equal to ZSPI^VAL^NULL^TOKENCODE and *index* is either omitted or zero, then #SSGET returns the length of the current occurrence of the current token.

If *l-token-id* is a token map, this operation returns the length contained within the specified token map; the actual value in the buffer can be longer or shorter than this length. To get the actual length of the token value in the buffer, call #SSGET with ZSPI^TKN^LEN, a token code made up of ZSPI^TYP^STRUCT, and the token number from the token map. This returns the length of the structure value, including 2 bytes for the length field. Then subtract 2 from this value to get the length of the value itself.

ZSPI^TKN^NEXTCODE

directs #SSGET to return the next token code that is different from the current token code, followed by the subsystem ID.

The subsystem ID returned in the expansion always has a version field of zero (null).

The *index* parameter has no effect on this operation, but if supplied, it must be equal to zero.

Note. The special operations ZSPI^TKN^NEXTCODE and ZSPI^TKN^NEXTTOKEN return only token codes. In particular, tokens added to the buffer by using #SSPUTV with a token map are carried in the buffer with a token code of type ZSPI^TYP^STRUCT. The NEXTCODE and NEXTTOKEN operations return this token code, not the token code used with #SSPUTV.

ZSPI^TKN^NEXTTOKEN

directs #SSGET to return the very next token code, followed by the subsystem ID. This operation differs from ZSPI^TKN^NEXTCODE in that it always returns the token code of the next token, whether or not it is the same as that of the current token, and whether or not the token is within a list. The operation returns multiple occurrences of the same token code in the same order as they were added to the buffer with #SSPUT or #SSPUTV.

The subsystem ID returned in the expansion always has a version field of zero (null).

The *index* and *count* parameters have no effect on this operation. However, if *index* is supplied, it must be equal to zero.

See the NOTE at the end of ZSPI^TKN^NEXTCODE on page 8-21.

ZSPI^TKN^OFFSET *o-token-id*

directs #SSGET to return the byte offset of the token specified by the token code or token map *o-token-id*. The value returned is the offset from the start of the buffer to the value associated with the specified token code and index. (For variable-length values, the token value begins with the length word; the offset given is the offset to that length word.)

If *o-token-id* is either omitted or equal to ZSPI^VAL^NULL^TOKENCODE and *index* is either omitted or zero, then #SSGET returns the length of the current occurrence of the current token.

Expansion

#SSGET expands to a numeric status code indicating the outcome of the operation. If the status code is 0 (no error), it is followed by a space and a space-separated list of the relevant results in TACL's external representation.

The status code has one of these values:

```
  0    No error
 -1    Invalid buffer format
 -2    Illegal parameter value
 -3    Missing parameter
 -4    Illegal parameter address
 -5    Buffer full
 -6    Invalid checksum
 -7    Internal error
 -8    Token not found
 -9    Illegal token code or map
-10    Invalid subsystem ID
-11    Operation not supported
-12    Insufficient stack space
```

For more information about nonzero status codes, see Appendix A, Errors. If the status is 0, these results are returned following the status:

- If you specified *token-code*, the number of token values returned, followed by a space-separated list of the token values in external form. Variable-length token values are returned in two parts—the byte length followed by the actual value—separated by a space.
- If you specified ZSPI^TKN^COUNT *c-token-id*, the total number of occurrences of the token specified by the token code or token map *c-token-id*, starting at *index*.
- If you specified ZSPI^TKN^LEN *l-token-id*, the length of the token specified by the token code or token map *l-token-id*.
- If you specified ZSPI^TKN^NEXTCODE, the next token code that is different from the current token code, followed by the subsystem ID.
- If you specified ZSPI^TKN^NEXTTOKEN, the next token code, regardless of whether it is different from the current token code, followed by the subsystem ID.
- If you specified ZSPI^TKN^OFFSET *o-token-id*, the byte offset of the token is specified by the token code or token map *o-token-id*.

Considerations

- When *token-code* is ZSPI^TKN^ENDLIST, the *index* and *count* parameters have no effect. However, if supplied, *index* must be equal to zero or 1, and the count in the expansion is always returned as 1.

Header Tokens and Special Operation for #SSGET and #SSGETV

The *index* parameter you supply with these token codes must be 0 or 1.

Table 8-3. Header Token Values Retrieved by #SSGET and #SSGETV

Token Code	Type	Value Retrieved
ZSPI^TKN^BUFLEN	UINT	Buffer length
ZSPI^TKN^CHECKSUM	INT	Checksum flag
ZSPI^TKN^COMMAND	ENUM	Command number
ZSPI^TKN^HDRTYPE	UINT	Header type
ZSPI^TKN^LASTERR	ENUM	Last nonzero SPI status code
ZSPI^TKN^LASTERRCODE	INT2	Token involved in last procedure error
ZSPI^TKN^LASTPOSITION	BYTE:8	Position of last token added with SSPUT
ZSPI^TKN^MAX^FIELD^VERSION	UINT	Maximum field version
ZSPI^TKN^MAXRESP	INT	Maximum response records to return
ZSPI^TKN^OBJECT^TYPE	ENUM	Object-type number
ZSPI^TKN^POSITION	BYTE:8	Current position for #SSGET
ZSPI^TKN^SERVER^VERSION	UINT	Server release version
ZSPI^TKN^SSID	SSID	Subsystem ID used with #SSINIT
ZSPI^TKN^USEDLEN	UINT	Number of bytes used in the buffer

If you specify ZSPI^TKN^LASTPOSITION or ZSPI^TKN^POSITION, the value returned is an 8-byte position descriptor that you can later use to reset the position with the #SSPUT or #SSPUTV special operation ZSPI^TKN^POSITION. For #SSGET, the position descriptor is returned as a space-separated list of eight 1-byte values. For #SSGETV, the value is returned in a STRUCT consisting of 8 bytes.

In addition, you can specify the special operation code ZSPI^TKN^DEFAULT^SSID. This obtains the default subsystem ID at the current position, preceded by the number of token values returned—which in this case is always 1. #SSPUT, #SSPUTV, #SSGET, and #SSGETV use this value whenever the *ssid* parameter is omitted or null.

If the default subsystem ID comes from a list token, the version field of the returned subsystem ID value is set to ZSPI^VAL^NULL^VERSION. Therefore, when comparing subsystem ID values for equality, your program should omit the version field from the test.

#SSGETV

Use #SSGETV to obtain binary token values from an SPI buffer and put them into a STRUCT. You can use #SSGETV with any type of token. With tokens of type ZSPI^TYP^STRUCT and extensible structured tokens, you must use #SSGETV.

```
#SSGETV [ / option [ , option ]... / ]
        buffer-var get-op result-var
```

option

is any of:

```
COUNT count
INDEX index
SSID ssid
```

These options are the same as those described under #SSGET, substituting *token-id* for all references to *token-code*.

buffer-var

is the same as described for #SSGET.

get-op

is one of these:

token-id

is either a token code or a token map. It directs #SSGETV to return the token value or values associated with *token-id*.

If *token-code* is a token that marks the beginning of a list (ZSPI^TKN^DATALIST, ZSPI^TKN^ERRLIST, ZSPI^TKN^SEGLIST, or ZSPI^TKN^LIST), #SSGET selects the list so that subsequent calls can retrieve tokens within the list. If *token-code* is ZSPI^TKN^ENDLIST, #SSGET deselects (pops out of) the list.

```
ZSPI^TKN^COUNT c-token-id
ZSPI^TKN^LEN l-token-id
ZSPI^TKN^NEXTCODE
ZSPI^TKN^NEXTTOKEN
ZSPI^TKN^OFFSET o-token-id
```

are the same as defined for #SSGET, except that #SSGETV returns the results in *result-var* rather than in the expansion.

result-var

is the name of the writable STRUCT in which #SSGETV is to store the result. The original contents of the STRUCT are lost.

If the status code in the expansion is 0 (no error), the result stored in the STRUCT is:

- If you specified *token-id*, the result is the value of the token.
- If you specified ZSPI^TKN^COUNT *c-token-id*, the result is an INT giving the total number of occurrences of the token specified by the token code or token map *c-token-id*, starting at *index*.
- If you specified ZSPI^TKN^LEN *l-token-id*, the result is an INT giving the length of the token specified by the token code or token map *l-token-id*.
- If you specified ZSPI^TKN^NEXTCODE, the result is an INT2, an INT, and an SSID giving the next token code that is different from the current token code, the number of contiguous occurrences of that token code, and the subsystem ID.
- If you specified ZSPI^TKN^NEXTTOKEN, the result is an INT2 and an SSID giving the next token code (regardless of whether it is different from the current token code), followed by the subsystem ID.
- If you specified ZSPI^TKN^OFFSET *o-token-id*, the result is an INT2 giving the byte offset of the token specified by the token code or token map *o-token-id*.

To extract individual fields of the token code or the subsystem ID returned by #SSGETV with the ZSPI^TKN^NEXTCODE or ZSPI^TKN^NEXTTOKEN option, see the example at the end of Considerations on page 8-26.

Expansion

#SSGETV expands to a numeric status code indicating the outcome of the operation. This status code has one of these values:

```
  0    No error
 -1    Invalid buffer format
 -2    Illegal parameter value
 -3    Missing parameter
 -4    Illegal parameter address
 -5    Buffer full
 -6    Invalid checksum
 -7    Internal error
 -8    Token not found
 -9    Illegal token code or map
-10    Invalid subsystem ID
-11    Operation not supported
-12    Insufficient stack space
```

If the status is 0 and the *get-op* is *token-id*, the status is followed by a space and the number of token values returned. For more information about nonzero status codes, see Appendix A, Errors.

Considerations

- This example shows how to declare STRUCTs that allow you to extract individual fields of the token code or the subsystem ID returned by #SSGETV with the ZSPI^TKN^NEXTCODE or ZSPI^TKN^NEXTTOKEN option:

```
?SECTION decompose_ssid STRUCT
   BEGIN
      SSID   ss;
      STRUCT z^ssid REDEFINES ss;
         BEGIN
            CHAR  z^owner(0:7);
            INT   z^number;
            UINT  z^version;
         END;
   END;

?SECTION nexttoken_return STRUCT
   BEGIN
      STRUCT tkn ; LIKE zspi^ddl^tokencode;
      STRUCT ssid; LIKE decompose_ssid;
   END;

?SECTION nextcode_return STRUCT
   BEGIN
      STRUCT tkn ; LIKE zspi^ddl^tokencode;
      INT    contiguous_occurences;
      STRUCT ssid; LIKE decompose_ssid;
   END;
```

This routine uses these STRUCT declarations to compare two subsystem IDs returned by #SSGETV with ZSPI^TKN^NEXTCODE or ZSPI^TKN^NEXTTOKEN, ignoring the version field:

```
?SECTION same_ssid ROUTINE  == <ssid1> <ssid2>
== Returns TRUE if two SSIDs are the same except for
== the version field
#FRAME
#PUSH sstext
#DEF ss1 STRUCT LIKE decompose_ssid;
#DEF ss2 STRUCT LIKE decompose_ssid;
#IF{SINK} [#ARGUMENT/VALUE sstext/ SUBSYSTEM]
#SET ss1 [sstext]
#IF{SINK} [#ARGUMENT/VALUE sstext/ SUBSYSTEM]
#SET ss2 [sstext]
#RESULT [#COMPUTE [#COMPAREV ss1:z^ssid:z^owner(0:7)
                             ss2:z^ssid:z^owner(0:7)]
              AND [#COMPAREV ss1:z^ssid:z^number
                             ss2:z^ssid:z^number] ]
```

```
#UNFRAME
{same_ssid}
```

- Tokens extracted by #SSGETV are not deleted or removed from the buffer.
- When the current position is within a particular list, all #SSGETV calls pertain only to tokens within that list, except that header tokens are always accessible. Your program can exit the list by calling #SSGET with the ZSPI^TKN^ENDLIST token.
- When *token-id* is ZSPI^TKN^ENDLIST, the *index* and *count* parameters have no effect. However, if supplied, *index* must be equal to zero or 1.
- When using #SSGETV with a token map for the *token-id* parameter, the map can specify a structure version that is longer or shorter than the structure contained in the buffer. If the requested version is longer than the version in the buffer, #SSGETV calls SSNULL to set to null values the new fields that are not obtained from the buffer. If the requested version is shorter than the one in the buffer, #SSGETV returns only the requested length.
- If the data returned by #SSGETV is longer than the data area of the STRUCT specified by *result-var*, the excess bytes are discarded without any indication. If the data is shorter than the data area of the STRUCT, the entire STRUCT is set to its default values, and the data returned overwrites the beginning of the data bytes of the STRUCT. No type conversions are performed. For instance, if the token retrieved is of type ZSPI^TYP^INT and the *result-var* STRUCT consists of a single field of type INT2, the token value ends up in the high-order 16 bits of the INT2 field, not the low-order 16 bits.
- If the COUNT *count* option is specified, all occurrences of the token value are put into the STRUCT one after the other, just as they are extracted from the buffer, subject to the size corrections explained in the previous consideration. If the tokens are variable-length tokens, each token value consists of a length word followed by the actual value, and the actual value is word-aligned.
- Header tokens and one special operation can be passed in *token-id* to get the corresponding values. (See Header Tokens and Special Operation for #SSGET and #SSGETV on page 8-23.)

#SSMOVE

Use #SSMOVE to copy tokens from one SPI buffer to another. #SSMOVE performs a sequence of #SSGETV and #SSPUTV operations.

```
#SSMOVE [ / option [ , option ]... / ]
        source-var dest-var token-id
```

option

can be any of:

COUNT *count*

gives the maximum number of token values to copy, unless *token-id* is a list token; in the latter case, it gives the maximum number of lists to copy. If you do not supply this option, *count* defaults to 1.

DINDEX *dest-index*

if *dest-index* is greater than zero, identifies the first occurrence of *token-id* to be replaced in the destination buffer. A value of 1 specifies that replacement should start with the first occurrence of the token code, a value of 2 specifies the second occurrence, and so on. If the specified occurrences are not found in the destination buffer, the tokens being copied are added to the end of the buffer.

if *dest-index* is zero or if you do not supply this option, directs #SSMOVE to add the tokens from the source buffer to the end of the destination buffer.

SINDEX *source-index*

if *source-index* is greater than zero, identifies the first occurrence of *token-id* to be copied from the source buffer. (One or more than one occurrence can be copied, depending on the value of *count*.) A *source-index* value of 1 specifies that the copy is to start with the first occurrence of the token code, a value of 2 specifies the second occurrence, and so on.

if *source-index* is zero or if you do not supply this option, directs #SSMOVE to start with the next occurrence of the token code after the current position in the source buffer.

SSID *ssid*

is a subsystem ID, as described in Section 4, ZSPI Data Definitions, that qualifies the token ID. If not supplied or equal to zero (0.0.0), *ssid* defaults to the subsystem ID of the current list, or if the current position is not in a list, then to the subsystem ID specified in the SPI message header (ZSPI^TKN^SSID). The version field of *ssid* is not used in searching the source buffer.

source-var

is the name of the source message buffer variable—that is, the SPI buffer variable from which the specified token or tokens are to be copied.

dest-var

is the name of the destination message buffer variable—that is, the SPI buffer variable to which the specified token or tokens are to be copied.

token-id

is a token code or a token map that identifies the token to be copied. This token must be present in the source buffer. The *token-id* can identify a simple token,

an extensible structured token, or a list token. If a list token is specified, the list token, its associated end-list token, and all tokens in between are moved.

Expansion

#SSMOVE expands to a numeric status code indicating the outcome of the operation. If the status code is 0 (no error), it is followed by a space and the number of token values or lists moved.

The status code has one of these values:

```
  0    No error
 -1    Invalid buffer format
 -2    Illegal parameter value
 -3    Missing parameter
 -4    Illegal parameter address
 -5    Buffer full
 -6    Invalid checksum
 -7    Internal error
 -8    Token not found
 -9    Illegal token code or map
-10    Invalid subsystem ID
-11    Operation not supported
-12    Insufficient stack space
```

For more information about nonzero status codes, see Appendix A, Errors.

Considerations

- Tokens copied by #SSMOVE are not deleted or removed from the source buffer.
- After a successful #SSMOVE operation, the current-token pointer in the source buffer is changed to the position of the last token that was moved.
- When #SSMOVE copies a token identified by a token map, the value obtained from the source buffer is truncated or padded according to the map specifications, and the ZSPI^TKN^MAX^FIELD^VERSION header token of the destination buffer is appropriately adjusted.
- #SSMOVE can be used to copy an incomplete list (a list with no end-list token) if and only if *dest-index* is not supplied or zero. If a nonzero destination index is specified, meaning that a replacement operation is being requested, an incomplete list causes #SSMOVE to return a “token not found” (-8) status code.
- If an error occurs on #SSMOVE, the ZSPI^TKN^LASTERR and ZSPI^TKN^LASTERRCODE indications can be set in either the source buffer or the destination buffer, depending on whether the error occurred on the logical #SSGETV or #SSPUTV part of the move.

Interprocess Communication

An application requester written in TACL should open the management process for each subsystem using the READ option of #REQUESTER, and should use #APPENDV and #EXTRACTV to send commands and decode responses.

If you are writing a subsystem, it is recommended that you run TACL specifying its IN and OUT files as $RECEIVE, and then use an #INPUTV/#REPLYV loop protected by a suitable exception handler.

For more information about these features, see the *TACL Programmer's Guide*.

Example: Printing or Displaying the Status Structure of the Subsystem Control Point (SCP)

Example 8-1 is a TACL program that prints or displays the status structure of the Subsystem Control Point (SCP). The SCP process was chosen because it is commonly found running on most nodes under the name $ZNET. Because SCP adheres to the extended SPI protocol, it uses ZCOM data definitions. These definitions are described in the *SPI Common Extensions Manual*.

The first step is to load the appropriate definition files. This task takes the greatest amount of execution time. If you frequently use TACL for SPI, you should preload the ZSPIDEF.*subsys*TACL definitions so that your TACL functions do not need to do so.

The second step is to create an SPI buffer and initialize the SSID STRUCT with the correct value. Then the macro constructs the request, using #SSINIT and #SSPUT. Note that OBJNAME is a variable-length character string, so that a length preceding the value is required.

The third step opens the SPI server, sends the request, gets the reply, and then closes the server. (If you are going to do many requests, you should keep the server open rather than continually opening and closing it for each request.)

The fourth step prepares the buffer for use again by using ZSPI^TKN^RESET^BUFFER. Then the macro checks the return token, ZSPI^TKN^RETCODE, for successful completion of the command. The #SETMANY call that separates the results returned from #SSGET, and the INDEX 1 option is specified so that it does not matter where in the buffer the return token has been placed. The COUNT parameter is skipped (_), because this value is always 1.

Finally the desired data is extracted from the reply and, in this case, displayed. Here #SSGETV is used because a STRUCT is retrieved.

The error handling in this example is rudimentary. The error text causes TACL to stop executing the macro and point to the error text.

Example 8-1. Printing or Displaying the Status Structure of the SCP (page 1 of 2)

```
?TACL MACRO == SCPSTAT <scpname>
== Display SCP's status structure.
#FRAME
#PUSH zspi_subvol    == ZSPI definitions subvolume
#PUSH err            == Error return value
#PUSH retcode        == Return code from server
#PUSH io_err         == I/O error return value
#PUSH request        == Request I/O variable
#PUSH reply          == Reply I/O variable

== Locate and load the SPI definition files.
#SET zspi_subvol [#FILENAMES/MAXIMUM 1/ $*.ZSPIDEF.ZSCPTACL]
#SET zspi_subvol [#FILEINFO/VOLUME/[zspi_subvol]].ZSPIDEF
#LOAD/LOADED err/ [zspi_subvol].ZSPITACL
#LOAD/LOADED err/ [zspi_subvol].ZCOMTACL
#LOAD/LOADED err/ [zspi_subvol].ZSCPTACL

== Define the message buffer.
#DEF spi_buf STRUCT LIKE ZCOM^DDL^MSG^BUFFER;

== Assign SCP's subsystem ID.
#SET ZSCP^VAL^SSID [ZSPI^VAL^TANDEM].[ZSPI^SSN^ZSCP].&
[ZSCP^VAL^VERSION]

== Initialize the buffer for a STATUS command on
== the SCP subsystem PROCESS object type.
#SET err [#SSINIT spi_buf [ZSCP^VAL^SSID] ZCOM^CMD^STATUS
                                  /OBJECT ZCOM^OBJ^PROCESS/]
[#IF err |THEN| *** ERROR [err] from #SSINIT]

== Add the object-name token to the message.  The value
== of the token is the name of the SCP process.
#SET err [#SSPUT spi_buf ZCOM^TKN^OBJNAME [_longest %1%] %1%]
[#IF err |THEN| *** ERROR [err] from #SSPUT ZCOM^TKN^OBJNAME]

== Open the SCP process for SPI communications.
#SET err [#REQUESTER /WAIT [ZCOM^VAL^BUFLEN]/
                     READ %1%.#ZSPI io_err reply request]
[#IF err |THEN| *** ERROR [err] opening %1%.#ZSPI]

== Send the command and await the response.
#APPENDV request spi_buf
#EXTRACTV reply spi_buf
[#IF NOT [#EMPTYV io_err] |THEN|&
 *** ERROR [io_err] sending to %1%.#ZSPI]

== Close the SCP process.
#SET err [#REQUESTER CLOSE request]
[#IF err |THEN| *** ERROR [err] closing %1%.#ZSPI]
```

Example 8-1. Printing or Displaying the Status Structure of the SCP (page 2 of 2)

```
== Reset the buffer after receiving the response.
#SET err [#SSPUT spi_buf ZSPI^TKN^RESET^BUFFER
                         ZCOM^VAL^BUFLEN]
[#IF err |THEN| *** ERROR [err] from #SSPUT RESET^BUFFER]

== Retrieve the return code and test its value.
#SETMANY err _{count 1} retcode &
       , [#SSGET /INDEX 1/ spi_buf ZSPI^TKN^RETCODE]
[#IF err |THEN| *** ERROR [err] from #SSGET RETCODE]
[#IF retcode <> ZSPI^ERR^OK |THEN|
     *** ERROR [err] FROM %1%.#ZSPI on STATUS command]

== Retrieve the SCP status structure from the response.
#SETMANY err, [#SSGETV /INDEX 1/ spi_buf ZSCP^MAP^STATUS^PROC
                                         ZSCP^DDL^STATUS^PROC]
[#IF err |THEN| *** ERROR [err] from #SSGETV MAP^STATUS^PROC]

== Display the SCP response structure.
#OUTPUTV ZSCP^DDL^STATUS^PROC

#UNFRAME
{scpstat}
```

For a larger example illustrating the use of SPI in TACL, see the *Distributed Name Service (DNS) Management Programming Manual*.

9 SPI Programming in TAL

This section provides language-specific information for the programmer who is using the TAL to write an SPI requester or server:

Definition Names in TAL

Symbolic names in this section are in the TAL form, using circumflex (^) symbols rather than hyphens. For example, the DDL token code ZSPI-TKN-RETCODE is expressed as ZSPI^TKN^RETCODE in TAL.

TAL Definition Files

Each TAL module of your application that uses the Subsystem Programmatic Interface (SPI) must begin with ?SOURCE directives to include the TAL versions of the SPI standard definitions and the definitions for all subsystems with which your program communicates. The TAL version of the SPI standard definitions is in the file named ZSPIDEF.ZSPITAL on the disk volume chosen by your site.

For NonStop Kernel subsystems, the TAL versions of the subsystem definitions have file names of the form ZSPIDEF.*subsys*TAL, where *subsys* is the 4-character subsystem abbreviation given in Appendix D, NonStop Kernel Subsystem Numbers and Abbreviations. You can include these ?SOURCE directives in any order, but they must precede any of your own declarations that refer to them.

Declarations Needed in TAL Programs

In addition to the declarations already provided in the definition files, you must add these declarations to your TAL programs:

SPI Buffer

The ZSPIDEF.*subsys*TAL definition file for each NonStop Kernel subsystem includes a buffer declaration named *subsys*^DDL^MSG^BUFFER^DEF, which has the

structure described in “SPI Buffer” in Section 4. Use this declaration to allocate a buffer variable of the recommended size (*subsys*^VAL^BUFLEN):

```
STRUCT .buf(subsys^DDL^MSG^BUFFER^DEF);
```

Then you can easily refer to the Z^MSGCODE (-28), Z^BUFLEN, and Z^OCCURS fields of this structure as needed.

Some subsystems provide additional buffer declarations allocating different recommended buffer-size values for different commands. For details, see the individual subsystem management programming manuals.

If you wish to define a buffer larger than the recommended size in order to handle a large number of response records per reply, you can write your own buffer declaration, following the pattern of *subsys*^DDL^MSG^BUFFER^DEF.

Subsystem ID

In TAL, before the first time you call SSINIT to send a command to a subsystem, you must initialize the subsystem ID. For NonStop Kernel subsystems, the name of the subsystem ID structure in the TAL definition file is *subsys*^VAL^SSID^DEF. To initialize it, use ZSPI^VAL^TANDEM for the Z^OWNER field, ZSPI^SSN^*subsys* for the Z^NUMBER field, and *subsys*^VAL^VERSION for the Z^VERSION field. For example, if your application sends commands to TMF, your program would use the definitions supplied with TMF (in the file ZSPIDEF.ZTMFTAL) and could then declare and initialize the subsystem ID:

```
STRUCT .ZTMF^VAL^SSID (ZTMF^VAL^SSID^DEF);

ZTMF^VAL^SSID ':=' [ZSPI^VAL^TANDEM,ZSPI^SSN^ZTMF,
                                    ZTMF^VAL^VERSION];
```

If you are sending a command to a subsystem provided by a company other than HP, you must make the appropriate, different entries for the Z^OWNER, Z^NUMBER, and Z^VERSION fields.

Defining Token Maps

Defining and initializing token maps in TAL requires that you use the LITERAL and DEFINE names produced by DDL:

For every token map you define, the DDL name is

subsys-MAP-*name*

This name is translated to TAL as

```
DEFINE subsys^MAP^name = initialization-list-for-TAL-array #;
LITERAL subsys^MAP^name^WLN = n ;
```

where *n* is the size of the TAL array to be initialized.

Your TAL module must include this source directive and declaration:

```
?SOURCE subsysTAL
    .
    .
INT .user^chosen [0:subsys^MAP^name^WLN-1] := subsys^MAP^name ;
```

This code declares *user^chosen* and initializes it with the proper value for the token map. Then you can refer to *user^chosen* as the token code in SPI calls such as SSPUT and SSGET to operate on *subsys*-MAP-*name* tokens.

Interprocess Communication

An application requester written in TAL should use the file-system procedure WRITEREAD, rather than WRITE, to send the command. The program should specify a read count that is the length of the buffer as initialized with SSINIT.

If the subsystem to which you are sending commands honors assign and param messages, and you want to allow the user of your application to provide these to the server, your application must explicitly save these messages and pass them on.

Subsystems written in TAL should open $RECEIVE to receive SPI messages, and then use READUPDATE to read the messages and REPLY to send the replies.

SPI Procedure Syntax in TAL

These pages give the syntax and semantics of the SPI procedures SSGET and SSGETTKN, SSINIT, SSMOVE and SSMOVETKN, SSNULL, and SSPUT and SSPUTTKN. For descriptions of the SPI procedures and their parameters, see Section 3, The SPI Procedures.

Passing Token Parameters by Value or by Reference

The way your TAL applications pass tokens as parameters to the SPI procedures depends on the token and on personal preference: You must pass token maps, which refer to extensible structured tokens, by reference. Pass token codes by value or by reference. (Passing a token code by reference usually requires storing it in a temporary variable.)

Use the SSGET, SSMOVE, and SSPUT procedures to pass the *token-id* parameter by reference; use the SSGETTKN, SSMOVETKN, and SSPUTTKN procedures to pass it by value.

If you bind into the system library any routines that use SSGETTKN, SSPUTTKN, or SSMOVETKN, the Binder issues parameter mismatch warnings in SYSGEN. You can ignore these warnings. To eliminate them, use SSGET, SSPUT, and SSMOVE.

SSINIT

```
{ status := }  SSINIT   ( buffer                    ! o
{ CALL      }           , buffer-length             ! i
                        , ssid                       ! i
                        , header-type                ! i
                        , command                    ! i
                        , [ object-type       ]      ! i
                        , [ max-resp          ]      ! i
                        , [ server-version    ]      ! i
                        , [ checksum          ]      ! i
                        , [ max-field-version ] ) ; ! i
```

SSNULL

```
{ status := }  SSNULL   ( token-map                  ! i
{ CALL      }           , struct ) ;                 ! o
```

SSPUT and SSPUTTKN

The SSPUT and SSPUTTKN procedures are identical except for the type of the *token-id* parameter (SSPUT passes *token-id* by reference and SSPUTTKN passes it by value) and the consequent fact that SSPUTTKN cannot be used with a token map. In TAL programs, you can use SSPUTTKN when supplying a token code for the *token-id* parameter; doing so avoids the need to store the token code in a temporary variable before passing it to SSPUT. You must use SSPUT when supplying a token map.

```
{ status := }  { SSPUT    }   (   buffer             ! i/o
{ CALL      }  { SSPUTTKN }   ,   token-id           ! i
                              , [ token-value ]      ! i
                              , [ count ]            ! i
                              , [ ssid  ] ) ;        ! i
```

SSGET and SSGETTKN

The SSGET and SSGETTKN procedures are identical except for the type of the *token-id* parameter (SSGET passes *token-id* by reference and SSGETTKN passes it by value) and the consequent fact that SSGETTKN cannot be used with a token map. In TAL programs, you can use SSGETTKN when supplying a token code for the *token-id* parameter; doing so avoids the need to store the token code in a temporary variable before passing it to SSGET. You must use SSGET when supplying a token map.

```
{ status := }   { SSGET    }    ( buffer              ! i/o
{ CALL      }   { SSGETTKN }    , token-id            ! i
                                , [ token-value ]     ! i/o
                                , [ index       ]     ! i
                                , [ count       ]     ! i/o
                                , [ ssid        ] ) ; ! i/o
```

SSMOVE and SSMOVETKN

The SSMOVE and SSMOVETKN procedures are identical except for the type of the *token-id* parameter (SSMOVE passes *token-id* by reference and SSMOVETKN passes it by value) and the consequent fact that SSMOVETKN cannot be used with a token map. In TAL programs, you can use SSMOVETKN when supplying a token code for the *token-id* parameter; doing so avoids the need to store the token code in a temporary variable before passing it to SSMOVE. You must use SSMOVE when supplying a token map.

```
{ status := }   { SSMOVE    }    (   token-id          ! i
{ CALL      }   { SSMOVETKN }    ,   source-buffer     ! i/o
                                 , [ source-index  ]   ! i
                                 ,   dest-buffer       ! i/o
                                 , [ dest-index ]      ! i
                                 , [ count      ]      ! i/o
                                 , [ ssid       ] ) ; ! i
```

Examples

For example programs written in TAL, see Appendix E, SPI Programming Examples.

A Errors

This appendix lists all the error numbers defined by SPI. The two categories of errors are:

- Error numbers returned in the *status* parameter on calls to the SPI procedures (errors 0 through –12)
- General errors returned in SPI messages (errors –13 through –37)

The ZSPI errors are listed by number in Table A-1 and by name in Table A-2. When any of these errors (except error 0 or error –1) occurs, the header token ZSPI-TKN-LASTERR is set to the error number.

Table A-1. ZSPI Errors, by Number (page 1 of 2)

Number	Name	Meaning
0	ZSPI-ERR-OK	No error
–1	ZSPI-ERR-INVBUF	Invalid buffer format
–2	ZSPI-ERR-ILLPARM	Invalid parameter value
–3	ZSPI-ERR-MISPARM	Missing parameter
–4	ZSPI-ERR-BADADDR	Invalid parameter address
–5	ZSPI-ERR-NOSPACE	Buffer full
–6	ZSPI-ERR-XSUMERR	Invalid checksum
–7	ZSPI-ERR-INTERR	Internal error
–8	ZSPI-ERR-MISTKN	Token not found
–9	ZSPI-ERR-ILLTKN	Invalid token code or map
–10	ZSPI-ERR-BADSSID	Invalid subsystem ID
–11	ZSPI-ERR-NOTIMP	Operation not supported
–12	ZSPI-ERR-NOSTACK	Insufficient stack space
–13	ZSPI-ERR-ZFIL-ERR	File-system error
–14	ZSPI-ERR-ZGRD-ERR	Error from ZGRD procedure
–15	ZSPI-ERR-INV-FILE	Template file invalid
–16	ZSPI-ERR-CONTINUE	More text is available
–17	ZSPI-ERR-NEW-LINE	More text on new line
–18	ZSPI-ERR-NO-MORE	No more fields
–19	ZSPI-ERR-MISS-NAME	Name not found
–20	ZSPI-ERR-DUP-NAME	Name ambiguous
–21	ZSPI-ERR-MISS-ENUM	Enumeration not found
–22	ZSPI-ERR-MISS-STRUCT	Structure not found
–23	ZSPI-ERR-MISS-OFFSET	Offset not found

Table A-1. ZSPI Errors, by Number (page 2 of 2)

Number	Name	Meaning
–24	ZSPI-ERR-TOO-LONG	Text longer than maximum
–25	ZSPI-ERR-MISS-FIELD	Field past end of token value
–26	ZSPI-ERR-NO-SCANID	No scan ID available
–27	ZSPI-ERR-NO-FORMATID	No format ID available
–28	ZSPI-ERR-OCCURS-DEPTH	OCCURS nested too deep
–29	ZSPI-ERR-MISS-LABEL	No labeled dump info
–30	ZSPI-ERR-BUF-TOO-LARGE	Specified buffer size > maximum
–31	ZSPI-ERR-OBJFORM	Invalid object name format
–32	ZSPI-ERR-OBJCLASS	Invalid object class
–33	ZSPI-ERR-BADNAME	Invalid encoded name
–34	ZSPI-ERR-TEMPLATE	Encoded name is a template
–35	ZSPI-ERR-ILL-CHAR	Invalid character in name
–36	ZSPI-ERR-NO-TKNDEFID	No tkndef ID available
–37	ZSPI-ERR-INCOMP-RESP	Incomplete response

Table A-2. ZSPI Errors, by Name (page 1 of 2)

Name	Number	Meaning
ZSPI-ERR-BADADDR	–4	Invalid parameter address
ZSPI-ERR-BADNAME	–33	Invalid encoded name
ZSPI-ERR-BADSSID	–10	Invalid subsystem ID
ZSPI-ERR-BUF-TOO-LARGE	–30	Specified buffer size > maximum
ZSPI-ERR-CONTINUE	–16	More text is available
ZSPI-ERR-DUP-NAME	–20	Name ambiguous
ZSPI-ERR-ILL-CHAR	–35	Invalid character in name
ZSPI-ERR-ILLPARM	–2	Invalid parameter value
ZSPI-ERR-ILLTKN	–9	Invalid token code or map
ZSPI-ERR-INCOMP-RESP	–37	Incomplete response
ZSPI-ERR-INTERR	–7	Internal error
ZSPI-ERR-INV-FILE	–15	Template file invalid
ZSPI-ERR-INVBUF	–1	Invalid buffer format
ZSPI-ERR-MISPARM	–3	Missing parameter
ZSPI-ERR-MISS-ENUM	–21	Enumeration not found
ZSPI-ERR-MISS-FIELD	–25	Field past end of token value
ZSPI-ERR-MISS-LABEL	–29	No labeled dump info

Table A-2. ZSPI Errors, by Name (page 2 of 2)

Name	Number	Meaning
ZSPI-ERR-MISS-NAME	–19	Name not found
ZSPI-ERR-MISS-OFFSET	–23	Offset not found
ZSPI-ERR-MISS-STRUCT	–22	Structure not found
ZSPI-ERR-MISTKN	–8	Token not found
ZSPI-ERR-NEW-LINE	–17	More text on new line
ZSPI-ERR-NO-FORMATID	–27	No format ID available
ZSPI-ERR-NO-MORE	–18	No more fields
ZSPI-ERR-NO-SCANID	–26	No scan ID available
ZSPI-ERR-NO-TKNDEFID	–36	No tkndef ID available
ZSPI-ERR-NOSPACE	–5	Buffer full
ZSPI-ERR-NOSTACK	–12	Insufficient stack space
ZSPI-ERR-NOTIMP	–11	Operation not supported
ZSPI-ERR-OBJCLASS	–32	Invalid object class
ZSPI-ERR-OBJFORM	–31	Invalid object name format
ZSPI-ERR-OCCURS-DEPTH	–28	OCCURS nested too deep
ZSPI-ERR-OK	0	No error
ZSPI-ERR-TEMPLATE	–34	Encoded name is a template
ZSPI-ERR-TOO-LONG	–24	Text longer than maximum
ZSPI-ERR-XSUMERR	–6	Invalid checksum
ZSPI-ERR-ZFIL-ERR	–13	File-system error
ZSPI-ERR-ZGRD-ERR	–14	Error from ZGRD procedure

0: ZSPI-ERR-OK

Cause. Successful operation.

Effect. The requested operation is completed.

Recovery. No recovery is necessary.

–1: ZSPI-ERR-INVBUF

Cause. The buffer supplied in the procedure is considered to be improperly formatted for one of these reasons:

- The first word of the buffer does not contain the SPI message code (ZSPI-VAL-MSGCODE ¦ –28).
- The current version of SPI does not recognize the buffer format.

- The length of the used portion of the buffer (ZSPI-TKN-USEDLEN) is greater than the maximum buffer length (Z-BUFLEN). SSPUT might have been called with ZSPI-TKN-RESET-BUFFER and a *maxlen* value that was smaller than ZSPI-TKN-USEDLEN.
- The buffer contains ZSPI-TKN-ENDLIST but no corresponding list token.
- The position descriptor within the buffer (ZSPI-TKN-POSITION) indicates a current list that does not begin with a list token. Perhaps an incorrect position descriptor—for instance, one saved from another buffer—was restored to ZSPI-TKN-POSITION.

Effect. The requested operation is not completed. Because the buffer format is invalid, the last error is not saved.

Recovery. Check for an incorrect *buffer* parameter or a corrupted buffer; or if this error was returned from a call to SSGET with ZSPI-TKN-RESET-BUFFER, check for a short read and allocate a larger buffer.

–2: ZSPI-ERR-ILLPARM

Cause. An invalid parameter was supplied in the procedure call. Possibilities include:

- SSINIT was called with an invalid header type.
- A negative *index* or *count* parameter was supplied.
- An attempt was made to use SSPUT or SSGET on a token using a *count* of zero.
- A call was made to one of the special operations that return attributes of a token, but the operation was applied to the special token itself. The special tokens that return attributes are ZSPI-TKN-COUNT, ZSPI-TKN-LEN, ZSPI-TKN-OFFSET, and ZSPI-TKN-ADDR.
- The program supplied an SPI-defined token code that was invalid for this procedure call. For example, ZSPI-TKN-DELETE was specified in a call to SSGET, or ZSPI-TKN-COMMAND was specified in a call to SSPUT.
- An invalid position descriptor was supplied with ZSPI-TKN-POSITION.

Effect. ZSPI-TKN-LASTERR is set to ZSPI-ERR-ILLPARM and the requested operation is not completed.

Recovery. Correct the parameter in error.

–3: ZSPI-ERR-MISPARM

Cause. This error number indicates that a required parameter was not supplied. Certain parameters are required only under certain circumstances:

- The *ssid* parameter is required when calling SSGET with ZSPI-TKN-NEXTCODE or ZSPI-TKN-NEXTTOKEN if the next token code in the buffer is not qualified by the default subsystem ID. Always supply a variable for *ssid* when calling SSGET

with ZSPI-TKN-NEXTCODE or ZSPI-TKN-NEXTTOKEN unless you are certain that all tokens the program could encounter are qualified by the default subsystem ID.

- The *value* parameter is required when calling SSGET with certain standard token codes (such as ZSPI-TKN-LEN and ZSPI-TKN-OFFSET) or when calling SSPUT with a token code that has a value (a nonzero token length).

Effect. ZSPI-TKN-LASTERR is set to ZSPI-ERR-MISPARM (unless the *buffer* parameter is missing), and the requested operation is not performed.

Recovery. Supply the missing parameter.

–4: ZSPI-ERR-BADADDR

Cause. A reference parameter has an invalid address. Possibilities include:

- A parameter has a starting address that is invalid or out of bounds.
- A parameter has an absolute extended address, and the caller is nonprivileged.
- A parameter's starting address and length are such that the parameter overlaps the stack space required by the SPI procedure.

