WSim Workload Simulator

IBM

Script Guide and Reference

Version 1 Release 1

SC31-8946-01

WSim Workload Simulator

IBM

Script Guide and Reference

Version 1 Release 1

SC31-8946-01

Note!

Before using this information and the product it supports, be sure to read the general information under “Notices” on page 517.

Second Edition (October 2015)

This document applies to the Workload Simulator Version 1 Release 1 (program number 5655-I39), an IBM licensed program, which runs under the following operating systems:

MVS/370 (MVS/SP Version 1 or later)

MVS/Extended Architecture (MVS/SP Version 2 or later)

MVS/Enterprise System Architecture (MVS/SP Version 3 or later)

OS/390

Publications are not stocked at the address given below. If you want more IBM publications, ask your IBM representative or write to the IBM branch office serving your locality.

Contents

Tables

Figures

About this book

This book is a guide and reference for the formats of the Workload Simulator (WSim) language statements. This book is intended to help you write network definitions and message generation decks.

This book also provides information about a programming language, the Structured Translator Language (STL), that enables you to create message generation decks for your simulated network. These message generation decks enable your simulated terminals to send and receive messages. This book describes the STL Translator, a utility that converts STL programs into the message generation decks that WSim uses to generate messages. The STL Translator also stores network definitions that WSim uses to define your simulated network. Additionally, this book contains information about how to debug STL programs.

Who should read this book

This book is intended for people responsible for coding message generation decks for a simulated network. It provides detailed syntax and special considerations for coding scripting language statements. It also provides instructions for using the STL programming language and includes a reference section giving details on syntax and usage for specific keywords and functions.

For an overview of WSim and how to begin using it, refer to *WSim User's Guide*. Refer to *Creating WSim Scripts* for information about defining your simulated network. You should also be familiar with the terminals you are simulating and the applications you plan to test.

How to use this book

This book contains three parts: the language statements and their associated operands, an introduction to STL concepts and programming procedures, and a reference section that provides detailed information about STL keywords and functions.

This book contains the following sections:

Part 1, "WSim language statements," on page 1 explains the language statements and their operands. It also describes the order in which you should code the statements when defining a network for simulation. It includes the following chapters:

- Chapter 1, "Introduction," on page 3, describes how the language statements are presented in this book.

  Chapter 2, "Understanding network hierarchies," on page 9, describes the different network hierarchies you use to define a simulated terminal.
- Chapter 3, "Defining a network configuration," on page 17, describes the statements you use to define network configurations.
- Chapter 4, "Defining general simulation statements," on page 23, describes the statements you use to define the general network configuration.

- Chapter 5, “Defining CPI-C simulation statements,” on page 59, describes the network configuration definition statements you use to define the simulation of a Common Programming Interface Communications (CPI-C) transaction program.
- Chapter 6, “Defining VTAMAPPL simulation statements,” on page 71, describes the network configuration definition statements you use to define the simulation of SNA logical units through the VTAM application interface.
- Chapter 7, “Defining TCP/IP client simulation statements,” on page 87, describes the network configuration definition statements you use to define simulations of Telnet 3270, 3270E, 5250, and Network Virtual Terminal client devices, File Transfer Program (FTP) clients, or Simple TCP or UDP clients.
- Chapter 8, “Defining the message generation deck,” on page 103, describes the scripting language statements you use to define the data and control message generation.
- Chapter 9, “Data field options,” on page 199, describes the data field options used to insert variable data into messages or data fields that are constructed duringWSim processing.
- Chapter 10, “Data locations,” on page 209, describes the locations used to evaluate logic tests.
- Chapter 11, “Terminal, device, and logical unit types,” on page 211, lists the types of terminals that WSim can simulate.
- Chapter 12, “Counters and switches,” on page 213, lists which set of counters are allocated to each simulated device.
- Chapter 13, “Format control statements,” on page 215, lists the statements that you can use to format network listings for easier readability.
- Chapter 14, “Conditions logic test not evaluated,” on page 217, lists the conditions under which logic tests are not evaluated.