Effect. The header token ZSPI-TKN-LASTERR is set to this error number (unless the bounds error occurs on the *buffer* parameter), and the requested operation is not performed. If the bounds error occurs on the *buffer* parameter, SPI is unable to find the buffer, so it cannot set ZSPI-TKN-LASTERR.

Recovery. Correct the parameter declarations to allocate the required amount of storage.

–5: ZSPI-ERR-NOSPACE

Cause. One of:

- The buffer is full; it cannot contain any more tokens or header information.
- SSPUT was called with ZSPI-TKN-RESET-BUFFER, but *maxlen* was smaller than the used length of the buffer. In this case, some information at the end of the message was lost. Subsequent SPI calls for this buffer return error –1 (invalid buffer format).

Effect. The last error is set to this error number, and the requested operation is not completed.

Recovery. For the first, use a larger buffer. For the second, the corrective action depends on the application.

–6: ZSPI-ERR-XSUMERR

Cause. The current buffer checksum does not match the checksum computed on return from the last SPI call. This error suggests that the buffer has been modified or damaged.

Effect. The last error is set to this error number, and the requested operation is not completed.

Recovery. Using a debugging tool such as Inspect, check for inadvertent corruption of the buffer contents.

–7: ZSPI-ERR-INTERR

Cause. This internal error should not occur unless the SPI software malfunctions. Specific causes include:

- SSGET attempted to return a token value when the program had requested a token attribute (such as length, offset, address, or count).
- SSGET attempted to return an undefined token attribute.
- On returning, SSGET or SSPUT attempted to set a used length greater than the buffer length.
- SSPUT received an error when calling SSGET to obtain the default subsystem ID (ZSPI-TKN-DEFAULT-SSID) from the SPI message header.

Effect. The last error is set to this error number, and the requested operation is not completed.

Recovery. Report the problem to your HP representative, supplying a reproducible test case.

–8: ZSPI-ERR-MISTKN

Cause. One of:

- The token requested in a call to SSGET was not found.
- An attempt was made to put a ZSPI-TKN-ENDLIST token into the buffer, but no corresponding list token was found.

Effect. The last error is set to this error number, and the requested operation is not completed.

Recovery. Corrective action depends on the application. Check for program logic errors in scanning the buffer. Also check to see if the token is positioned outside the current list or preceding the current position.

–9: ZSPI-ERR-ILLTKN

Cause. An invalid token code or token map was supplied in the procedure call. The possibilities include:

- The token data type was not recognized. For example, a program used a token data type not included in the SPI standard definitions. (The only token data types permitted by SPI are those defined by SPI.)
- The token length was not a multiple of the basic length associated with the token data type.
- The token map contained an invalid null-value specification.
- The sum of the lengths of the null-value specifications in the token map was not equal to the total structure length specified by the map.
- A token map was supplied as a parameter to SSGETTKN, SSPUTTKN, or SSMOVETKN.

Any of these situations might arise if the program accidentally corrupted the variable holding the token code or token map.

Effect. The last error is set to this error number, and the requested operation is not completed.

Recovery. Correct the token code or token map causing the error.

–10: ZSPI-ERR-BADSSID

Cause. A subsystem ID with an invalid name was supplied as a parameter. The owner-name field of a subsystem ID must contain an owner name that

- Begins with an alphabetic character
- Contains only alphabetic characters, hyphens, or numeric characters
- Is left-justified and padded on the right with blanks

Effect. The last error is set to this error number, and the requested operation is not completed.

Recovery. Check the *ssid* parameter being supplied in the call, and correct it as necessary.

–11: ZSPI-ERR-NOTIMP

Cause. This operation is not supported in the version of the SPI definitions being used. (For instance, the operation ZSPI-TKN-ADDR was called to get the address of a header token.)

Effect. The last error is set to this error number, and the requested operation is not completed.

Recovery. Check that the token code being supplied in the call is defined for the version of the SPI definitions being used.

–12: ZSPI-ERR-NOSTACK

Cause. An SPI procedure was called with fewer than 768 words of data stack left.

Effect. The last error is set to this error number, and the requested operation is not completed.

Recovery. Increase the number of stack pages available or reduce the amount of stack space used. The methods available for doing this depend on the programming language.

–13 Through –37: General SPI Errors

These error numbers report general SPI errors. The precise interpretation of each error depends on the subsystem or procedure that generates the error. See the description provided by the generating subsystem.

B Summary of DDL for SPI

This appendix reviews features of DDL that pertain to SPI data definitions. If you are not familiar with DDL, this information will help you read and understand the DDL source code in this and other related manuals.

If you are already familiar with DDL, parts of this appendix will give you an overview of the clauses and statements added to DDL to support SPI:

If you are writing your own subsystem, you will need more information than is given in this appendix. For more detailed information about DDL and its syntax, see the *Data Definition Language (DDL) Reference Manual*.

The Role of DDL in SPI

DDL is a program and database development tool with capabilities that include creating database schemas, creating data dictionaries, generating FUP commands necessary to create the corresponding database, revising the source schemas and data dictionaries, and generating programming-language source code for data declarations that correspond to the source schemas.

For SPI, DDL is used to generate equivalent data declarations (definitions) in different programming languages—TAL, C, COBOL, and TACL. Because the data declarations are equivalent, application program modules coded in one language can communicate with HP modules and other application modules coded in another language.

HP provides definition files containing data declarations used both by HP software (including SPI and NonStop Kernel subsystems) and by applications that use SPI. For each set of declarations, HP supplies equivalent definition files in TAL, C, COBOL, TACL, and DDL. The TAL, C, COBOL, and TACL definition files are generated by the DDL compiler, which translates the DDL source code into these other languages.

This manual and others describing programmatic interfaces based on SPI describe the various data structures related to SPI messages containing commands, responses, error lists, and event descriptions. Because these data structures can be in any one of four languages, the manuals use DDL source code from the definition files as a common notation to define the data structures.

You can make the information in the DDL definition file available to the system procedures that derive display text from SPI messages. The DDL information helps the EMSTEXT procedure and the SPI_BUFFER_FORMAT procedures produce more readable display text. EMSTEXT presents event messages to operators. The SPI_BUFFER_FORMAT procedures help Inspect to display SPI messages.

To make DDL information available at run time to the formatting procedures, you write a template source file. This file enables the template compiler to encode DDL-clause information in template form. For information about how to write and compile a template source file, see the *DSM Template Services Manual*.

General Language Rules for DDL

In DDL (as in TAL, COBOL, and TACL, though not in C), alphabetic characters in names and keywords are not case-sensitive; that is, corresponding uppercase and lowercase letters are equivalent. The convention used in the SPI manuals was chosen to emphasize the SPI definition names, because these names, rather than the DDL keywords, are of primary importance. The DDL keywords here merely serve as part of the notation.

Periods in DDL serve as separators. They separate one statement from the next; they also separate subdivisions of a DEF statement. Blanks, carriage returns, and tab characters can occur within a statement or a statement subdivision. Therefore, a statement or statement subdivision can continue over several lines.

DEFINITION (DEF) Statement

The DEFINITION statement (shortened here to its legal abbreviation DEF) defines the structure of a data item or a group of items. It specifies the name, data type, and other characteristics of each data item or group. The DDL compiler translates a DEF statement into a declaration of an equivalent data structure in TAL, C, COBOL, and TACL.

The examples in Figure B-1 on page B-3 show the use of the DEF statement, and are referred to in descriptions throughout this section.

Figure B-1. DEF Statement Examples

```
1   def   ZSPI-DDL-BOOLEAN   type   logical   spi-null   "  " .


2   def   ZSPI-DDL-ENUM   pic   s9(4)   comp   spi-null   255
                                                  tacl   enum .


3   def   ZFUP-DDL-PART-RENAME-OPTS .
        02   Z-PART-ONLY          type   ZSPI-DDL-INT .
        02   Z-PART-NAME          type   ZSPI-DDL-DEVICE .
        02   Z-PRIEXT-SIZE        type   ZSPI-DDL-INT2   spi-null   255 .
        02   Z-SECEXT-SIZE        type   ZSPI-DDL-INT2   spi-null   255 .
    end .


4   def   ZSPI-DDL-CHAR5
        02   Z-C                  pic   x(5)              spi-null   "  " .
        02   Z-S   redefines Z-C .
             03   Z-I             type binary 16          occurs 2 times .
             03   filler          pic   x .
        02   Z-B   redefines Z-C  pic   x                occurs 5  times .
    end .
```

VST022.vsd

Examples 3 and 4 in Figure B-1 are group DEF statements, which define internal data structures, including component fields or groups of fields. Each field or group of fields is described by a subdivision of the DEF statement, starting with a level number such as 02 or 03 and ending with a period. The level numbers establish a hierarchy of fields or groups of fields. The keyword END marks the end of the group DEF statement.

DEF statements contain clauses beginning with DDL keywords such as TYPE, PIC, OCCURS, REDEFINES, FILLER, SPI-NULL, TACL, SSID, HEADING, and DISPLAY:

TYPE Clause

The TYPE clause defines the data type and size, or the internal structure, associated with a DEF or with a field in a group DEF. This clause consists of the keyword TYPE followed by either the keyword of a DDL type (such as LOGICAL or BINARY 16) or the name of another DEF that was given earlier (such as ZSPI-DDL-INT).

If the name of another DEF appears in the TYPE clause, the DEF containing the TYPE clause defines its data structure, or a portion of its data structure, based on another DEF statement. (This is called the reference form of the DEF statement.)

In DDL source code, a DEF statement referred to in a TYPE clause must precede the DEF statement that refers to it. In manuals, likewise, DEF statements are usually presented so that each DEF precedes all the DEFs that refer to it. Subsystem DEFs referred to by other subsystem DEFs usually appear in the “Common Definitions” section of the appropriate subsystem management programming manual.

For instance, in example 3 in Figure B-1, the definition ZFUP-DDL-PART-RENAME-OPTS refers to the definitions ZSPI-DDL-INT, ZSPI-DDL-DEVICE, and ZSPI-DDL-INT2. The definition ZFUP-DDL-PART-RENAME-OPTS is described in the *File Utility Program (FUP) Management Programming Manual*. The three definitions to which it refers are standard SPI definitions contained in the SPI standard definition file that you must source in, copy in, or load (depending on your programming language). These standard definitions are described in Section 4, ZSPI Data Definitions.

When the DDL compiler translates a reference-form DEF statement into TAL, C, COBOL, or TACL, the structures are combined, and the resulting programming-language structure declaration reflects the information in all the DEFs to which the translated DEF refers, directly or indirectly. (In TACL, an exception to this rule is when the TACL clause appears. For more information, see TACL Clause on page B-5.)

PICTURE (PIC) Clause

The PICTURE clause (shortened here to its legal abbreviation PIC) defines the data type and size associated with a DEF or with a field in a group DEF. The notation that follows the PIC keyword corresponds to the PICTURE (PIC) notation in COBOL. A DEF or field defined by a PIC clause translates to an ASCII character string with certain characteristics defined by the notation. The TYPE and PIC clauses are mutually exclusive for a particular variable or field.

OCCURS Clause

The OCCURS clause specifies a subscripted array of like fields or groups. In example 4 in Figure B-1, the level-03 field Z-I represents an array of two 16-bit signed integers. Likewise, the level-02 field Z-B represents an array of five ASCII characters.

REDEFINES Clause

The REDEFINES clause assigns a new name and, optionally, a new structure to a previously defined data storage area. REDEFINES clauses are used in the SPI definitions to allow variables to be addressed in several different ways depending on the programming language and the needs of the program. For instance, example 4 in Figure B-1 is an array of 5 ASCII characters that can be addressed as a TAL or TACL STRUCT, as five bytes, or as two integers. (If the array is addressed as two integers, the fifth byte is inaccessible because it is an unnamed filler byte.)

FILLER Clause

A clause beginning with the keyword FILLER specifies an unnamed place-holder field. The definition files include FILLER clauses where needed to ensure alignment of adjacent fields on word boundaries. Example 4 in Figure B-1 illustrates the use of the FILLER clause.

SPI-NULL Clause

When an SPI-NULL clause appears in a DEF or a field of a DEF, it specifies the null value to be assigned to a variable or structure field based on that DEF when a program calls the SSNULL procedure. Example 1 in Figure B-1 directs SSNULL to assign a blank to each byte of each field of a structure defined as type ZSPI-DDL-BOOLEAN. The second example directs SSNULL to assign the byte value 255 (all binary ones) to each byte of each field of a structure defined as type ZSPI-DDL-ENUM.

An SPI-NULL clause specified in a group DEF is inherited by each of the fields within the group. When you reference one DEF from another, any SPI-NULL clauses in the referring DEF override corresponding SPI-NULL clauses in the referenced DEF.

If a field has no SPI-NULL clause, SSNULL assigns 255 to that field.

TACL Clause

When a TACL clause appears in a DEF or a field of a DEF, it names one of the high-level TACL data types that can be used in simple data-item declarations within TACL STRUCT statements. The DDL compiler then translates that field or structure into a STRUCT field of the corresponding type. Examples of high-level TACL data types that can appear in the TACL clause are ENUM, CRTPID, DEVICE, and SSID; Section 8 gives the complete list.

If a DEF or field includes a TACL clause but also refers to another DEF that itself includes a TACL clause, the TACL clause in the higher level DEF overrides the clause in the referenced DEF.

Example 2 in Figure B-1 directs DDL, when producing TACL output, to translate the definition ZSPI-DDL-ENUM, or any field defined by reference to ZSPI-DDL-ENUM, to a simple data item field of type ENUM.

SSID Clause

This DDL clause identifies the subsystem associated with a token code or token map. When the template compiler compiles the template source file for a subsystem, it copies the token information from the DDL dictionary to templates if the token code or token map contains an SSID clause that refers to the subsystem. Without SSID, the template compiler and the SPI_BUFFER_FORMAT procedures lose access to token names and to other DDL clauses, limiting the display text they produce.

HEADING Clause

This clause provides the SPI_BUFFER_FORMAT procedures and Inspect with information they need to label a token or field value when displaying the contents of an SPI message.

DISPLAY Clause

This clause provides the edit code that is used when a token or field value is represented in display text. Include this clause when the DISPLAY clause might make the value more readable or when the default edit code is not suitable for the token or field.

Constants

DDL allows the definition of named constants by means of the CONSTANT statement. DDL translates CONSTANT statements into literals or defines in TAL, **#define** directives in C, level-01 variables in COBOL, and text variables in TACL. This feature lets the definition files provide symbolic names for frequently used values.

The definition files supplied by HP use CONSTANT statements to define subsystem numbers (in the SPI definitions only), token data types (in the SPI definitions only), token numbers, command numbers, object-type numbers, error numbers, and various other commonly used values.

Type ENUM DEFs

Many integer tokens or fields contain code values with text-string equivalents (for example, 0, 1, and 2 for stopped, slow, and fast). You can define such a token as a type ENUM DEF and place the text strings in AS clauses. Then text strings that represent the integer values can be returned to system procedures that refer to templates.

Token Types, Token Codes, and Token Maps

Token types, token codes, and token maps are defined in DDL by TOKEN-TYPE, TOKEN-CODE, and TOKEN-MAP statements, respectively.

In descriptions of commands, responses, event messages, error lists, and common definitions in the manuals, the internal structure and component data types of each extensible structured token are shown by giving the associated DEF statement, which includes a TYPE or PIC clause for each field. Similar information for each simple token is indicated by including information extracted from the associated TOKEN-CODE statement, but in a form similar to that of part of a DEF statement. For instance, the simple token ZSPI-TKN-CONTEXT is listed in command descriptions as:

```
ZSPI-TKN-CONTEXT     token-type ZSPI-TYP-BYTESTRING.
```

This notation is not DDL code, but is a shortened notation designed to give the information needed. For instance, the example above reflects this TOKEN-CODE statement in the definition file ZSPIDEF.ZSPIDDL:

```
TOKEN-CODE zspi-tkn-context   VALUE IS zspi-tnm-context
                         TOKEN-TYPE IS zspi-typ-bytestring.
```

DDL Data Translation

The DDL compiler can translate any DEF statement into data-declaration source code in TAL, C, COBOL, or TACL. The only restriction is that not all data types are supported in all four languages. Whenever a declared data type is not supported in a particular language, DDL attempts to translate the data type into a declaration of a compatible data type. For example, DDL structures described with PIC X or PIC 9 clauses are translated into STRING BYTE types in TAL; a structure described as PIC S9(4) COMP is translated into a TAL or TACL INT data type or a COBOL NATIVE-2 data type.

When no compatible data type is available, DDL translates the data type into a character-string declaration. For example, a structure described as TYPE FLOAT, which is the REAL data type in TAL, is translated into PIC X(4) in COBOL.

For additional examples of DDL DEF source and the resulting output in TAL, C, COBOL, and TACL, see the information about DDL data translation in the *Data Definition Language (DDL) Reference Manual*.

C SPI Internal Structures

This appendix describes these internal formats of SPI data structures:

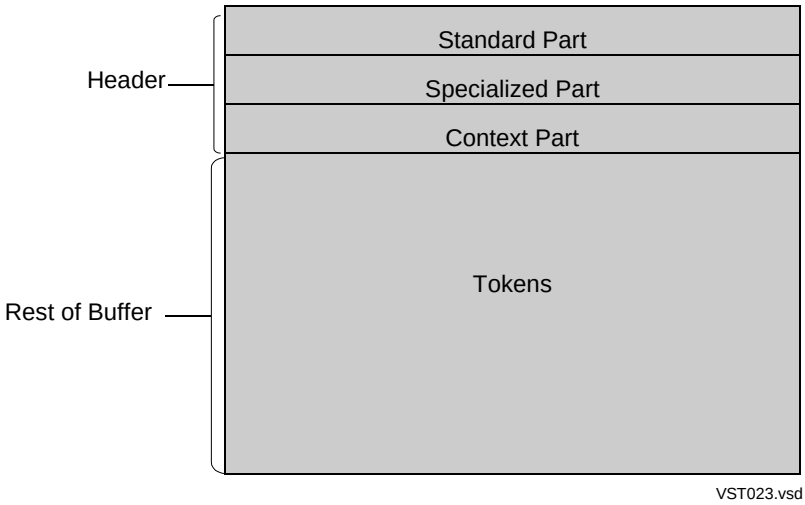
The detailed formats are expressed as TAL structures. This information is for debugging purposes only.

Note. The information in this appendix might change in future RVUs. Application programs and subsystems you write should not depend on these formats. Always work with these structures using their symbolic names with the SPI procedures.

SPI Buffer Format

An SPI buffer consists of an SPI message header and zero or more buffer tokens (tokens that are not header tokens). The header consists of three parts: the standard part, the specialized part, and the context part.

Figure C-1. SPI Buffer Format



Each buffer token consists of a token code and a token value, as described in Section 2, SPI Concepts and Protocol.

The three parts of the header contain information that includes the values of the header tokens. Each of the three parts is implemented as a structure, and the values of the header tokens in that part are fields in the structure. Thus the header tokens are

not implemented as true tokens—they are identified in the buffer by their positions rather than by their token codes. However, application programs and subsystems you write retrieve and change the values of header tokens using SSGET and SSPUT as if they were true tokens.

Standard Part of Header

The first data structure in the buffer contains the standard part of the header. This structure contains information that is common to all SPI buffers. For example:

```
STRUCT ZSPI^DDL^STDHDR^DEF(*);
BEGIN
   INT     Z^MSGCODE;              ! -28
   INT     Z^BUFLEN;               ! Buffer length in bytes
   INT     Z^OCCURS;               ! # bytes in rest of buffer
   STRUCT  Z^SSID;                 ! Subsys ID used w/SSINIT
   BEGIN
      INT     Z^SSOWNER[0:3];      ! Subsystem owner
      INT     Z^SSNUM;             ! Subsystem number
      INT     Z^SSVERSION;         ! Subsystem version
   END;
   INT     Z^XSUM;                 ! Buffer checksum word
   INT     Z^MAX^FIELD^VERSION;    ! Maximum field version
   INT     Z^FLAGS;                ! Flag word
   STRUCT  Z^FLAG = ZFLAGS;
   BEGIN
      UNSIGNED(1) Z^XSUM;          ! .<0>     = Checksum flag
      BIT_FILLER 11;               ! .<1:11>  = Reserved
      UNSIGNED(4) Z^VERSION;       ! .<12:15> = SPI version
   END;
END;
```

The first field, called Z-MSGCODE, is a signed integer whose value is always -28; this number identifies the message as an SPI buffer. Z-BUFLEN is an unsigned integer field whose value is the maximum buffer length—that is, the length in bytes of the entire buffer as declared. Z-OCCURS is an unsigned integer included to facilitate the declaration of COBOL I/O buffers.

As programs using the SPI procedures see it, the remaining portion of the buffer contains tokens. However, the declared length of the buffer might not always be filled with token information. The used length of the buffer is the length in bytes of the currently used portion. Programs can call the SSGET procedure with the token code ZSPI-TKN-USEDLEN to get the used length.

Programs should call the SPI procedures to perform all operations on the buffer; they should not attempt to modify the buffer directly. The SPI procedures return an error if a program attempts to modify some parts of the buffer directly. Although programs can modify the Z-BUFLEN field, it is recommended that they not do so, but instead call SSPUT with the token code ZSPI-TKN-RESET-BUFFER to reset the maximum buffer length when they receive an SPI buffer from another process.

Specialized Part of Header

The standard part of the header is followed by the specialized part, a structure that differs depending on the header type. To users of the SPI procedures, the header type is a header token with an unsigned integer value of either 0 for a standard command header, or 1 for an event-message header. Internally, the header type is identified by a token code that appears in the first two words of the specialized part. This token code has token data type ZSPI-TDT-STRUCT, a token length equal to the length of the remainder of the specialized part (not including the token code itself), and a token number of either ZSPI-TNM-CMDHDR or ZSPI-TNM-EVTHDR. (The two token-number definitions are part of the SPI standard definition files but are not described in Section 4, ZSPI Data Definitions. They are for HP internal use only.)

Specialized Part of Standard Command Header

The structure for the specialized part of a standard command header (for a command or response) is:

```
STRUCT ZSPI^DDL^CMDHDR^DEF(*);
BEGIN
   INT(32) Z^TKNCODE;            ! Token code for standard
   INT Z^TKNTYPE = Z^TKNCODE;    ! command header
   STRUCT  Z^TKN = Z^TKNCODE;
   BEGIN
      STRING      Z^DATATYPE;    ! Token data type =
                                 !    ZSPI-TDT-STRUCT
      STRING      Z^BYTELEN;     ! Token length = length of
                                 !    specialized part
      INT         Z^NUMBER;      ! Token number =
                                 !    ZSPI-TNM-CMDHDR
   END;
   INT     Z^COMMAND;            ! Command number
   INT     Z^OBJECT^TYPE;        ! Object-type number
   INT     Z^MAXRESP;            ! Maximum responses
   INT     Z^SERVER^VERSION;     ! Server version
END;
```

Specialized Part of Event-Message Header

Like the specialized part of a standard command header, the specialized part of an event-message header begins with the token code, described earlier, that identifies the header type. For an event-message header, the token number of this token code is ZSPI-TNM-EVTHDR. The structure for the specialized fields *following* this token code—for event messages, the declared structure does not include the token code—is:

```
STRUCT ZEMS^DDL^EVENTHDR^DEF(*);
BEGIN
   INT     ZEVENT^NUMBER;         ! Event number
   FIXED   ZGENTIME;              ! Event generation time
   FIXED   ZLOGTIME;              ! Logging time
   STRUCT  ZUSERID;               ! User ID of generator
   BEGIN
```

```
      STRING  ZGROUP;
      STRING  ZUSER;
      INT     ZUSERGROUP = ZGROUP;
   END;
   INT     ZSYSTEM;
   STRUCT  ZCRTPID;
   BEGIN
      INT     ZNAME [0:2];
      STRING  ZCPU,
              ZPIN;
   END;
   INT     ZFLAGS;               ! Console print, emphasis
END;
```

For more information about event messages, see the *EMS Manual*.

Context Part of Header

This structure, if present, follows the specialized part and contains context information used by SSGET and SSPUT, including position descriptors and SPI error information.

The context part of the header is needed only while the SPI buffer is being built (with SSPUT or SSMOVE calls) or examined (with SSGET or SSMOVE calls). Under some circumstances, HP software can save space and communication time by deleting this part of the header when it is not needed and then adding it again when it is needed. This is one reason that programs must never assume that an SSGET call leaves the buffer unchanged. It also explains why a call to SSGET can result in an SPI error -5 (buffer full).

```
STRUCT ZSPI^DDL^CONTEXT^HDR^DEF(*);
BEGIN
   INT (32) Z^TKNCODE;     ! Token code for context part
   INT     Z^TKNTYPE = Z^TKNCODE;   !    fields
   STRUCT  Z^TKN = Z^TKNCODE;
   BEGIN
      STRING      Z^DATATYPE;  ! Token data type
      STRING      Z^BYTELEN;   ! Token byte length
      INT         Z^NUMBER;    ! Token number
   END;
   INT     Z^LASTERR;      ! Last nonzero status returned
   INT(32) Z^LASTERRCODE;  ! Token code used on call
                           !    resulting in Z^LASTERR
   STRUCT  Z^POS;          ! Position info for SSGET
   BEGIN
      INT     Z^LIST;      ! Byte offset to current list
      INT     Z^CURTKN;    ! Byte offset to current token
      INT     Z^INDEX;     ! Current index
   END;
   INT     Z^LASTLIST;     ! Byte offset to current SSPUT list
   INT     Z^LASTTOKEN;    ! Byte offset to the last token
END;
```

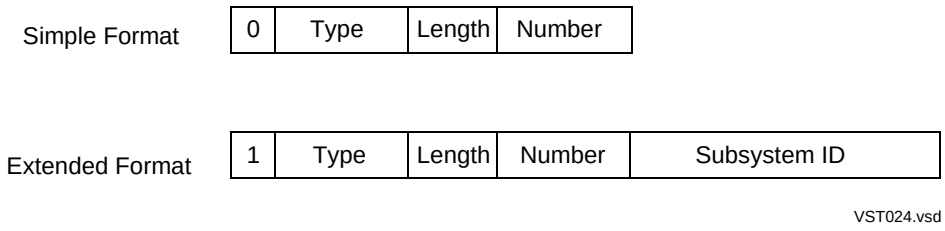
Token Structure

A token consists of a token code (qualified by a subsystem ID) and a token value.

Token Code

The token code is stored in either a simple format (when qualified by the default subsystem ID), or an extended format (when qualified by a subsystem ID other than the default).

Figure C-2. Internal Format of Token Code



The token-code structure is:

```
STRUCT ZSPI^DDL^TKNCODE^DEF(*);
BEGIN
   INT(32) Z^TKNCODE;
   INT     Z^TKNTYPE = Z^TKNCODE;
   STRUCT  Z^TKN = Z^TKNCODE;
   BEGIN
      UNSIGNED(1) Z^EXTENDED;    ! True if extended format
      UNSIGNED(7) Z^DATATYPE;    ! Token data type
      UNSIGNED(8) Z^BYTELEN;     ! Token length
      INT         Z^NUMBER;      ! Token number
   END;
   STRUCT  Z^TKNSSID;            ! Optional subsystem ID
   BEGIN
      INT   Z^SSOWNER[0:3];
      INT   Z^SSNUM;
      INT   Z^SSVERSION;
   END;
END;
```

If the token-length field (ZTKN^BYTELEN) of the token code is less than 255, it specifies the length in bytes of the associated token value. If the token-length field is

255, then the first word of the value gives the (noninclusive) length in bytes of the rest of the value.

Single-Occurrence Tokens

Single occurrences of token codes and token values are stored contiguously in the SPI buffer (see Figure C-3). A padding byte is emitted where necessary so that each token code begins on a word boundary.

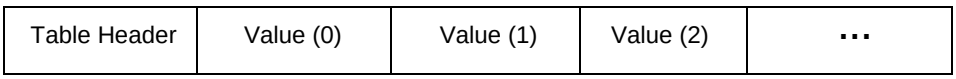
Figure C-3. Single-Occurrence Tokens as Stored in the Buffer

VST025.vsd

Multiple-Occurrence Tokens

To save space, multiple occurrences of the same token code are prefixed by a table header (see Figure C-4). The first two words of the table header are a token code with a token type of ZSPI-TYP-SSTBL (consisting of token data type ZSPI-TDT-SSTBL and token length 255) and a token number of ZSPI-TNM-TBLHDR. (The definitions just named are in the SPI standard definition files but are not described in Section 4, ZSPI Data Definitions. They are for HP internal use only.)

Figure C-4. Multiple-Occurrence Tokens as Stored in the Buffer



VST026.vsd

The table header for multiple-occurrence tokens has this structure:

```
STRUCT ZSPI^DDL^TBLHDR^DEF(*);
BEGIN
   INT(32)    Z^TBL^TKNCODE;     ! Token code for table
   INT        Z^TBL^BYTELEN;     ! Length of table (in bytes)
   INT        Z^TBL^COUNT;       ! Number of occurrences
   STRUCT     Z^TBL^TKN[0:-1];   ! Token code for tbl values
   BEGIN
      INT(32) Z^TKNCODE;
      STRUCT  Z^TKN = Z^TKNCODE;
      BEGIN
```

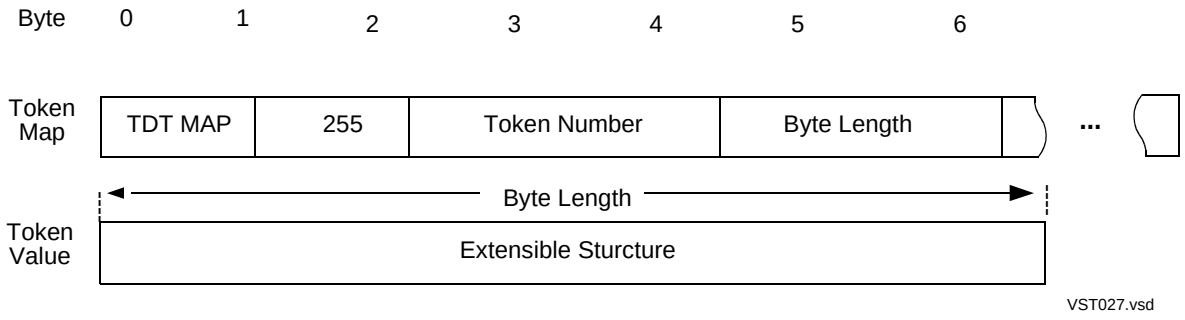
```
            UNSIGNED(1) Z^EXTENDED;    ! True if extended-
                                       !   format token code
            UNSIGNED(7) Z^DATATYPE;    ! Token data type
            UNSIGNED(8) Z^BYTELEN;     ! Token length
            INT         Z^NUMBER;      ! Token number
         END;
         STRUCT  Z^TKN^SSID;  ! Optional subsystem ID
         BEGIN
            INT   Z^SSOWNER[0:3];
            INT   Z^SSNUM;
            INT   Z^SSVERSION;
         END;
      END;
END;
```

This table header is followed by Z^TBL^COUNT values of the type associated with Z^TBL^TKN. (See Figure C-4 on page C-6.) Fixed-length string values are aligned on byte boundaries; all other types of values are aligned on word boundaries.

Token-Map Structure

Token maps are used to describe version and null-value information for the values of extensible structured tokens. The format of a token map and its associated token value are shown in Figure C-5.

Figure C-5. Token Map and Its Token Value



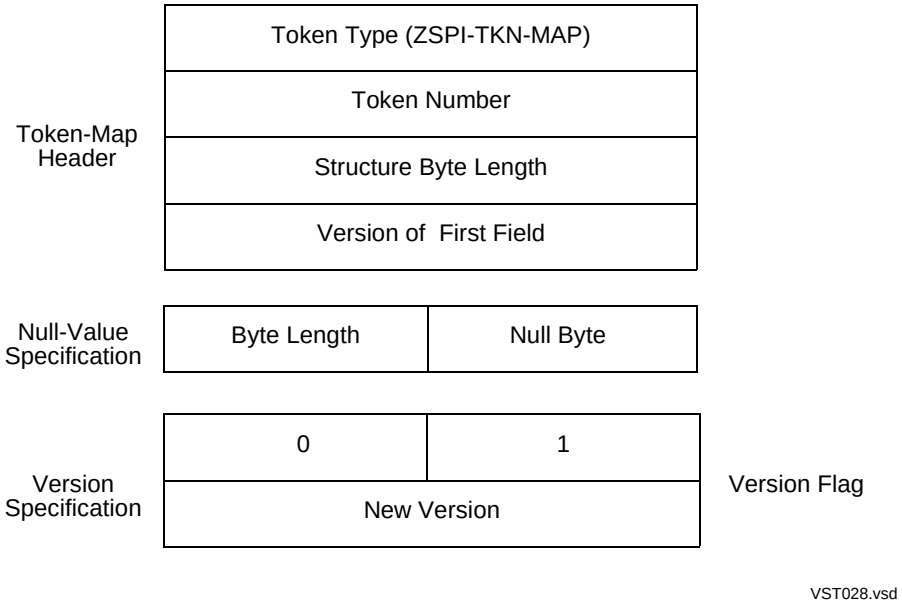
The token map contains:

- A token type of ZSPI-TYP-MAP (consisting of the token data type ZSPI-TDT-MAP and the token length 255)
- A subsystem-defined token number
- The byte length of the structure and its base version number (the version of the first field in the structure)

- One or more null-value specifications
- As new fields are added, additional version specifications for the new fields

In more detail, a token map can be seen as a token-map header followed by one or more null-value specifications and zero or more version specifications. Figure C-6 shows the formats of the token-map header, a null-value specification, and a version specification.

Figure C-6. Structures Within a Token Map



Null-value specifications and version specifications can occur in any order and can be intermixed. The end of the map is indicated when the byte lengths of the null-value specifications add up to the total byte length of the structure value.

The SSNULL procedure constructs the null value of the structure by processing the null-value specifications in the order that they appear in the map. For each null-value specification, SPI copies the indicated null value into the next *n* bytes of the structure, where *n* is the byte length from the null-value specification. The version of any byte in the structure is that of the most recently encountered version when the null-construction process reaches that byte of the structure.

Token-Map Example

These DDL definitions:

```
DEFINITION zbat-ddl-jobinfo.
  02  znumber      TYPE zspi-ddl-int.
  02  zpriority    TYPE zspi-ddl-int.
  02  zlocation    TYPE zspi-ddl-char8  SPI-NULL "X".
END
```

```
CONSTANT  zbat-tnm-jobinfo   VALUE IS 63.

TOKEN-MAP zbat-map-jobinfo   VALUE IS zbat-tnm-jobinfo
                             DEF   IS zbat-ddl-jobinfo.
   VERSION "C00" FOR znumber THROUGH zlocation.
END
```

generate this TAL output:

```
?SECTION ZBAT^TNM^JOBINFO
Literal ZBAT^TNM^JOBINFO = 63;

?SECTION ZBAT^DDL^JOBINFO
STRUCT    ZBAT^DDL^JOBINFO^DEF (*);
   BEGIN
   INT       ZNUMBER;
   INT       ZPRIORITY;
   STRUCT    ZLOCATION;
      BEGIN
      STRUCT    Z^C;
         BEGIN STRING BYTE [0:7]; END;
      STRUCT    Z^S = Z^C;
         BEGIN
         INT       Z^I[0:3];
         END;
      STRING    Z^B[0:7] = Z^C;
      END;
   END;

?SECTION ZBAT^TNM^JOBINFO
Literal ZBAT^TNM^JOBINFO = 63;

?SECTION ZBAT^MAP^JOBINFO
Define ZBAT^MAP^JOBINFO = [ 2303, 63, 12, 17152, 1024,
                            2136 ]#;
Literal ZBAT^MAP^JOBINFO^WLN = 6;
```

Structures can be changed in a version-compatible way only by adding fields at the end of the structure. Changing the DDL to add new fields for user ID and job class would look like:

```
?DICT
?TAL ZTAL !

DEFINITION zbat-ddl-jobinfo.
   02  znumber             TYPE zspi-ddl-int.
   02  zpriority           TYPE zspi-ddl-int.
   02  zlocation           TYPE zspi-ddl-char8
                                      SPI-NULL "X".
   02  zuserid-is-present  TYPE zspi-ddl-boolean.
   02  zuserid             TYPE zspi-ddl-userid.
   02  zjobclass           TYPE zspi-ddl-int.
END

CONSTANT  zbat-tnm-jobinfo   VALUE IS 63.
```

```
TOKEN-MAP zbat-map-jobinfo   VALUE IS zbat-tnm-jobinfo
                             DEF   IS zbat-ddl-jobinfo.
   VERSION "C00" FOR znumber THROUGH zlocation.
   VERSION "C10" FOR zuserid-is-present.
   NOVERSION     FOR zuserid.
   VERSION "C10" FOR zjobclass.
END
```

The resulting TAL code generated by the DDL compiler is:

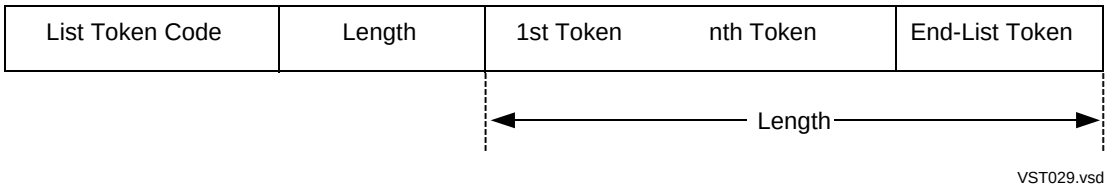
```
?SECTION ZBAT^DDL^JOBINFO
STRUCT    ZBAT^DDL^JOBINFO^DEF (*);
   BEGIN
   INT       ZNUMBER;
   INT       ZPRIORITY;
   STRUCT    ZLOCATION;
      BEGIN
      STRUCT    Z^C;
         BEGIN STRING BYTE [0:7]; END;
      STRUCT    Z^S = Z^C;
         BEGIN
         INT       Z^I[0:3];
         END;
      STRING    Z^B[0:7] = Z^C;
      END;
   INT       ZUSERID^IS^PRESENT;
   STRUCT    ZUSERID;
      BEGIN
      STRING    ZBYTE[0:1];
      END;
   INT       ZJOBCLASS;
   END;

?SECTION ZBAT^TNM^JOBINFO
Literal ZBAT^TNM^JOBINFO = 63;

?SECTION ZBAT^MAP^JOBINFO
Define ZBAT^MAP^JOBINFO = [ 2303, 63, 18, 17152, 1024,
                            2136, 1, 17162, 544, 1, 0, 512,
                            1, 17162, 512 ]#;
Literal ZBAT^MAP^JOBINFO^WLN = 15;
```

In this case, NOVERSION is specified for the user ID field because the presence or absence of the field is actually indicated by the IS-PRESENT Boolean.

List Structure

Each list token (each token of token type ZSPI-TYP-LIST and token length 0) is stored in the buffer as a variable-length token (token type ZSPI-TKN-LIST and token length ZSPI-TLN-VARIABLE, or 255). Its length—which follows the token code, as for all variable-length tokens—includes all tokens that are part of the list, including the end-list token.

Figure C-7. Structure of a List in the Buffer



When stored in the buffer, the token-number portion of the ZSPI-TKN-ENDLIST token is replaced with the length of the list. This is used for a boundary-tag consistency check.

D

NonStop Kernel Subsystem Numbers and Abbreviations

This appendix lists the NonStop Kernel subsystem numbers and abbreviations.

Table D-1 lists the subsystem numbers, abbreviations, and mnemonics assigned to NonStop Kernel subsystems. The integer numbers themselves are provided for debugging purposes. Under normal circumstances, you should refer to a subsystem number by its symbolic name, which has the form ZSPI-SSN-*ssss*.

Table D-2 on page D-12 lists the subsystem abbreviations that appear in NonStop Kernel subsystem definition files. Data items from these files can appear in command messages, response messages, and event messages. All NonStop Kernel subsystem abbreviations start with *Z*. If you are writing an SPI server, start your definition names with a letter other than *Z* to avoid name conflicts with software that HP provides for the NonStop server.

For some subsystems and other system components, HP defines a mnemonic (short name) of up to eight characters:

- Displayed by the ViewPoint console application to identify the source of event messages
- Accepted and displayed by TACL in the subsystem-number portion of subsystem IDs (for example, EXPAND in TANDEM.EXPAND.D40)
- Used in the HELP command and error-message displays of the Subsystem Control Facility (SCF)

Table D-1 and Table D-2 on page D-12 list the mnemonic for any subsystem or other system component for which a mnemonic is defined. The mnemonics are case-sensitive; all alphabetic characters must be uppercase. (TACL does not upshift them.)

Table D-1. NonStop Kernel Subsystem Numbers (page 1 of 11)

Subsystem Number	Subsystem Abbreviation	Mnemonic	Description
0			ZSPI-SSN-NULL
1	ZSPI	SPI	Subsystem Programmatic Interface
2	ZODP	ODP	Optical disk process
3	ZTAC	TACL	HP Tandem Advanced Command Language
4	ZTAP	TAPE	Tape process
5	ZDNS	DNS	Distributed Name Service

1 Subsystem 265 is supported only on systems running H-series RVUs.

2 Subsystems, 266 and 267 are supported only on systems running G-series RVUs.

Table D-1. NonStop Kernel Subsystem Numbers (page 2 of 11)

Subsystem Number	Subsystem Abbreviation	Mnemonic	Description
6	ZFUP	FUP	File Utility Program
7	ZPUP	PUP	Peripheral Utility Program
8	ZPWY	PATHWAY	HP NonStop TS/MP (TS/MP) and HP NonStop Pathway/iTS
9	ZBAT	BAT	NetBatch batch processing system
10	ZTMF	TMF	HP NonStop Transaction Management Facility
11	ZTUT		Tandem SQL utility
12	ZEMS	EMS	Event Management Service
13	ZFOX	FOX	Fiber Optic Extension
14	ZMDS	MDS	Remote Maintenance Interface (RMI) event messages sent to HP Tandem Maintenance and Diagnostic System (TMDS)
15	ZCPU	CPU	Central processor microcode and memory manager
16	ZIPB	IPB	Interprocessor bus interrupt handler
17	ZCAB		Cabinet
18	ZCAT		Catalyst
19	ZLAM	TLAM	HP Tandem LAN Access Method (TLAM)/Multilan connectivity tool
20	ZTMD	TMDS	HP Tandem Maintenance and Diagnostic System (TMDS)
21	ZCOM	ZCOM	SPI common extensions definitions (Items with shared ZCOM definitions take the subsystem ID of the subsystem with which they are used.)
22	ZAM3	AM3270	AM3270
23	ZAM6	AM6520	AM6520
24	ZATP	ATP	ATP6100 terminal and printer processes
25	ZSCP	SCP	Subsystem Control Point
26	ZCP6		CP6100
27	ZCSM	CSM	Communications Subsystem Manager
28	ZART	ARCHTAPE	Archived Tape
29	ZAPC	SNAX/APC	SNAX/Advanced Program Communication (SNAX/APC) communications services

1 Subsystem 265 is supported only on systems running H-series RVUs.

2 Subsystems, 266 and 267 are supported only on systems running G-series RVUs.

Table D-1. NonStop Kernel Subsystem Numbers (page 3 of 11)

Subsystem Number	Subsystem Abbreviation	Mnemonic	Description
30	ZEXP	EXPAND	Expand networking tool
31	ZDSK	DISK	Disk process
32	ZDCS	DCOMDSAP	DCOM and Disk Space Analysis Program (DSAP) utilities
33	ZGDS	GDS	General Device Support
34	ZDSC	DSC	Dynamic System Configuration
35	ZSTN	SAFENET	Safe-T-Net
36	ZSX1	SNAX	SNAX Advanced Peer Networking (SNAX/APN) and SNAX Extended Facility (SNAX/XF)
37	ZCDF	SNAXCDF	SNAX Cross-Domain Facility (SNAX/CDF)
38	ZTIL	TIL	HP Tandem-to-IBM Link (TIL)
39	ZTHL	THL	HP Tandem HyperLink (THL)
40	ZTR3	TR3271	TR3721
41	ZX25	X25AM	X.25 Access Method
42	ZSO4		SBS4
43			*reserved*
44	ZDDN		Defense Data Network (DDN)
45	ZMHS	OSI/MHS	Open Systems Interconnection (OSI) /Message Handling System (MHS)
46	ZGRD	GUARDLIB	NonStop operating system
47	ZGP1	EDITREAD	Editread
48	ZGP2	SIO	Sequential I/O
49	ZGP3	FMTR	Formatter
50	ZGP4	IOEDIT	*reserved*
51	ZGP5		*reserved*
52	ZGP6		*reserved*
53	ZGP7		*reserved*
54	ZGP8		*reserved*
55	ZGP9		*reserved*
56	ZSRT		Sort and FastSort utilities
57	ZSPL		Spooler
58	ZENF		Enform

1 Subsystem 265 is supported only on systems running H-series RVUs.

2 Subsystems, 266 and 267 are supported only on systems running G-series RVUs.

Table D-1. NonStop Kernel Subsystem Numbers (page 4 of 11)

Subsystem Number	Subsystem Abbreviation	Mnemonic	Description
59	ZVCS		Sorceress
60	ZVPT	VIEWPT	ViewPoint console application
61	ZS16	TMDS6106	SBS16
62	ZOSI	OSI/AS	Open Systems Interconnection/Application Services
63	ZOS4	OSI/TS	Open Systems Interconnection/Transport Services
64	ZCLX		HP NonStop CLX processors
65	ZTLN	TLN	Talon
66	ZCLK	CLOCK	System clock low-level software
67	ZSRV		Surveyor
68	ZFIL	FILESYS	File system
69	ZGIO	GUARDIO	Low-level I/O
70	ZCMS	CMS	Configuration Management System
71	ZCSS	CSS	HP Tandem Maintenance and Diagnostic System (TMDS) for communications subsystem (CSS)
72	ZMON	MONITOR	System monitor
73	ZMSG	MSGSYS	Message system
74	ZBKU	BACKUP	Backup
75	ZRST	RESTORE	Restore
76	ZBKC	BACKCOPY	Backcopy
77	ZNFS	NFS	Network File System
78	ZCDG	CDG	Common Data Communications Diagnostics
79	ZRPC	RPC	Remote Procedure Call (RPC)
80	ZTCI	ZTCI	Transmission Control Protocol/Internet Protocol (TCP/IP)
81	ZPMT	DSMPM	Problem Management and Tracking
82	ZDSN	DSNM	Distributed Systems Network Management
83	ZCMK	COM-KRNL	Common messages
84	ZDGN		Diskgen
85	ZCL2		SNAX/EnvoyACPXF CL2
86	ZSCM		SQL compiler

1 Subsystem 265 is supported only on systems running H-series RVUs.

2 Subsystems, 266 and 267 are supported only on systems running G-series RVUs.

Table D-1. NonStop Kernel Subsystem Numbers (page 5 of 11)

Subsystem Number	Subsystem Abbreviation	Mnemonic	Description
87	ZORS	ORSERV	Online reload server
88	ZEXF		EnvoyACP/XF
89	ZPWR		CLX autopower utility
90	ZREP		TMDS Replace
91	ZTNT	TELSERV	Telserv subsystem
92	ZFIR		TMDS FRU Information Record (FIR)
93	ZAPS		OSI/AS
94	ZSFG	SFG	Safeguard security product
95	ZSYS		Operating system
96	ZTAS		Safeguard Trusted Audit Service
97	ZOSA	OSIAPLMG	Open Systems Interconnection/Application Manager
98	ZFTM	FTAM	Open Systems Interconnection/ File Transfer, Access, and Management
99	ZMMS	MMS	OSI Manufacturing Message Spec
100	ZDUA	DUA	OSI Directory user agent
101	ZNMA	NMA	OSI Network Management Agent
102	ZTRC	TRACE	DSM Trace
103	ZDNI		DSM NetView Interconnect
104	ZRPO	REPO	Repository
105	ZSDN	ISDN	Integrated Services Digital Network (ISDN)
106	ZACS	ACS	Atalla Cryptographic Subsystem
107	ZENV	ENVOY	Envoy
108	ZCCM	CCM	Call Center Management
109	ZEM3	EM3270	EM3270
110	ZHLS	SNAXHLS	SNAX High-Level Support (SNAX/HLS)
111	ZMCS	MediaMgt	Media Management Catalog
112	ZTLK	TTALK	TandemTalk
113	ZPNA	PNA	Programmatic Network Administrator (PNA)
114	ZING	INGRES	Ingres
115	ZT21		IBM node type 2.1
116	ZSYT		Surveyor to TCM Interface

1 Subsystem 265 is supported only on systems running H-series RVUs.

2 Subsystems, 266 and 267 are supported only on systems running G-series RVUs.

Table D-1. NonStop Kernel Subsystem Numbers (page 6 of 11)

Subsystem Number	Subsystem Abbreviation	Mnemonic	Description
117	ZNNM	NNM	NonStop NET/MASTER
118	ZSWY	SYSWAY	SysWay
119	ZTWY	TRANSWAY	TransWay
120	ZNST	TEXTBASE	TEXTBASE
121	ZSNM		TCP/IP
122	ZSCS		SQL Communications Subsystem
123			*reserved*
124	ZLEG	LEGHOST	*reserved*
125	ZTN3	TN32SERV	TN32SERV access method
126	ZEMA		EMS Analyzer
127	ZOMF	OMF	Object Monitoring Facility
128	ZEDF		Event Distribution Facility
129	ZMCM		Measure TCM Interface
130	ZRMT		RMI remote port
131	ZSYH	SYSH	Syshealth
132	ZLDS		T1002 Template
133	ZCRE	SNAXCRE	SNAX Creator-2
134	ZRMI		RMIACCES XMIOP
135	ZPHI	DSM/SCM	DSM/Software Configuration Manager
136	ZOSF	OSI/FTAM	OSI/File Transfer, Access, and Maintenance System
137	ZSGN	SYSGEN	System Generation EMS Interface
138	ZVHS	VHS	NonStop Virtual Hometerm Subsystem
139	ZOSN		OSI/NM Network Management
140	ZSUI		User interface
141	ZMSR	MEASURE	Measure
142	ZBRT	BRT	Tape Reader/BACKUP Subsystem
143	ZOSS	OSS	HP NonStop Kernel Open System Services (OSS)
144	ZNAR		Network Control Language (NCL) Automation Rules Set
145	ZXCR		Exchange/remote job entry (RJE)

1 Subsystem 265 is supported only on systems running H-series RVUs.

2 Subsystems, 266 and 267 are supported only on systems running G-series RVUs.

Table D-1. NonStop Kernel Subsystem Numbers (page 7 of 11)

Subsystem Number	Subsystem Abbreviation	Mnemonic	Description
146	ZXCS		Exchange/Systems Network Architecture (SNA)
147	ZSXC	SXCM	SNAX Connection Manager (SNAX/CM)
148	ZMWR		FastConnect/MessageWare
149	ZGPA		Performance Analyzer
150	ZSSI	SSI	Storage Server Interface
151	ZDMP	DMP	HP Tandem Failure Data System (TFDS)
152	ZQAT		Online transaction processing (OLTP) QA TESTWARE
153	ZNDS		OSI/NonStop Directory Services (NSDS)
154	ZDTS		DSM Template Services
155	ZSMP	SMP	Simple Network Management Protocol (SNMP)
156	ZGSX		SWEDS
157	ZGSP		GoldSend
158	ZTBL		SCF IF GEN
159	ZLBL		Tapestry Label
160	ZRSC		Remote Server Call
161	ZSQL	SQL	HP NonStop SQL/MP (SQL/MP)
162	ZAUD	SQLAUDIT	HP NonStop SQL/MP Audit
163	ZNSX	NSX	Network Statistics Extended
164	ZIPX	IPXSPX	Interprocessor Extended/Subsystem Processor Extended
165	ZWMS	WMS	Workload Measurement System
166	ZQIO	QIO	Shared I/O for Transmission Control Protocol/Internet Protocol
167	ZLMN	LINKMON	LINKMON Subsystem
168	ZSMF	SMF	HP NonStop Storage Management Foundation
169	ZTIF	TIF	Tandem Instrumentation Facility
170	ZGSS	GSS	Guided System Services
171	ZCIC	CICS	Parallel Transaction Processing (PTP) for Customer Information Control System (CICS)

1 Subsystem 265 is supported only on systems running H-series RVUs.

2 Subsystems, 266 and 267 are supported only on systems running G-series RVUs.

Table D-1. NonStop Kernel Subsystem Numbers (page 8 of 11)

Subsystem Number	Subsystem Abbreviation	Mnemonic	Description
172	ZATM	ATM	ATM
173	ZOPS	OPSCON	DSM/MultiView
174	ZEIC	ElInv	Electronic Inventory
175	ZRAL	RAL	Resource access layer
176	ZWAN	WAN	Wide Area Network subsystem
177	ZLSN	LSNTTTCP	Listener Tandem Talk
178	ZNBX	NBX	NETBIOS over IPX/SPX
179	ZNBT	NBT	NETBIOS over TCP/IP
180	ZHRM	HRM	Host Resources Subagent
181	ZCCC	CCC	Common Call Catcher
182	ZONS	ONS	Open Notification Service
183	ZNSK	NSK	HP NonStop Operating System/Kernel Managed Swap Events
184	ZSRL	SRL	Shared run-time library
185	ZTMX	TMX	Simple Network Management Protocol (SNMP) Trap Multiplexer
186	ZMEV	MEV	Multi-Event Viewer
187	ZTAG	TAG	Transport Agent
188	ZDCE	DCE	Distributed Computing Environment (DCE)
189	ZSTO	STORAGE	Storage Subsystem
190	ZSCZ	SCSI	Open SCSI
191	ZTSE	TSE	HP NonStop TS/MP for HP NonStop Tuxedo users
192	ZSQA		System QA
193	ZLAN	LAN	Local area network
194	ZTSM	TSM	HP TSM Transfer
195	ZSAP	SAP	Service access point
196	ZISA	IPXSA	IPX-SPV Subsystem (NetWare on NonStop operating system)
197	ZYMP	YMP	MIOP (maintenence I/O process) subsystem
198	ZCEV	CEV	Common Event Viewer/EMS Event Viewer
199	ZKRN	KRN	Kernel Subsystem

1 Subsystem 265 is supported only on systems running H-series RVUs.

2 Subsystems, 266 and 267 are supported only on systems running G-series RVUs.

Table D-1. NonStop Kernel Subsystem Numbers (page 9 of 11)

Subsystem Number	Subsystem Abbreviation	Mnemonic	Description
200	ZXIO	XIO	HP NonStop Kernel XIO (extensible input/output)
201	ZSNT	SNT	ServerNet Error Handler
202	ZTSA	TSA	TCP/IP SNMP SubAgent
203	ZIPC	IPC	HP NonStop Kernel Message Subsystem
204	ZNSC	NSC	HP NonStop C Multitasking Engine
205	ZSMD	SMD	Open SCSI Module Driver
206	ZSPR	SPR	Server Processor
207	ZSIM	SIM	Secure Internet Messaging Services
208	ZTUX	TUX	NonStop Tuxedo
209	ZDIO	DIO	Direct-bulk I/O
210	ZWEB	WEB	WebServer
211	ZRDF	RDF	Remote Duplicate Database Facility
212	ZESM	ESM	Enterprise Storage Manager
213	ZASM	ASM	Automated Storage Manager
214	ZPRT	PRT	TCP/IP Utilities
215	ZPAM	PAM	Port Access Method
216	ZDOM	DOM	NS-DOM
217	ZESA	ETHSA	SNMP Ethernet/ Token-Ring Subagent
218	ZSCL	SCL	SuperClusters
219	ZNIM		HP NonStop Internet Messaging
220	ZTCP	TCPMAN	TCP/IP Manager
221	ZETN	ETN32SRV	NonStop Operating System Enhanced TN3270 Server
222			*reserved*
223			*reserved*
224			*reserved*
225	ZDSL	DSL	Dynamic Update of System Library (DUSL)
226	ZASP	ASAP	Availability, Statistics, and Performance
227	ZKRF		Key Repository Facility
228			*reserved*
229			*reserved*

1 Subsystem 265 is supported only on systems running H-series RVUs.

2 Subsystems, 266 and 267 are supported only on systems running G-series RVUs.

Table D-1. NonStop Kernel Subsystem Numbers (page 10 of 11)

Subsystem Number	Subsystem Abbreviation	Mnemonic	Description
230			*reserved*
231	ZMXO	ODBC/MX	HP NonStop ODBC/MX
232	ZNNS	NNSERVER	Network Node Server
233	ZDPA		Data Path Adapter
234	ZNOS		HP NonStop ODBC/MX server
235	ZNDC		
236	ZMXS	SQL/MX	HP NonStop SQL/MX
237	ZSMN	SANMAN	External system area network manager process
238	ZFTP	FTP	File Transfer Protocol
239	ZNUL	TNOS	TNOS Utility
240	ZFSP		TCP/IP LAN Print Spooler
241	ZINS		Inspect
242	ZESC		HP NonStop Auto TMF software
243	ZLSG		Lockstop Gateway
244	ZXMN	Expand/TCP	Expand/TCP Manager
245	ZTTY	OSSTTY	OSS teletype
246	ZTC6	TCP/IPv6	HP NonStop TCP/IPv6
247	ZMQI	WMQI	WebSphere MQI for NonStop Kernel
248	ZMQS		MQSeries V5.1
249	ZASY	ASY	HP NonStop AutoSYNC software
250	ZOSM	OSM	HP NonStop Open System Management (OSM) Interface
251	ZE2A	E2A	J2EE 1.3 Compliant Application Server
252	ZACL	ACL	Application Cluster Services
253	ZOVN	OVAN	OpenView Agent for HP NonStop servers
254	ZOVM	OVAN MON	OVAN Monitor
255	ZBRU	BR2	Backup Restore 2
256	ZTPM	TPM	Tandem/Total Performance Management
257	ZWVP	WVPT	Web ViewPoint
258	ZOPM	OVNPM	OpenView Performance Monitor for HP NonStop servers
259	ZCIP	CIP	Constellation IP

1 Subsystem 265 is supported only on systems running H-series RVUs.

2 Subsystems, 266 and 267 are supported only on systems running G-series RVUs.

Table D-1. NonStop Kernel Subsystem Numbers (page 11 of 11)

Subsystem Number	Subsystem Abbreviation	Mnemonic	Description
260	ZDLL	DLL	DLL Subsystem
261	ZWPY	WPY	HP Web ViewPoint Pathway
262	ZFSM	FSM	Fibre Channel Storage Monitor
263	ZCMP	CMP	HP NonStop Operating System - Complex Manager Auxiliary Process
264	ZZXA	ZXA	XA Broker Subsystem
265	ZL4J	L4J	Java Logging Subsystem[1]
266	ZCPS	CPS	Matrix SMLC CBB 4.2 Subsystem[2]
267	ZDLD	DLD	Deadlock Detector Subsystem[2]

1 Subsystem 265 is supported only on systems running H-series RVUs.

2 Subsystems, 266 and 267 are supported only on systems running G-series RVUs.

Table D-2. NonStop Kernel Subsystem Abbreviations (page 1 of 9)

Subsystem Abbreviation	Subsystem Number	Mnemonic	Description
ZACL	252	ACL	Application Cluster Services
ZACS	106	ACS	Atalla Cryptographic Subsystem
ZAM3	22	AM3270	AM3270
ZAM6	23	AM6520	AM6520
ZAPC	29	SNAXAPC	SNAX/APC communications services
ZAPS	93		OSI/AS
ZART	28	ARCHTAPE	Archived Tape
ZASM	213	ASM	Automated Storage Manager
ZASP	226	ASAP	Availability, Statistics, and Performance
ZASY	249	ASY	HP NonStop AutoSYNC software
ZATM	172	ATM	ATM
ZATP	24	ATP	ATP6100 terminal and printer processes
ZAUD	162	SQLAUDIT	SQL/MP Audit
ZBAT	9	BAT	NetBatch batch processing system
ZBKC	76	BACKCOPY	Backcopy
ZBKU	74	BACKUP	Backup
ZBRT	142	BRT	Tape Reader/BACKUP Subsystem
ZCAB	17		Cabinet
ZCAT	18		Catalyst
ZCCC	181	CCC	Common Call Catcher
ZCCM	108	CCM	Call Center Management
ZCDF	37	SNAXCDF	SNAX Cross-Domain Facility (SNAX/CDF)
ZCDG	78	CDG	Common Data Communications Diagnostics
ZCEV	198	CEV	Common Event Viewer/EMS Event Viewer
ZCIC	171	CICS	PTP for CICS
ZCIP	259	CIP	Constellation IP
ZCL2	85		SNAX/EnvoyACPXF CL2
ZCLK	66	CLOCK	System clock low-level software
ZCLX	64		NonStop CLX processors
ZCMK	83	COM-KRNL	Common messages
ZCMP	263	CMP	HP NonStop Operating System - Complex Manager Auxiliary Process

1 Subsystems, 266 and 267 are supported only on systems running G-series RVUs.

2 Subsystem 265 is supported only on systems running H-series RVUs.

Table D-2. NonStop Kernel Subsystem Abbreviations (page 2 of 9)

Subsystem Abbreviation	Subsystem Number	Mnemonic	Description
ZCMS	70	CMS	Configuration Management System
ZCOM	21	ZCOM	SPI common extensions definitions
ZCP6	26		CP6100
ZCPS	266	CPS	Matrix SMLC CCB 4.2 Subsystem[1]
ZCPU	15	CPU	Central processor microcode and memory manager
ZCRE	133	SNAXCRE	SNAX Creator-2
ZCSM	27	CSM	Communications Subsystem Manager
ZCSS	71	CSS	HP Tandem Maintenance and Diagnostic System (TMDS) for communications subsystem (CSS)
ZDCE	188	DCE	Distributed Computing Environment
ZDCS	32	DCOMDSAP	DCOM and DSAP utilities
ZDGN	84		Diskgen
ZDIO	209	DIO	Direct-bulk I/O
ZDLD	267	DLD	Deadlock Detector Subsystem[1]
ZDLL	260	DLL	DLL Subsystem
ZDMP	151	DMP	HP Tandem Failure Data System (TFDS)
ZDNI	103		DSM NetView Interconnect
ZDNS	5	DNS	DNS
ZDOM	216	DOM	NS-DOM
ZDPA	233		Data Path Adapter
ZDSC	34	DSC	Dynamic System Configuration
ZDSK	31	DISK	Disk process
ZDSL	225	DSL	Dynamic Update of System Library (DUSL)
ZDSN	82	DSNM	Distributed Systems Network Management
ZDTS	154		DSM Template Services
ZDUA	100	DUA	OSI Directory User Agent
ZE2A	251	E2A	J2EE 1.3 Compliant Application Server
ZEDF	128		Event Distribution Facility
ZEM3	109	EM3270	EM3270
ZEMA	126		EMS Analyzer
ZEMS	12	EMS	EMS

1 Subsystems, 266 and 267 are supported only on systems running G-series RVUs.

2 Subsystem 265 is supported only on systems running H-series RVUs.

Table D-2. NonStop Kernel Subsystem Abbreviations (page 3 of 9)

Subsystem Abbreviation	Subsystem Number	Mnemonic	Description
ZENF	58		Enform
ZENV	107	ENVOY	Envoy
ZESA	217	ETHSA	SNMP Ethernet/Token-Ring Subagent
ZESC	242		HP NonStop Auto TMF software
ZESM	212	ESM	Enterprise Storage Manager
ZETN	221	ETN32SRV	NonStop Operating System Enhanced TN3270 Server
ZEXF	88		EnvoyACP/XF
ZEXP	30	EXPAND	Expand networking tool
ZFIL	68	FILESYS	File system
ZFIR	92		TMDS FRU Information Record (FIR)
ZFOX	13	FOX	Fiber Optic Extension
ZFSM	262	FSM	Fibre Channel Storage Monitor
ZFSP	240		TCP/IP LAN Print Spooler
ZFTM	98	FTAM	OSI File Transfer/Access Management
ZFTP	238	FTP	File Transfer Protocol
ZFUP	6	FUP	File Utility Program
ZGDS	33	GDS	General Device Support
ZGIO	69	GUARDIO	Low-level I/O
ZGP1	47	EDITREAD	Editread
ZGP2	48	SIO	Sequential I/O
ZGP3	49	FMTR	Formatter
ZGPA	149		Performance Analyzer
ZGRD	46	GUARDLIB	NonStop operating system
ZGSP	157		GoldSend
ZGSS	170	GSS	Guided System Services
ZGSX	156		SWEDS
ZHLS	110	SNAXHLS	SNAX High-Level Support (SNAX/HLS)
ZHRM	180	HRM	Host Resources Subagent
ZING	114	INGRES	Ingres
ZINS	241		Inspect
ZIPB	16	IPB	Interprocessor bus interrupt handler

1 Subsystems, 266 and 267 are supported only on systems running G-series RVUs.

2 Subsystem 265 is supported only on systems running H-series RVUs.

Table D-2. NonStop Kernel Subsystem Abbreviations (page 4 of 9)

Subsystem Abbreviation	Subsystem Number	Mnemonic	Description
ZIPC	203	IPC	NonStop Kernel Message Subsystem
ZIPX	164	IPXSPX	Interprocessor Extended/Subsystem Processor Extended
ZISA	196	IPXSSA	IPX-SPV Subsystem (NetWare on NonStop operating system)
ZKRF	227		Key Repository Facility
ZKRN	199	KRN	Kernel Subsystem
ZL4J	265	L4J	Java Logging Subsystem[2]
ZLAM	19	TLAM	TLAM/Multilan connectivity tool
ZLAN	193	LAN	Local area network
ZLBL	159		Tapestry Label
ZLDS	132		T1002 Template
ZLEG	124	LEGHOST	*reserved*
ZLMN	167	LINKMON	LINKMON Subsystem
ZLSG	243		Lockstop Gateway
ZMCM	129		Measure TCM Interface
ZMCS	111	MediaMgt	Media Management Catalog
ZMDS	14	MDS	RMI event messages sent to TMDS
ZMEV	186	MEV	Multi-Event Viewer
ZMHS	45	MHS	Open Systems Interconnection/Message Handling Services
ZMMS	99	MMS	OSI Manufacturing Message Spec
ZMON	72	MONITOR	System monitor
ZMQI	247	WMQI	WebSphere MQI for NonStop Kernel
ZMQS	248		MQSeries V5.1
ZMSG	73	MSGSYS	Message system
ZMSR	141	MEASURE	Measure
ZMWR	148		FastConnect/MessageWare
ZMXO	231	ODBC/MX	NonStop Open Database Connectivity/MX
ZMXS	236		
ZNAR	144		Network Control Language (NCL) Automation Rules Set
ZNBT	179	ZNBT	NETBIOS over TCP/IP

1 Subsystems, 266 and 267 are supported only on systems running G-series RVUs.

2 Subsystem 265 is supported only on systems running H-series RVUs.

Table D-2. NonStop Kernel Subsystem Abbreviations (page 5 of 9)

Subsystem Abbreviation	Subsystem Number	Mnemonic	Description
ZNBX	178	NBX	NETBIOS over IPX/SPX
ZNDC	235		
ZNDS	153		OSI/NSDS
ZNFS	77	NFS	Network File System
ZNIM	219		NonStop Internet Messaging
ZNMA	101	NMA	OSI Network Management Agent
ZNNM	117	NNM	NonStop NET/MASTER
ZNNS	232	NNSERVER	Network Node Server
ZNOS	234		NonStop ODBC/MX server
ZNSC	204	NSC	NonStop C Multitasking Engine
ZNSK	183	NSK	NonStop operating system/Kernel Managed Swap Events
ZNST	120	TEXTBASE	TEXTBASE
ZNSX	163	NSX	Network Statistics Extended
ZNUL	239	TNOS	TNOS Utility
ZODP	2	ODP	Optical disk process
ZOMF	127	OMF	Object Monitoring Facility
ZONS	182	ONS	Open Notification Service
ZOPM	258	OVNPM	OpenView Performance Monitor for HP NonStop servers
ZOPS	173	OPSCON	DSM/MultiView
ZORS	87	ORSERV	Online reload server
ZOS4	63	OSI/TS	Open Systems Interconnection/Transport Services
ZOSA	97	OSIAPLMG	Open Systems Interconnection/Application Manager
ZOSF	136	OSIFTAM	OSI/File Transfer, Access, and Maintenance System
ZOSI	62	OSI/AS	Open Systems Interconnection/Application Services
ZOSM	250	OSM	HP NonStop Open System Management (OSM) Interface
ZOSN	139		OSI/NM Network Management
ZOSS	143	OSS	NonStop Open System Services

1 Subsystems, 266 and 267 are supported only on systems running G-series RVUs.

2 Subsystem 265 is supported only on systems running H-series RVUs.

Table D-2. NonStop Kernel Subsystem Abbreviations (page 6 of 9)

Subsystem Abbreviation	Subsystem Number	Mnemonic	Description
ZOVM	254	OVANMON	OVAN Monitor
ZOVN	253	OVAN	OpenView Agent for HP NonStop servers
ZPAM	215	PAM	Port Access Method
ZPHI	135	DSM/SCM	DSM/Software Configuration Manager
ZPMT	81	DSMPM	Problem Management and Tracking
ZPNA	113	PNA	Programmatic Network Administrator
ZPRT	214	PRT	TCP/IP Utilities
ZPUP	7	PUP	Peripheral Utility Program
ZPWR	89		CLX autopower utility
ZPWY	8	PATHWAY	HP NonStop TS/MP and HP NonStop Pathway/iTS
ZQAT	152		Online transaction processing (OLTP) QA TESTWARE
ZQIO	166	QIO	Shared I/O for Transmission Control Protocol/Internet Protocol (TCP/IP)
ZRAL	175	RAL	Resource access layer
ZRDF	211	RDF	Remote Duplicate Database Facility
ZREP	90		TMDS Replace
ZRMI	134		RMIACCES XMIOP
ZRMT	130		RMI remote port
ZRPC	79	RPC	Remote Procedure Call (RPC)
ZRPO	104	REPO	Repository
ZRSC	160		Remote Server Call
ZRST	75	RESTORE	Restore
ZSCM	86		SQL compiler
ZS16	61	TMDS6106	SBS16
ZSAP	195	SAP	Service access point
ZSCL	218	SCL	SuperClusters
ZSCP	25	SCP	Subsystem Control Point
ZSCS	122		SQL Communications Subsystem
ZSCZ	190	SCSI	Open SCSI
ZSDN	105	ISDN	Integrated Services Digital Network
ZSFG	94	SFG	Safeguard security product

1 Subsystems, 266 and 267 are supported only on systems running G-series RVUs.

2 Subsystem 265 is supported only on systems running H-series RVUs.

Table D-2. NonStop Kernel Subsystem Abbreviations (page 7 of 9)

Subsystem Abbreviation	Subsystem Number	Mnemonic	Description
ZSGN	137		SYSGEN EMS Interface
ZSIM	207	SIM	Secure Internet Messaging Services
ZSMD	205	SMD	Open SCSI Module Driver
ZSMF	168	SMF	System Managed Storage
ZSMN	237	SANMAN	System Area Network Manager
ZSMP	155	SMP	NonStop SNMP (Simple Network Management Protocol)
ZSNM	121		TCP/IP
ZSNT	201	SNT	ServerNet Error Handler
ZSO4	42		SBS4
ZSPI	1	SPI	Subsystem Programmatic Interface
ZSPL	57		Spooler
ZSPR	206	SPR	Server Processor
ZSQA	192		System QA
ZSQL	161	SQL	NonStop SQL/MP
ZSRL	184	SRL	Shared run-time library
ZSRT	56		Sort and FastSort utilities
ZSRV	67		Surveyor
ZSSI	150		Storage Service Interface
ZSTN	35	SAFENET	Safe-T-Net
ZSTO	189	STORAGE	Storage Subsystem
ZSUI	140		User interface
ZSWY	118	SYSWAY	SysWay
ZSX1	36	SNAX	SNAX Advanced Peer Networking (SNAX/APN) and SNAX Extended Facility (SNAX/XF)
ZSXC	147	SXCM	SNAX Connection Manager
ZSYH	131	SYSH	Syshealth
ZSYS	95		Operating system
ZSYT	116		Surveyor to TCM Interface
ZT21	115		IBM node type 2.1
ZTAC	3	TACL	HP Tandem Advanced Command Language
ZTAG	187	TAG	Transport Agent

1 Subsystems, 266 and 267 are supported only on systems running G-series RVUs.

2 Subsystem 265 is supported only on systems running H-series RVUs.

Table D-2. NonStop Kernel Subsystem Abbreviations (page 8 of 9)

Subsystem Abbreviation	Subsystem Number	Mnemonic	Description
ZTAP	4	TAPE	Tape process
ZTAS	96		SafeGuard Trusted Audit Service
ZTBL	158		SCF IF GEN
ZTCI	80	ZTCI	Transmission Control Protocol/Internet Protocol (TCP/IP)
ZTC6	246	TCP/IPv6	HP NonStop TCP/IPv6
ZTCP	220	TCPMAN	TCP/IP Manager
ZTHL	39	THL	HP Tandem HyperLink
ZTIF	169	TIF	Tandem Instrumentation Facility
ZTIL	38	TIL	HP Tandem-to-IBM Link (TIL)
ZTLK	112	TTALK	TandemTalk
ZTLN	65	TLN	Talon
ZTMD	20	TMDS	Tandem Maintenance and Diagnostic System
ZTMF	10	TMF	NonStop Transaction Management Facility
ZTMX	185	TMX	Simple Network Management Protocol (SNMP) Trap Multiplexer
ZTN3	125	TN32SERV	TN32SERV access method
ZTNT	91	TELSERV	Telserv subsystem
ZTPM	256	TPM	Tandem/Total Performance Management
ZTR3	40	TR3271	TR3721
ZTRC	102	TRACE	DSM Trace
ZTSA	202	TSA	TCP/IP SNMP SubAgent
ZTSE	191	TSE	NonStop TS/MP for NonStop Tuxedo users
ZTSM	194	TSM	HP TSM Transfer
ZTTY	245	OSSTTY	OSS teletype
ZTUT	11		Tandem SQL utility
ZTUX	208	TUX	NonStop Tuxedo
ZTWY	119	TRANSWAY	TransWay
ZVCS	59		Sorceress
ZVHS	138	VHS	NonStop Virtual Hometerm Subsystem
ZVPT	60	VIEWPT	ViewPoint console application
ZWAN	176	WAN	Wide Area Network subsystem

1 Subsystems, 266 and 267 are supported only on systems running G-series RVUs.

2 Subsystem 265 is supported only on systems running H-series RVUs.

Table D-2. NonStop Kernel Subsystem Abbreviations (page 9 of 9)

Subsystem Abbreviation	Subsystem Number	Mnemonic	Description
ZWEB	210	WEB	WebServer
ZWMS	165	WMS	Workload Measurement System
ZWVP	257	WVPT	Web ViewPoint
ZX25	41	X25	X.25 Access Method
ZZXA	264	ZXA	XA Broker Subsystem
ZXCR	145		Exchange/RJE
ZXCS	146		Exchange/SNA
ZXIO	200	XIO	NonStop Kernel XIO (extensible input/output)
ZXMN	244	Expand/TCP	Expand/TCP Manager
ZXSC	147	SCXM	SNAX Connection Manager
ZYMP	197	YMP	MIOP (maintenence I/O process) subsystem

1 Subsystems, 266 and 267 are supported only on systems running G-series RVUs.

2 Subsystem 265 is supported only on systems running H-series RVUs.

E SPI Programming Examples

This appendix lists the TAL source code and C source code for six programs that demonstrate basic SPI concepts and programming techniques. In addition to these twelve working example programs, there are eight supporting files (four in TAL and four in C), one TACL command file for compiling the TAL programs, and one TACL command file for compiling the C programs.

The source code presented in this appendix is for working programs, supporting files, and one command file each for compiling the TAL and C programs.

Source files for these example programs are distributed in the subvolume ZSPIEXAM.

Working Programs

Example	Description	Source File	Page
E-1	A TAL program that demonstrates the basic buffer operations shown in Figure 2-4 on page 2-18. This example shows how to initialize a buffer, put tokens into the buffer, reset the buffer, and get tokens from the buffer.	SET0204	E-4
E-2	The C program equivalent of Example E-1	SEC0204	E-7
E-3	A TAL program that demonstrates the basic buffer operations shown in Figure 2-5 on page 2-22. This example shows how to move in and out of lists and retrieve tokens from within a list.	SET0205	E-9
E-4	The C program equivalent of Example E-3	SEC0205	E-12
E-5	A TAL program that performs the basic buffer operations shown in Figure 2-6 on page 2-24. This example demonstrates the behavior of the special SSGET operation ZSPI-TKN-NEXTTOKEN in and around lists.	SET0206	E-15
E-6	The C program equivalent of Example E-5	SEC0206	E-18
E-7	A TAL program that performs the basic buffer operations shown in Figure 2-7 on page 2-26. This example demonstrates the behavior of the special SSGET operation ZSPI-TKN-NEXTCODE in and around lists.	SET0207	E-21
E-8	The C program equivalent of Example E-7	SEC0207	E-24
E-9	A simple SPI requester that interacts with the server program in Example E-11. This example demonstrates fundamental SPI requester activities, including response continuation.	SETREQR	E-27

Supporting Files

Compile Command Files

Compiling the Example Programs

Compiling the TAL Programs

1. Set your default subvolume to the subvolume that contains the example source files. (If you plan to modify the programs for experimentation, copy the source files from the distribution subvolume to a working subvolume.)
2. If necessary, edit the ASSIGN statements in the compile command file for the TAL example source files (SETBUILD) to point to the location of the SPI definition files on your node. These are usually installed in $SYSTEM.ZSPIDEF but can be installed elsewhere.
3. TACL> obey setbuild

Compiling the C Programs

1. Set your default subvolume to the subvolume that contains the example source files. (If you plan to modify the programs for experimentation, copy the source files from the distribution subvolume to a working subvolume.)
2. If necessary, edit the search subvolume (ssv1) list statement in the compile command file (SECBUILD) for the C example source files to point to the location of the SPI definition files on your node. These are usually installed in $SYSTEM.ZSPIDEF but can be installed elsewhere.
3. TACL> obey secbuild

Running the Example Programs

Running the TAL Programs

The SETBUILD macro creates six runnable object files:

Example	Object File
E-1	SET0204O
E-3	SET0205O
E-5	SET0206O
E-7	SET0207O
E-9	SETREQRO
E-11	SETSERVO*

* Run automatically by SETREQO, the requester from E-9. Choose from the first five programs.

Running the C Programs

The SECBUILD macro creates six runnable object files:

Example	Object File
E-2	SEC0204O
E-4	SEC0205O
E-6	SEC0206O
E-8	SEC0207O
E-10	SECREQRO
E-12	SECSERVO*

* Run automatically by SECREQO, the requester from E-10. Choose from the first five programs.

A Note on Program Output

The example programs use DSM Template Services formatting procedures to display the contents of SPI buffers. Template Services uses an asterisk (*) to mark the current-token pointer and a hyphen (-) to mark the last-put-token pointer. When both point to the same token, only the hyphen is displayed. For more information about these procedures, see the *DSM Template Services Manual*.

Source File Examples

Example E-1: Basic Buffer Manipulations in TAL

The TAL source file in Example E-1 on page E-5 demonstrates the basic buffer manipulation activities illustrated in Figure 2-4 on page 2-18. The program uses DSM Template Services to display the contents of the buffer after each manipulation. Run the program to see the effect of SPI procedure calls on the contents of the buffer and the locations of the buffer pointers.