Part 2, “Guide to using STL and the STL Translator,” on page 219 provides an overview of STL concepts and functions. Read this part to obtain general information about the elements and capabilities of the language and also to learn how to program for specific situations and terminals. It includes the following chapters:

- Chapter 15, “Introducing the Structured Translator Language,” on page 221 introduces STL.
- Chapter 16, “Designing STL programs,” on page 227 discusses factors to consider when designing an STL program.
- Chapter 17, “Understanding the elements of an STL program,” on page 233 describes the various programming elements that compose an STL program.
- Chapter 18, “Controlling STL program flow,” on page 255 explains how to control the order in which your program is executed.
- Chapter 19, “Generating messages for an STL program,” on page 267 describes how to provide data to be used in messages sent by your simulated terminals.
- Chapter 20, “Transmitting and receiving messages from an STL program,” on page 285 explains how to transmit and receive messages, how to control intermessage delays, and how to use events to synchronize terminal activities.
- Chapter 21, “Monitoring and automating your test,” on page 307 discusses ways to operate and monitor your simulation.
- Chapter 22, “Using the STL Translator,” on page 311 explains how to use the STL Translator to translate your STL programs into message generation statements and to store your network definitions.

- Chapter 23, “Combining STL programs and network definitions,” on page 325 provides information about integrating network definitions with your STL program.
- Chapter 24, “Debugging your STL programs,” on page 337 describes how to use printed listings to correct errors in your STL programs and explains how you can use the output from the Loglist Utility to examine how your STL program worked.

Part 3, “Reference to STL statements and functions,” on page 353 contains reference information about STL statements and functions. Statements and functions are presented alphabetically to provide quick access to details about the language. It includes the following chapters:

- Chapter 25, “Reference to STL statements,” on page 355 provides syntax and a detailed description for each statement used in the STL programming language.
- Chapter 26, “Reference to STL functions,” on page 449 provides syntax and a detailed description for each function used in the STL programming language.
- Chapter 27, “Keys valid for particular devices,” on page 501, lists device key statements and AID keys that can be used for particular devices.
- Chapter 28, “Expressions not allowed in asynchronous conditions,” on page 503, lists the on types of expressions are not allowed on ONIN, ONOUT, or ON SIGNALED statements.
- Chapter 29, “STL reserved words,” on page 507, lists STL keywords, functions, and reserved variable names that may not be used as user-defined variable or constant names.
- Chapter 30, “STL Variable and Named Constant Declarations for CPI-C Verb Parameters,” on page 511, contains a comprehensive list of STL variable declarations for CPI Communications (CPI-C) verb parameters.

The Glossary lists the terms that are used in this book.

The Bibliography lists the related publications that you can use to find more information on networks.

Typographic conventions

The following typographic conventions are used in this book:

Convention	Meaning
UPPERCASE	Used for STL reserved words, including keywords, function names, and reserved variables. These words must be entered with the characters shown, but they do not have to be entered in uppercase.
italic	Indicates items for which you must fill in information.
{ }	Indicates that the stacked items enclosed in braces are alternatives. You must include one of the items.
[ ]	Indicates that the items enclosed in brackets are optional.
. . .	Indicates that part of the text or example has been omitted. The missing text is irrelevant to the information being discussed. The three dots may be either vertical or horizontal.

Where to find more information

The following list shows the books in the WSim library. For more information about related publications, see the "Bibliography" on page 529.

Planning, Installation, and Operation	
WSim User's Guide	SC31-8948
WSim Messages and Codes	SC31-8951
WSim Test Manager User's Guide and Reference	SC31-8949
Resource and Message Traffic Definition	
Creating WSim Scripts	SC31-8945
WSim Script Guide and Reference	SC31-8946
WSim Utilities Guide	SC31-8947
Customization	
WSim User Exits	SC31-8950

Part 1. WSim language statements

Chapter 1. Introduction

This chapter introduces Workload Simulator (WSim) and explains the format used to code language statements.

What is Workload Simulator?

WSim is a terminal and network simulation tool. You can use WSim to determine system performance and response time, to evaluate network design, to perform functional testing, and to automate regression testing. Used as a basic tool in a comprehensive test plan, WSim increases the effectiveness of the overall test by providing automated, repeatable test conditions with minimal resources.

WSim runs on any IBM host processor that supports:
- MVS/370 (MVS/SP Version 1 or later)
- MVS/XA (MVS/SP Version 2 or later)
- MVS/ESA (MVS/SP Version 3 or later)
- OS/390

In this book, MVS is any environment running MVS, MVS/XA, or MVS/ESA (unless explicitly stated otherwise).

Coding scripting language statements

WSim scripting language statements consist of network configuration statements and message generation statements. To code scripting language statements, you can use the following three fields:

Name	The Name field begins in column 1 and contains a label to be associated with the statement. Unless otherwise noted, the Name field can contain a maximum of eight characters.
Statement	The Statement field is separated from the Name field by at least one blank and contains the actual scripting language statement. **Note:** Regardless of whether or not you code the Name field, the Statement field cannot begin in column 1.
Operand	The Operand field is separated from the Statement field by at least one blank and contains the operands or text data for the statement. The Operand field cannot extend beyond column 71.

Note: You can code scripting language statements in mixed case.

Coding the name field

Normally, the Name field can contain a maximum of eight characters. For each type of statement, the following characters are valid:
- Network configuration statements. The valid characters are a - z, A - Z, 0 - 9, and all symbols except WSim delimiters, which are ( ) + - = , . | & and blank.
- Message generation statements. The valid characters are a - z, A - Z, 0 - 9, and the symbols #, @, and $.

Note: See the specific statement for any other restrictions that apply to the Name field. For example, the first letter of a name may have to be alphabetic, or a name may have to be completely numeric.

Coding the operand field

There must be at least one blank between a statement and its operands. See the specific statement for a list of the available operands for the statement. The following conventions are used in this manual to describe statement operands:

- Required operands are not enclosed in brackets and are listed first in alphabetical order. Optional operands follow the required operands. They are enclosed in brackets and are also listed in alphabetical order.
- An operand is described by its keyword name in uppercase, followed by an equal sign and a variable value that you can choose.
- You must code the following symbols exactly as they appear for the statement description:

Asterisk	*
Comma	,
Dollar sign	$
Equal sign	=
Hyphen	-
Parentheses	( )
Period	.
Plus sign	+
Quote	'

- Lowercase italicized letters and words represent variables for which you can supply specific information. The operand description, which follows each statement description, lists the valid variable values and any restrictions for the values.
- The following symbols are used for the operand format and should never be coded as part of an operand value:

Braces	{ }	A stack of items, each contained within braces, represents a set of alternatives, one of which must be chosen. For example: **{YES}** **{NO}**
Vertical Bar	|	Items separated by vertical bars represents a set of alternatives, one of which must be chosen. For example: **{YES|NO}**
Brackets	[ ]	Information contained within brackets represents an option that can be included or omitted when coding the operand. For example: [,**DISPLAY=**(*a*,*b*[,*c*,*d*])]
Ellipsis	...	An ellipsis indicates that a number of items may be coded. For example: (*integer*,...)

Underscore _	An underscored item represents a default value that need not be coded. For example: **[,RTR={YES | NO}]**

If a set of stacked items or items separated by vertical bars, each contained within braces, is included within a set of brackets, one of the items must be chosen if the bracketed operand is coded.

Comment statements

You can include comments by coding an asterisk (*) in statement column 1 or by including text, separated by at least one blank, after the last keyword operand. Statements that have no defined operands can contain comments after the Statement field. Statements that have operands must have at least one operand specified before a comment appears on the statement.

To ensure that comments are stored and listed with the appropriate network configuration or message generation deck, they should follow, rather than precede, the NTWRK, MSGTXT, or MSGUTBL statement.

Continuing statements

Statements can be continued between operands. Continuation is indicated with a comma after the last operand on the line, followed by a blank. The rest of the operands may be specified on the next line anywhere after the first column, which must be blank. You must specify at least one operand to continue a statement to the next line.

Continuing text data

You can also continue most statement operands that specify text data. Use any of the following methods to continue operand data.

1. Continue the data through column 71 and begin again in column 2 of the next line. Text data past column 71 will be ignored.
2. Close the text on one line with an ending delimiter and comma and begin again on the next line with an opening delimiter in any column after column 1 and preceding column 72.
3. Close the text on one line with an ending delimiter and plus sign and begin again on the next line with an opening delimiter in any column after column 1 and preceding column 72.

The following table shows which operands can be continued and which continuation method is valid:

Table 1. Operand continuation methods

Statement	Operand	Continuation Method
CMND	DATA=(*data*)	1,2,3
CMND	RESP=(*data*)	1,3
DATASAVE	INSERT=(*data*)	1,2,3
DATASAVE	TABLEI=(*data*)	1,2,3
DATASAVE	TABLEO=(*data*)	1,2,3
DATASAVE	TEXT=(*data*)	1,2,3

Table 1. Operand continuation methods (continued)

Statement	Operand	Continuation Method
EXIT	PARM=(*data*)	1,2,3
FE	COMMAND=(*data*)	1,2,3
IF	LOCTEXT=(*data*)	1,2,3
IF	LOG=(*data*)	1,2,3
IF	TEXT=(*data*)	1,2,3
IF	THEN | ELSE=VERIFY(*data*)	1,2,3
LOG	(*data*)	1,2,3
MSGUTBL	(*entry*)	1,3
ON	THEN=VERIFY(*data*)	1,2,3
OPCMND	(*data*)	1,2,3
STRIPE	(*data*)	1,2,3
TEXT	RESP=(*data*)	1,3
TEXT	(*data*)	1,2,3
UTBL	(*entry*)	1,3
WTO	(*data*)	1,2,3
WTOABRHD	(*data*)	1,2,3

Coding literal text DBCS data

Table 2 lists the statements and operands where you can code literal text DBCS data.