Source File

SET0204

Object File

SET0204O

Example E-1. TAL File: Basic Buffer Manipulations (page 1 of 2)

```
--   File name: SET0204
--   SPI EXAMPLE TAL 2-4
--   Figure 2-4. Pointer Manipulation
?SYMBOLS, INSPECT
LITERAL
        max^bufsize   = 256;     ! in bytes

?SOURCE SETCDECS
?NOLIST, SOURCE $system.system.extdecs0 (
?        abend, debug, dnumout, fileinfo, initializer, myterm, numout,
?        open, stop, writex,
?        spi_buffer_formatfinish_, spi_buffer_formatnext_,
?        spi_buffer_formatstart_,
?        ssinit, ssgettkn, ssmovetkn,
?        ssput, ssputtkn )
?LIST
?SOURCE SETCUTIL
?PAGE "PROC spitest MAIN"
PROC spitest MAIN;
BEGIN

  bufsize := max^bufsize;
  CALL initializer;
  CALL myterm(termname);
  CALL open(termname,term);
  IF <> THEN CALL abend;

  !
  ! Initialize the SPI buffer "b1"
  !
  IF (err := ssinit (b1, bufsize, ssid, zspi^val^cmdhdr)) THEN
    CALL display^spi^error (err, zspi^val^ssinit, 0d, true);

  sline ':=' "After SSINIT: " -> @sp;
  CALL writex (term, sline, @sp '-' @sline);
  CALL dump^buf (b1);

  !
  ! Put four tokens in the SPI buffer "b1"
  !
  val := "A";
  IF (err := ssputtkn (b1, tkn^1, val)) THEN
    CALL display^spi^error (err, zspi^val^ssputtkn, tkn^1, true);
  val := "B";
  IF (err := ssputtkn (b1, tkn^2, val)) THEN
    CALL display^spi^error (err, zspi^val^ssputtkn, tkn^2, true);
  val := "C";
  IF (err := ssputtkn (b1, tkn^3, val)) THEN
    CALL display^spi^error (err, zspi^val^ssputtkn, tkn^3, true);
  val := "D";
  IF (err := ssputtkn (b1, tkn^3, val)) THEN
    CALL display^spi^error (err, zspi^val^ssputtkn, tkn^3, true);

  sline ':=' "After SSPUT of second TKN^3: " -> @sp;
  CALL writex (term, sline, @sp '-' @sline);
  CALL dump^buf (b1);
```

Example E-1. TAL File: Basic Buffer Manipulations (page 2 of 2)

```
  !
  ! Reset the SPI buffer "b1"
  !
  IF (err := ssputtkn (b1, zspi^tkn^reset^buffer, bufsize)) THEN
    CALL display^spi^error (err, zspi^val^ssputtkn,
                            zspi^tkn^reset^buffer, true);

  sline ':=' "After RESET BUFFER: " -> @sp;
  CALL writex (term, sline, @sp '-' @sline);
  CALL dump^buf (b1);

  !
  ! Get the tokens in the SPI buffer "b1"
  !
  IF (err := ssgettkn (b1, tkn^1, val, 1)) THEN
    CALL display^spi^error (err, zspi^val^ssgettkn, tkn^1, true);

  sline ':=' "After GETTKN TKN^1: " -> @sp;
  CALL writex (term, sline, @sp '-' @sline);
  CALL dump^buf (b1);

  IF (err := ssgettkn (b1, zspi^tkn^nexttoken, tkn^code,,, ssid)) THEN
    CALL display^spi^error (err, zspi^val^ssgettkn,
                            zspi^tkn^nexttoken, true);

  sline ':=' "After GETTKN NEXTTOKEN: " -> @sp;
  CALL writex (term, sline, @sp '-' @sline);
  CALL dump^buf (b1);

  IF (err := ssgettkn (b1, tkn^2, val, 1)) THEN
    CALL display^spi^error (err, zspi^val^ssgettkn, tkn^2, true);

  sline ':=' "After GETTKN TKN^2: " -> @sp;
  CALL writex (term, sline, @sp '-' @sline);
  CALL dump^buf (b1);

  IF (err := ssgettkn (b1, tkn^3, val, 2)) THEN
    CALL display^spi^error (err, zspi^val^ssgettkn, tkn^3, true);

  sline ':=' "After GETTKN second TKN^3: " -> @sp;
  CALL writex (term, sline, @sp '-' @sline);
  CALL dump^buf (b1);

  sline ':=' "Program finished." -> @sp;
  CALL writex (term, sline, @sp '-' @sline);
  CALL stop;

END;
```

Example E-2: Basic Buffer Manipulations in C

The C source file in Example E-2 demonstrates the basic buffer manipulation activities illustrated in Figure 2-4 on page 2-18. The program uses DSM Template Services to display the contents of the buffer after each manipulation. Run the program to see the effect of SPI procedure calls on the contents of the buffer and the locations of the buffer pointers.

Source File

SEC0204C

Object File

SEC0204O

Example E-2. C File: Basic Buffer Manipulations (page 1 of 2)

```
/*   File name: sec0204c
 *   SPI EXAMPLE C 2-4
 *   Figure 2-4. Pointer Manipulation
 */
#pragma symbols
#pragma inspect
#pragma nomap
#pragma nolmap

#define max_bufsize  256      /* in bytes */

#include "secc.h"
#pragma list
#include "seccutlc"
#pragma page "MAIN"
main(/* int argc, char *argv[] */)
{

  bufsize = max_bufsize;

  /*
   * Initialize the SPI buffer "b1"
   */
  if (err = SSINIT (b1, bufsize, (short *) &ssid, ZSPI_VAL_CMDHDR))
    display_spi_error (err, ZSPI_VAL_SSINIT, 0L, true);

  printf ("After SSINIT: \n");
  dump_buf (b1);
  /*
   * Put four tokens in the SPI buffer "b1"
   */
  val = 'A';
  if (err = SSPUTTKN (b1, tkn_1, &val))
    display_spi_error (err, ZSPI_VAL_SSPUTTKN, tkn_1, true);
  val = 'B';
  if (err = SSPUTTKN (b1, tkn_2, &val))
    display_spi_error (err, ZSPI_VAL_SSPUTTKN, tkn_2, true);
  val = 'C';
```

Example E-2. C File: Basic Buffer Manipulations (page 2 of 2)

```
  if (err = SSPUTTKN (b1, tkn_3, &val))
    display_spi_error (err, ZSPI_VAL_SSPUTTKN, tkn_3, true);
  val = 'D';
  if (err = SSPUTTKN (b1, tkn_3, &val))
    display_spi_error (err, ZSPI_VAL_SSPUTTKN, tkn_3, true);

  printf ("After SSPUT of second TKN_3: \n");
  dump_buf (b1);

  /*
   * Reset the SPI buffer "b1"
   */
  if (err = SSPUTTKN (b1, ZSPI_TKN_RESET_BUFFER, (char *) &bufsize))
    display_spi_error (err, ZSPI_VAL_SSPUTTKN,
                            ZSPI_TKN_RESET_BUFFER, true);

  printf ("After RESET BUFFER: \n");
  dump_buf (b1);

  /*
   * Get the tokens in the SPI buffer "b1"
   */
  if (err = SSGETTKN (b1, tkn_1, &val, 1))
    display_spi_error (err, ZSPI_VAL_SSGETTKN, tkn_1, true);

  printf ("After GETTKN TKN_1: \n");
  dump_buf (b1);

  get_count = 1;
  if (err = SSGETTKN (b1, ZSPI_TKN_NEXTTOKEN, (char *) &tkn_code,
                      0, &get_count, (short *) &ssid))
    display_spi_error (err, ZSPI_VAL_SSGETTKN,
                            ZSPI_TKN_NEXTTOKEN, true);

  printf ("After GETTKN NEXTTOKEN: \n");
  dump_buf (b1);

  if (err = SSGETTKN (b1, tkn_2, &val, 1))
    display_spi_error (err, ZSPI_VAL_SSGETTKN, tkn_2, true);
  printf ("After GETTKN TKN_2: \n");
  dump_buf (b1);

  if (err = SSGETTKN (b1, tkn_3, &val, 2))
    display_spi_error (err, ZSPI_VAL_SSGETTKN, tkn_3, true);

  printf ("After GETTKN second TKN_3: \n");
  dump_buf (b1);

  printf ("Program finished.\n");
}
```

Example E-3: Working With Lists in TAL

Example E-3 on page E-10 demonstrates the basic buffer manipulation activities illustrated in Figure 2-5 on page 2-22. The TAL program uses DSM Template Services to display the contents of the buffer after each manipulation. Run the program to see SPI procedure calls used to move in and out of an SPI list.

Source File

SET0205

Object File

SET0205O

Example E-3. TAL File: Working With Lists (page 1 of 3)

```
--   File name: SET0205
--   SPI EXAMPLE TAL 2-5
--   Figure 2-5. Pointer Manipulation and lists
?SYMBOLS, INSPECT
LITERAL
        max^bufsize   = 256;     ! in bytes

?SOURCE SETCDECS
?NOLIST, SOURCE $system.system.extdecs0 (
?        abend, debug, dnumout, fileinfo, initializer, myterm, numout,
?        open, stop, writex,
?        spi_buffer_formatfinish_, spi_buffer_formatnext_,
?        spi_buffer_formatstart_,
?        ssinit, ssgettkn, ssmovetkn,
?        ssput, ssputtkn )
?LIST
?SOURCE SETCUTIL
?PAGE "PROC spitest MAIN"
PROC spitest MAIN;
BEGIN

  bufsize := max^bufsize;
  CALL initializer;
  CALL myterm(termname);
  CALL open(termname,term);
  IF <> THEN CALL abend;

  !
  ! Create the SPI buffer "b1"
  !
  IF (err := ssinit (b1, bufsize, ssid,zspi^val^cmdhdr)) THEN
    CALL display^spi^error (err, zspi^val^ssinit, 0d, true);

  val := "A";
  IF (err := ssputtkn (b1, tkn^1, val)) THEN
    CALL display^spi^error (err, zspi^val^ssputtkn, tkn^1, true);
  val := "B";
  IF (err := ssputtkn (b1, tkn^2, val)) THEN
    CALL display^spi^error (err, zspi^val^ssputtkn, tkn^2, true);
  IF (err := ssputtkn (b1, zspi^tkn^datalist)) THEN
    CALL display^spi^error (err, zspi^val^ssputtkn, zspi^tkn^datalist, true);
  val := "C";
  IF (err := ssputtkn (b1, tkn^3, val)) THEN
```

Example E-3. TAL File: Working With Lists (page 2 of 3)

```
   CALL display^spi^error (err, zspi^val^ssputtkn, tkn^3, true);
   val := "D";
  IF (err := ssputtkn (b1, tkn^4, val)) THEN
    CALL display^spi^error (err, zspi^val^ssputtkn, tkn^4, true);
  val := "E";
  IF (err := ssputtkn (b1, tkn^5, val)) THEN
    CALL display^spi^error (err, zspi^val^ssputtkn, tkn^5, true);
  IF (err := ssputtkn (b1, zspi^tkn^endlist)) THEN
    CALL display^spi^error (err, zspi^val^ssputtkn, zspi^tkn^endlist, true);

  val := "F";
  IF (err := ssputtkn (b1, tkn^6, val)) THEN
    CALL display^spi^error (err, zspi^val^ssputtkn, tkn^6, true);
sline ':=' "After RESET BUFFER: " -> @sp;
  CALL writex (term, sline, @sp '-' @sline);
  CALL dump^buf (b1);

  !
  ! Get the tokens in the SPI buffer "b1"
  !
  IF (err := ssgettkn (b1, tkn^1, val, 1)) THEN
    CALL display^spi^error (err, zspi^val^ssgettkn, tkn^1, true);

  sline ':=' "After GETTKN TKN^1: " -> @sp;
  CALL writex (term, sline, @sp '-' @sline);
  CALL dump^buf (b1);

  IF (err := ssgettkn (b1, tkn^2, val, 1)) THEN
    CALL display^spi^error (err, zspi^val^ssgettkn, tkn^2, true);

  sline ':=' "After GETTKN TKN^2: " -> @sp;
  CALL writex (term, sline, @sp '-' @sline);
  CALL dump^buf (b1);

  -- This call should get an error (missing token)
  sline ':=' "After GETTKN TKN^3: " -> @sp;
  CALL writex (term, sline, @sp '-' @sline);
  IF (err := ssgettkn (b1, tkn^3, val, 1)) THEN
    CALL display^spi^error (err, zspi^val^ssgettkn, tkn^3, false);

  IF (err := ssgettkn (b1, zspi^tkn^datalist)) THEN
    CALL display^spi^error (err, zspi^val^ssgettkn, zspi^tkn^datalist, true);

  sline ':=' "After GETTKN DATALIST: " -> @sp;
  CALL writex (term, sline, @sp '-' @sline);
  CALL dump^buf (b1);

  IF (err := ssgettkn (b1, tkn^3, val, 1)) THEN
    CALL display^spi^error (err, zspi^val^ssgettkn, tkn^3, true);

  sline ':=' "After GETTKN TKN^3: " -> @sp;
  CALL writex (term, sline, @sp '-' @sline);
  CALL dump^buf (b1);

  IF (err := ssgettkn (b1, tkn^5, val, 1)) THEN
    CALL display^spi^error (err, zspi^val^ssgettkn, tkn^5, true);

  sline ':=' "After GETTKN TKN^5: " -> @sp;
  CALL writex (term, sline, @sp '-' @sline);
  CALL dump^buf (b1);

  -- This call should get an error (missing token)
  sline ':=' "After GETTKN TKN^6: " -> @sp;
```

Example E-3. TAL File: Working With Lists (page 3 of 3)

```
  CALL writex (term, sline, @sp '-' @sline);
  IF (err := ssgettkn (b1, tkn^6, val, 1)) THEN
    CALL display^spi^error (err, zspi^val^ssgettkn, tkn^6, false);

  IF (err := ssgettkn (b1, zspi^tkn^endlist)) THEN
    CALL display^spi^error (err, zspi^val^ssgettkn, zspi^tkn^endlist, true);
  !
  ! Reset the SPI buffer "b1"
  !
  IF (err := ssputtkn (b1, zspi^tkn^reset^buffer, bufsize)) THEN
    CALL display^spi^error (err, zspi^val^ssputtkn,
                            zspi^tkn^reset^buffer, true);
  sline ':=' "After GETTKN ENDLIST: " -> @sp;
  CALL writex (term, sline, @sp '-' @sline);
  CALL dump^buf (b1);

  IF (err := ssgettkn (b1, tkn^6, val, 1)) THEN
    CALL display^spi^error (err, zspi^val^ssgettkn, tkn^6, true);


 sline ':=' "After GETTKN TKN^6: " -> @sp;
  CALL writex (term, sline, @sp '-' @sline);
  CALL dump^buf (b1);

  sline ':=' "Program finished." -> @sp;
  CALL writex (term, sline, @sp '-' @sline);
  CALL stop;

END;
```

Example E-4: Working With Lists in C

Example E-4 on page E-13 demonstrates the basic buffer manipulation activities illustrated in Figure 2-5 on page 2-22. The C program uses DSM Template Services to display the contents of the buffer after each manipulation. Run the program to see SPI procedure calls used to move in and out of an SPI list.

Source File

SEC0205C

Object File

SEC0205O

Example E-4. C File: Working With Lists (page 1 of 2)

```
/*   File name: sec0205c
 *   SPI EXAMPLE C 2-5
 *   Figure 2-5. Pointer Manipulation and lists
 */
#pragma symbols
#pragma inspect
#pragma nomap
#pragma nolmap

#define max_bufsize  256      /* in bytes */

#include "secc.h"
#pragma list
#include "seccutlc"
#pragma PAGE "MAIN"
main(/* int argc, char *argv[] */)
{

  bufsize = max_bufsize;

  /*
   * Create the SPI buffer "b1"
   */
  if (err = SSINIT (b1, bufsize, (short *) &ssid, ZSPI_VAL_CMDHDR))
    display_spi_error (err, ZSPI_VAL_SSINIT, 0L, true);

  val = 'A';
  if (err = SSPUTTKN (b1, tkn_1, &val))
    display_spi_error (err, ZSPI_VAL_SSPUTTKN, tkn_1, true);
  val = 'B';
  if (err = SSPUTTKN (b1, tkn_2, &val))
    display_spi_error (err, ZSPI_VAL_SSPUTTKN, tkn_2, true);
  if (err = SSPUTTKN (b1, ZSPI_TKN_DATALIST))
    display_spi_error (err, ZSPI_VAL_SSPUTTKN, ZSPI_TKN_DATALIST, true);
  val = 'C';
  if (err = SSPUTTKN (b1, tkn_3, &val))
    display_spi_error (err, ZSPI_VAL_SSPUTTKN, tkn_3, true);
  val = 'D';
  if (err = SSPUTTKN (b1, tkn_4, &val))
    display_spi_error (err, ZSPI_VAL_SSPUTTKN, tkn_4, true);
  val = 'E';
  if (err = SSPUTTKN (b1, tkn_5, &val))
    display_spi_error (err, ZSPI_VAL_SSPUTTKN, tkn_5, true);
  if (err = SSPUTTKN (b1, ZSPI_TKN_ENDLIST))
    display_spi_error (err, ZSPI_VAL_SSPUTTKN, ZSPI_TKN_ENDLIST, true);
  val = 'F';
  if (err = SSPUTTKN (b1, tkn_6, &val))
    display_spi_error (err, ZSPI_VAL_SSPUTTKN, tkn_6, true);
  /*
   * Reset the SPI buffer "b1"
   */
  if (err = SSPUTTKN (b1, ZSPI_TKN_RESET_BUFFER, (char *) &bufsize))
    display_spi_error (err, ZSPI_VAL_SSPUTTKN,
                            ZSPI_TKN_RESET_BUFFER, true);
```

Example E-4. C File: Working With Lists (page 2 of 2)

```
  printf ("After RESET BUFFER: \n");
  dump_buf (b1);

  /*
   * Get the tokens in the SPI buffer "b1"
   */
  if (err = SSGETTKN (b1, tkn_1, &val, 1))
    display_spi_error (err, ZSPI_VAL_SSGETTKN, tkn_1, true);

  printf ("After GETTKN TKN_1: \n");
  dump_buf (b1);

  if (err = SSGETTKN (b1, tkn_2, &val, 1))
    display_spi_error (err, ZSPI_VAL_SSGETTKN, tkn_2, true);

  printf ("After GETTKN TKN_2: \n");
  dump_buf (b1);

  /* This should get an error (missing token) */
  printf ("After GETTKN TKN_3: \n");
  if (err = SSGETTKN (b1, tkn_3, &val, 1))
    display_spi_error (err, ZSPI_VAL_SSGETTKN, tkn_3, false);

  if (err = SSGETTKN (b1, ZSPI_TKN_DATALIST))
    display_spi_error (err, ZSPI_VAL_SSGETTKN, ZSPI_TKN_DATALIST, true);

  printf ("After GETTKN DATALIST: \n");
  dump_buf (b1);

  if (err = SSGETTKN (b1, tkn_3, &val, 1))
    display_spi_error (err, ZSPI_VAL_SSGETTKN, tkn_3, true);

  printf ("After GETTKN TKN_3: \n");
  dump_buf (b1);

  if (err = SSGETTKN (b1, tkn_5, &val, 1))
    display_spi_error (err, ZSPI_VAL_SSGETTKN, tkn_5, true);

  printf ("After GETTKN TKN_5: \n");
  dump_buf (b1);

  /* This should get an error (missing token) */
  printf ("After GETTKN TKN_6: \n");
  if (err = SSGETTKN (b1, tkn_6, &val, 1))
    display_spi_error (err, ZSPI_VAL_SSGETTKN, tkn_6, false);

  if (err = SSGETTKN (b1, ZSPI_TKN_ENDLIST))
    display_spi_error (err, ZSPI_VAL_SSGETTKN, ZSPI_TKN_ENDLIST, true);

  printf ("After GETTKN ENDLIST: \n");
  dump_buf (b1);


  if (err = SSGETTKN (b1, tkn_6, &val, 1))
    display_spi_error (err, ZSPI_VAL_SSGETTKN, tkn_6, true);

  printf ("After GETTKN TKN_6: \n");
  dump_buf (b1);

  printf ("Program finished.\n");
}
```

Example E-5: Displaying SPI Buffer Contents With TAL

Example E-5 demonstrates the basic buffer manipulation activities illustrated in Figure 2-6 on page 2-24. This TAL program uses DSM Template Services to display the contents of the buffer after each manipulation. Run the program to see the behavior of the SSGET special operation ZSPI-TKN-NEXTTOKEN.

Source File

SET0206

Object File

SET0206O

Example E-5. TAL File: Pointers, Lists, and ZSPI-TKN-NEXTTOKEN (page 1 of 3)

```
--   File name: SET0206
--   SPI EXAMPLE TAL 2-6
--   Figure 2-6. Pointer Manipulation and lists
?SYMBOLS, INSPECT
LITERAL
        max^bufsize   = 256;     ! in bytes

?SOURCE SETCDECS
?NOLIST, SOURCE $system.system.extdecs0 (
?         abend, debug, dnumout, fileinfo, initializer, myterm, numout,
?         open, stop, writex,
?         spi_buffer_formatfinish_, spi_buffer_formatnext_,
?         spi_buffer_formatstart_,
?         ssinit, ssgettkn, ssmovetkn,
?         ssput, ssputtkn )
?LIST
?SOURCE SETCUTIL
?PAGE "PROC spitest MAIN"
PROC spitest MAIN;
BEGIN

  bufsize := max^bufsize;
  CALL initializer;
  CALL myterm(termname);
  CALL open(termname,term);
  IF <> THEN CALL abend;
!
```

Example E-5. TAL File: Pointers, Lists, and ZSPI-TKN-NEXTTOKEN (page 2 of 3)

```
! Create the SPI buffer "b1"
!
IF (err := ssinit (b1, bufsize, ssid,zspi^val^cmdhdr)) THEN
  CALL display^spi^error (err, zspi^val^ssinit, 0d, true);

val := "A";
IF (err := ssputtkn (b1, tkn^1, val)) THEN
  CALL display^spi^error (err, zspi^val^ssputtkn, tkn^1, true);
IF (err := ssputtkn (b1, zspi^tkn^datalist)) THEN
  CALL display^spi^error (err, zspi^val^ssputtkn, zspi^tkn^datalist, true);
val := "B";
IF (err := ssputtkn (b1, tkn^2, val)) THEN
  CALL display^spi^error (err, zspi^val^ssputtkn, tkn^2, true);
val := "C";
IF (err := ssputtkn (b1, tkn^3, val)) THEN
  CALL display^spi^error (err, zspi^val^ssputtkn, tkn^3, true);
val := "D";
IF (err := ssputtkn (b1, tkn^3, val)) THEN
  CALL display^spi^error (err, zspi^val^ssputtkn, tkn^3, true);
IF (err := ssputtkn (b1, zspi^tkn^endlist)) THEN
  CALL display^spi^error (err, zspi^val^ssputtkn, zspi^tkn^endlist, true);
val := "E";
IF (err := ssputtkn (b1, tkn^4, val)) THEN
  CALL display^spi^error (err, zspi^val^ssputtkn, tkn^4, true);

!
! Reset the SPI buffer "b1"
!
IF (err := ssputtkn (b1, zspi^tkn^reset^buffer, bufsize)) THEN
  CALL display^spi^error (err, zspi^val^ssputtkn,
                          zspi^tkn^reset^buffer, true);

sline ':=' "After RESET BUFFER: " -> @sp;
CALL writex (term, sline, @sp '-' @sline);
CALL dump^buf (b1);

!
! Get the tokens in the SPI buffer "b1"
!
IF (err := ssgettkn (b1, zspi^tkn^nexttoken, tkn^code,,, ssid)) THEN
  CALL display^spi^error (err, zspi^val^ssgettkn,
                          zspi^tkn^nexttoken, true);

sline ':=' "After 1st GETTKN NEXTTOKEN: Token = " -> @sp;
@sp := @sp '+' dnumout (sp, tkn^code, 10);
CALL display^token (tkn^code);
CALL writex (term, sline, @sp '-' @sline);
CALL dump^buf (b1);

IF (err := ssgettkn (b1, zspi^tkn^nexttoken, tkn^code,,, ssid)) THEN
  CALL display^spi^error (err, zspi^val^ssgettkn,
                          zspi^tkn^nexttoken, true);

sline ':=' "After 2nd GETTKN NEXTTOKEN: Token = " -> @sp;
@sp := @sp '+' dnumout (sp, tkn^code, 10);
CALL display^token (tkn^code);
CALL writex (term, sline, @sp '-' @sline);
CALL dump^buf (b1);

IF (err := ssgettkn (b1, zspi^tkn^nexttoken, tkn^code,,, ssid)) THEN
  CALL display^spi^error (err, zspi^val^ssgettkn,
                          zspi^tkn^nexttoken, true);
```

Example E-5. TAL File: Pointers, Lists, and ZSPI-TKN-NEXTTOKEN (page 3 of 3)

```
  sline ':=' "After 3rd GETTKN NEXTTOKEN: Token = " -> @sp;
  @sp := @sp '+' dnumout (sp, tkn^code, 10);
  CALL display^token (tkn^code);
  CALL writex (term, sline, @sp '-' @sline);
  CALL dump^buf (b1);

  -- This call should get an error (missing token)
  sline ':=' "After GETTKN TKN^4: " -> @sp;
  CALL writex (term, sline, @sp '-' @sline);
  IF (err := ssgettkn (b1, tkn^4, val, 1)) THEN
    CALL display^spi^error (err, zspi^val^ssgettkn, tkn^4, false);

  IF (err := ssgettkn (b1, tkn^3, val, 2)) THEN
    CALL display^spi^error (err, zspi^val^ssgettkn, tkn^3, true);

  sline ':=' "After GETTKN of 2nd TKN^3: " -> @sp;
  @sp := @sp '+' dnumout (sp, tkn^code, 10);
  CALL display^token (tkn^code);
  CALL writex (term, sline, @sp '-' @sline);
  CALL dump^buf (b1);

  IF (err := ssgettkn (b1, zspi^tkn^nexttoken, tkn^code,,, ssid)) THEN
    CALL display^spi^error (err, zspi^val^ssgettkn,
                            zspi^tkn^nexttoken, true);

  sline ':=' "After 4th GETTKN NEXTTOKEN: Token = " -> @sp;
  @sp := @sp '+' dnumout (sp, tkn^code, 10);
  CALL display^token (tkn^code);
  CALL writex (term, sline, @sp '-' @sline);
  CALL dump^buf (b1);

  IF (err := ssgettkn (b1, zspi^tkn^nexttoken, tkn^code,,, ssid)) THEN
    CALL display^spi^error (err, zspi^val^ssgettkn,
                            zspi^tkn^nexttoken, true);

  sline ':=' "After 5th GETTKN NEXTTOKEN: Token = " -> @sp;
  @sp := @sp '+' dnumout (sp, tkn^code, 10);
  CALL display^token (tkn^code);
  CALL writex (term, sline, @sp '-' @sline);
  CALL dump^buf (b1);

  -- This call should get an error (missing token)
  sline ':=' "After 6th GETTKN NEXTTOKEN: " -> @sp;
  CALL writex (term, sline, @sp '-' @sline);
  IF (err := ssgettkn (b1, zspi^tkn^nexttoken, tkn^code,,, ssid)) THEN
    CALL display^spi^error (err, zspi^val^ssgettkn,
                            zspi^tkn^nexttoken, false);

  sline ':=' "Program finished." -> @sp;
  CALL writex (term, sline, @sp '-' @sline);
  CALL stop;

END;
```

Example E-6: Displaying SPI Buffer Contents With C

Example E-6 on page E-19 demonstrates the basic buffer manipulation activities illustrated in Figure 2-6 on page 2-24. This C program uses DSM Template Services to display the contents of the buffer after each manipulation. Run the program to see the behavior of the SSGET special operation ZSPI-TKN-NEXTTOKEN.

Source File

SEC0206C

Object File

SEC0206O

Example E-6. C File: Pointers, Lists, and ZSPI-TKN-NEXTTOKEN (page 1 of 3)

```
/*   File name: sec0206c
 *   SPI EXAMPLE C 2-6
 *   Figure 2-6. Pointer Manipulation and lists
 */
#pragma symbols
#pragma inspect
#pragma nomap
#pragma nolmap

#define max_bufsize  256      /* in bytes */

#include "secc.h"
#pragma list
#include "seccutlc"
#pragma PAGE "MAIN"
main(/* int argc, char *argv[] */)
{

  bufsize = max_bufsize;

  /*
   * Create the SPI buffer "b1"
   */
  if (err = SSINIT (b1, bufsize, (short *) &ssid, ZSPI_VAL_CMDHDR))
    display_spi_error (err, ZSPI_VAL_SSINIT, 0L, true);

  val = 'A';
  if (err = SSPUTTKN (b1, tkn_1, &val))
    display_spi_error (err, ZSPI_VAL_SSPUTTKN, tkn_1, true);
  if (err = SSPUTTKN (b1, ZSPI_TKN_DATALIST))
    display_spi_error (err, ZSPI_VAL_SSPUTTKN, ZSPI_TKN_DATALIST, true);
  val = 'B';
  if (err = SSPUTTKN (b1, tkn_2, &val))
    display_spi_error (err, ZSPI_VAL_SSPUTTKN, tkn_2, true);
  val = 'C';

  if (err = SSPUTTKN (b1, tkn_3, &val))
    display_spi_error (err, ZSPI_VAL_SSPUTTKN, tkn_3, true);
  val = 'D';
  if (err = SSPUTTKN (b1, tkn_3, &val))
    display_spi_error (err, ZSPI_VAL_SSPUTTKN, tkn_3, true);
  if (err = SSPUTTKN (b1, ZSPI_TKN_ENDLIST))
    display_spi_error (err, ZSPI_VAL_SSPUTTKN, ZSPI_TKN_ENDLIST, true);
```

Example E-6. C File: Pointers, Lists, and ZSPI-TKN-NEXTTOKEN (page 2 of 3)

```
 val = 'E';
 if (err = SSPUTTKN (b1, tkn_4, &val))
  display_spi_error (err, ZSPI_VAL_SSPUTTKN, tkn_4, true);

/*
 * Reset the SPI buffer "b1"
 */
if (err = SSPUTTKN (b1, ZSPI_TKN_RESET_BUFFER, (char *) &bufsize))
  display_spi_error (err, ZSPI_VAL_SSPUTTKN,
                          ZSPI_TKN_RESET_BUFFER, true);

printf ("After RESET BUFFER: \n");
dump_buf (b1);

/*
 * Get the tokens in the SPI buffer "b1"
 */
get_count = 1;
if (err = SSGETTKN (b1, ZSPI_TKN_NEXTTOKEN, (char *) &tkn_code,
                        0, &get_count, (short *) &ssid))
  display_spi_error (err, ZSPI_VAL_SSGETTKN,
                          ZSPI_TKN_NEXTTOKEN, true);

printf ("After 1st GETTKN NEXTTOKEN: Token = %ld", tkn_code);
display_token (tkn_code);
printf ("\n");
dump_buf (b1);

if (err = SSGETTKN (b1, ZSPI_TKN_NEXTTOKEN, (char *) &tkn_code,
                        0, &get_count, (short *) &ssid))
  display_spi_error (err, ZSPI_VAL_SSGETTKN,
                          ZSPI_TKN_NEXTTOKEN, true);

printf ("After 2nd GETTKN NEXTTOKEN: Token = %ld", tkn_code);
display_token (tkn_code);
printf ("\n");
dump_buf (b1);

if (err = SSGETTKN (b1, ZSPI_TKN_NEXTTOKEN, (char *) &tkn_code,
                        0, &get_count, (short *) &ssid))
  display_spi_error (err, ZSPI_VAL_SSGETTKN,
                          ZSPI_TKN_NEXTTOKEN, true);

printf ("After 3rd GETTKN NEXTTOKEN: Token = %ld", tkn_code);
display_token (tkn_code);
printf ("\n");
dump_buf (b1);

/* This should get an error (missing token) */
printf ("After GETTKN TKN_4: \n");
if (err = SSGETTKN (b1, tkn_4, &val, 1))
  display_spi_error (err, ZSPI_VAL_SSGETTKN, tkn_4, false);

if (err = SSGETTKN (b1, tkn_3, &val, 2))
  display_spi_error (err, ZSPI_VAL_SSGETTKN, tkn_3, true);

printf ("After GETTKN of 2nd TKN_3: %ld", tkn_code);
display_token (tkn_code);
printf ("\n");
dump_buf (b1);

if (err = SSGETTKN (b1, ZSPI_TKN_NEXTTOKEN, (char *) &tkn_code,
                        0, &get_count, (short *) &ssid))
  display_spi_error (err, ZSPI_VAL_SSGETTKN,
                          ZSPI_TKN_NEXTTOKEN, true);
```

Example E-6. C File: Pointers, Lists, and ZSPI-TKN-NEXTTOKEN (page 3 of 3)

```
  printf ("After 4th GETTKN NEXTTOKEN: Token = %ld", tkn_code);
  display_token (tkn_code);
  printf ("\n");
  dump_buf (b1);

  if (err = SSGETTKN (b1, ZSPI_TKN_NEXTTOKEN, (char *) &tkn_code,
                      0, &get_count, (short *) &ssid))
    display_spi_error (err, ZSPI_VAL_SSGETTKN,
                            ZSPI_TKN_NEXTTOKEN, true);

  printf ("After 5th GETTKN NEXTTOKEN: Token = %ld", tkn_code);
  display_token (tkn_code);
  printf ("\n");
  dump_buf (b1);

  /* This should get an error (missing token) */
  printf ("After 6th GETTKN NEXTTOKEN: \n");
  if (err = SSGETTKN (b1, ZSPI_TKN_NEXTTOKEN, (char *) &tkn_code,
                      0, &get_count, (short *) &ssid))
    display_spi_error (err, ZSPI_VAL_SSGETTKN,
                            ZSPI_TKN_NEXTTOKEN, false);

  printf ("Program finished.\n");
}
```

Example E-7: Special SSGET Operation in TAL

Example E-7 on page E-22 demonstrates the basic buffer manipulation activities illustrated in Figure 2-7 on page 2-26. This TAL program uses DSM Template Services to display the contents of the buffer after each manipulation. Run the program to see the behavior of the SSGET special operation ZSPI-TKN-NEXTCODE.

Source File

SET0207

Object File

SET0207O

Example E-7. TAL File: Pointers, Lists, and ZSPI-TKN-NEXTCODE (page 1 of 3)

```
--   File name: SET0207
--   SPI EXAMPLE TAL 2-7
--   Figure 2-7. Pointer Manipulation and lists
?SYMBOLS, INSPECT
LITERAL
        max^bufsize   = 256;     ! in bytes

?SOURCE SETCDECS
?NOLIST, SOURCE $system.system.extdecs0 (
?        abend, debug, dnumout, fileinfo, initializer, myterm, numout,
?        open, stop, writex,
?        spi_buffer_formatfinish_, spi_buffer_formatnext_,
?        spi_buffer_formatstart_,
?        ssinit, ssgettkn, ssmovetkn,
?        ssput, ssputtkn )
?LIST
?SOURCE SETCUTIL
?PAGE "PROC spitest MAIN"
PROC spitest MAIN;
BEGIN

  bufsize := max^bufsize;
  CALL initializer;
  CALL myterm(termname);
  CALL open(termname,term);
  IF <> THEN CALL abend;

  !
  ! Create the SPI buffer "b1"
  !
  IF (err := ssinit (b1, bufsize, ssid, zspi^val^cmdhdr)) THEN
    CALL display^spi^error (err, zspi^val^ssinit, 0d, true);

  val := "A";
  IF (err := ssputtkn (b1, tkn^1, val)) THEN
    CALL display^spi^error (err, zspi^val^ssputtkn, tkn^1, true);
  IF (err := ssputtkn (b1, zspi^tkn^datalist)) THEN
    CALL display^spi^error (err, zspi^val^ssputtkn,
                            zspi^tkn^datalist, true);
  val := "B";
  IF (err := ssputtkn (b1, tkn^2, val)) THEN
    CALL display^spi^error (err, zspi^val^ssputtkn, tkn^2, true);
  val := "C";
  IF (err := ssputtkn (b1, tkn^3, val)) THEN
    CALL display^spi^error (err, zspi^val^ssputtkn, tkn^3, true);
  val := "D";
  IF (err := ssputtkn (b1, tkn^3, val)) THEN
    CALL display^spi^error (err, zspi^val^ssputtkn, tkn^3, true);
  IF (err := ssputtkn (b1, zspi^tkn^endlist)) THEN
    CALL display^spi^error (err, zspi^val^ssputtkn,
                            zspi^tkn^endlist, true);
  val := "E";
  IF (err := ssputtkn (b1, tkn^4, val)) THEN
    CALL display^spi^error (err, zspi^val^ssputtkn, tkn^4, true);
```

Example E-7. TAL File: Pointers, Lists, and ZSPI-TKN-NEXTCODE (page 2 of 3)

```
 !
! Reset the SPI buffer "b1"
!
IF (err := ssputtkn (b1, zspi^tkn^reset^buffer, bufsize)) THEN
  CALL display^spi^error (err, zspi^val^ssputtkn,
                          zspi^tkn^reset^buffer, true);

sline ':=' "After RESET BUFFER: " -> @sp;
CALL writex (term, sline, @sp '-' @sline);
CALL dump^buf (b1);

!
! Get the tokens in the SPI buffer "b1"
!
IF (err := ssgettkn (b1, zspi^tkn^nextcode, tkn^code,,, ssid)) THEN
  CALL display^spi^error (err, zspi^val^ssgettkn,
                          zspi^tkn^nextcode, true);

sline ':=' "After 1st GETTKN NEXTCODE: Token = " -> @sp;
@sp := @sp '+' dnumout (sp, tkn^code, 10);
CALL display^token (tkn^code);
CALL writex (term, sline, @sp '-' @sline);
CALL dump^buf (b1);

IF (err := ssgettkn (b1, zspi^tkn^nextcode, tkn^code,,, ssid)) THEN
  CALL display^spi^error (err, zspi^val^ssgettkn,
                          zspi^tkn^nextcode, true);

sline ':=' "After 2nd GETTKN NEXTCODE: Token = " -> @sp;
@sp := @sp '+' dnumout (sp, tkn^code, 10);
CALL display^token (tkn^code);
CALL writex (term, sline, @sp '-' @sline);
CALL dump^buf (b1);

IF (err := ssgettkn (b1, zspi^tkn^nextcode, tkn^code,,, ssid)) THEN
  CALL display^spi^error (err, zspi^val^ssgettkn,
                          zspi^tkn^nextcode, true);

sline ':=' "After 3rd GETTKN NEXTCODE: Token = " -> @sp;
@sp := @sp '+' dnumout (sp, tkn^code, 10);
CALL display^token (tkn^code);
CALL writex (term, sline, @sp '-' @sline);
CALL dump^buf (b1);

IF (err := ssgettkn (b1, zspi^tkn^datalist,, 1)) THEN
  CALL display^spi^error (err, zspi^val^ssgettkn,
                          zspi^tkn^datalist, true);

sline ':=' "After GETTKN of DATALIST: " -> @sp;
@sp := @sp '+' dnumout (sp, tkn^code, 10);
CALL writex (term, sline, @sp '-' @sline);
CALL dump^buf (b1);

IF (err := ssgettkn (b1, zspi^tkn^nextcode, tkn^code,,, ssid)) THEN
  CALL display^spi^error (err, zspi^val^ssgettkn,
                          zspi^tkn^nextcode, true);
```

Example E-7. TAL File: Pointers, Lists, and ZSPI-TKN-NEXTCODE (page 3 of 3)

```
  sline ':=' "After 4th GETTKN NEXTCODE: Token = " -> @sp;
  @sp := @sp '+' dnumout (sp, tkn^code, 10);
  CALL display^token (tkn^code);
  CALL writex (term, sline, @sp '-' @sline);
  CALL dump^buf (b1);

  IF (err := ssgettkn (b1, zspi^tkn^nextcode, tkn^code,,, ssid)) THEN
    CALL display^spi^error (err, zspi^val^ssgettkn,
                            zspi^tkn^nextcode, true);

  sline ':=' "After 5th GETTKN NEXTCODE: Token = " -> @sp;
  @sp := @sp '+' dnumout (sp, tkn^code, 10);
  CALL display^token (tkn^code);
  CALL writex (term, sline, @sp '-' @sline);
  CALL dump^buf (b1);

  IF (err := ssgettkn (b1, zspi^tkn^nextcode, tkn^code,,, ssid)) THEN
    CALL display^spi^error (err, zspi^val^ssgettkn,
                            zspi^tkn^nextcode, false);

  sline ':=' "After 6th GETTKN NEXTCODE: Token = " -> @sp;
  @sp := @sp '+' dnumout (sp, tkn^code, 10);
  CALL display^token (tkn^code);
  CALL writex (term, sline, @sp '-' @sline);
  CALL dump^buf (b1);

  -- This call should get an error (missing token)
  sline ':=' "After GETTKN TKN^4: " -> @sp;
  CALL writex (term, sline, @sp '-' @sline);
  IF (err := ssgettkn (b1, tkn^4, val, 1)) THEN
    CALL display^spi^error (err, zspi^val^ssgettkn, tkn^4, false);

  sline ':=' "Program finished." -> @sp;
  CALL writex (term, sline, @sp '-' @sline);
  CALL stop;

END;
```

Example E-8: Special SSGET Operation in C

Example E-8 on page E-25 demonstrates the basic buffer manipulation activities illustrated in Figure 2-7 on page 2-26. This C program uses DSM Template Services to display the contents of the buffer after each manipulation. Run the program to see the behavior of the SSGET special operation ZSPI-TKN-NEXTCODE.

Source File

SEC0207C

Object File

SEC0207O

Example E-8. C File: Pointers, Lists, and ZSPI-TKN-NEXTCODE (page 1 of 3)

```
/*   File name: sec0207c
 *   SPI EXAMPLE C 2-7
 *   Figure 2-7. Pointer Manipulation and lists
 */
#pragma symbols
#pragma inspect
#pragma nomap
#pragma nolmap

#define max_bufsize  256      /* in bytes */

#include "secc.h"
#pragma list
#include "seccutlc"
#pragma PAGE "MAIN"
main(/* int argc, char *argv[] */)
{

  bufsize = max_bufsize;

  /*
   * Create the SPI buffer "b1"
   */
  if (err = SSINIT (b1, bufsize, (short *) &ssid, ZSPI_VAL_CMDHDR))
    display_spi_error (err, ZSPI_VAL_SSINIT, 0L, true);

  val = 'A';
  if (err = SSPUTTKN (b1, tkn_1, &val))
    display_spi_error (err, ZSPI_VAL_SSPUTTKN, tkn_1, true);
  if (err = SSPUTTKN (b1, ZSPI_TKN_DATALIST))
    display_spi_error (err, ZSPI_VAL_SSPUTTKN,
                            ZSPI_TKN_DATALIST, true);
  val = 'B';
  if (err = SSPUTTKN (b1, tkn_2, &val))
    display_spi_error (err, ZSPI_VAL_SSPUTTKN, tkn_2, true);
  val = 'C';
  if (err = SSPUTTKN (b1, tkn_3, &val))
    display_spi_error (err, ZSPI_VAL_SSPUTTKN, tkn_3, true);
  val = 'D';
  if (err = SSPUTTKN (b1, tkn_3, &val))
    display_spi_error (err, ZSPI_VAL_SSPUTTKN, tkn_3, true);
  if (err = SSPUTTKN (b1, ZSPI_TKN_ENDLIST))
    display_spi_error (err, ZSPI_VAL_SSPUTTKN,
                            ZSPI_TKN_ENDLIST, true);
  val = ''E';
  if (err = SSPUTTKN (b1, tkn_4, &val))
    display_spi_error (err, ZSPI_VAL_SSPUTTKN, tkn_4, true);

  /*
   * Reset the SPI buffer "b1"
   */
  if (err = SSPUTTKN (b1, ZSPI_TKN_RESET_BUFFER, (char *) &bufsize))
    display_spi_error (err, ZSPI_VAL_SSPUTTKN,
                            ZSPI_TKN_RESET_BUFFER, true);

  printf ("After RESET BUFFER: \n");
  dump_buf (b1);
```

Example E-8. C File: Pointers, Lists, and ZSPI-TKN-NEXTCODE (page 2 of 3)

```
/*
 * Get the tokens in the SPI buffer "b1"
 */
get_count = 1;
if (err = SSGETTKN (b1, ZSPI_TKN_NEXTCODE, (char *) &tkn_code,
                        0, &get_count, (short *) &ssid))
  display_spi_error (err, ZSPI_VAL_SSGETTKN,
                          ZSPI_TKN_NEXTCODE, true);

printf ("After 1st GETTKN NEXTCODE: Token = %ld", tkn_code);
display_token (tkn_code);
printf ("\n");
dump_buf (b1);

if (err = SSGETTKN (b1, ZSPI_TKN_NEXTCODE, (char *) &tkn_code,
                        0, &get_count, (short *) &ssid))
  display_spi_error (err, ZSPI_VAL_SSGETTKN,
                          ZSPI_TKN_NEXTCODE, true);

printf ("After 2nd GETTKN NEXTCODE: Token = %ld", tkn_code);
display_token (tkn_code);
printf ("\n");
dump_buf (b1);

if (err = SSGETTKN (b1, ZSPI_TKN_NEXTCODE, (char *) &tkn_code,
                        0, &get_count, (short *) &ssid))
  display_spi_error (err, ZSPI_VAL_SSGETTKN,
                          ZSPI_TKN_NEXTCODE, true);

printf ("After 3rd GETTKN NEXTCODE: Token = %ld", tkn_code);
display_token (tkn_code);
printf ("\n");
dump_buf (b1);

if (err = SSGETTKN (b1, ZSPI_TKN_DATALIST, (char *) &tkn_code, 1))
  display_spi_error (err, ZSPI_VAL_SSGETTKN,
                          ZSPI_TKN_DATALIST, true);

printf ("After GETTKN of DATALIST: %d\n", tkn_code);
dump_buf (b1);

if (err = SSGETTKN (b1, ZSPI_TKN_NEXTCODE, (char *) &tkn_code,
                        0, &get_count, (short *) &ssid))
  display_spi_error (err, ZSPI_VAL_SSGETTKN,
                          ZSPI_TKN_NEXTCODE, true);

printf ("After 4th GETTKN NEXTCODE: Token = %ld", tkn_code);
display_token (tkn_code);
printf ("\n");
dump_buf (b1);

if (err = SSGETTKN (b1, ZSPI_TKN_NEXTCODE, (char *) &tkn_code,
                        0, &get_count, (short *) &ssid))
  display_spi_error (err, ZSPI_VAL_SSGETTKN,
                          ZSPI_TKN_NEXTCODE, true);

printf ("After 5th GETTKN NEXTCODE: Token = %ld", tkn_code);
display_token (tkn_code);
printf ("\n");
dump_buf (b1);
```

Example E-8. C File: Pointers, Lists, and ZSPI-TKN-NEXTCODE (page 3 of 3)

```
  if (err = SSGETTKN (b1, ZSPI_TKN_NEXTCODE, (char *) &tkn_code,
                      0, &get_count, (short *) &ssid))
    display_spi_error (err, ZSPI_VAL_SSGETTKN,
                            ZSPI_TKN_NEXTCODE, false);

  printf ("After 6th GETTKN NEXTCODE: Token = %ld", tkn_code);
  display_token (tkn_code);
  printf ("\n"');
  dump_buf (b1);

  /* This should get an error (missing token) */
  printf (""After GETTKN TKN_4: \n");
  if (err = SSGETTKN (b1, tkn_4, &val, 1))
    display_spi_error (err, ZSPI_VAL_SSGETTKN, tkn_4, false);

  printf ("Program finished.\n");
}
```

Example E-9: A Simple SPI Requester in TAL

Example E-9 on page E-28 is a simple SPI requester that sends commands to the server shown in Example E-11 on page E-45. The requester starts and stops the server automatically. The requester's commands have the server perform simple string manipulations on a text string that you enter. This TAL program gives you the option of displaying the contents of the SPI messages that are exchanged by the requester and the server. The second option, which shifts a string to upper case, uses response continuation to return the results one character per message. Run the program and enter one of the displayed options:

```
1 - Reverse string,
2 - Shift string to upper case,
3 - Shift string to lower case,
4 - Display SPI messages,
5 - Don't display SPI messages,
6 - Exit
Enter command:
```

Source File

SETREQR

Object File

SETREQRO

Example E-9. TAL File: A Simple SPI Requester (page 1 of 8)

```
--   File name: SETREQR
--   SPI EXAMPLE TAL Basic Requester model.
--
?SYMBOLS, INSPECT
LITERAL
        max^bufsize   = 560,    ! in bytes
        version       = %H4414; ! Set to the value: "D20"

?SOURCE SETCDECS
?SOURCE SETRDECS

INT
         buflen,
         debug^flag,                ! Flag to startup server in INSPECT
         dest^idx,                  ! Destination index for SSMOVETKN
         display^spi^buffer := false,
         file^error,
         file^num,
         open^flags,
         process^id [0:11] := "                          ",
         process^name [0:3]  := "$SPIX   ",            ! Server's process
name.
         read^count,
         server^name  [0:11] := "$NONE   NONE    SETSERVO", ! Server's object
         server^up,
         source^idx,                ! Source index for SSMOVETKN
         spi^command      := 0,
         srvr^file^num,
         srvr^retry^count := 0,
         tkn^count,
         tkn^retcode;

INT(32)
         time^to^wait := 1000D;     ! In centi-seconds = 10 seconds

DEFINE   sav^buffer = b2#;   ! Saved command buffer (same as b2)

?NOLIST, SOURCE $SYSTEM.SYSTEM.EXTDECS0 (AWAITIOX, CANCEL, CLOSE,
? DEBUG, DNUMOUT, FILEINFO, FNAMECOLLAPSE,
? MYTERM, NEWPROCESS, NUMIN, NUMOUT,
? OPEN, READUPDATEX, REPLYX,
? SPI_BUFFER_FORMATFINISH_, SPI_BUFFER_FORMATNEXT_,
? SPI_BUFFER_FORMATSTART_,
? SSGET, SSGETTKN, SSINIT, SSMOVETKN, SSPUT, SSPUTTKN,
? STOP, WRITEREADX, WRITEX)
?LIST

?PAGE "FORWARD DECLARATIONS"
PROC open^server;
FORWARD;

?SOURCE SETCUTIL
?SOURCE SETRUTIL
?PAGE "get^string"
!============================================================================!
! Proc      : get^string                                                     !
! Function  : This procedure will prompt the home term for the function    !
!           and the string data on which to perform the                    !
!           function.                                                      !
!============================================================================!
```

Example E-9. TAL File: A Simple SPI Requester (page 2 of 8)

```
PROC get^string (p^buffer, p^count^read);
STRING .p^buffer;
INT    .p^count^read;    ! This will be modified
BEGIN
  INT   l^work^to^do;
  INT   l^size;
  INT   l^status;

  l^work^to^do := false;
  DO
  BEGIN
    sline ':=' " " -> @sp;
    CALL writex (term, sline, @sp '-' @sline);
    sline ':=' " 1 - Reverse string," -> @sp;
    CALL writex (term, sline, @sp '-' @sline);
    sline ':=' " 2 - Shift string to upper case,  " -> @sp;
    CALL writex (term, sline, @sp '-' @sline);
    sline ':=' " 3 - Shift string to lower case,  " -> @sp;
    CALL writex (term, sline, @sp '-' @sline);
    sline ':=' " 4 - Display SPI messages," -> @sp;
    CALL writex (term, sline, @sp '-' @sline);
    sline ':=' " 5 - Don't display SPI messages,  " -> @sp;
    CALL writex (term, sline, @sp '-' @sline);
    sline ':=' " 6 - Exit" -> @sp;
    CALL writex (term, sline, @sp '-' @sline);

    sline ':=' "Enter command: " -> @sp;
    CALL writereadx (term, sline, @sp '-' @sline, 1, l^size);
    IF < THEN CALL get^file^error (term);
    IF > THEN                                    ! CTL-Y entered
    BEGIN
      CALL stop (process^id);     ! Stop the server.
      CALL stop;
    END;
    sline[l^size] := 0;  -- Ignore the rest of the buffer
    CALL numin (sline, spi^command, 10, l^status);
    IF l^status THEN
       spi^command := 999;    ! Invalid data entered.
    CASE spi^command OF
    BEGIN
      1, 2, 3 ->
        sline ':=' " " -> @sp;
        CALL writex (term, sline, @sp '-' @sline);
        p^buffer ':=' "Enter string: " -> @sp;
        CALL writereadx (term, p^buffer, @sp '-' @p^buffer, 40, l^size);

        IF < THEN CALL get^file^error (term);
        IF > THEN                                    ! CTL-Y entered
        BEGIN
          CALL stop (process^id);     ! Stop the server.
          CALL stop;
        END;
        p^count^read := l^size;
        p^buffer [l^size] := 0;  -- Ignore the rest of the buffer
        l^work^to^do := true;
```

Example E-9. TAL File: A Simple SPI Requester (page 3 of 8)

```
      4 ->
        display^spi^buffer := true;
      5 ->
        display^spi^buffer := false;
      6 ->
        sline ':=' " " -> @sp;
        CALL writex (term, sline, @sp '-' @sline);
        CALL stop (process^id);     ! Stop the server.
        CALL stop;
      OTHERWISE ->
        sline ':=' " " -> @sp;
        CALL writex (term, sline, @sp '-' @sline);
        sline ':=' "Invalid option. Try again." -> @sp;
        CALL writex (term, sline, @sp '-' @sline);
    END;

  END
  UNTIL l^work^to^do = true;
END; -- of PROC get^string

?PAGE "initialization"
!============================================================================!!
Proc      : initialization                                                  !
! Function  : This procedure will open $RECEIVE and the home term.          !
!       It then starts the server.                    !
!============================================================================
PROC initialization;
BEGIN
  STRING
         .l^start^ptr;              ! Scan pointer
  INT     l^termname      [0:11],
          l^filename      [0:11] := ["$RECEIVE", 8*["  "]],
          l^recv^file^num,
          l^count^read,
          l^init^complete;

  ! Open $RECEIVE (sys msgs)
  CALL open (l^filename, l^recv^file^num, %40000, 1);
  IF <> THEN CALL get^file^error (-1);

  ! Read $RECEIVE messages
  l^init^complete := false;
  WHILE l^init^complete = false DO
  BEGIN
    CALL readupdatex (l^recv^file^num, start^buffer, $LEN(start^buffer),
                      l^count^read);
    IF <> THEN
    BEGIN
      CALL fileinfo (l^recv^file^num, file^error);
      IF file^error <> 6 THEN CALL debug;     ! Not a system message
    END;
    CASE start^buffer.msgcode OF
    BEGIN
      -1 -> ! Process STARTUP message
        startup^msg ':=' start^buffer FOR $LEN (start^buffer) BYTES;
```

Example E-9. TAL File: A Simple SPI Requester (page 4 of 8)

```
          ! Put in the current volume/subvolume.
        server^name ':=' start^buffer.default FOR 8 WORDS;
        ! Parse parameters for 'debug'
        debug^flag := false;


        SCAN start^buffer.param WHILE " " -> @l^start^ptr;
        IF NOT $CARRY THEN      ! Found a non-null string
        BEGIN
          IF l^start^ptr = "D" OR l^start^ptr = "d" THEN
          BEGIN
            debug^flag := true;
          END;
        END;
        CALL replyx ( ,,,, 70); ! Must reply and get PARAMs and ASSIGNs
        IF <> THEN CALL get^file^error (l^recv^file^num);

      -31 -> ! Process CLOSE message
        l^init^complete := true;
        CALL replyx (start^buffer, l^count^read); ! Must reply
        IF <> THEN CALL get^file^error (l^recv^file^num);

      OTHERWISE ->
        CALL replyx (start^buffer, l^count^read); ! Must reply
        IF <> THEN CALL get^file^error (l^recv^file^num);

    END; -- of CASE
  END; -- of WHILE l^init^complete = false DO

  ! Open the terminal
  CALL myterm (l^termname);                  ! Get the terminal name
  CALL open (l^termname, term);    ! Open the terminal
  IF <> THEN CALL get^file^error (-1);

  ! Do the NEWPROCESS call of the server
  server^up := false;
  DO
  BEGIN
    CALL restart^server;
  END
  UNTIL server^up = true;

END; -- of PROC initialize

?PAGE "open^server"
!============================================================================!
! Proc      :open^server                                                     !
! Function  :This procedure will open the server.                            !
!============================================================================!

PROC open^server;
BEGIN
  ! Open the server to send the STARTUP msg
    CALL open (process^id, srvr^file^num);
  IF <> THEN
  BEGIN
    CALL fileinfo (-1, file^error);
    sline ':=' "File system error (" -> @sp;
    CALL numout (sp, file^error, 10, 3);
```

Example E-9. TAL File: A Simple SPI Requester (page 5 of 8)

```
 sp[3] ':=' ") on OPEN of the SERVER" -> @sp;
    CALL writex (term, sline, @sp '-' @sline);
    IF <> THEN CALL get^file^error (term);
    RETURN;
  END;

  ! Now write the STARTUP msg to the server
  CALL writex (srvr^file^num, startup^msg, $LEN(startup^msg));
  IF <> THEN
  BEGIN
  CALL fileinfo (srvr^file^num, file^error);
  ! Ignore error 70 (continue operation)
  IF file^error <> 70 THEN
  BEGIN
    sline ':=' "File system error (" -> @sp;
    CALL numout (sp, file^error, 10, 3);
    sp[3] ':=' ") on WRITE to the SERVER" -> @sp;
    CALL writex (term, sline, @sp '-' @sline);
    IF <> THEN CALL get^file^error (term);
    RETURN;
   END;
  END;
  CALL close (srvr^file^num);
  IF <> THEN
  BEGIN
    CALL fileinfo (srvr^file^num, file^error);
    sline ':=' "File system error (" -> @sp;
    CALL numout (sp, file^error, 10, 3);
    sp[3] ':=' ") on CLOSE of the SERVER" -> @sp;
    CALL writex (term, sline, @sp '-' @sline);
    IF <> THEN CALL get^file^error (term);
    RETURN;
  END;
  ! Re-open the server
  open^flags := 0;
  open^flags.<12:15> := 1;  ! NOWAIT IO !
  ! Open the server using SPI
  process^id [4] ':=' "#ZSPI   ";
  CALL open (process^id, srvr^file^num, open^flags);

  IF <> THEN
  BEGIN
    CALL fileinfo (-1, file^error);
    sline ':=' "File system error (" -> @sp;
    CALL numout (sp, file^error, 10, 3);
    sp[3] ':=' ") on REOPEN of the SERVER" -> @sp;
    CALL writex (term, sline, @sp '-' @sline);
    IF <> THEN CALL get^file^error (term);
    RETURN;
  END;
  process^id [4] ':=' "        ";  ! Fix the process ID
  server^up := true;

END; ! -- of PROC open^server;
?PAGE "PROC requester MAIN"
!============================================================================!
!     MAINLINE ROUTINE STARTS HERE                                           !
!                                                                            !
!============================================================================!
```

Example E-9. TAL File: A Simple SPI Requester (page 6 of 8)

```
PROC requester MAIN;
BEGIN
LABEL SEND^IT;

CALL initialization;                          ! Open files

my^ssid ':=' [zspi^val^tandem,
              zspi^ssn^null, version];


spi^command := zspi^cmd^getversion;
!Send a GETVERSION
IF err := ssinit (req^buffer, max^bufsize, my^ssid,
                  zspi^val^cmdhdr, spi^command) THEN
  CALL display^spi^error (err, zspi^val^ssinit, 0d, true);

IF display^spi^buffer THEN
BEGIN
  sline ':=' "SPI buffer sent:" -> @sp;
  CALL writex (term, sline, @sp '-' @sline);
  CALL dump^buf (req^buffer);
END;

CALL writereadx (srvr^file^num, req^buffer, max^bufsize,
                 $OCCURS (req^buffer), read^count);
IF <> THEN CALL get^file^error (srvr^file^num);
file^num := -1;              ! Don't Cancel

CALL awaitiox (file^num, !buffer!, read^count, !tag!, time^to^wait);
IF < THEN
BEGIN
  CALL get^file^error (file^num);
END;

IF display^spi^buffer THEN
BEGIN
  sline ':=' "SPI buffer received:" -> @sp;
  CALL writex (term, sline, @sp '-' @sline);
  CALL dump^buf (req^buffer);
END;

! Reset the buffer
buflen := max^bufsize;
IF err := ssputtkn (req^buffer, zspi^tkn^reset^buffer,
                    buflen) THEN
BEGIN
  CALL display^spi^error (err, zspi^val^ssputtkn,
                          zspi^tkn^reset^buffer, false);
  sline ':=' "Bad SPI buffer returned! Cannot reset the buffer." -> @sp;
  CALL writex (term, sline, @sp '-' @sline);
  CALL stop (process^id);     ! Stop the server.
  CALL stop;
END;

! Get and display the BANNER.
IF err := ssgettkn (req^buffer, zspi^tkn^server^banner,
                    server^banner, 1) THEN
BEGIN
  sline ':=' "No Server Banner found!" -> @sp;
END ELSE
BEGIN
  sline ':=' "Using: " -> @sp;
  sp ':=' server^banner FOR 50 BYTES;
  @sp := @sp '+' 50;
```

Example E-9. TAL File: A Simple SPI Requester (page 7 of 8)

```
    END;


  CALL writex (term, sline, @sp '-' @sline);

  WHILE 1=1 DO                                  ! Do forever
  BEGIN
    CALL get^string (in^string.data, in^string.len);


!Send a SPI message
    IF err := ssinit (req^buffer, max^bufsize, my^ssid,
                      zspi^val^cmdhdr, spi^command) THEN
      CALL display^spi^error (err, zspi^val^ssinit, 0d, true);
    IF err := ssputtkn (req^buffer, zspi^tkn^comment, in^string) THEN
      CALL display^spi^error (err, zspi^val^ssputtkn,
                              zspi^tkn^comment, true);

    ! Save the original buffer in case of continuation.
    sav^buffer ':=' req^buffer FOR max^bufsize/2 WORDS;

SEND^IT:
    IF display^spi^buffer THEN
    BEGIN
      sline ':=' "SPI buffer sent:" -> @sp;
      CALL writex (term, sline, @sp '-' @sline);
      CALL dump^buf (req^buffer);
    END;

    CALL write^read^server;

    IF display^spi^buffer THEN
    BEGIN
      sline ':=' "SPI buffer received:" -> @sp;
      CALL writex (term, sline, @sp '-' @sline);
      CALL dump^buf (req^buffer);
    END;

    ! Reset the buffer
    buflen := max^bufsize;
    IF err := ssputtkn (req^buffer, zspi^tkn^reset^buffer,
                            buflen) THEN
    BEGIN
      CALL display^spi^error (err, zspi^val^ssputtkn,
                              zspi^tkn^reset^buffer, false);
      sline ':=' "Bad SPI buffer returned! Cannot reset the buffer." -> @sp;
      CALL writex (term, sline, @sp '-' @sline);
    END;

    IF NOT err THEN
    BEGIN
      tkn^retcode := 0;
      IF err := ssgettkn (req^buffer, zspi^tkn^retcode, tkn^retcode, 1) THEN
      BEGIN
        CALL display^spi^error (err, zspi^val^ssgettkn,
                                zspi^tkn^retcode, false);
        sline ':=' "Bad SPI buffer returned! Missing RETCODE." -> @sp;
        CALL writex (term, sline, @sp '-' @sline);
      END ELSE

      BEGIN
        err := tkn^retcode; ! To simplify the rest of the error
        checking code.
```

Example E-9. TAL File: A Simple SPI Requester (page 8 of 8)

```
        IF tkn^retcode THEN
        BEGIN
          sline ':=' "Error returned. RETCODE = (" -> @sp;
          IF tkn^retcode < 0 THEN
          BEGIN
            sp ':=' "-" -> @sp;
            @sp := @sp '+' dnumout (sp, $DBL (-tkn^retcode), 10);
          END ELSE
          BEGIN
            @sp := @sp '+' dnumout (sp, $DBL (tkn^retcode), 10);
          END;
          sp ':=' ")" -> @sp;
          CALL writex (term, sline, @sp '-' @sline);
        END;
      END;
    END;


IF NOT err THEN
    BEGIN
      ! retrieve and validate the context, if any
      IF (err := ssgettkn (req^buffer, zspi^tkn^context, the^context, 1) =
                 zspi^err^ok) THEN
      BEGIN
        ! This message has a context. Add it to the original message and
        ! send it again.
        !
        ! This is what is happening. "sav^buffer" has the original msg sent.
        ! I will move the "context" token from the "req^buffer" to the
        ! "sav^buffer". Then, I will move "sav^buffer" to "req^buffer".
        ! Now "req^buffer" is ready to send, but "sav^buffer" has the
        ! "context" token added to the original msg sent. Thus, I will
        ! delete the "context" token from "sav^buffer".
        !
        source^idx := 1;
        dest^idx := 1;
        tkn^count := 1;
        CALL ssmovetkn (zspi^tkn^context, req^buffer, source^idx,
                        sav^buffer, dest^idx, tkn^count);
        req^buffer ':=' sav^buffer FOR max^bufsize/2 WORDS;
        ! Now delete the context token from the saved buffer.
        tkn^code := zspi^tkn^context;
        IF err := ssputtkn (sav^buffer, zspi^tkn^delete,
                            tkn^code) THEN
          CALL display^spi^error (err, zspi^val^ssputtkn,
                                  tkn^code, true);
        GOTO SEND^IT;
      END;

      ! No context. Get the string from the COMMENT token
      IF err := ssgettkn (req^buffer, zspi^tkn^comment, in^string, 1) THEN
      BEGIN
        CALL display^spi^error (err, zspi^val^ssgettkn,
                                zspi^tkn^comment, false);
        sline ':=' "Bad SPI buffer returned! Missing TKN^COMMENT." -> @sp;
        CALL writex (term, sline, @sp '-' @sline);
      END ELSE
      BEGIN
        sline ':=' in^string.data FOR in^string.len BYTES -> @sp;
        CALL writex (term, sline, @sp '-' @sline);
      END;
    END; -- of IF NOT err
  END; -- of WHILE 1=1 DO
END; -- OF PROC requester
```

Example E-10: A Simple SPI Requester in C

Example E-10 on page E-37 is a simple SPI requester that sends commands to the server shown in Example E-11 on page E-45. The requester starts and stops the server automatically. The requester's commands have the server perform simple string manipulations on a text string that you enter. This TAL program gives you the option of displaying the contents of the SPI messages that are exchanged by the requester and the server. The second option, which shifts a string to upper case, uses response continuation to return the results one character per message. Run the program and enter one of the displayed options:

```
1 - Reverse string,
2 - Shift string to upper case,
3 - Shift string to lower case,
4 - Display SPI messages,
5 - Don't display SPI messages,
6 - Exit
Enter command:
```

Source File

SECREQRC

Object File

SECREQRO

Example E-10. C File: A Simple SPI Requester (page 1 of 8)

```
/*   File name: secreqrc
 *   SPI EXAMPLE C Basic Requester model.
 */
#pragma symbols
#pragma inspect
#pragma nomap
#pragma nolmap

#define max_bufsize  560     /* in bytes */
#define version      0x4414u /* Set to the value: "D20" */

#include "secc.h"
#include "secr.h"

short
         buflen,
         debug_flag,            /* Flag to startup server in INSPECT */
         dest_idx,              /* Destination index for SSMOVETKN */
         display_spi_buffer = false,
         file_error,
         file_num,
         open_flags,
         read_count,
         server_up,
         source_idx,           /* Source index for SSMOVETKN */
         spi_command      = 0,
         srvr_file_num,
         srvr_retry_count = 0,
         tkn_count,
         tkn_retcode;

_lowmem short
         process_id [13],
         process_name [5],     /* Server's process name.*/
         server_name [13];     /* Server's object*/
long
         time_to_wait = 1000L; /* In centi-seconds = 10 seconds */

#define  sav_buffer  b2        /* Saved command buffer (same as b2) */

#pragma list

#pragma PAGE "FORWARD DECLARATIONS"
void open_server(void);

#include "seccutlc"
#include "secrutlc"

#pragma PAGE "get_string"
```

Example E-10. C File: A Simple SPI Requester (page 2 of 8)

```
/*
 *=============================================================================
 * Proc      : get_string                                                      =
 * Function  : This procedure will prompt the home term for the function =
 *             and the string data on which to perform the function.     =
 *=============================================================================
 */
void get_string (char* p_buffer, short* p_count_read)
{
  short  l_work_to_do;
  short  l_input;
  short  l_temp;

  l_work_to_do = false;
  do
  {
    printf (" \n");
    printf (" 1 - Reverse string,\n");
    printf (" 2 - Shift string to upper case,  \n");
    printf (" 3 - Shift string to lower case,  \n");
    printf (" 4 - Display SPI messages,\n");
    printf (" 5 - Don't display SPI messages,  \n");
    printf (" 6 - Exit\n");

    printf ("Enter command: \n");
    l_input = getchar();
    if (l_input == EOF)                          /* CTL-Y entered */
    {
       STOP(process_id);     /* Stop the server. */
       STOP();
    }
    /* Now eat all the characters upto and including the ENTER. */
    while ((l_temp = getchar()) != '\n') {}
    spi_command = 0;
    switch (l_input)
    {
      case '3':
         ++spi_command;
      case '2':
         ++spi_command;
      case '1':
         ++spi_command;
        printf (" \n");
        printf ("Enter string: \n");
        gets (p_buffer);
        *p_count_read = strlen(p_buffer);
        if (p_count_read == 0)                    /* no input entered */
        {
           STOP(process_id);     /* Stop the server. */
           STOP();
        }
        l_work_to_do = true;
        break;
```

Example E-10. C File: A Simple SPI Requester (page 3 of 8)

```
      case '4':
        display_spi_buffer = true;
        break;

      case '5':
        display_spi_buffer = false;
        break;

      case '6':
        printf (" \n");
        STOP(process_id);     /* Stop the server. */
        STOP();
        break;

      default:
        printf (" \n");
        printf ("Invalid option. Try again.\n");
        break;

    } /* end of switch */

  }
  while (l_work_to_do == false);
}  /* of get_string() */

#pragma PAGE "initialization"
/*
 *=============================================================================
 * Proc      : initialization                                                  =
 * Function  : This procedure will start the server.                           =
 *=============================================================================
 */
void initialization(void)
{
  short     len;
 /* Get the startup msg */
  if (get_startup_msg(&startup_msg, &len))
  {
    printf ("No STARTUP message received.");
    return;
  }
  /* Put in the current volume/subvolume. */
  memcpy((char *) &server_name[0], &startup_msg.defaults.whole, 16);
  /* Parse parameters for 'debug' */
  debug_flag = false;
  if ((strcmp(startup_msg.param, "D") == 0)
  ||  (strcmp(startup_msg.param, "d") == 0))
  {
    debug_flag = true;
    time_to_wait = 60000L; /* In centi-seconds = 600 seconds */
  }
  /* Do the NEWPROCESS of the server */
  server_up = false;
  do
  {
    restart_server();
  }
  while (server_up == false);

}  /* of PROC initialize */
```

Example E-10. C File: A Simple SPI Requester (page 4 of 8)

```
#pragma PAGE "open_server"
/*
 *====================================================================
 * Proc      : open_server                                           =
 * Function  : This procedure will open the server.                  =
 *====================================================================
 */
 void open_server(void)
{
  short   l_status;
  char    l_process_id_zspi [24];

  open_flags = 0;
  /* Open the server to send the STARTUP msg */
  memset (l_process_id_zspi, ' ', 24);
  memcpy (l_process_id_zspi, (char *) &process_id[0], 6);

  /* Open the server */
  l_status = OPEN ((short *) &l_process_id_zspi[0],
                   &srvr_file_num, open_flags);
  if (l_status != CCE)
  {
    FILEINFO (-1, &file_error);
    printf ("File system error (%d) on OPEN of the SERVER\n", file_error);
    return;
  }
  /* Now write the STARTUP msg to the server */
  /* ********************************************************/
  /* But first, I will blank out the INFILE and OUTFILE so a  */
  /* server written in C (which uses the CRE) will not use    */
  /* the standard files, but accept open requests.            */
  /* ********************************************************/
  memset(&startup_msg.infile, ' ', sizeof(startup_msg.infile));
  memset(&startup_msg.outfile, ' ', sizeof(startup_msg.outfile));

  l_status = WRITEX (srvr_file_num, (char *) &startup_msg,
                     sizeof(startup_msg));
  if (l_status != CCE)
  {
    FILEINFO (srvr_file_num, &file_error);
    /* Ignore error 70 (continue operation) */
    if (file_error != 70)
    {
      printf ("File system error (%d) on WRITE to the SERVER\n", file_error);
      return;
    }
  }
```

Example E-10. C File: A Simple SPI Requester (page 5 of 8)

```
  l_status = CLOSE (srvr_file_num);
  if (l_status != CCE)
  {
    FILEINFO (srvr_file_num, &file_error);
    printf ("File system error (%d) on CLOSE of the SERVER\n", file_error);
    return;
  }
  /* Re-open the server */
  open_flags = 0x0001;     /* NOWAIT IO */
  /* Open the server using SPI */
  memcpy (&l_process_id_zspi[8], "#ZSPI   ", 8);
  l_status = OPEN ((short *) &l_process_id_zspi[0],
                   &srvr_file_num, open_flags);
  if (l_status != CCE)
  {
    FILEINFO (-1, &file_error);
    printf ("File system error (%d) on REOPEN to the SERVER\n", file_error);
    return;
   }
  /* Fix the process ID */
  memcpy (&l_process_id_zspi[8], "        ", 8);
  server_up = true;

}  /* of PROC open_server  */

#pragma PAGE "PROC requester MAIN"
/*
 *============================================================================
 *     MAINLINE ROUTINE STARTS HERE.
 *
 *============================================================================
 */
main(/* int argc, char *argv[] */)
{
  short     l_status;

  debug_flag = false;
  /* Server's process name. */
  memcpy ((char *) &process_name[0], "$SPIX   ", 6);
  /* Server's object */
  memcpy ((char *) &server_name[8], "SECSERVO", 8);

  initialization();                        /* Open files */

  spi_command = ZSPI_CMD_GETVERSION;
  /* Send a GETVERSION */
  if (err = SSINIT (req_buffer, max_bufsize, (short *) &my_ssid,
                    ZSPI_VAL_CMDHDR, spi_command))
    display_spi_error (err, ZSPI_VAL_SSINIT, 0L, true);

  if (display_spi_buffer)
  {
    printf ("SPI buffer sent:\n");
    dump_buf (req_buffer);
  }
```

Example E-10. C File: A Simple SPI Requester (page 6 of 8)

```
  l_status = WRITEREADX (srvr_file_num, (char *) &req_buffer[0],
                         max_bufsize, sizeof (req_buffer), &read_count);
  if (l_status != CCE)  get_file_error (srvr_file_num);
  file_num = -1;              /* Don't Cancel */

  l_status = AWAITIOX (&file_num, /*buffer*/, &read_count,
                       /*tag*/, time_to_wait);
  if (l_status == CCL)
  {
     get_file_error (file_num);
  }

  if (display_spi_buffer)
  {
    printf ("SPI buffer received:\n");
    dump_buf (req_buffer);
  }

  /* Reset the buffer */
  buflen = max_bufsize;
  if (err = SSPUTTKN (req_buffer, ZSPI_TKN_RESET_BUFFER,
                      (char *) &buflen))
  {
    display_spi_error (err, ZSPI_VAL_SSPUTTKN,
                            ZSPI_TKN_RESET_BUFFER, false);
    printf ("Bad SPI buffer returned! Cannot reset the buffer.\n");
    STOP (process_id);     /* Stop the server. */
    STOP();
  }

  /* Get and display the BANNER. */
  if (err = SSGETTKN (req_buffer, ZSPI_TKN_SERVER_BANNER,
                      (char *) &server_banner, 1))
  {
    printf ("No Server Banner found! \n");
  } else
  {
    printf ("Using: %.50s\n", (char*) &server_banner);
  }

  do                                  /* Do forever */
  {
    get_string (in_string.data, &in_string.len);

    /* Send a SPI message */
    if (err = SSINIT (req_buffer, max_bufsize, (short *) &my_ssid,
                      ZSPI_VAL_CMDHDR, spi_command))
      display_spi_error (err, ZSPI_VAL_SSINIT, 0L, true);
    if (err = SSPUTTKN (req_buffer, ZSPI_TKN_COMMENT,
                        (char *) &in_string))
      display_spi_error (err, ZSPI_VAL_SSPUTTKN,
                              ZSPI_TKN_COMMENT, true);

    /* Save the original buffer in case of continuation. */
    memcpy(sav_buffer, req_buffer, max_bufsize);
SEND_IT:
    if (display_spi_buffer)
    {
      printf ("SPI buffer sent:\n");
      dump_buf (req_buffer);
    }
```

Example E-10. C File: A Simple SPI Requester (page 7 of 8)

```
write_read_server ();

if (display_spi_buffer)
{
  printf ("SPI buffer received:\n");
  dump_buf (req_buffer);
}

/* Reset the buffer */
buflen = max_bufsize;
if (err = SSPUTTKN (req_buffer, ZSPI_TKN_RESET_BUFFER,
                    (char *) &buflen))
{
  display_spi_error (err, ZSPI_VAL_SSPUTTKN,
                          ZSPI_TKN_RESET_BUFFER, false);
  printf ("Bad SPI buffer returned! Cannot reset the buffer.\n");
}

if (! err)
{
  tkn_retcode = 0;
  if (err = SSGETTKN (req_buffer, ZSPI_TKN_RETCODE,
                      (char *) &tkn_retcode, 1))
  {
    display_spi_error (err, ZSPI_VAL_SSGETTKN,
                            ZSPI_TKN_RETCODE, false);
    printf ("Bad SPI buffer returned! Missing RETCODE.\n");
  } else
  {
    err = tkn_retcode; /* To simplify the error checking code. */
    if (tkn_retcode)
    {
      printf ("Error returned. RETCODE = (%d)\n", tkn_retcode);
    }
  }
}

if (! err)
{
  /* retrieve and validate the context, if any */
  if ((err = SSGETTKN (req_buffer, ZSPI_TKN_CONTEXT,
              (char *) &the_context, 1) == ZSPI_ERR_OK))
  {        /* This message has a context. Add it to the original message
    * and send it again.
    *
    * This is what is happening. "sav_buffer" has the original msg sent.
    * I will move the "context" token from the "req_buffer" to the
    * "sav_buffer". Then, I will move "sav_buffer" to "req_buffer".
    * Now "req_buffer" is ready to send, but "sav_buffer" has the
    * "context" token added to the original msg sent. Thus, I will
    * delete the "context" token from "sav_buffer".
    */
    source_idx = 1;
    dest_idx = 1;
    tkn_count = 1;
    SSMOVETKN (ZSPI_TKN_CONTEXT, req_buffer, source_idx,
                      sav_buffer, dest_idx, &tkn_count);
```

Example E-10. C File: A Simple SPI Requester (page 8 of 8)

```
        memcpy (req_buffer, sav_buffer, max_bufsize);
        /* Now delete the context token from the saved buffer. */
        tkn_code = ZSPI_TKN_CONTEXT;
        if (err = SSPUTTKN (sav_buffer, ZSPI_TKN_DELETE,
                            (char *) &tkn_code))
          display_spi_error (err, ZSPI_VAL_SSPUTTKN,
                                  tkn_code, true);
        goto SEND_IT;
      }
      /* No context. Get the string from the COMMENT token */
      if (err = SSGETTKN (req_buffer, ZSPI_TKN_COMMENT,
                          (char *) &in_string, 1))
      {
        display_spi_error (err, ZSPI_VAL_SSGETTKN,
                                ZSPI_TKN_COMMENT, false);
        printf ("Bad SPI buffer returned! Missing TKN_COMMENT.\n");
      } else
      {
        in_string.data [in_string.len] = 0; /* terminate the string */
        printf ("%s", in_string.data);
      }
    } /* of if (! err)  */
  }
  while (true);
}  /* OF PROC requester */
```

Example E-11: A Simple SPI Server in TAL

Example E-11 on page E-45 is a simple SPI server that performs simple string manipulations on strings provided by the requester shown in Example E-9 on page E-28. The server is started automatically when you run the requester, SETREQRO.

Source File

SETSERV

Object File

SETSERVO

Example E-11. TAL File: A Simple SPI Server (page 1 of 10)

```
--   File name: SETSERV
--   SPI EXAMPLE TAL Basic Server model.
--
?SYMBOLS, INSPECT
LITERAL
        max^bufsize   = 1010,   ! in bytes
        version       = %H4414; ! Set to the value: "D20"

?SOURCE SETCDECS
?SOURCE SETRDECS

INT
         context^count,             ! Number of CONTEXT tokens
         dest^idx,                  ! Destination index for SSMOVETKN
         file^error,
         max^resp,                  ! From the SPI message
         object^type,               ! Object type
         rcv^file^num,              ! $RECEIVE's file number
         resp^type,                 ! From the SPI message
         source^idx,                ! Source index for SSMOVETKN
         spi^buffer^size,           ! Size of last SPI buffer read.
         spi^command,               ! From the SPI message
         startup^recvd,             ! Indicates if startup msg received
         tkn^count,                 ! The number of tokens
         tkn^retcode;

STRUCT  .out^string (string^template);   ! output string

DEFINE   res^buffer = b2#;   ! Response buffer (same as b2)

?NOLIST, SOURCE ZCOMTAL
?LIST
?NOLIST, SOURCE $SYSTEM.SYSTEM.EXTDECS0 (CLOSE, DEBUG, DNUMOUT, FILEINFO,
? OPEN, READUPDATEX, REPLYX, SHIFTSTRING,
? SPI_BUFFER_FORMATFINISH_, SPI_BUFFER_FORMATNEXT_, SPI_BUFFER_FORMATSTART_,
? SSGET, SSGETTKN, SSINIT, SSMOVETKN, SSPUT, SSPUTTKN,
? STOP, WRITEX)
?LIST

?PAGE "FORWARD DECLARATIONS"
PROC build^hdr^response;
FORWARD;

PROC error^response (p^err^num);
INT     p^err^num;    ! error number
FORWARD;

PROC initialization;
FORWARD;

PROC process^spi^buffer;
FORWARD;

PROC process^requests;
FORWARD;

INT PROC validate^tokens;
FORWARD;

INT PROC verify^msg (p^count);
INT     p^count;
FORWARD;
```

Example E-11. TAL File: A Simple SPI Server (page 2 of 10)

```
?SOURCE SETCUTIL
?PAGE "PROC build^hdr^response"
PROC build^hdr^response;
!=========================================================================!
! Proc      : build^hdr^response                                          !
! Function  : This procedure will build the header for responses.         !
!=========================================================================!

BEGIN

  object^type := zspi^val^null^object^type;

  !Initialize response buffer
  IF err := ssinit (res^buffer, max^bufsize, my^ssid,
                    zspi^val^cmdhdr, spi^command, object^type) THEN
    CALL display^spi^error (err, zspi^val^ssinit, 0d, true);

  ! Put in server version token
  IF err := ssputtkn (req^buffer, zspi^tkn^server^version,
                      my^version) THEN
    CALL display^spi^error (err, zspi^val^ssputtkn,
                            zspi^tkn^server^version, true);

END; ! End of build^hdr^response procedure
?PAGE "PROC error^response"
PROC error^response (p^err^num);
!=========================================================================!
! Proc      : error^response                                              !
! Function  : This procedure will format responses for unsuccessful       !
!             processing of spi^command.                                  !
!=========================================================================!

INT p^err^num;    ! error number
! max^resp token from SPI buffer must be previously set.
! tkn^code of the token which caused error must be previously set.
BEGIN

  STRUCT .EXT l^err^def (zspi^ddl^error^def);
  STRUCT .EXT l^parm^err^def (zspi^ddl^parm^err^def);

  CALL build^hdr^response;

  source^idx := 1;
  dest^idx := 1;
  tkn^count := 1;

  ! Start Data list only IF zspi^tkn^maxresp is not = 0
  IF max^resp THEN
     IF err := ssputtkn (res^buffer, zspi^tkn^datalist) THEN
     BEGIN
       CALL display^spi^error (err, zspi^val^ssputtkn,
                               zspi^tkn^datalist, false);
       RETURN;
     END;

  !Put return code token in response buffer
  IF err := ssputtkn (res^buffer, zspi^tkn^retcode, p^err^num) THEN
  BEGIN
    CALL display^spi^error (err, zspi^val^ssputtkn,
                            zspi^tkn^retcode, false);
    RETURN;
  END;
```

Example E-11. TAL File: A Simple SPI Server (page 3 of 10)

```
  ! Put error list token
  IF err := ssputtkn (res^buffer, zspi^tkn^errlist) THEN
  BEGIN
    CALL display^spi^error (err, zspi^val^ssputtkn,
                            zspi^tkn^errlist, false);
    RETURN;
  END;

  l^err^def.z^ssid ':=' my^ssid FOR 12 BYTES;
  l^err^def.z^error := p^err^num; !Error number returned from validate^tokens

  IF err := ssputtkn (res^buffer, zspi^tkn^error, l^err^def) THEN
  BEGIN
    CALL display^spi^error (err, zspi^val^ssputtkn,
                            zspi^tkn^error, false);
    RETURN;
  END;

  !Put endlist token for end of response (End list of Error list)
  IF err := ssputtkn (res^buffer, zspi^tkn^endlist) THEN
  BEGIN
    CALL display^spi^error (err, zspi^val^ssputtkn,
                            zspi^tkn^endlist, false);
    RETURN;
  END;

  !Put endlist token for end of response (End list of Data list)
  IF max^resp THEN
    IF err := ssputtkn (res^buffer, zspi^tkn^endlist) THEN
    BEGIN
      CALL display^spi^error (err, zspi^val^ssputtkn,
                              zspi^tkn^endlist, false);
      RETURN;
    END;

END; ! End of error^response procedure

?PAGE "PROC initialization"
!============================================================================!
! Proc      : initialization                                                 !
! Function  : This procedure will open $RECEIVE and the home term. It also   !
!             sets the server^banner and the SSID.                           !
!============================================================================!

PROC initialization;
BEGIN
  INT    l^filename [0:11] := ["$RECEIVE",8 * ["  "]];
  INT    l^err;

  server^banner.z^b ':=' " " & server^banner.z^b [0] FOR 49 BYTES;
  server^banner.z^b ':=' "TAL SERVER Version 1.01 (14APR95)";
  !Assign values to ssid definitions

  my^ssid ':=' [zspi^val^tandem,
                zspi^ssn^null, version];

  bufsize := max^bufsize;
  rcv^file^num := -1;
  !open $RECEIVE
  startup^recvd :=0;
  WHILE rcv^file^num = -1 DO       !retry opening $RECEIVE until ok
  BEGIN
    CALL open (l^filename, rcv^file^num, %40000, 5); ! Recv depth = 5
  END;
```

Example E-11. TAL File: A Simple SPI Server (page 4 of 10)

```
  IF <> THEN                !error handling
  BEGIN
    IF > THEN
    BEGIN
      CALL fileinfo (rcv^file^num, l^err);
      IF l^err > 0 THEN
      BEGIN
        CALL debug;
        continue := false;
      END;
    END ELSE
    BEGIN
      CALL debug;
      continue := false;
    END;
  END;

END;   !initialization

?PAGE "PROC process^spi^buffer"
PROC process^spi^buffer;
!============================================================================!
! Proc      : process^spi^buffer                                             !
! Function  : This procedure will format responses for successful            !
!             processing of spi^command.                                     !
!============================================================================!
BEGIN
  INT
      l^err,
      l^idx,
      l^len,
      l^start^idx;

  ! determine if this is a valid SPI message
  IF (l^err := verify^msg (spi^buffer^size)) THEN
  BEGIN
    CALL error^response (l^err);
    RETURN;
  END;

  ! get the spi^command from the request buffer!
  IF (err := ssgettkn (req^buffer, zspi^tkn^command, spi^command)) THEN
  BEGIN
    CALL display^spi^error (err, zspi^val^ssgettkn,
                            zspi^tkn^command, true);
  END;

  IF (l^err := validate^tokens) THEN          ! This is a procedure call
  BEGIN
    CALL error^response (l^err);
    RETURN;
  END;

  ! Security checks on the command/user could be done here.

  CALL build^hdr^response;

  source^idx := 1;
  dest^idx := 1;
  tkn^count := 1;
```

Example E-11. TAL File: A Simple SPI Server (page 5 of 10)

```
! Start Data list only IF zspi^tkn^maxresp is not = 0
IF max^resp THEN
BEGIN
  IF err := ssputtkn (res^buffer, zspi^tkn^datalist) THEN
    CALL display^spi^error (err, zspi^val^ssputtkn,
                            zspi^tkn^datalist, true);
END;
l^start^idx := 0;
! Check for CONTEXT token.
IF context^count THEN
BEGIN
  ! Here the CONTEXT token was sent. Copy the passed context.
  out^string.len := the^context.con^string.len;
  out^string.data ':=' the^context.con^string.data FOR
                       out^string.len BYTES;
  l^start^idx := the^context.index;
END;

! Now perform the spi^command
tkn^retcode := zspi^err^ok;
CASE spi^command OF
BEGIN
  zspi^cmd^getversion  ->
    ! Put SERVER BANNER token in response buffer
    IF (err := ssputtkn (res^buffer, zspi^tkn^server^banner,
                         server^banner)) THEN
      CALL display^spi^error (err, zspi^val^ssputtkn,
                              zspi^tkn^server^banner, true);

  1                    ->    ! STRING to reverse.
    l^len := in^string.len;
    FOR l^idx := l^start^idx TO l^len - 1 DO
    BEGIN
      out^string.data [l^len - l^idx - 1] := in^string.data [l^idx];
    END;
    out^string.len := in^string.len;
    IF err := ssputtkn (res^buffer, zspi^tkn^comment,
                        out^string) THEN
      CALL display^spi^error (err, zspi^val^ssputtkn,
                              zspi^tkn^comment, true);

  2                    ->    ! Shift string to uppercase
    ! This could be done in one operation, but to show the use of
    ! the CONTEXT token, it will be done one character at a time.
    l^len := in^string.len;
    out^string.data [l^start^idx] := in^string.data [l^start^idx];
    CALL shiftstring (out^string.data [l^start^idx],
                      1 ! only one byte at a time!,
                      0 !upshift!);   ! Change to upper-case
   l^start^idx := l^start^idx + 1;
    out^string.len := l^start^idx;
```

Example E-11. TAL File: A Simple SPI Server (page 6 of 10)

```
      IF out^string.len < in^string.len THEN
      BEGIN
        ! update the context and add it to the returned buffer.
        the^context.command := spi^command;
        the^context.index := l^start^idx;
        the^context.con^string.len := out^string.len;

        the^context.con^string.data ':=' out^string.data FOR
                                          out^string.len BYTES;
        the^context.len := $offset (context^template.con^string) +
                           out^string.len + 2;
        IF err := ssputtkn (res^buffer, zspi^tkn^context,
                            the^context) THEN
          CALL display^spi^error (err, zspi^val^ssputtkn,
                                  zspi^tkn^context, true);
      END;
      ! Here I have the option to add the work that has been done to
      ! returned buffer. This depends on the function being performed.
      ! I will add it here just for illustration purposes.
      !
      IF err := ssputtkn (res^buffer, zspi^tkn^comment,
                          out^string) THEN
        CALL display^spi^error (err, zspi^val^ssputtkn,
                                zspi^tkn^comment, true);

    3                     ->    ! Shift string to lowercase
      out^string.len := in^string.len;
      out^string.data ':=' in^string.data FOR in^string.len BYTES;

      CALL shiftstring (out^string.data,
                        out^string.len,
                        1 !downshift!);   ! Change to lower-case
      IF err := ssputtkn (res^buffer, zspi^tkn^comment,
                          out^string) THEN
        CALL display^spi^error (err, zspi^val^ssputtkn,
                                zspi^tkn^comment, true);

    OTHERWISE             ->    ! (invalid command)
      tkn^retcode := zspi^err^notimp;

  END; ! End of CASE (spi^command) !

  !Put return code token in response buffer
  IF err := ssputtkn (res^buffer, zspi^tkn^retcode, tkn^retcode) THEN
    CALL display^spi^error (err, zspi^val^ssputtkn,
                            zspi^tkn^retcode, true);

  ! Put endlist token for end of response
  IF max^resp AND (resp^type <> zspi^val^err^and^warn) THEN
    IF err := ssputtkn (res^buffer, zspi^tkn^endlist) THEN
      CALL display^spi^error (err, zspi^val^ssputtkn,
                              zspi^tkn^endlist, true);

END; ! End of process^spi^buffer procedure

?PAGE "PROC process^requests"
!=========================================================================!
! Proc      : process^requests                                            !
! Function  : This procedure will process the requests received           !
!             on $RECEIVE                                                 !
!=========================================================================!
```

Example E-11. TAL File: A Simple SPI Server (page 7 of 10)

```
PROC process^requests;
BEGIN

  ! Read in the spi^command received

  CALL readupdatex (rcv^file^num, req^buffer, max^bufsize,
                    spi^buffer^size);
  CALL fileinfo (rcv^file^num, last^file^err);

  CASE (last^file^err) OF          ! data was found in the buffer
  BEGIN
    6 -> ! system message
      ! first word of req^buffer is message type.
      CASE (req^buffer[0]) OF
      BEGIN
        -30 -> ! OPEN message
          !Don't reject the OPEN for sending the STARTUP msg.
          IF (req^buffer [9] <> "#ZSPI ") AND
             (startup^recvd := -1) THEN
          BEGIN
            ! Reject the open with file error 11.
            file^error := 11;
          END;
        OTHERWISE ->
          res^buffer ':=' req^buffer FOR max^bufsize/2 WORDS;
      END;
   0 -> ! non-system message
      ! first word of msg (Z^MSGCODE) is a -28 for a SPI msg.
      CASE (req^buffer[0]) OF
      BEGIN
        -1 -> ! Process Startup message
          res^buffer ':=' req^buffer FOR max^bufsize/2 WORDS;
         startup^recvd := -1;
        -28 ->
          CALL process^spi^buffer;
      OTHERWISE ->
          res^buffer ':=' req^buffer FOR max^bufsize/2 WORDS;
      END;

    OTHERWISE ->   !unexpected message not a SPI or system message
      res^buffer ':=' req^buffer FOR max^bufsize/2 WORDS;
      continue := false;
  END;  -- of CASE (last^file^err)

END; -- of PROC process^requests

?PAGE "INT PROC validate^tokens"
!=========================================================================!
! Proc      : validate^tokens                                             !
! Function  : This procedure will determine if the "req^buffer" contains  !
!             a valid SPI command. All required tokens must be present.   !
!             Duplicate tokens, invalid tokens, invalid token values are  !
!             rejected.                                                   !
! Returns   : An error code indicating the error found in the command     !
!             buffer or zspi^err^ok (0) which indicates no error was found.!
!=========================================================================!
```

Example E-11. TAL File: A Simple SPI Server (page 8 of 10)

```
 INT PROC validate^tokens;
BEGIN
  STRUCT .l^ssid (zspi^ddl^ssid^def);

  ! set default token values
  ! IF any of these have to appear for a command set them to
  ! null values here & check them in the command^ code
  ! reset token counts
  context^count := 0;
  tkn^count := 1;

  ! get the header tokens-  validate that they were retrieved ok
  IF err := ssgettkn (req^buffer, zspi^tkn^ssid, l^ssid) THEN
    CALL display^spi^error (err, zspi^val^ssgettkn,
                            zspi^tkn^ssid, true);

  ! check if the SSID matches mine, but don't check the version.
  IF l^ssid <> my^ssid FOR 5 WORDS THEN
  BEGIN
    tkn^code := zspi^tkn^ssid;
    RETURN (zcom^err^tkn^val^inv);
  END;

  IF err := ssgettkn (req^buffer, zspi^tkn^maxresp, max^resp) THEN
    CALL display^spi^error (err, zspi^val^ssgettkn,
                            zspi^tkn^maxresp, true);

  IF max^resp < -1 THEN
  BEGIN
    tkn^code := zspi^tkn^maxresp;
    RETURN (zcom^err^tkn^val^inv);
  END;

  ! reposition to head of SPI buffer
  tkn^value := 0;

  IF err := ssputtkn (req^buffer, zspi^tkn^initial^position,
                      tkn^value) THEN
    CALL display^spi^error (err, zspi^val^ssputtkn,
                            zspi^tkn^initial^position, true);
 ! walk through the buffer pulling out tokens
  WHILE (err := ssgettkn (req^buffer, zspi^tkn^nextcode, tkn^code,,
                          tkn^count)) = zspi^err^ok
  DO
  BEGIN
    CASE $INT (tkn^code) OF
    BEGIN
      zspi^tnm^comment ->
        ! Check the entire token code, just to be sure.
        IF tkn^code <> zspi^tkn^comment THEN
          RETURN (zcom^err^tkn^code^inv);
        IF err := ssgettkn (req^buffer, zspi^tkn^comment, in^string) THEN
          CALL display^spi^error (err, zspi^val^ssgettkn,
                                  zspi^tkn^comment, true);

      zspi^tnm^context ->
        ! Check the entire token code, just to be sure.
        IF tkn^code <> zspi^tkn^context THEN
          RETURN (zcom^err^tkn^code^inv);
```

Example E-11. TAL File: A Simple SPI Server (page 9 of 10)

```
        tkn^code := zspi^tkn^context;
        context^count := context^count + tkn^count;
        IF context^count <> 1 THEN
        BEGIN
          RETURN (zcom^err^tkn^dup);
        END ELSE
        BEGIN
          ! retrieve and validate the context
          IF err := ssgettkn (req^buffer, zspi^tkn^context, the^context) THEN
            CALL display^spi^error (err, zspi^val^ssgettkn,
                                    zspi^tkn^context, true);
          IF (the^context.len > $len(context^template)) OR
             (the^context.len <> ($offset(context^template.con^string) +
                                  the^context.con^string.len + 2)) OR
             (the^context.command <> spi^command) OR
             (the^context.index > the^context.con^string.len) THEN
          BEGIN
            RETURN (zcom^err^tkn^cntxt^code^inv);
          END;
        END;

      OTHERWISE ->
        RETURN (zcom^err^tkn^code^inv);

    END; -- of CASE

  END; -- of WHILE LOOP

  RETURN (zspi^err^ok);       ! no errors found, RETURN ok
END; ! validate^tokens
PAGE "INT PROC verify^msg"
!=====================================================================!
! Proc      : verify^msg                                              !
! Function  : This procedure will determine if a valid SPI buffer was  !
!             received.                                               !
! Returns   : An error code indicating the error found,               !
!             or zspi^err^ok (0) which indicates no error was found.  !
!=====================================================================!

INT PROC verify^msg (p^count);
INT p^count;  !size of data read must be at least 6!
BEGIN
  IF p^count < 6 THEN
    RETURN (zspi^err^invbuf);
  ! Reset the buffer
  tkn^count := 1;
  IF err := ssputtkn (req^buffer, zspi^tkn^reset^buffer,
                      bufsize) THEN
  BEGIN
    CALL display^spi^error (err, zspi^val^ssputtkn,
                            zspi^tkn^reset^buffer, false);
    RETURN (zspi^err^invbuf);
  END;
```

Example E-11. TAL File: A Simple SPI Server (page 10 of 10)

```
  ! header type must be a complete header
  ! SSGET of zspi^tkn^hdrtype returns a token value of zspi^val^cmdhdr
  IF err := ssgettkn (req^buffer, zspi^tkn^hdrtype, tkn^value,
                            ,tkn^count) THEN
  BEGIN
    CALL display^spi^error (err, zspi^val^ssgettkn,
                            zspi^tkn^hdrtype, false);
    RETURN (zspi^err^invbuf);
  END;

  IF tkn^value <> zspi^val^cmdhdr THEN
    RETURN (zspi^err^invbuf);

  tkn^code := zspi^tkn^hdrtype;
  RETURN (err);
END;  ! procedure verify^msg
?PAGE "PROC server MAIN"
!============================================================================!
!     MAINLINE ROUTINE STARTS HERE.                                          !
!                                                                            !
!============================================================================!

PROC server MAIN;

BEGIN
  INT  l^reply^length;

  CALL initialization;

  continue := true;
  WHILE continue = true DO
  BEGIN
    ! Clear important fields
    res^buffer ':=' 0 & res^buffer FOR $OCCURS (res^buffer) - 1 WORDS;
    req^buffer ':=' 0 & req^buffer FOR $OCCURS (req^buffer) - 1 WORDS;
    ! Read a request message from $RECEIVE
    CALL process^requests;

    ! SEND reply buffer to $RECEIVE
    l^reply^length := max^bufsize;
    IF file^error THEN
    BEGIN
       l^reply^length := 0;
    END;

    CALL replyx (res^buffer, l^reply^length, !count^sent!,
                 !tag!, file^error);
    IF < THEN
    BEGIN
      CALL get^file^error (rcv^file^num);
      continue := false;
    END;

    ! Clear important fields
    file^error := 0;

  END;  ! End of WHILE continue DO

  CALL stop;

END; -- of PROC server
```

Example E-12: A Simple SPI Server in C

Example E-12 is a simple SPI server that performs simple string manipulations on strings provided by the requester shown in Example E-9 on page E-28. The server is started automatically when you run the requester, SECREQRO.

Source File

SECSERVC

Object File

SECSERVO

Example E-12. C File: A Simple SPI Server (page 1 of 12)

```
/*   File name: secservc
 *   SPI EXAMPLE C Basic Server model.
 */
#pragma symbols
#pragma inspect
#pragma nomap
#pragma nolmap
#pragma nostdfiles

#define max_bufsize  1010    /* in bytes */
#define version      0x4414u /* Set to the value: "D20" */

#include "secc.h"
#include "secr.h"

short
         context_count,             /* Number of CONTEXT tokens          */
         dest_idx,                  /* Destination index for SSMOVETKN   */
         file_error,
         max_resp,                  /* From the SPI message              */
         object_type,               /* Object type                       */
         rcv_file_num,              /* $RECEIVE's file number            */
         resp_type,                 /* From the SPI message              */
         source_idx,                /* Source index for SSMOVETKN        */
         spi_buffer_size,           /* Size of last SPI buffer read.     */
         spi_command,               /* From the SPI message              */
         tkn_count,                 /* The number of tokens              */
         tkn_retcode;

_lowmem string_template  out_string;   /* output string */

#define  res_buffer  b2     /* Response buffer (same as b2) */

#include "zcomc (constants)" nolist

#pragma PAGE "FORWARD DECLARATIONS"
void build_hdr_response(void);

void error_response(short p_err_num);

void initialization(void);
```

Example E-12. C File: A Simple SPI Server (page 2 of 12)

```
void process_spi_buffer(void);

void process_requests(void);

short validate_tokens(void);

short verify_msg (short p_count);

#include "seccutlc"
#pragma PAGE "PROC build_hdr_response"
void build_hdr_response(void)
/*
 *=====================================================================
 * Proc      : build_hdr_response                                      =
 * Function  : This procedure will build the header for responses.     =
 *=====================================================================
 */
{

  object_type = ZSPI_VAL_NULL_OBJECT_TYPE;

  /* Initialize response buffer */
  if (err = SSINIT (res_buffer, max_bufsize, (short *) &my_ssid,
                    ZSPI_VAL_CMDHDR, spi_command, object_type))
    display_spi_error (err, ZSPI_VAL_SSINIT, 0L, true);

  /* Put in server version token */
  if (err = SSPUTTKN (req_buffer, ZSPI_TKN_SERVER_VERSION,
                      (char *) &my_version))
    display_spi_error (err, ZSPI_VAL_SSPUTTKN,
                            ZSPI_TKN_SERVER_VERSION, true);

}  /* End of build_hdr_response procedure */

#pragma PAGE "PROC error_response"
void error_response (short p_err_num)
/*
 *=====================================================================
 * Proc      : error_response                                          =
 * Function  : This procedure will format responses for unsuccessful   =
 *             processing of spi_command.                              =
 *=====================================================================
  */
/* max_resp token from SPI buffer must be previously set. */
/* tkn_code of the token which caused error must be previously set. */
{

  zspi_ddl_error_def     l_err_def;

  build_hdr_response();

  source_idx = 1;
  dest_idx = 1;
  tkn_count = 1;
  }
```

Example E-12. C File: A Simple SPI Server (page 3 of 12)

```
  /* Start Data list only if (ZSPI_TKN_MAXRESP is not = 0 */
  if (max_resp)
    if (err = SSPUTTKN (res_buffer, ZSPI_TKN_DATALIST))
    {
      display_spi_error (err, ZSPI_VAL_SSPUTTKN,
                              ZSPI_TKN_DATALIST, false);
      return;
    }

  /* Put return code token in response buffer */
  if (err = SSPUTTKN (res_buffer, ZSPI_TKN_RETCODE, (char *) &p_err_num))
  {
    display_spi_error (err, ZSPI_VAL_SSPUTTKN,
                            ZSPI_TKN_RETCODE, false);
    return;
  }

  /* Put error list token */
  if (err = SSPUTTKN (res_buffer, ZSPI_TKN_ERRLIST))
  {
    display_spi_error (err, ZSPI_VAL_SSPUTTKN,
                            ZSPI_TKN_ERRLIST, false);
    return;
  }

  memcpy (&l_err_def.z_ssid, &my_ssid, 12);
  l_err_def.z_error = p_err_num; /* Error number from validate_tokens */

  if (err = SSPUTTKN (res_buffer, ZSPI_TKN_ERROR, (char *) &l_err_def))
  {
    display_spi_error (err, ZSPI_VAL_SSPUTTKN,
                            ZSPI_TKN_ERROR, false);
    return;

  /* Put endlist token for end of response (End list of Error list) */
  if (err = SSPUTTKN (res_buffer, ZSPI_TKN_ENDLIST))
  {
    display_spi_error (err, ZSPI_VAL_SSPUTTKN,
                            ZSPI_TKN_ENDLIST, false);
    return;
  }

  /* Put endlist token for end of response (End list of Data list) */
  if (max_resp)
    if (err = SSPUTTKN (res_buffer, ZSPI_TKN_ENDLIST))
    {
      display_spi_error (err, ZSPI_VAL_SSPUTTKN,
                              ZSPI_TKN_ENDLIST, false);
      return;
    }

}  /* End of error_response procedure */

#pragma PAGE "PROC initialization"
```

Example E-12. C File: A Simple SPI Server (page 4 of 12)

```
 /*
 *===========================================================================
 * Proc      : initialization                                               =
 * Function  : This procedure will open $RECEIVE. It also sets the          =
 *             server_banner.                                               =
 *===========================================================================
 */
void initialization(void)
{
  char    l_filename [24];
  short   l_err;
  short   l_status;

  memset (&server_banner.u_z_c.z_b, ' ', sizeof (server_banner));
  memcpy (server_banner.u_z_c.z_b, "C SERVER Version 1.01 (27MAR95)", 31);
  /* Assign values to ssid definitions */
  memcpy (l_filename, "$RECEIVE                ", 24);
  bufsize = max_bufsize;
  rcv_file_num = -1;
  /* open $RECEIVE */
  do                                /* retry opening $RECEIVE until ok */
  {
    l_status = OPEN ((short *) &l_filename[0], &rcv_file_num,
                     040000, 5); /* Recv depth = 5 */
  }
  while (rcv_file_num == -1);
  if (l_status != CCE)            /* error handling */
  {
    if (l_status == CCG)
    {
      FILEINFO (rcv_file_num, &l_err);
      if (l_err > 0)
      {
        DEBUG();
        continue_flag = false;
      }
    } else
    {
      DEBUG();
      continue_flag = false;
    }
  }
}    /* initialization */

#pragma PAGE "PROC process_spi_buffer"
void process_spi_buffer(void)
/*
 *===========================================================================
 * Proc      : process_spi_buffer                                           =
 * Function  : This procedure will format responses for successful          =
 *             processing of spi_command.                                   =
 *===========================================================================
 */
{
  short
      l_err,
      l_idx,
      l_len,
      l_start_idx;
```

Example E-12. C File: A Simple SPI Server (page 5 of 12)

```
/* determine if this is a valid SPI message */
if ((l_err = verify_msg (spi_buffer_size)))
{
  error_response (l_err);
  return;
}

/* get the spi_command from the request buffer */
if ((err = SSGETTKN (req_buffer, ZSPI_TKN_COMMAND, (char *) &spi_command)))
{
  display_spi_error (err, ZSPI_VAL_SSGETTKN,
                          ZSPI_TKN_COMMAND, true);
}

if ((l_err = validate_tokens()))
{
  error_response (l_err);
  return;
}

/* Security checks on the command/user could be done here. */

build_hdr_response ();

source_idx = 1;
dest_idx = 1;
tkn_count = 1;

/* Start Data list only if (ZSPI_TKN_MAXRESP is not = 0 */
if (max_resp)
{
  if (err = SSPUTTKN (res_buffer, ZSPI_TKN_DATALIST))
    display_spi_error (err, ZSPI_VAL_SSPUTTKN,
                            ZSPI_TKN_DATALIST, true);
}

l_start_idx = 0;
/* Check for CONTEXT token. */
if (context_count)
{
  /* Here the CONTEXT token was sent. Copy the passed context. */
  out_string.len = the_context.con_string.len;
  memcpy (out_string.data, the_context.con_string.data,
          out_string.len);
  l_start_idx = the_context.index;
}

/* Now perform the spi_command */
tkn_retcode = ZSPI_ERR_OK;
switch (spi_command)
{
  case ZSPI_CMD_GETVERSION :
    /* Put SERVER BANNER token in response buffer */
    if ((err = SSPUTTKN (res_buffer, ZSPI_TKN_SERVER_BANNER,
                         (char *) &server_banner)))
       display_spi_error (err, ZSPI_VAL_SSPUTTKN,
                               ZSPI_TKN_SERVER_BANNER, true);
    break;
```

Example E-12. C File: A Simple SPI Server (page 6 of 12)

```
case 1 :    /* string to reverse. */
  l_len = in_string.len;
  for (l_idx = l_start_idx; l_idx < l_len; l_idx++)
  {
    out_string.data [l_len - l_idx - 1] = in_string.data [l_idx];
  }
  out_string.len = in_string.len;
  if (err = SSPUTTKN (res_buffer, ZSPI_TKN_COMMENT,
                      (char *) &out_string))
     display_spi_error (err, ZSPI_VAL_SSPUTTKN,
                             ZSPI_TKN_COMMENT, true);
  break;

case 2  :                       /* Shift string to uppercase */
  /*
   * This could be done in one operation, but to show the use of
   * the CONTEXT token, it will be done one character at a time.
   */
  l_len = in_string.len;
  out_string.data [l_start_idx] = in_string.data [l_start_idx];
  SHIFTSTRING (&out_string.data [l_start_idx],
                   1 /* only one byte at a time */,
                   0 /*upshift*/);   /* Change to upper-case */
  l_start_idx = l_start_idx + 1;
  out_string.len = l_start_idx;

  if (out_string.len < in_string.len)
  {
    /* update the context and add it to the returned buffer.
    */
    the_context.command = spi_command;
    the_context.index = l_start_idx;
    the_context.con_string.len = out_string.len;
    memcpy (the_context.con_string.data, out_string.data,
            out_string.len);
    the_context.len = offsetof (context_template, con_string) +
                      out_string.len + 2;
    if (err = SSPUTTKN (res_buffer, ZSPI_TKN_CONTEXT,
                        (char *) &the_context))
      display_spi_error (err, ZSPI_VAL_SSPUTTKN,
                              ZSPI_TKN_CONTEXT, true);
  }
```

Example E-12. C File: A Simple SPI Server (page 7 of 12)

```
      /*
       * Here I have the option to add the work that has been done to
       * returned buffer. This depends on the function being performed.
       * I will add it here just for illustration purposes.
       */
      if (err = SSPUTTKN (res_buffer, ZSPI_TKN_COMMENT,
                          (char *) &out_string))
        display_spi_error (err, ZSPI_VAL_SSPUTTKN,
                                ZSPI_TKN_COMMENT, true);
      break;

    case 3 :                        /* Shift string to lowercase */
      out_string.len = in_string.len;
      memcpy (out_string.data, in_string.data, in_string.len);

      SHIFTSTRING (out_string.data,
                         out_string.len,
                         1 /*downshift*/);   /* Change to lower-case */
      if (err = SSPUTTKN (res_buffer, ZSPI_TKN_COMMENT,
                          (char *) &out_string))
        display_spi_error (err, ZSPI_VAL_SSPUTTKN,
                                ZSPI_TKN_COMMENT, true);
      break;

    default :                    /* (invalid command) */
      tkn_retcode = ZSPI_ERR_NOTIMP;
      break;

  } /* End of CASE (spi_command) */

  /* Put return code token in response buffer */
  if (err = SSPUTTKN (res_buffer, ZSPI_TKN_RETCODE, (char *) &tkn_retcode))
    display_spi_error (err, ZSPI_VAL_SSPUTTKN,
                            ZSPI_TKN_RETCODE, true);

  /* Put endlist token for end of response */
  if (max_resp && (resp_type != ZSPI_VAL_ERR_AND_WARN))
    if (err = SSPUTTKN (res_buffer, ZSPI_TKN_ENDLIST))
      display_spi_error (err, ZSPI_VAL_SSPUTTKN,
                              ZSPI_TKN_ENDLIST, true);

}  /* End of process_spi_buffer procedure */
```

Example E-12. C File: A Simple SPI Server (page 8 of 12)

```
#pragma PAGE "PROC process_requests"
/*
 *===================================================================
 * Proc      : process_requests                                     =
 * Function  : This procedure will process the requests received    =
 *             on $RECEIVE.                                         =
 *===================================================================
 */
void process_requests(void)
{

  /* Read in the spi_command received */

  READUPDATEX (rcv_file_num, (char *) &req_buffer[0], max_bufsize,
                  &spi_buffer_size);
  FILEINFO (rcv_file_num, &last_file_err);

  switch (last_file_err)        /* data was found in the buffer */
  {
    case 6 : /* system message */
      /* first word of req_buffer is message type. */
      switch (req_buffer[0])
      {
        case -30 :  /* OPEN message */
          if (memcmp((char *) &req_buffer[9], "#ZSPI   ", 8))
          {
            /* Reject the open with file error 11. */
            file_error = 11;
          }
          break;
        default :
          memcpy (res_buffer, req_buffer, max_bufsize);
          break;
      }
      break;

    case 0 : /* non-system message */
      /* first word of msg (Z_MSGCODE) is a -28 for a SPI msg. */
      switch (req_buffer[0])
      {
        case -28 :
          process_spi_buffer();
          break;
        default :
          memcpy (res_buffer, req_buffer, max_bufsize);
          break;
      }
      break;

    default :      /* unexpected message not a SPI or system message */
      memcpy (res_buffer, req_buffer, max_bufsize);
      continue_flag = false;
      break;
  }  /* of switch (last_file_err) */

}  /* of PROC process_requests */
```

Example E-12. C File: A Simple SPI Server (page 9 of 12)

```
#pragma PAGE "validate_tokens"
/*
 *=====================================================================
 * Proc      : validate_tokens                                        =
 * Function  : This procedure will determine if the "req_buffer"      =
 *             contains a valid SPI command. All required tokens      =
 *             must be present. Duplicate tokens, invalid tokens,     =
 *             invalid token values are rejected                      =
 * Returns   : An error code indicating the error found in the        =
 *              command buffer or ZSPI_ERR_OK (0) which indicates     =
 *              no error was found.                                   =
 *=====================================================================
 */
short validate_tokens(void)
{
  zspi_ddl_ssid_def  l_ssid;

  /* set default token values
   * if (any of these have to appear for a command set them to
   * null values here & check them in the command_ code
   * reset token counts
   */
  context_count = 0;
  tkn_count = 1;

  /* get the header tokens-  validate that they were retrieved ok */
  if (err = SSGETTKN (req_buffer, ZSPI_TKN_SSID, (char *) &l_ssid))
    display_spi_error (err, ZSPI_VAL_SSGETTKN,
                            ZSPI_TKN_SSID, true);

  /* check if the SSID matches mine, but don't check the version. */
  if (memcmp (&l_ssid, &my_ssid, 10) != 0)
  {
    tkn_code = ZSPI_TKN_SSID;
    return (ZSPI_ERR_TKN_VAL_INV);
  }

  if (err = SSGETTKN (req_buffer, ZSPI_TKN_MAXRESP, (char *) &max_resp))
    display_spi_error (err, ZSPI_VAL_SSGETTKN,
                            ZSPI_TKN_MAXRESP, true);

  if (max_resp < -1)
  {
    tkn_code = ZSPI_TKN_MAXRESP;
    return (ZSPI_ERR_TKN_VAL_INV);
  }

  /* reposition to head of SPI buffer */
  tkn_value = 0;

  if (err = SSPUTTKN (req_buffer, ZSPI_TKN_INITIAL_POSITION,
                      (char *) &tkn_value))
    display_spi_error (err, ZSPI_VAL_SSPUTTKN,
                            ZSPI_TKN_INITIAL_POSITION, true);
```

Example E-12. C File: A Simple SPI Server (page 10 of 12)

```
  /* walk through the buffer pulling out tokens */
  while (err = SSGETTKN (req_buffer, ZSPI_TKN_NEXTCODE,
                         (char *) &tkn_code,, &tkn_count) == ZSPI_ERR_OK)
 {
    switch (tkn_code)
    {
      case ZSPI_TKN_COMMENT :
        /* Check the entire token code, just to be sure. */
        if (tkn_code != ZSPI_TKN_COMMENT)
          return (ZCOM_ERR_TKN_CODE_INV);
        if (err = SSGETTKN (req_buffer, ZSPI_TKN_COMMENT,
                            (char *) &in_string))
          display_spi_error (err, ZSPI_VAL_SSGETTKN,
                                  ZSPI_TKN_COMMENT, true);
        break;

      case ZSPI_TKN_CONTEXT :
        /* Check the entire token code, just to be sure. */
        if (tkn_code != ZSPI_TKN_CONTEXT)
          return (ZCOM_ERR_TKN_CODE_INV);

        tkn_code = ZSPI_TKN_CONTEXT;
        context_count = context_count + tkn_count;
        if (context_count != 1)
        {
          return (ZCOM_ERR_TKN_DUP);
        } else
        {
          /* retrieve and validate the context */
          if (err = SSGETTKN (req_buffer, ZSPI_TKN_CONTEXT,
                              (char *) &the_context))
            display_spi_error (err, ZSPI_VAL_SSGETTKN,
                                    ZSPI_TKN_CONTEXT, true);
          if ((the_context.len > sizeof(context_template)) ||
             (the_context.len != (offsetof (context_template, con_string) +
                                  the_context.con_string.len + 2)) ||
             (the_context.command != spi_command) ||
             (the_context.index > the_context.con_string.len))
          {
            return (ZCOM_ERR_TKN_CNTXT_CODE_INV);
          }
        }
        break;

      default :
        return (ZCOM_ERR_TKN_CODE_INV);
        break;

    } /* of switch */
  } /* of while */

  return (ZSPI_ERR_OK);       /* no errors found, return ok */
}  /* validate_tokens */
```

Example E-12. C File: A Simple SPI Server (page 11 of 12)

```
#pragma PAGE "verify_msg"
/*
 *=========================================================================
 * Proc      : verify_msg                                                  =
 * Function  : This procedure will determine if a valid SPI buffer was     =
 *             received.                                                   =
 * Returns   : An error code indicating the error found,                   =
 *             or ZSPI_ERR_OK (0) which indicates no error was found.      =
 *=========================================================================
 */
short verify_msg (short p_count)
/* size of data read must be at least 6 */
{
  if (p_count < 6)
    return (ZSPI_ERR_INVBUF);
  /* Reset the buffer */
  tkn_count = 1;
  if (err = SSPUTTKN (req_buffer, ZSPI_TKN_RESET_BUFFER,
                      (char *) &bufsize))
  {
    display_spi_error (err, ZSPI_VAL_SSPUTTKN,
                            ZSPI_TKN_RESET_BUFFER, false);
    return (ZSPI_ERR_INVBUF);
  }

  /* header type must be a complete header */
  /* SSGET of ZSPI_TKN_HDRTYPE returns a token value of ZSPI_VAL_CMDHDR */
  if (err = SSGETTKN (req_buffer, ZSPI_TKN_HDRTYPE, (char *) &tkn_value,
                            , &tkn_count))
  {
    display_spi_error (err, ZSPI_VAL_SSGETTKN,
                            ZSPI_TKN_HDRTYPE, false);
    return (ZSPI_ERR_INVBUF);
  }

  if (tkn_value != ZSPI_VAL_CMDHDR)
    return (ZSPI_ERR_INVBUF);

  tkn_code = ZSPI_TKN_HDRTYPE;
  return (err);
  }   /* procedure verify_msg */
```

Example E-12. C File: A Simple SPI Server (page 12 of 12)

```
#pragma PAGE "PROC server MAIN"
/*
 *=========================================================================
 *     MAINLINE ROUTINE STARTS HERE.                                      =
 *                                                                        =
 *=========================================================================
 */
main(/* int argc, char *argv[] */)
{
  short l_reply_length;
  short l_status;

  initialization();

  continue_flag = true;
  do
  {
    /* Clear important fields */
    memset (res_buffer, '0', sizeof (res_buffer));
    memset (req_buffer, '0', sizeof (req_buffer));
    /* Read a request message from $RECEIVE */
    process_requests();

    /* SEND reply buffer to $RECEIVE */

    l_reply_length = max_bufsize;
    if (file_error)
    {
       l_reply_length = 0;
    }

    l_status = REPLYX ((char *) &res_buffer[0], l_reply_length,
                       /*count_sent*/, /*tag*/, file_error);
    if (l_status == CCL)
    {
      get_file_error (rcv_file_num);
      continue_flag = false;
    }

    /* Clear important fields */
    file_error = 0;

  } while (continue_flag == true);

  STOP();

}  /* of main */
```

Example E-13: Common Declarations for TAL Examples

This TAL code contains common declarations used by the other TAL example programs.

Source File

SETCDECS

Example E-13. TAL File: SETDECS Supporting Code

```
?SYMBOLS, INSPECT, NOCODE, NOMAP, NOLMAP, DATAPAGES 64
?NOLIST, SOURCE ZSPITAL
?LIST
--
--   File name: SETCDECS
--   SPI EXAMPLE TAL Common Declarations and variables.
--
LITERAL  tkn^1 = 1D '<<' 24 + 1D '<<' 16 + 1D;
LITERAL  tkn^2 = 1D '<<' 24 + 1D '<<' 16 + 2D;
LITERAL  tkn^3 = 1D '<<' 24 + 1D '<<' 16 + 3D;
LITERAL  tkn^4 = 1D '<<' 24 + 1D '<<' 16 + 4D;
LITERAL  tkn^5 = 1D '<<' 24 + 1D '<<' 16 + 5D;
LITERAL  tkn^6 = 1D '<<' 24 + 1D '<<' 16 + 6D;

LITERAL  false = 0,
         true = -1;
-- SPI related variables
INT     .b1 [0:max^bufsize/2];    ! SPI buffer 1
INT     .b2 [0:max^bufsize/2];    ! SPI buffer 2
INT      bufsize;
INT      err := 0;
INT      last^file^err;           ! Set by "get^file^error"
INT     .ssid [0:5]          := ["EXAMPLES", 1, 0 ];
INT(32)  tkn^code;
STRUCT   tkn^code^def (zspi^ddl^tokencode^def) = tkn^code;
INT     .tkn^buffer^i [0:49];                          ! For GET and PUT
INT     .tkn^value           := @tkn^buffer^i;         ! For GET and PUT
INT(32) .tkn^value^2         := @tkn^buffer^i;         ! For GET and PUT
STRING  .tkn^buffer          := @tkn^buffer^i '<<' 1; ! For GET and PUT
STRING   val;

-- Program misc. variables
INT      continue;               ! flag
INT      line [0:50];            ! output buffer for the home term.
STRING  .sline := @line '<<' 1;
STRING  .sp;                     ! String pointer.
INT      term;                   ! The home term's file number.
INT     .termname [0:11];        ! The home term's name.
```

Example E-14: Common Declarations for C Examples

This C code contains common declarations used by the other C example programs.

Source File

SECCH

Example E-14. C File: SECCH Supporting Code

```
#include <tal.h>    nolist
#include <cextdecs> nolist
#include <ctype.h>  nolist
#include <stdio.h>  nolist
#include <stdlib.h> nolist
#include <stddef.h> nolist
#include <time.h>   nolist
#include <string.h> nolist
#include <memory.h> nolist
#include <fcntl.h>  nolist
#include "zspic"   nolist
#pragma list
/*
**   File name: secch
**   SPI EXAMPLE C Common Declarations and variables.
*/
#define  tkn_1  16842753lu  /* 1D '<<' 24 + 1D '<<' 16 + 1D */
#define  tkn_2  16842754lu  /* 1D '<<' 24 + 1D '<<' 16 + 2D */
#define  tkn_3  16842755lu  /* 1D '<<' 24 + 1D '<<' 16 + 3D */
#define  tkn_4  16842756lu  /* 1D '<<' 24 + 1D '<<' 16 + 4D */
#define  tkn_5  16842757lu  /* 1D '<<' 24 + 1D '<<' 16 + 5D */
#define  tkn_6  16842758lu  /* 1D '<<' 24 + 1D '<<' 16 + 6D */

#define  false  0
#define  true  -1

/* SPI related variables */
short    b1 [max_bufsize/2];    /* SPI buffer 1 */
short    b2 [max_bufsize/2];    /* SPI buffer 2 */
short    bufsize;
short    err = 0;
short    last_file_err;         /* Set by "get_file_error" */
short    get_count;             /* Count param to SSGET/SSGETTKN
*/
long     tkn_code;
short    tkn_buffer_i [50];                  /* For GET and PUT */
short*   tkn_value;      /*  = @tkn_buffer_i;  /* For GET and PUT */
long*    tkn_value_2;    /*  = @tkn_buffer_i;      /* For GET and PUT */
char*    tkn_buffer;     /*  = @tkn_buffer_i '<<' 1; /8 For GET and PUT */
char     val;
zspi_ddl_ssid_def       ssid  = {{'E','X','A','M','P','L','E','S'}, 1, 0};
zspi_ddl_tokencode_def  tkn_code_def; /* = tkn_code; */

/* Program misc. variables */
short    continue_flag;          /* flag */
```

Example E-15: Common Routines for TAL Examples

This TAL code contains common routines used by the other TAL example programs.

Source File

SETCUTIL

Example E-15. TAL File: SETCUTIL Supporting Code (page 1 of 5)

```
--   File name: SETCUTIL
--   SPI EXAMPLE TAL Common Utility procedures.
--
?PAGE "FORWARD declarations"
PROC display^token (p^tkn^code);
INT(32) p^tkn^code;
FORWARD;
------------------------------------------------------------------------------
?PAGE "PROC display^spi^error"
!============================================================================!
! Proc      : display^spi^error                                              !
! Function  : This procedure will format the SPI error that is passed into   !
!             a text message and display it on the home term. It will call    !
!             DEBUG if the "p^call^debug" parameter is true.                 !
!============================================================================!

PROC display^spi^error (p^spi^err, p^spi^proc, p^tkn^code, p^call^debug);
INT     p^spi^err;
INT     p^spi^proc;
INT(32) p^tkn^code;
INT     p^call^debug;
BEGIN
  sline ':=' "Error from " -> @sp;
  CASE p^spi^proc OF
  BEGIN
    ZSPI^VAL^SSINIT              ->      sp ':=' "SSINIT"            -> @sp;
    ZSPI^VAL^SSGET               ->      sp ':=' "SSGET"             -> @sp;
    ZSPI^VAL^SSGETTKN            ->      sp ':=' "SSGETTKN"          -> @sp;
    ZSPI^VAL^SSMOVE              ->      sp ':=' "SSMOVE"            -> @sp;
    ZSPI^VAL^SSMOVETKN           ->      sp ':=' "SSMOVETKN"         -> @sp;
    ZSPI^VAL^SSNULL              ->      sp ':=' "SSNULL"            -> @sp;
    ZSPI^VAL^SSPUT               ->      sp ':=' "SSPUT"             -> @sp;
    ZSPI^VAL^SSPUTTKN            ->      sp ':=' "SSPUTTKN"          -> @sp;
    ZSPI^VAL^BUFFER^FORMATSTART  ->      sp ':=' "FORMATSTART"       -> @sp;
    ZSPI^VAL^BUFFER^FORMATNEXT   ->      sp ':=' "FORMATNEXT"        -> @sp;
    ZSPI^VAL^BUFFER^FORMATFINISH ->      sp ':=' "FORMATFINISH"      -> @sp;
    ZSPI^VAL^FORMAT^CLOSE        ->      sp ':=' "FORMATCLOSE"       -> @sp;
    OTHERWISE                    ->      sp ':=' "???Unknown???"     -> @sp;
  END; -- of CASE p^spi^proc
  IF p^tkn^code <> 0d THEN
    CALL display^token (p^tkn^code);
  sp ':=' " (" -> @sp;
  IF p^spi^err < 0 THEN
  BEGIN
    sp ':=' "-" -> @sp;
    @sp := @sp '+' dnumout (sp, $DBL(-p^spi^err), 10);
  END ELSE
  BEGIN
    @sp := @sp '+' dnumout (sp, $DBL(p^spi^err), 10);
  END;
```

Example E-15. TAL File: SETCUTIL Supporting Code (page 2 of 5)

```
  sp ':=' ", " -> @sp;
  CASE p^spi^err OF
  BEGIN
    ZSPI^ERR^INVBUF       ->      sp ':=' "Invalid Buffer"        -> @sp;
    ZSPI^ERR^ILLPARM      ->      sp ':=' "Illegal Param"         -> @sp;
    ZSPI^ERR^MISPARM      ->      sp ':=' "Missing Param"         -> @sp;
    ZSPI^ERR^BADADDR      ->      sp ':=' "Illegal Address"       -> @sp;
    ZSPI^ERR^NOSPACE      ->      sp ':=' "Buffer full"           -> @sp;
    ZSPI^ERR^XSUMERR      ->      sp ':=' "Invalid Checksum"      -> @sp;
    ZSPI^ERR^INTERR       ->      sp ':=' "Internal Error"        -> @sp;
    ZSPI^ERR^MISTKN       ->      sp ':=' "Missing Token"         -> @sp;
    ZSPI^ERR^ILLTKN       ->      sp ':=' "Illegal Token"         -> @sp;
    ZSPI^ERR^BADSSID      ->      sp ':=' "Bad SSID"              -> @sp;
    ZSPI^ERR^NOTIMP       ->      sp ':=' "Not implemented"       -> @sp;
    ZSPI^ERR^NOSTACK      ->      sp ':=' "Insufficient Stack"    -> @sp;
    ZSPI^ERR^ZFIL^ERR     ->      sp ':=' "File system error"     -> @sp;
    ZSPI^ERR^ZGRD^ERR     ->      sp ':=' "OS Kernel error"       -> @sp;
    ZSPI^ERR^INV^FILE     ->      sp ':=' "Invalid template file" -> @sp;
    ZSPI^ERR^CONTINUE     ->      sp ':=' "Continue"              -> @sp;
    ZSPI^ERR^NEW^LINE     ->      sp ':=' "New line"              -> @sp;
    ZSPI^ERR^NO^MORE      ->      sp ':=' "No more"               -> @sp;
    ZSPI^ERR^MISS^NAME    ->      sp ':=' "Missing name"          -> @sp;
    ZSPI^ERR^DUP^NAME     ->      sp ':=' "Duplicate name"        -> @sp;
    ZSPI^ERR^MISS^ENUM    ->      sp ':=' "Missing enumeration"   -> @sp;
    ZSPI^ERR^MISS^STRUCT  ->      sp ':=' "Missing STRUCT"        -> @sp;
    ZSPI^ERR^MISS^OFFSET  ->      sp ':=' "Missing offset"        -> @sp;
    ZSPI^ERR^TOO^LONG     ->      sp ':=' "Too long"              -> @sp;
    ZSPI^ERR^MISS^FIELD   ->      sp ':=' "Missing field"         -> @sp;
    ZSPI^ERR^NO^SCANID    ->      sp ':=' "No SCAN ID"            -> @sp;
    ZSPI^ERR^NO^FORMATID  ->      sp ':=' "No Format ID"          -> @sp;
    ZSPI^ERR^OCCURS^DEPTH ->      sp ':=' "Occurs depth"          -> @sp;
    ZSPI^ERR^MISS^LABEL   ->      sp ':=' "Missing label"         -> @sp;
    ZSPI^ERR^BUF^TOO^LARGE->      sp ':=' "Buffer is too big"     -> @sp;
    ZSPI^ERR^OBJFORM      ->      sp ':=' "Object form"           -> @sp;
    ZSPI^ERR^OBJCLASS     ->      sp ':=' "Object class"          -> @sp;
    ZSPI^ERR^BADNAME      ->      sp ':=' "Bad name"              -> @sp;
    ZSPI^ERR^TEMPLATE     ->      sp ':=' "Template"              -> @sp;
    ZSPI^ERR^ILL^CHAR     ->      sp ':=' "Illegal character"     -> @sp;
    ZSPI^ERR^NO^TKNDEFID  ->      sp ':=' "No TKNDEF ID"          -> @sp;
    ZSPI^ERR^INCOMP^RESP  ->      sp ':=' "Incomplete response"   -> @sp;
    OTHERWISE             ->      sp ':=' "???Unknown???"         -> @sp;
  END; -- of CASE p^spi^err
  sp ':=' ")" -> @sp;
  CALL writex (term, sline, @sp '-' @sline);
  ! Write a blank line for output clarity
  sline ':=' " " & sline FOR 79 BYTES;
  CALL writex (term, sline, 2);
  IF p^call^debug THEN CALL DEBUG;
END;
```

Example E-15. TAL File: SETCUTIL Supporting Code (page 3 of 5)

```
?PAGE "PROC display^token"
!=========================================================================!
! Proc      : display^token                                               !
! Function  : This procedure will add the token name of the passed        !
!             token code to the current position pointer to by "sp".      !
!             Nothing is written to the home term.                        !
!=========================================================================!

PROC display^token (p^tkn^code);
INT(32) p^tkn^code;
BEGIN
  sp ':=' " " -> @sp;
  IF p^tkn^code = tkn^1 THEN
    sp ':=' "(TKN^1)" -> @sp;
  IF p^tkn^code = tkn^2 THEN
    sp ':=' "(TKN^2)" -> @sp;
  IF p^tkn^code = tkn^3 THEN
    sp ':=' "(TKN^3)" -> @sp;
  IF p^tkn^code = tkn^4 THEN
    sp ':=' "(TKN^4)" -> @sp;
  IF p^tkn^code = tkn^5 THEN
    sp ':=' "(TKN^5)" -> @sp;
  IF p^tkn^code = tkn^6 THEN
    sp ':=' "(TKN^6)" -> @sp;
  IF p^tkn^code = zspi^tkn^command THEN
    sp ':=' "(COMMAND)" -> @sp;
  IF p^tkn^code = zspi^tkn^comment THEN
    sp ':=' "(COMMENT)" -> @sp;
  IF p^tkn^code = zspi^tkn^context THEN
    sp ':=' "(CONTEXT)" -> @sp;
  IF p^tkn^code = zspi^tkn^datalist THEN
    sp ':=' "(DATALIST)" -> @sp;
  IF p^tkn^code = zspi^tkn^endlist THEN
    sp ':=' "(ENDLIST)" -> @sp;
  IF p^tkn^code = zspi^tkn^errlist THEN
    sp ':=' "(ERRLIST)" -> @sp;
  IF p^tkn^code = zspi^tkn^error THEN
    sp ':=' "(ERROR)" -> @sp;
  IF p^tkn^code = zspi^tkn^hdrtype THEN
    sp ':=' "(HDRTYPE)" -> @sp;
  IF p^tkn^code = zspi^tkn^initial^position THEN
    sp ':=' "(INITIAL^POSITION)" -> @sp;
  IF p^tkn^code = zspi^tkn^manager THEN
    sp ':=' "(MANAGER)" -> @sp;
  IF p^tkn^code = zspi^tkn^maxresp THEN
    sp ':=' "(MAXRESP)" -> @sp;
  IF p^tkn^code = zspi^tkn^nextcode THEN
    sp ':=' "(NEXTCODE)" -> @sp;
  IF p^tkn^code = zspi^tkn^nexttoken THEN
    sp ':=' "(NEXTTOKEN)" -> @sp;
  IF p^tkn^code = zspi^tkn^object^type THEN
    sp ':=' "(OBJECT^TYPE)" -> @sp;
  IF p^tkn^code = zspi^tkn^parm^err THEN
    sp ':=' "(PARM^ERR)" -> @sp;
  IF p^tkn^code = zspi^tkn^proc^err THEN
    sp ':=' "(PROC^ERR)" -> @sp;
  IF p^tkn^code = zspi^tkn^reset^buffer THEN
    sp ':=' "(RESET^BUFFER)" -> @sp;
  IF p^tkn^code = zspi^tkn^retcode THEN
    sp ':=' "(RETCODE)" -> @sp;
  IF p^tkn^code = zspi^tkn^server^banner THEN
    sp ':=' "(SERVER^BANNER)" -> @sp;
```

Example E-15. TAL File: SETCUTIL Supporting Code (page 4 of 5)

```
   IF p^tkn^code = zspi^tkn^server^version THEN
    sp ':=' "(SERVER^VERSION)" -> @sp;
  IF p^tkn^code = zspi^tkn^ssid THEN
    sp ':=' "(SSID)" -> @sp;
END;

?PAGE "PROC dump^buf"
PROC dump^buf (p^spi^buf);
!============================================================================!
! Proc      : dump^buf                                                       !
! Function  : This procedure will perform a labeled dump of the passed SPI   !
!             buffer and display it on the home term.                        !
!============================================================================!

INT    .p^spi^buf;
BEGIN
  LITERAL numeric^format = 10,    ! Decimal
          string^format = 0,      ! Char codes 32 - 126
          type^override = 0,      ! No TYPE overrides
          show^redef = 0,         ! Don't show redefines
          show^hidden = 1,        ! Show hidden fields
          no^header = 0,          ! Show header and data fields
          name^or^label = 1,      ! Use DDL names
          ems^or^ss = 1,
          tkn^label^len = 30,
          field^label^len = 0,
          max^lines = 10,
          max^line^len = 80;
  ! Note that the SPI_formatnext procedure can modify the spi buffer.
  !  Thus I will move it to l^spi^buf and display it from here.
  INT    .l^spi^buf [0:max^bufsize]; ! buffer to display
  INT    .l^format^buf [0:max^lines * max^line^len / 2];
  STRING .l^sformat^buf := @l^format^buf '<<' 1;
  INT    .l^lengths [0:max^lines];
  INT     l^buf^len,
          l^idx,
          l^cmd^num,
          l^done := 0,
          l^format^id,  !used by formatting routines to keep track of info.
          l^status,
          l^status1,
          l^status2;

  l^spi^buf ':=' p^spi^buf FOR $OCCURS (l^spi^buf) WORDS;
  ! get a format area reserved
  err := spi_buffer_formatstart_ (l^format^id, numeric^format, string^format,
                                  type^override, show^redef, show^hidden,
                                  no^header, name^or^label, ems^or^ss,
                                  tkn^label^len, field^label^len,
                                  l^status1, l^status2);
  IF err THEN
    CALL display^spi^error (err, zspi^val^buffer^formatstart, 0d, true);
  err := 0;
  WHILE NOT l^done DO
  BEGIN
    err := spi_buffer_formatnext_ (l^format^id,
                                   l^spi^buf,
                                   l^sformat^buf: max^lines*max^line^len,
                                   max^lines,
                                   l^lengths,
                                   l^status1,
                                   l^status2);
```

Example E-15. TAL File: SETCUTIL Supporting Code (page 5 of 5)

```
    USE l^idx2;
    IF err = 0 OR err = zspi^err^continue THEN
    BEGIN
      l^idx2 := 0;
      FOR l^idx2 := 0 TO max^lines - 1 DO
      BEGIN
        IF l^lengths [l^idx2] > -1 THEN
        BEGIN
          CALL writex (term, l^sformat^buf [l^idx2 * max^line^len],
                       l^lengths [l^idx2]);
        END; -- of IF l^lengths [l^idx2] > -1
      END; -- of FOR l^idx2 := 0 to max^lines - 1 DO
    END ELSE -- IF err = 0 OR err ... ELSE
    BEGIN
      ! an error found, l^done printing
      l^done := true;
      CALL display^spi^error (err, zspi^val^buffer^formatnext, 0d, true);
    END; -- of IF err = 0 OR err .... ELSE

    ! we are l^done printing IF all found (err = 0)
    IF NOT err THEN l^done := true;

  END; -- of WHILE NOT l^done DO

  ! release the format area
  CALL spi_buffer_formatfinish_ (l^format^id, l^status1, l^status2);
  ! Write a blank line for output clarity
  l^sformat^buf ':=' " " & l^sformat^buf FOR 79 BYTES;
  CALL writex (term, l^sformat^buf, 2);

END;  ! End of dump^buf PROC

?PAGE "PROC get^file^error"
!=============================================================================!
! Proc      : get^file^error                                                  !
! Function  : This procedure will get the file error from the file            !
!             number that is passed and store it in "last^file^err".          !
!=============================================================================!

PROC get^file^error (p^file^num);
INT p^file^num;
BEGIN
  CALL fileinfo (p^file^num, last^file^err);
  CALL debug;
END;  -- of PROC get^file^error
```

Example E-16: Common Routines for C Examples

This C code contains common routines used by the other C example programs.

Source File

SECCUTLC

Example E-16. C File: SECCUTLC Supporting Code (page 1 of 5)

```
/*   File name: seccutlc
**   SPI EXAMPLE C Common Utility procedures.
*/
#pragma PAGE "FORWARD declarations"
void display_spi_error (short p_spi_err,
                        short p_spi_proc,
                        long  p_tkn_code,
                        short p__debug);

void display_token (long p_tkn_code);

void dump_buf (short* p_spi_buf);

void get_file_error (short p_file_num);

/*====================================================================*
#pragma PAGE "display_spi_error"
/*
======================================================================
== Proc      : display_spi_error                                     =
== Function  : This procedure will format the SPI error that is passed   =
==             into a text message and display it on the home term. It   =
==             will call DEBUG if the "p__debug" parameter is true.      =
======================================================================
*/
void display_spi_error (short p_spi_err,
                        short p_spi_proc,
                        long  p_tkn_code,
                        short p__debug)
{
  printf ("Error from ");
  switch (p_spi_proc)
  {
    case ZSPI_VAL_SSINIT              : printf("SSINIT");        break;
    case ZSPI_VAL_SSGET               : printf("SSGET");         break;
    case ZSPI_VAL_SSGETTKN            : printf("SSGETTKN");      break;
    case ZSPI_VAL_SSMOVE              : printf("SSMOVE");        break;
    case ZSPI_VAL_SSMOVETKN           : printf("SSMOVETKN");     break;
    case ZSPI_VAL_SSNULL              : printf("SSNULL");        break;
    case ZSPI_VAL_SSPUT               : printf("SSPUT");         break;
    case ZSPI_VAL_SSPUTTKN            : printf("SSPUTTKN");      break;
    case ZSPI_VAL_BUFFER_FORMATSTART  : printf("FORMATSTART");   break;
    case ZSPI_VAL_BUFFER_FORMATNEXT   : printf("FORMATNEXT");    break;
    case ZSPI_VAL_BUFFER_FORMATFINISH : printf("FORMATFINISH");  break;
    case ZSPI_VAL_FORMAT_CLOSE        : printf("FORMATCLOSE");   break;
    default                           : printf("???Unknown???"); break;
  } /* of switch (p_spi_proc) */
  if (p_tkn_code != 0L)
     display_token (p_tkn_code);
  printf(" (%d, ", p_spi_err);
  switch (p_spi_err)
  {
    case ZSPI_ERR_INVBUF       : printf("Invalid Buffer");        break;
    case ZSPI_ERR_ILLPARM      : printf("Illegal Param");         break;
    case ZSPI_ERR_MISPARM      : printf("Missing Param");         break;
    case ZSPI_ERR_BADADDR      : printf("Illegal Address");       break;
    case ZSPI_ERR_NOSPACE      : printf("Buffer full");           break;
```

Example E-16. C File: SECCUTLC Supporting Code (page 2 of 5)

```
    case ZSPI_ERR_XSUMERR      : printf("Invalid Checksum");      break;
    case ZSPI_ERR_INTERR       : printf("Internal Error");        break;
    case ZSPI_ERR_MISTKN       : printf("Missing Token");         break;
    case ZSPI_ERR_ILLTKN       : printf("Illegal Token");         break;
    case ZSPI_ERR_BADSSID      : printf("Bad SSID");              break;
    case ZSPI_ERR_NOTIMP       : printf("Not implemented");       break;
    case ZSPI_ERR_NOSTACK      : printf("Insufficient Stack");    break;
    case ZSPI_ERR_ZFIL_ERR     : printf("File system error");     break;
    case ZSPI_ERR_ZGRD_ERR     : printf("OS Kernel error");       break;
    case ZSPI_ERR_INV_FILE     : printf("Invalid template file"); break;"
    case ZSPI_ERR_CONTINUE     : printf("Continue");              break;
    case ZSPI_ERR_NEW_LINE     : printf("New line");              break;
    case ZSPI_ERR_NO_MORE      : printf("No more");               break;
    case ZSPI_ERR_MISS_NAME    : printf("Missing name");          break;
    case ZSPI_ERR_DUP_NAME     : printf("Duplicate name");        break;
    case ZSPI_ERR_MISS_ENUM    : printf("Missing enumeration");   break;
    case ZSPI_ERR_MISS_STRUCT  : printf("Missing STRUCT");        break;
    case ZSPI_ERR_MISS_OFFSET  : printf("Missing offset");        break;
    case ZSPI_ERR_TOO_LONG     : printf("Too long");              break;
    case ZSPI_ERR_MISS_FIELD   : printf("Missing field");         break;
    case ZSPI_ERR_NO_SCANID    : printf("No SCAN ID");            break;
    case ZSPI_ERR_NO_FORMATID  : printf("No Format ID");          break;
    case ZSPI_ERR_OCCURS_DEPTH : printf("Occurs depth");          break;
    case ZSPI_ERR_MISS_LABEL   : printf("Missing label");         break;
    case ZSPI_ERR_BUF_TOO_LARGE: printf("Buffer is too big");     break;
    case ZSPI_ERR_OBJFORM      : printf("Object form");           break;
    case ZSPI_ERR_OBJCLASS     : printf("Object class");          break;
    case ZSPI_ERR_BADNAME      : printf("Bad name");              break;
    case ZSPI_ERR_TEMPLATE     : printf("Template");              break;
    case ZSPI_ERR_ILL_CHAR     : printf("Illegal character");     break;
    case ZSPI_ERR_NO_TKNDEFID  : printf("No TKNDEF ID");          break;
    case ZSPI_ERR_INCOMP_RESP  : printf("Incomplete response");   break;
    default                    : printf("???Unknown???");         break;
  } /* of switch (p_spi_err) */
  printf(")\n");

 /* Write a blank line for output clarity */
  printf (" \n");
  if (p__debug)  DEBUG();
}

#pragma PAGE "display_token"
/*
============================================================================
== Proc      : display_token                                               =
== Function  : This procedure will add the token name of the passed        =
==             token code and write its name.                               =
============================================================================
*/
void display_token (long p_tkn_code)
{
  printf(" ");
  if (p_tkn_code == tkn_1)
    printf("(TKN_1)");
  if (p_tkn_code == tkn_2)
    printf("(TKN_2)");
  if (p_tkn_code == tkn_3)
    printf("(TKN_3)");
```

Example E-16. C File: SECCUTLC Supporting Code (page 3 of 5)

```
  if (p_tkn_code == tkn_4)
    printf("(TKN_4)");
  if (p_tkn_code == tkn_5)
    printf("(TKN_5)");
  if (p_tkn_code == tkn_6)
    printf("(TKN_6)");
  if (p_tkn_code == ZSPI_TKN_COMMAND)
    printf("(COMMAND)");
  if (p_tkn_code == ZSPI_TKN_COMMENT)
    printf("(COMMENT)");
  if (p_tkn_code == ZSPI_TKN_CONTEXT)
    printf("(CONTEXT)");
  if (p_tkn_code == ZSPI_TKN_DATALIST)
    printf("(DATALIST)");
  if (p_tkn_code == ZSPI_TKN_ENDLIST)
    printf("(ENDLIST)");
  if (p_tkn_code == ZSPI_TKN_ERRLIST)
    printf("(ERRLIST)");
  if (p_tkn_code == ZSPI_TKN_ERROR)
    printf("(ERROR)");
  if (p_tkn_code == ZSPI_TKN_HDRTYPE)
    printf("(HDRTYPE)");
  if (p_tkn_code == ZSPI_TKN_INITIAL_POSITION)
    printf("(INITIAL_POSITION)");
  if (p_tkn_code == ZSPI_TKN_MANAGER)
    printf("(MANAGER)");
  if (p_tkn_code == ZSPI_TKN_MAXRESP)
    printf("(MAXRESP)");
  if (p_tkn_code == ZSPI_TKN_NEXTCODE)
    printf("(NEXTCODE)");
  if (p_tkn_code == ZSPI_TKN_NEXTTOKEN)
    printf("(NEXTTOKEN)");
  if (p_tkn_code == ZSPI_TKN_OBJECT_TYPE)
    printf("(OBJECT_TYPE)");
  if (p_tkn_code == ZSPI_TKN_PARM_ERR)
    printf("(PARM_ERR)");
  if (p_tkn_code == ZSPI_TKN_PROC_ERR)
    printf("(PROC_ERR)");
  if (p_tkn_code == ZSPI_TKN_RESET_BUFFER)
    printf("(RESET_BUFFER)");
  if (p_tkn_code == ZSPI_TKN_RETCODE)
    printf("(RETCODE)");
  if (p_tkn_code == ZSPI_TKN_SERVER_BANNER)
    printf("(SERVER_BANNER)");
  if (p_tkn_code == ZSPI_TKN_SERVER_VERSION)
    printf("(SERVER_VERSION)");
  if (p_tkn_code == ZSPI_TKN_SSID)
    printf("(SSID)");
}
```

Example E-16. C File: SECCUTLC Supporting Code (page 4 of 5)

```
#pragma PAGE "dump_buf"
void dump_buf (short* p_spi_buf)
/*
============================================================================
== Proc      : dump_buf                                                    =
== Function  : This procedure will perform a labeled dump of the passed   =
==             SPI buffer and display it on the home term.                 =
============================================================================
*/
{
  #define numeric_format  10     /* Decimal */
  #define string_format   0      /* Char codes 32 - 126 */
  #define type_override   0      /* No TYPE overrides */
  #define show_redef      0      /* Don't show redefines */
  #define show_hidden     1      /* Show hidden fields */
  #define no_header       0      /* Show header and data fields */
  #define name_or_label   1      /* Use DDL names */
  #define ems_or_ss       1
  #define tkn_label_len   30
  #define field_label_len 0
  #define max_lines       10
  #define max_line_len    80
  /*
  ** Note that the SPI_formatnext procedure can modify the spi buffer.
  **  Thus I will move it to l_spi_buf and display it from here.
  */
  short   l_spi_buf [max_bufsize];     /* buffer to display */
  short   l_format_buf [max_lines * max_line_len / 2];
  char    l_line [max_line_len + 1];   /* Add 1 for terminator */
  short   l_lengths [max_lines];
  short   l_idx,
          l_done = 0,
          l_format_id, /* used by formatting routines to keep track of
info.*/
          l_status1,
          l_status2;

  memcpy (&l_spi_buf, p_spi_buf, (max_bufsize * 2)); /* Copy the buffer */
  /* get a format area reserved */
  err = SPI_BUFFER_FORMATSTART_ (&l_format_id,
                                 numeric_format,
                                 string_format,
                                 type_override,
                                 show_redef,
                                 show_hidden,
                                 no_header,
                                 name_or_label,
                                 ems_or_ss,
                                 tkn_label_len,
                                 field_label_len,
                                 &l_status1,
                                 &l_status2);
  if (err)
    display_spi_error (err, ZSPI_VAL_BUFFER_FORMATSTART, 0L, true);
```

Example E-16. C File: SECCUTLC Supporting Code (page 5 of 5)

```
  err = 0;
  while (! l_done)
  {
    err = SPI_BUFFER_FORMATNEXT_ (l_format_id,
                                  l_spi_buf,
                                  (char *) &l_format_buf,
                                  max_lines * max_line_len,
                                  max_lines,
                                  l_lengths,
                                  &l_status1,
                                  &l_status2);
    if (err == 0
     || err == ZSPI_ERR_CONTINUE)
    {
      for (l_idx = 0; l_idx < max_lines; ++l_idx)
      {
        if (l_lengths [l_idx] > -1)
        {
          memcpy (&l_line, &l_format_buf [l_idx * max_line_len / 2],
                  l_lengths [l_idx]);       /* Copy the output line */
          l_line [l_lengths [l_idx]] = 0;   /* terminate the string */
          printf("%s\n", l_line);
        } /* of if */
      } /* of for */
    } else /* if */
    {
      /* an error found, done printing */
      l_done = true;
      display_spi_error (err, ZSPI_VAL_BUFFER_FORMATNEXT, 0L, true);
    } /* of if */

    /* we are done printing if all found (err = 0) */
    if (! err)  l_done = true;

  } /* of while */

  /* release the format area */
  SPI_BUFFER_FORMATFINISH_ (&l_format_id, &l_status1, &l_status2);
  /* Write a blank line for output clarity */
  printf(" \n");

}   /* End of dump_buf */

#pragma PAGE "get_file_error"
/*
============================================================================
== Proc      : get_file_error                                              =
== Function  : This procedure will get the file error from the file        =
==             number that is passed and store it in "last_file_err".      =
============================================================================
*/
void get_file_error (short p_file_num)
{
   FILEINFO (p_file_num, &last_file_err);
   DEBUG ();
}   /* of get_file_error */
```

Example E-17: Declarations for TAL Requesters and Servers

This TAL code contains common declarations used by the TAL requester and server example programs.

Source File

SETRDECS

Example E-17. TAL File: SETRDECS Supporting Code

```
--   File name: SETRDECS
--   SPI EXAMPLE TAL Requester Declarations and variables.
--
STRUCT ci^startup^def (*);
  BEGIN
  INT msgcode;
  STRUCT default;
    BEGIN
    INT volume [0:3],
        subvol [0:3];
    END; -- of STRUCT
  STRUCT infile;
    BEGIN
    INT volume [0:3],
        subvol [0:3],
        dname  [0:3];
    END; -- of STRUCT
  STRUCT outfile;
    BEGIN
    INT volume [0:3],
        subvol [0:3],
        dname  [0:3];
    END; -- of STRUCT
  STRING param [0:255];
  END;  -- of STRUCT

STRUCT      .start^buffer (ci^startup^def);
STRUCT      .param^msg = start^buffer;
  BEGIN
    INT        msg^code,
               param^count;
    STRING     param [0:1023];
  END;
STRUCT  .startup^msg  (ci^startup^def);

STRUCT  string^template(*);
BEGIN
  INT    len;
  STRING data [0:199];
END;

STRUCT  context^template(*);
BEGIN
  INT    len;
  INT    command;
  INT    index;                                ! Current index into the string
  STRUCT con^string (string^template);         ! The converted string, so far
END;
STRUCT  .in^string  (string^template);         ! Input string
STRUCT  .the^context (context^template);       ! continuation context
STRUCT  .server^banner (zspi^ddl^char50^def);
STRUCT  .my^ssid (zspi^ddL^ssid^def);
INT      my^version  := version;

DEFINE   req^buffer = b1#;   ! Request buffer (same as b1)
```

Example E-18: Declarations for C Requesters and Servers

This C code contains common declarations used by the C requester and server example programs.

Source File

SECRH

Example E-18. C File: SECRH Supporting Code

```
/*   File name: secrh
**   SPI EXAMPLE C Requester Declarations and variables.
*/

typedef struct
{
  short    msg_code,
           param_count;
  char     param [1024];
} param_msg;

typedef struct
{
  short   len;
  char    data [200];
} string_template;

typedef struct
{
  short           len;
  short           command;
  short           index;        /* Current index into the string */
  string_template con_string;   /* The converted string, so far */
} context_template;

startup_msg_type    startup_msg;
string_template     in_string;        /* Input string */
context_template    the_context;      /* continuation context */
zspi_ddl_char50_def server_banner;
zspi_ddl_ssid_def   my_ssid = {{ZSPI_VAL_TANDEM},
                               ZSPI_SSN_NULL, version};
short my_version  = version;

#define  req_buffer   b1     /* Request buffer (same as b1) */
```

Example E-19: Routines for TAL Requesters and Servers

This TAL code contains common routines used by the TAL requester and server example programs.

Source File

SETRUTIL

Example E-19. TAL File: SETRUTIL Supporting Code (page 1 of 4)

```
--   File name: SETRUTIL
--   SPI EXAMPLE TAL Requester Utility procedures.
--
?PAGE "report^newprocess^error"
!===========================================================================!
! Proc      : report^newprocess^error                                       !
! Function  : This procedure will format a NEWPROCESS error and write it   !
!             to the home term.                                             !
!===========================================================================!

PROC report^newprocess^error (p^program^fname, p^error);
INT .p^program^fname,
    p^error;
BEGIN
  INT    l^len;
  STRING l^err^sbuf [0:79];

  SUBPROC format^file^error;
  BEGIN
    sp ':=' " (ERROR " -> @sp;
    CALL numout (sp, p^error.<8:15>, 10, 3);
    sp [3] ':=' ")" -> @sp;
  END;  -- of SUBPROC

  l^err^sbuf ':=' " " & l^err^sbuf [0] FOR $OCCURS(l^err^sbuf) - 1;
  l^err^sbuf ':=' "NEWPROCESS ERROR #" -> @sp;
  CALL numout (sp, p^error.<0:7>, 10, 2);
  sp[2] ':=' "," -> @sp;
  CALL numout (sp, p^error.<8:15>, 10, 3);
  @sp := @sp[3];
  CALL writex (term, l^err^sbuf, @sp '-' @l^err^sbuf);
  IF <> THEN CALL get^file^error (term);

  CASE p^error.<0:7> OF
  BEGIN
    0 -> l^err^sbuf ':=' "No error" -> @sp;
    1 -> l^err^sbuf ':=' "Undefined Externals" -> @sp;
    2 -> l^err^sbuf ':=' "No PCB Available" -> @sp;
    3 -> l^err^sbuf ':=' "File System Error on Program File " -> @sp;
         l^len := fnamecollapse (p^program^fname, sp);
         @sp := @sp[l^len];
         CALL format^file^error;
    4 -> l^err^sbuf ':=' "Unable to Allocate Map" -> @sp;
    5 -> l^err^sbuf ':=' "File System Error on Swap File " -> @sp;
         CALL format^file^error;
    6 -> l^err^sbuf ':=' "Illegal File Format for " -> @sp;
         l^len := fnamecollapse (p^program^fname, sp);
         @sp := @sp[l^len];
```

Example E-19. TAL File: SETRUTIL Supporting Code (page 2 of 4)

```
    7 -> l^err^sbuf ':=' "Unlicensed PRIV program " -> @sp;
         l^len := fnamecollapse (p^program^fname, sp);
         @sp := @sp[l^len];
    8 -> l^err^sbuf ':=' "Process Name Error " -> @sp;
         CALL format^file^error;
    9 -> l^err^sbuf ':=' "Library Conflict" -> @sp;
   10 -> l^err^sbuf ':=' "Unable to communicate with System Monitor" -> @sp;
   11 -> l^err^sbuf ':=' "File System Error on Library File " -> @sp;
         l^len := fnamecollapse (p^program^fname[12], sp);
         @sp := @sp[l^len];
         CALL format^file^error;
   12 -> l^err^sbuf ':=' "Program and Library Files are the Same" -> @sp;
   13 -> l^err^sbuf ':=' "Invalid Segment Size" -> @sp;
   14 -> l^err^sbuf ':=' "File System Error on Initial Setup of Swap " &
                         "File " -> @sp;
         CALL format^file^error;
   15 -> l^err^sbuf ':=' "Illegal Home Terminal " -> @sp;
         CALL format^file^error;
   16 -> l^err^sbuf ':=' "I/O Error on Home Terminal " -> @sp;
         CALL format^file^error;
   17 -> l^err^sbuf ':=' "DEFINE context propagation error" -> @sp;
   18 -> l^err^sbuf ':=' "OBJECT file with an illegal process Device " &
                         "subtype" -> @sp;
   19 -> l^err^sbuf ':=' "process device subtype specified in Backup " &
                         "Process not the same as that in primary " &
                         "process " -> @sp;
   OTHERWISE -> l^err^sbuf ':=' "Unknown Error " -> @sp;
  END; -- of CASE

  CALL writex (term, l^err^sbuf, @sp '-' @l^err^sbuf);
  IF <> THEN CALL get^file^error (term);

END; -- of PROC report^newprocess^error
? PAGE "restart^server"
!============================================================================!
! Proc      : restart^server                                                 !
! Function  : This procedure will start and re-start the server up to a      !
!             maximum number to times.                                       !
!============================================================================!

PROC restart^server;
BEGIN
  STRING l^err^msg^1 [0:30] := "Server restart retries exceeded",
         l^err^msg^2 [0:15] := "Server restarted";
  INT    l^process^flags := 0,
         l^priority := 0;          ! Flag to start up server in INSPECT

  IF srvr^retry^count > 0 THEN
  BEGIN
    CALL writex (term, l^err^msg^2, $OCCURS (l^err^msg^2));
    IF <> THEN CALL get^file^error (term);
  END;
  srvr^retry^count := srvr^retry^count + 1;
  IF srvr^retry^count > 3 THEN
  BEGIN
    CALL writex (term, l^err^msg^1, $OCCURS (l^err^msg^1));
    IF <> THEN CALL get^file^error (term);
    CALL stop;
  END;
```

Example E-19. TAL File: SETRUTIL Supporting Code (page 3 of 4)

```
  !
  ! If debug^flag is set, then bring up the server in DEBUG.
  !
  l^process^flags.<15> := debug^flag;
  l^priority.<0> := debug^flag;
  process^id ':=' "   " & process^id [0] FOR $OCCURS (process^id) - 1;
  CALL newprocess (server^name, l^priority, !memory pages!, !processor!,
                   process^id, file^error, process^name, !home term!,
                   l^process^flags);
  IF file^error THEN
  BEGIN
    CALL report^newprocess^error (server^name, file^error);
    RETURN;
  END;

  CALL open^server;

END; ! -- of PROC restart^server;
?PAGE "write^read^server"
!============================================================================!
! Proc      : write^read^server                                              !
! Function  : This procedure will write a message to the server and reason   !
!             the reply. It handles any file errors on the server's file.    !
!============================================================================!

PROC write^read^server;
BEGIN
  INT
           l^await^done,
           l^op^done,
           l^recoverable^err;

  l^op^done := false;
  DO
  BEGIN
    CALL writereadx (srvr^file^num, req^buffer, max^bufsize,
                     $OCCURS (req^buffer), read^count);
    IF <> THEN CALL get^file^error (srvr^file^num);

    l^await^done := false;
    DO
    BEGIN
      file^num := -1;              ! Don't Cancel

      CALL awaitiox (file^num, !buffer!, read^count, !tag!, time^to^wait);
      IF < THEN
      BEGIN
        CALL fileinfo (file^num, file^error);
        IF file^error = 40 THEN    ! TIMEOUT Error
        BEGIN
          sline ':=' "Waiting for the Server" -> @sp;
          CALL writex (term, sline, @sp '-' @sline);
          IF <> THEN CALL get^file^error (term);
        END ELSE ! IF file^error = 40 ELSE
```

Example E-19. TAL File: SETRUTIL Supporting Code (page 4 of 4)

```
        BEGIN
          l^await^done := true;
          server^up := false;
          l^recoverable^err := false;       ! Set default value
          IF file^error = 201 THEN l^recoverable^err := true;
          IF file^error = 211 THEN l^recoverable^err := true;
          IF file^error = 6 THEN            ! System message
          BEGIN
            start^buffer ':=' req^buffer FOR read^count BYTES;
            IF start^buffer.msgcode = -5 THEN  ! STOP message
              l^recoverable^err := true;
            IF start^buffer.msgcode = -6 THEN  ! ABEND message
              l^recoverable^err := true;
          END; -- of IF file^error = 6

          sline ':=' "File system error (" -> @sp;
          CALL numout (sp, file^error, 10, 3);
          sp [3] ':=' ") on WRITEREAD to the SERVER" -> @sp;
          CALL writex (term, sline, @sp '-' @sline);
          IF <> THEN CALL get^file^error (term);

          IF l^recoverable^err THEN
          BEGIN
            DO
            BEGIN
              CALL restart^server;
            END
            UNTIL server^up := true;
            CALL cancel (srvr^file^num); ! Cancel the IO
          END ELSE ! IF l^recoverable^err ELSE
          BEGIN
            CALL debug;
          END; -- of IF l^recoverable^err
        END; -- of IF file^error = 40

      END ELSE ! IF < ELSE
      BEGIN
        l^op^done := true;
        l^await^done := true;
      END; -- of IF <

    END
    UNTIL l^await^done = true;

  END
  UNTIL l^op^done = true;

END; ! -- of PROC write^read^server;
```

Example E-20: Routines for C Requesters and Servers

This C code contains common routines used by the C requester and server example programs.

Source File

SECRUTLC

Example E-20. C File: SECRUTLC Supporting Code (page 1 of 4)

```
/*   File name: secrutlc
 *   SPI EXAMPLE C Requester Utility procedures.
 */
#pragma PAGE "report_newprocess_error"
/*
 *=============================================================================
 * Proc      : report_newprocess_error                                        =
 * Function  : This procedure will format a NEWPROCESS error and write       =
 *             it to the home term.                                           =
 *=============================================================================
 */
void report_newprocess_error (short* p_program_fname, short p_error)
{
  short   l_len;
  char    l_err_sbuf [80];

  printf ("NEWPROCESS ERROR # %d, %d\n",
         ((p_error & 0xFF00) >> 8) /* Bits 0-7 */,
          (p_error & 0x00FF) /* Bits 8-15 */);

  switch (((p_error & 0xFF00) >> 8) /* Bits 0-7 */)
  {
    case  0 : printf ("No error");
              break;
    case  1 : printf ("Undefined Externals");
              break;
    case  2 : printf ("No PCB Available");
              break;
    case  3 : printf ("File System Error on Program File ");
              l_len = FNAMECOLLAPSE (p_program_fname, l_err_sbuf);
              l_err_sbuf [l_len] = 0;
              printf ("%s", l_err_sbuf);
              printf (" (ERROR %d)", (p_error & 0x00FF) /* Bits 8-15 */);
              break;
    case  4 : printf ("Unable to Allocate Map");
              break;
    case  5 : printf ("File System Error on Swap File ");
              printf (" (ERROR %d)", (p_error & 0x00FF) /* Bits 8-15 */);
              break;
    case  6 : printf ("Illegal File Format for ");
              l_len = FNAMECOLLAPSE (p_program_fname, l_err_sbuf);
              l_err_sbuf [l_len] = 0;
              printf ("%s", l_err_sbuf);
              break;
    case  7 : printf ("Unlicensed PRIV program ");
              l_len = FNAMECOLLAPSE (p_program_fname, l_err_sbuf);
              l_err_sbuf [l_len] = 0;
              printf ("%s", l_err_sbuf);
              break;
    case  8 : printf ("Process Name Error ");
              printf (" (ERROR %d)", (p_error & 0x00FF) /* Bits 8-15 */);
              break;
    case  9 : printf ("Library Conflict");
              break;
    case 10 : printf ("Unable to communicate with System Monitor");
              break;
```

Example E-20. C File: SECRUTLC Supporting Code (page 2 of 4)

```
    case 11 : printf ("File System Error on Library File ");
              l_len = FNAMECOLLAPSE (&p_program_fname[12], l_err_sbuf);
              l_err_sbuf [l_len] = 0;
              printf ("%s", l_err_sbuf);
              printf (" (ERROR %d)", (p_error & 0x00FF) /* Bits 8-15 */);
              break;
    case 12 : printf ("Program and Library Files are the Same");
              break;
    case 13 : printf ("Invalid Segment Size");
              break;
    case 14 : printf ("File System Error on Initial Setup of Swap File ");
              printf (" (ERROR %d)", (p_error & 0x00FF) /* Bits 8-15 */);
              break;
    case 15 : printf ("Illegal Home Terminal ");
              printf (" (ERROR %d)", (p_error & 0x00FF) /* Bits 8-15 */);
              break;
    case 16 : printf ("I/O Error on Home Terminal ");
              printf (" (ERROR %d)", (p_error & 0x00FF) /* Bits 8-15 */);
              break;
    case 17 : printf ("DEFINE context propagation error");
              break;
    case 18 : printf ("OBJECT file with an illegal process Device subtype");
              break;
    case 19 : printf ("process device subtype specified in Backup ");
              printf ("Process not the same as that in primary process ");
              break;
    default : printf ("Unknown Error ");
              break;
  } /* of switch */
  printf ("\n");
}  /* of report_newprocess_error() */

#pragma  PAGE "restart_server"
/*
 *=========================================================================
 * Proc      : restart_server                                              =
 * Function  : This procedure will start and re-start the server up to     =
 *             a maximum number to times.                                  =
 *=========================================================================
 */
void restart_server(void)
{
  short  l_process_flags = 0,
         l_priority = 0;          /* Flag to start up server in INSPECT */

  if (srvr_retry_count > 0)
  {
    printf ("Server restarted\n");
  }
  srvr_retry_count = srvr_retry_count + 1;
  if (srvr_retry_count > 3)
  {
    printf ("Server restart retries exceeded\n");
    STOP();
  }
```

Example E-20. C File: SECRUTLC Supporting Code (page 3 of 4)

```
  /*
   * If debug_flag is set, then bring up the server in DEBUG.
   */
  if (debug_flag)
  {
    l_process_flags |= 0x0001; /* Turn debug on */
    l_priority |= 0x8000;      /* Turn debug on */
  } else
  {
      l_process_flags &= 0xFFFE; /* Turn debug off */
    l_priority &= 0x7FFF;      /* Turn debug off */
  }
  strcpy ((char *) &process_id[0],   "                          ");
  NEWPROCESS (server_name, l_priority,
              /*memory pages*/, /*processor*/,
              process_id, &file_error,
              process_name, /*home_term*/,
              l_process_flags);
  if (file_error)
  {
    report_newprocess_error (server_name, file_error);
    return;
  }

  open_server();

}  /* of restart_server() */

#pragma PAGE "write_read_server"
/*
 *=============================================================================
 * Proc      : write_read_server                                               =
 * Function  : This procedure will write a message to the server and read =
 *             the reply. It handles any file errors on the server's file.=
 *=============================================================================
 */
void write_read_server(void)
{
  short
           l_await_done,
           l_op_done,
           l_recoverable_err,
           l_status;

  l_op_done = false;
  do
  {
    l_status = WRITEREADX (srvr_file_num, (char *) &req_buffer[0],
                           max_bufsize, max_bufsize, &read_count);
    if (l_status != CCE) get_file_error (srvr_file_num);

    l_await_done = false;
```

Example E-20. C File: SECRUTLC Supporting Code (page 4 of 4)

```
  do
  {
      file_num = -1;              /* Don't Cancel */

      l_status = AWAITIOX (&file_num, /*buffer*/, &read_count, /*tag*/,
                           time_to_wait);
      if (l_status == CCL)
      {
        FILEINFO (file_num, &file_error);
        if (file_error == 40)    /* TIMEOUT Error */
        {
          printf ("Waiting for the Server\n");
        } else /* if (file_error == 40) else */
        {
          l_await_done = true;
          server_up = false;
          l_recoverable_err = false;       /* Set default value */
          if (file_error == 201) l_recoverable_err = true;
          if (file_error == 211) l_recoverable_err = true;
          if (file_error == 6)               /* System message */
          {
            memcpy (&startup_msg, &req_buffer, read_count);
            if (startup_msg.msg_code == -5)  /* STOP message  */
              l_recoverable_err = true;
            if (startup_msg.msg_code == -6)  /* ABEND message */
              l_recoverable_err = true;
          } /* of if (file_error == 6) */
          printf ("File system error (%d) on WRITEREAD to the SERVER\n",
                   file_error);

          if (l_recoverable_err)
          {
            do
            {
               restart_server();
            }
            while (server_up == false);

            CANCEL (srvr_file_num); /* Cancel the IO */
          } else /* if (l_recoverable_err) else */
          {
            DEBUG();
          } /* of if (l_recoverable_err) */
        } /* of if (file_error == 40) */

      } else /* if (l_status == CCL) else */
      {
        l_op_done = true;
        l_await_done = true;
      } /* of if (l_status == CCL) */

    }
    while (l_await_done == false);

  }
  while (l_op_done == false);

}  /* of PROC write_read_server; */
```

Example E-21: TAL Examples Compiler

The TACL command file in Example E-21 compiles the TAL example programs. Modify the two file assignments to specify the location of the SPI definition files on your node. ($SYSTEM.SPIDEF is the default.)

Source File

SETBUILD

Example E-21. TACL Command File to Compile TAL Program Examples

```
comment This TACL obey file will compile all of the SPI Example (TAL) files.
comment Change these assigns to match your system's configuration.

ASSIGN ZSPITAL, $SYSTEM.SPIDEF.ZSPITAL
ASSIGN ZCOMTAL, $SYSTEM.SPIDEF.ZCOMTAL

TAL /in SET0204, out $s.#hold/ SET0204O
TAL /in SET0205, out $s.#hold/ SET0205O
TAL /in SET0206, out $s.#hold/ SET0206O
TAL /in SET0207, out $s.#hold/ SET0207O
TAL /in SETREQR, out $s.#hold/ SETREQRO
TAL /in SETSERV, out $s.#hold/ SETSERVO
```

Example E-22: C Examples Compiler

The TACL command file in Example E-22 compiles the C example programs. Modify the two file assignments to specify the location of the SPI definition files on your node. ($SYSTEM.SPIDEF is the default.)

Source File

SECBUILD

Example E-22. TACL Command File to Compile C Program Examples

```
comment This TACL obey file will compile all of the SPI Example (C) files.
comment Change these SSVs to match your system's configuration.

#push myParms

#set myParms  runnable, &
 ssv0 "[#defaults]",&
 ssv1 "$DSV.ZSPIDEF"

C /in SEC0204C, out $s.#SEC0204/ SEC0204O; [myParms]
C /in SEC0205C, out $s.#SEC0205/ SEC0205O; [myParms]
C /in SEC0206C, out $s.#SEC0206/ SEC0206O; [myParms]
C /in SEC0207C, out $s.#SEC0207/ SEC0207O; [myParms]
C /in SECREQRC, out $s.#SECREQR/ SECREQRO; [myParms]
C /in SECSERVC, out $s.#SECSERV/ SECSERVO; [myParms]

#pop myParms
```

Glossary

attribute. A characteristic of an object. For example, two attributes of a DNS alias are an alias type and domain. Two attributes of a communications line might be its baud rate and its retry count. In SPI messages, an attribute of an object is usually expressed as a simple token or a field within an extensible structured token. Tokens themselves have attributes: length, count, address, and offset. Programs can retrieve these through special SSGET operations.

buffer. A block of memory where data that is being moved from one location to another is stored temporarily. For instance, data to be sent in an interprocess message is encoded in a buffer. The file system copies the contents of this buffer to another buffer within the memory space of the recipient process. See also message buffer and SPI buffer.

built-in. A TACL primitive function or variable. Names of built-ins begin with a pound sign (#). TACL provides the #SSGET, #SSGETV, #SSINIT, #SSMOVE, #SSNULL, #SSPUT, and #SSPUTV built-ins for working with SPI messages.

command. A request 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 message. An SPI message, containing a command number and related tokens, that a requester sends to a subsystem manager process. See SPI message.

command number. A number that represents a particular command to a subsystem. Each subsystem with a token-oriented programmatic interface can have its own set of command numbers, which are represented in DDL by constants with names of the form *subsys*-CMD-*name*, and in programs by TAL literals or defines, COBOL level-01 variables, or TACL text variables. The command number is stored in the SPI message header.

compatibility. The ability of two or more elements in a system to work together correctly. See also version compatibility.

constant (DDL). A DDL declaration that associates a name with a number or string. A constant defined in DDL becomes a literal or define in TAL, a level-01 variable in COBOL, and a text variable in TACL. In the definition files supplied by HP for the NonStop server, constants are used for command numbers, object-type numbers, event numbers, error numbers, subsystem numbers, token data-type numbers, token numbers, and other values.

context information. The information required by a subsystem to continue processing a command for which a partial response was returned. Continuation of a response in multiple response messages requires the subsystem to send context information to the requester. The application program, in turn, must return that information to the

subsystem in a new command message so that the subsystem can continue processing. See context token.

context token. A token that indicates (by its presence or absence) whether or not a subsystem has more objects to process or more response messages to return. If this token is present in a response message, the response is incomplete and can be continued in another response message. To obtain the next message, the requester reissues the original command with the context token. When the server returns a message that does not contain a context token, the requester knows that the response is complete.

context-free server. A server that does not retain any information about command processing after returning a response. A context-free server stores any information it needs to continue processing an incomplete command in the context token, which it returns to the requester. The requester must resend the command with the context token for the server to continue processing. A context-free server allows the requester to interrupt or abandon the continuation of a series of response messages.

context-sensitive server. A server that retains information about previous processing. For instance, in performing a command on a list of objects, a context-sensitive subsystem might retain, between response messages, the name of the object it last processed. Context-sensitive servers limit or complicate the requester's ability to interrupt or abandon the continuation of a series of response messages.

continuation. A method of completing a response in multiple response messages. The subsystem uses a context token to indicate that the response is continued to another message. Each response message can contain multiple response records.

control and inquiry. Those aspects of 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). Compare with event management.

control operation. An action that affects the recording, processing, transmission, or interpretation of data. In SPI, an operation that modifies the contents of an SPI buffer not by adding a token, but by performing a housekeeping function, for instance, clearing the last SPI error number or flushing the buffer from the current position. A positioning operation is one kind of control operation.

current token. The token in the current position. See current-token position.

current-token position. The location in the SPI buffer of the token whose token code, token value, or attribute has just been retrieved. Compare with next-token position and initial position.

data list. A grouping of tokens used to delimit response records in an SPI response. A data list begins with ZSPI-TKN-DATALIST and ends with ZSPI-TKN-ENDLIST. See response record.

definition files. A set of files containing data declarations for items related to SPI messages and their processing. The core definitions required to use SPI are provided in a DDL file and in several language-specific definition files, one for each programming language that supports SPI. The DDL compiler generates the language-specific files from the DDL file. Subsystems that support SPI provide additional definition files containing subsystem-specific definitions. See also SPI standard definitions and subsystem definitions.

Distributed Systems Management. An architecture and a set of software tools that facilitate management of systems and networks. These tools include the ViewPoint console application, the Subsystem Control Facility (SCF), the Subsystem Programmatic Interface (SPI), the Event Management System (EMS), the Distributed Name Service (DNS), and token-oriented programmatic interfaces to the manager processes of various NonStop Kernel subsystems.

downward compatibility. The ability of a requester to operate properly with a server of a lower revision level. In this case, the requester is downward-compatible with the server. Compare with upward compatibility.

DSM. Distributed Systems Management.

empty list or variable. A list that has no members, or a variable that has no content.

empty response. A response record containing only the return token with a value that means “empty response.” See return token.

EMS. Event Management System.

end-list token. ZSPI-TKN-ENDLIST, which marks the end of a list in an SPI message. This same token is used to mark the end of all four types of SPI lists. Compare with list token.

enumerated type. A 16-bit signed data type with a value drawn from a specific list of values with designated meanings. The enumerated type is one of the standard token data types defined by SPI. The list of acceptable values and their meanings varies depending on the token number and is defined by the subsystem.

error. A condition that causes a command or other operation to fail. Compare with warning.

error list. A grouping of tokens used within a response record to provide error and warning information. An error list begins with ZSPI-TKN-ERRLIST, ends with ZSPI-TKN-ENDLIST, and contains an error token and other tokens explaining the error. Error lists can be nested within other error lists. The return token cannot be included in an error list. See return token.

error number. 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, the SPI extensions define many more, and still others are defined by subsystems.

error token. A response token, ZSPI-TKN-ERROR, that identifies an error that occurred during command processing. Every error list contains an error token. Its value is a structure consisting of the subsystem ID and an error number identifying the error. See error list, error number, and return token.

event. A significant change in some condition in the system or network. Events can be operational errors, notifications of limits exceeded, requests for action needed, and so on.

event management. The reporting and logging of important events that occur in a system or network, the distribution and retrieval of information concerning those events, and the actions taken by operations personnel or software in response to the events. Compare with control and inquiry.

Event Management System. A software facility that provides event-message collection, logging, and distribution facilities for the operating system (implemented as a major enhancement of the operator process, $0). It provides for different descriptions of events for people and for programs, lets an operator or application select conveniently from event-message data, and allows for flexible distribution of event messages within a system or network. It has programmatic interfaces based on SPI for both event reporting and event retrieval. See event message.

event message. A special kind of SPI message that describes an event occurring in the system or network.

extensible structured token. A token with a value that can be extended by appending new fields in later releases. Such structures are typically used to indicate the attributes of an object being operated on, and to return status and statistics information in responses. The token is accessed through reference to a token map that contains field version and null-value information that allows SPI to provide compatibility between different versions of the structure. Compare with simple token; see also structure, structured token, and token map.

flush. With reference to an SPI buffer, to erase all tokens, including tokens in lists, starting at the current position and continuing to the end of the buffer. To flush a buffer, call SSPUT with the token ZSPI-TKN-DATA-FLUSH.

generic list. The most general list construct supported by SPI. It starts with ZSPI-TKN-LIST and ends with ZSPI-TKN-ENDLIST. Generic lists can be nested. No NonStop Kernel subsystem provided by HP uses generic lists, but subsystems you write can do so. Compare with data list and error list.

GETVERSION command. An information command that retrieves the server version of the subsystem server, and possibly additional version information about objects defined by the subsystem. Every subsystem with an SPI command/response interface supports the GETVERSION command.

header. See SPI message header.

header token. A special token, present in every SPI message, that provides information about the message as a whole. The header tokens are typically items common to all or most messages of a specific kind. Header tokens differ from other tokens in several ways: they exist in the buffer at initialization and their values are usually set by SSINIT, they can occur only once in a buffer, they are never enclosed in a list, they cannot be moved to another buffer with SSMOVE, and programs cannot position to them or retrieve their values using the NEXTCODE or NEXTTOKEN operation. Programs retrieve the values of header tokens by passing appropriate token codes to SSGET, and can change the values of some header tokens by passing their token codes to SSPUT. Examples of header tokens for commands are the command, the object type, the maximum-response token, the server-version token, the maximum-field-version token, and the checksum token. Examples of header tokens for event messages are the event number, the event generation time, the logging time, the maximum-field-version token, and the checksum token.

header type. A header token in an SPI message that indicates whether the message is a command or response message or an event message.

information command. A command that retrieves information about an object but does not act on the object or change it in any way. Extended SPI classifies informational commands as nonsensitive for security purposes. Compare with command.

initial position. In an SPI buffer, the location just prior to the first token that is not a header token. Compare with current-token position and next-token position.

initialize. To prepare a data structure to have values assigned to it. For example, the SPI SSINIT procedure initializes the buffer by building the message header; the SSNULL procedure initializes an extensible structured token by assigning null values to the fields of the structure.

is-present field. A field in a structure that indicates whether the value in a related field was supplied by the program that sent the structure. In most cases, a field to which the program made no assignment has the null value that was set by SSNULL. A subsystem defines a separate is-present field when no null value can be defined that is not a valid value for the field.

list. In an SPI message, a group of tokens that defines a context for scanning the buffer and extracting tokens with the SSGET procedure. A list imposes a hierarchy on the buffer. To retrieve tokens from a list, a program must first enter the list by retrieving the initial list token, then retrieve tokens from the list, and then exit the list to the next higher level of tokens by retrieving the end-list token. SPI defines four kinds of lists: data lists, error lists, segment lists, and generic lists.

list token. A token that marks the beginning of a list in an SPI message. The four list tokens, each marking one of the four types of SPI lists are: ZSPI-TKN-DATALIST for data lists, ZSPI-TKN-ERRLIST for error lists, ZSPI-TKN-SEGLIST for segment lists, and ZSPI-TKN-LIST for generic lists. Compare with end-list token; see also syntax token.

management application. A program that manages a subsystem and its objects by issuing commands, retrieving event messages, or both. A management application is a requester with respect to the subsystem server to which it sends commands.

management interface. An interactive or programmatic interface through which one can manage a subsystem and its objects. In some subsystems, a specific process is dedicated to the management interface; in other subsystems, the process that provides the management interface also performs other functions.

manager process. A subsystem process that supports the SPI command interface for that subsystem. Management applications send commands to and receive responses from subsystem manager processes. The Expand manager process, $ZNET, and the X25AM line handlers are examples of manager processes.

maximum field version. In an SPI message, the latest version associated with any non-null field of any extensible structured token in the message. The maximum field version of the SPI message is contained in a header token. It corresponds to the version of the oldest server or requester that can successfully process the message.

message. A block of information, usually in the form of a structure, that is sent from one process to another. See also SPI message.

message buffer. A block of memory used for temporary storage of an interprocess message. See also SPI buffer.

message code. The contents of the first word of an interprocess message. A message code of -28 identifies the message as an SPI message.

nested error list. An error list contained in another error list. When an error in one subsystem or in a library procedure prevents another subsystem from performing a command, the calling subsystem reports this pass-through error by nesting it within its own error list. For instance, a response from FUP might include an error list explaining the FUP error, which in turn contains an error list explaining the SORT error that caused the FUP error. See also pass-through error.

next-token position. The location where the next SSGET operation will take place. Compare with current-token position and initial position.

nonsensitive command. A command that has no affect on the state of an object. Nonsensitive commands can be issued by any user or program that is allowed access to the subsystem. Most informational commands are nonsensitive. Compare with sensitive command.

null value. A value in a field of an extensible structure indicating that no value was assigned by the sending process. Null values are initialized by the SSNULL procedure.

object. In SPI, an entity subject to independent reference and control by a subsystem: for example, the disk volume $DATA or the data communications line $X2502. An object

typically has a name and a type known to the controlling subsystem. In DDL, an item in a dictionary. DDL assigns each object a unique object number for identification.

object type. A category of objects to which a specific object belongs: for example, a specific disk file might have the object type FILE and a specific terminal the object type SU. A subsystem identifies a set of object types for the objects it manages. The operator interface to the subsystem might have keywords to identify the types. Correspondingly, the programmatic interface would have object-type numbers suitable for passing to the SPI SSINIT procedure. In DDL, one of the six types of dictionary objects: records, DEFs, constants, token types, token codes, and token maps.

object-name template. An object name, provided in a command, which the subsystem compares with the names of its objects to identify those to which the command should be applied. Some subsystems allow wild-card characters in object-name templates.

object-name token. A parameter or response token that identifies, by name, a particular object of a given object type. An object-name token is a kind of object-selector token. See object-selector token.

object-selector token. A token that identifies one or more specific objects to operate on, of the object type given in the command. Typically, the value of such a token is either some form of object name or an object number. An object-name token is a kind of object-selector token. See object-name token.

object-type number. A number that represents an object type managed by a subsystem. Each subsystem with a token-oriented programmatic interface can have its own set of object-type numbers, which are represented in DDL by constants and in programs by TAL literals or defines, COBOL level-01 variables, or TACL text variables. (In some cases, object-type numbers are shared by several subsystems.) The object-type number is a header token in commands and responses. See object type.

owner. 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 FUP GIVE command. 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. In the case of a token or other definition, the subsystem that provided the definition. In the case of a subsystem, the company or organization that provides the subsystem, or the eight-character string identifying that company.

parameter token. In control and inquiry, a token that provides parameter information for a command. Most tokens in the SPI message for a command are parameter tokens; depending on the subsystem, they can include attribute tokens, object-selector or object-name tokens, and subsystem-control tokens. Compare with syntax token. In event management, a token representing a parameter passed by an application to an event-message filter; such tokens are kept in a parameter buffer. For further information, see the *EMS Manual*.

pass-through error. An error originally reported by one subsystem or system component but included in a response record produced by another subsystem. Typically, a subsystem passes an error from a second subsystem only if that error prevented the first subsystem from performing a command successfully. A pass-through error is expressed as an error list that is nested within another error list. See nested error list.

position descriptor. A four-word block of information that indicates a position within an SPI buffer. A position descriptor is used as a parameter to some of the special operations for SSGET and SSPUT.

positioning operation. An operation that gets, sets, or changes the current position in an SPI buffer, or that creates in the buffer a construct (like a list) that provides a scope for retrieval of data.

predefined value. A commonly used value that is given a symbolic name in an SPI definition file.

private token type. A token type defined by, and specific to, a particular subsystem. A private token type is built from standard SPI token data types, although it might have additional semantic connotations for the subsystem. For example, a subsystem might define a token type that looks to SPI like an integer but that implies to the subsystem a range of values smaller than an integer type would allow. See token type.

procedural interface. A means for obtaining services through procedure calls; also, the set of procedures through which services are obtained. For instance, an application has a procedural interface to SPI; that interface consists of the procedures SSINIT, SSNULL, SSPUT, SSPUTTKN, SSGET, SSGETTKN, SSMOVE, and SSMOVETKN.

programmatic command. A command issued by a program, rather than by a human operator.

programmatic interface. A means for a program to communicate with another program. On an HP NonStop system, a programmatic interface typically includes: a message format, a set of message formats, or a set of procedures (such as the SPI procedures) to build and decode messages; definitions of message elements (commands, data types, objects, parameters, response data, errors, and so on); rules for communication between the requester and the server; and software to receive and respond to messages defined for the interface.

programmed operator. A management application that performs functions that might otherwise be performed by a human operator.

qualified token code. A token code that includes a subsystem ID.

requester. An SPI requester. A management application that sends SPI commands to a subsystem server.

requester version. The software revision level of the definition files used in the compilation of a requester. Each subsystem has its own definitions, so the requester version can differ in requests to different subsystems.

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 (response messages) from a subsystem to an application.

response message. An SPI message that is sent from a subsystem to an application program, in response to a command message. See SPI message and command message.

response record. A set of response tokens, usually describing the results of performing a command on one object. A response can consist of multiple response records, spread across one or more response messages. If there are multiple response records in a response message, each response record is enclosed in a data list. See data list. Each response record is required to contain a return token; see return token.

response segment. A data list containing part of a segmented response record. See "segmented response."

response token. A token returned as an element of a response. Response tokens include information tokens (which contain response data of interest to the application), syntax tokens (such as list tokens), one special response-control token (the context token), the return token, and error tokens.

response-control token. A parameter token or response token that influences or reflects how a subsystem packages its response to a command. Response-control tokens are defined by SPI, rather than by subsystems; they include the maximum-response token, the response-type token, and the context token.

return token. The response token that indicates whether a command was successful and why it failed if it did. Every response record in a response from a NonStop Kernel subsystem contains a return token; a response record can also contain error lists that include error tokens. The token code for the return token is ZSPI-TKN-RETCODE. Its value consists of a single integer field. Compare with error token.

SCF. Subsystem Control Facility.

SCP. Subsystem Control Point.

segment list. An SPI list used in segmented responses to group the repeating tokens in the response record. A segment list starts with ZSPI-TKN-SEGLIST and ends with ZSPI-TKN-ENDLIST.

segmented response. A response in which a response record spans more than one response message. Segmented responses can be constructed when the response contains repeating groups of tokens. The response record contains the nonrepeating tokens followed by the repeating groups in segment lists. All but the last segment of

the response record contain ZSPI-TKN-MORE-DATA, indicating that the response record is incomplete.

sensitive command. A command that can be issued only by a restricted set of users, such as the super group, because the subsystem restricts access to the command. For extended SPI subsystems, the sensitive commands are those that can change the state or configuration of objects; for these subsystems, the sensitive commands are also action commands. Compare with nonsensitive command.

server. An SPI server. A subsystem process that accepts SPI commands from management applications.

server version. The software release version of the server to which a requester using SPI (such as a management application) is sending a command. If the server version is older than the maximum field version in a request, the server rejects the request. SPI puts the maximum field version into the command buffer; the server puts its own version into each response buffer. See maximum field version.

short read. An operation in which the application reads fewer bytes than are available in a message. In the context of SPI, the term implies that the number of bytes requested by the application is fewer than the number of used bytes in the SPI buffer, or that the application furnished a buffer too small to contain the response data produced by the subsystem.

simple token. A token consisting of a token code and a value that is either a single elementary field, such as an integer or a character string, or a fixed (nonextensible) structure. Compare with extensible structured token.

snapshot file. A file used by context-sensitive servers to record response data for a command. It is used to ensure a consistent response in cases where data—for instance, statistics—might change between one reply message and the next. NonStop Kernel subsystems provided by HP do not use snapshot files, but subsystems you write can do so.

special operation. An operation, such as a control operation or an operation that gets information from the buffer (rather than the header), performed by the SSGET or the SSPUT procedure. Special operations include obtaining the length or number of occurrences of a token, changing the current position, clearing the last-error information, or deleting a token from the buffer. A program directs SSGET or SSPUT to perform a special operation by passing to the procedure one of a set of special SPI token codes. These special token codes do not represent tokens in the buffer, but simply direct SSGET or SSPUT to perform the indicated operations.

SPI. Subsystem Programmatic Interface.

SPI buffer. A block of memory where SPI procedures create and manipulate SPI messages.

SPI definitions. See SPI standard definitions.

SPI error number. A number that indicates whether a call to an SPI procedure completed successfully and why it failed if it did. This number is returned in the status parameter on calls to the SPI procedures. The SPI error number does not reflect the success or failure of a command; it applies only to errors in the building and decoding of a message in an SPI buffer.

SPI message header. The initial part of an SPI message. The first word of this header always contains the value -28; the remainder of the header contains descriptive information about the SPI message, most of which is accessible as header tokens. The tokens in an SPI message header differ according to the header type: the header of a message that contains a command or response differs somewhat from the header of an event message. An application can use SSGET or EMSGET calls to retrieve the values of header tokens, and can use SSPUT calls to change the values of some; however, there are certain basic differences between header tokens and other tokens. See header token.

SPI message. A message specially formatted by the SPI procedures for communication between a management application and a subsystem, or between one subsystem and another. An SPI message consists of a collection of tokens. To retrieve a token from the message, the application passes a token code to SPI, which scans for the appropriate token and returns its value to the application. An SPI message is a single block of information sent at one time as one interprocess message. The two types of SPI messages are distinguished by SPI message header type: 1) command and response messages, and 2) event messages. See header type.

SPI procedures. The procedures used to build and decode SPI messages. These procedures are SSINIT, SSNULL, SSPUT, SSPUTTKN, SSGET, SSGETTKN, SSMOVE, SSMOVETKN, SSIDTOTEXT, and TEXTTOSSID. Corresponding TACL built-ins are also available.

SPI standard definitions. The set of declarations available for use with the SPI procedures regardless of the subsystem. Also a set of subsystem-specific declarations for each subsystem, and some sets of declarations that apply to multiple subsystems exist. An application using SPI needs the SPI standard definitions and also the subsystem definitions for all subsystems with which it communicates. See also definition files and subsystem definitions.

statistics token. A response token providing performance data about an object.

status token. A response token whose value indicates the status (current state) of an object.

structure. A data item with multiple fields, possibly of different types. This kind of data item corresponds to a DEF in DDL, to a STRUCT in TAL and TACL, and to a RECORD in COBOL.

structured token. A token whose value is a structure. Some structured tokens are simple tokens with fixed structures—for example, the error token, ZSPI-TKN-ERROR. Other

structured tokens are extensible structured tokens. See structure, simple token, and extensible structured token.

subject. In event management, a device, process, or other named entity about which a given event message is concerned.

subordinate names option. In extended SPI subsystems, the designation that the object name given in the command stands not just for itself, but for the names of all objects at the next-lower level in a 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 token.) When this option is present in a command, the subordinate names are implied even though they are not given explicitly.

subsystem. A program or set of processes that manages a cohesive set of objects. Each subsystem has a manager process (in some cases, this process is the entire subsystem) through which applications can request services by issuing commands defined by that subsystem. See manager process.

Subsystem Control Facility. The interactive interface for configuring, controlling, and collecting information from NonStop Kernel subsystems. The Subsystem Control Facility provides many of the same functions as CMI, CUP, and PUP, plus additional functions not available in CMI, CUP, or PUP.

Subsystem Control Point. An intermediate management process used by many NonStop Kernel subsystems. There can be several instances of this process. Applications send commands to an instance of this process, which in turn sends them on to the manager processes of the target subsystem. The Subsystem Control Point also processes a few commands itself. It provides security features, version compatibility, support for tracing, and support for applications implemented as process pairs.

subsystem definitions. The set of declarations available for use with a particular subsystem that supports a token-oriented programmatic interface. See also definition files and SPI standard definitions.

subsystem ID. A data structure that uniquely identifies a subsystem (including whether it is a NonStop Kernel subsystem or a subsystem you write). It consists of the name of the owner of the subsystem (the company that provides it), a subsystem number that denotes the subsystem within the scope of its owner, and a subsystem version number. The subsystem ID is an argument to most of the SPI procedures.

subsystem number. An integer that identifies a subsystem within the context of its owner. The subsystem owner, the subsystem number, and the subsystem version number make up the subsystem ID that uniquely identifies a subsystem.

subsystem owner. A value identifying the company that supplies a particular subsystem. It consists of a name of up to eight characters, blank-filled on the right. For example, the owner for all subsystems supplied by HP is “TANDEM~~bb~~”. The subsystem owner, the subsystem number, and the subsystem version number make up the subsystem ID that uniquely identifies the 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 (as described under SPI message); definition files in TAL, COBOL, and TACL for inclusion in programs, macros, and routines using the interface procedures; and definition files in DDL for programmers writing their own subsystems.

subsystem version number. A 16-bit integer representing the software release version of a subsystem. The subsystem version number is a field of the subsystem ID. If its value is null (zero), the subsystem ID refers to any and all versions of the subsystem. See version number.

subsystem-control token. A parameter token that influences how a subsystem performs a command. For instance, in the START PATHWAY programmatic command, the parameter ZPWY-TKN-DEF-PATHWAY is a subsystem-control token, because it determines whether a cold start or a cool start is performed. Likewise, the standard SPI token ZSPI-TKN-ALLOW-TYPE, supported by some subsystems, is a subsystem-control token; it determines under what conditions the subsystems will continue command processing on the next object in a sequence if errors or warnings occur. Compare with response-control token.

summary state. One of the generally defined possible conditions of an object with respect to the management of that object. A summary state differs from a state in two ways. First, a summary state pertains to the management of an object, whereas a state can convey other kinds of information about the object. Second, the set of summary states is a common list defined the same way for all extended SPI subsystems, whereas the set of possible states differs from subsystem to subsystem. The SPI extensions define a number of summary states, including STARTED, STOPPED, SUSPENDED, ABORTING, and DEFINED.

syntax token. A token whose function is not to provide information for a command or response, but to bracket or group other tokens; its use is analogous to that of a punctuation symbol. The tokens that begin and end lists (the list tokens) are syntax tokens. Compare with parameter token.

token. In SPI, a distinguishable unit in an SPI message. Programs place tokens in an SPI buffer using the SSPUT procedure (except for header tokens, which are a special case), and retrieve them from the buffer with the SSGET procedure. A token has two parts: an identifying code—a token code—and a token value. For command and response messages, a token normally represents a parameter to a command, an item of information in a response, or control information for the subsystem. For event messages, a token normally represents an item of information about an event or about the event message itself. In TACL, an entity recognized by the #ARGUMENT built-in function when parsing an argument string passed to a routine.

token code. In SPI, a 32-bit value that, as the first part of a token, allows any token to be identified and located within an SPI message. A token code consists of a token type

(16 bits) and a token number (16 bits). In TAL, TACL, and COBOL, names are used to represent token codes (ZSPI-TKN-SSID, for example). In DDL, a special definition (using the TOKEN-CODE statement) that the DDL compiler translates into an SPI token code. Token codes have symbolic names of the form *subsys*-TKN-*name*. See also token map and qualified token code.

token data type. The part of the token code that defines the kind of value (such as an integer or a file name) allowed for a token. Token data types have symbolic names of the form *subsys*-TDT-*name*.

token length. The part of a token code that indicates the length in bytes of the corresponding token value. A token length of 255 indicates that the token value has variable length or a length greater than 254, in which case the first word of the token value contains the (noninclusive) byte length of the rest of the token value.

token map. An SPI structure that describes the fields of an extensible structured token. Also, a variable name used to refer to an extensible structured token. The token map includes a token code and a description of the token value: its fields, the null values of those fields, and the versions of the fields. A token map defines a structure that might change in some later code version (by the addition of new fields at the end), and the information in the map allows SPI to provide compatibility between different structure versions. In DDL, a special definition (using the TOKEN-MAP statement) that the DDL compiler translates into an SPI token map. Token maps have symbolic names of the form *subsys*-MAP-*name*.

token number. The number used by a subsystem to identify each token that it defines. The token type and the token number together form the token code. Token numbers have symbolic names of the form *subsys*-TNM-*name*.

token type. In SPI, a combination of the token data type and token length; part of the token code. In DDL, a special definition (using the TOKEN-TYPE statement) that the DDL compiler will translate into an SPI token type. Token types have symbolic names of the form *subsys*-TYP-*name*.

token value. The value assigned to a token.

token-oriented. Programmatic interfaces that convey information in tokens, code-value pairs accessed by code rather than by address or ordinal position. SPI is a token-oriented programmatic interface.

upward compatibility. The ability of a requester to operate gracefully with a server of a higher version. In this case, the requester is upward-compatible with the server, and the server is downward-compatible with the requester. Compare with downward compatibility.

version compatibility. The ability of a requester and server of different revision levels to operate gracefully together.

version number. A 16-bit integer representation of a software version. For NonStop Kernel subsystems, this consists of an uppercase alphabetic character in its left half and a number in its right half.

warning. A condition, encountered in performing a command or other operation, that can be significant, but does not cause the command or operation to fail. A warning is less serious than an error. Compare with error.