Table 2. Literal text DBCS statements and operands

Statement	Operand
FE	COMMAND
IF (network- and message-level)	LOCTEXT LOG TEXT THEN | ELSE=VERIFY(*data*)
NTWRK	HEAD
UTBL	(*entry*)
CMND	DATA RESP
DATASAVE	INSERT TABLEI TABLEO TEXT
EXIT	PARM
LOG	(*data*)
MSGUTBL	(*entry*)
ON	THEN=VERIFY(*data*)
OPCMND	(*data*)
STRIPE	(*data*)

Table 2. Literal text DBCS statements and operands (continued)

Statement	Operand
TEXT	(*data*) RESP
WTO	(*data*)
WTOABRHD	(*data*)

Refer to , SC31-8945 for examples of coding DBCS data.

Chapter 2. Understanding network hierarchies

This chapter describes the network hierarchical concepts involved in defining a network configuration.

Network definition

You can use the following simulation methods to provide teleprocessing message traffic to the system being tested:

CPI-C transaction program simulation
: To simulate a Common Programming Interface Communications (CPI-C) transaction program, use the APPCLU and TP statements.

VTAM application simulation
: To simulate an SNA logical unit via the VTAM application interface, use the VTAMAPPL and LU statements.

TCP/IP simulation
: To simulate 3270, 3270E, 5250, or NVT devices having Telnet sessions, client applications that are using the File Transfer Protocol (FTP), or Simple TCP or UDP clients, use the TCPIP and DEV statements.

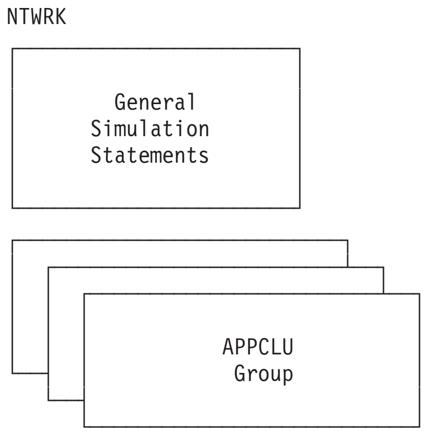
Note: You can use all simulation methods in the same network.

Sequence of network configuration definition statements

This section provides examples to show the general order of statements when configuring various types of networks. Each example contains boxes that represent a group of statements. Multiple boxes mean that you can code a group more than once. The boxes are broken down into the individual statements following the examples.

Refer to Table 3 on page 13 for more information on the order of statements.

CPI-C simulation



See "General simulation statements" on page 11 for information about coding these statements.

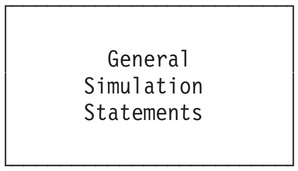
A CPI-C network consists of one or more APPCLU groups. See "APPCLU group" on page 12 for information about coding this group.

```
[FE] ...
```

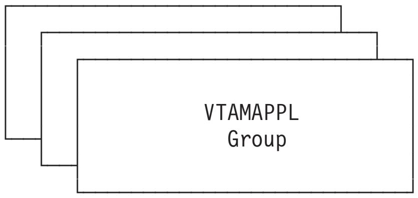
You can code one or more Future Event (FE) statements in a CPI-C simulation. See "FE - future event statement" on page 24 for more information about this statement.

VTAMAPPL simulation

```
NTWRK
```



See "General simulation statements" on page 11 for information about coding these statements.



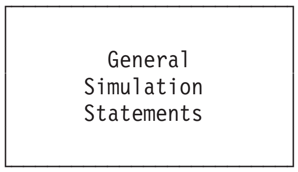
A VTAMAPPL network consists of one or more VTAMAPPL groups. See "VTAMAPPL group" on page 12 for information about coding this group.

```
[FE] ...
```

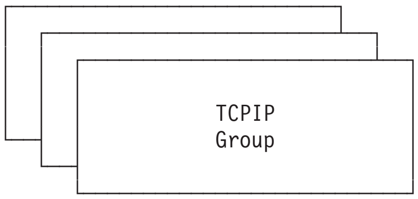
You can code one or more Future Event (FE) statements in a VTAMAPPL simulation. See "FE - future event statement" on page 24 for more information about this statement.

TCP/IP client simulation

```
NTWRK
```



See "General simulation statements" on page 11 for information about coding these statements.

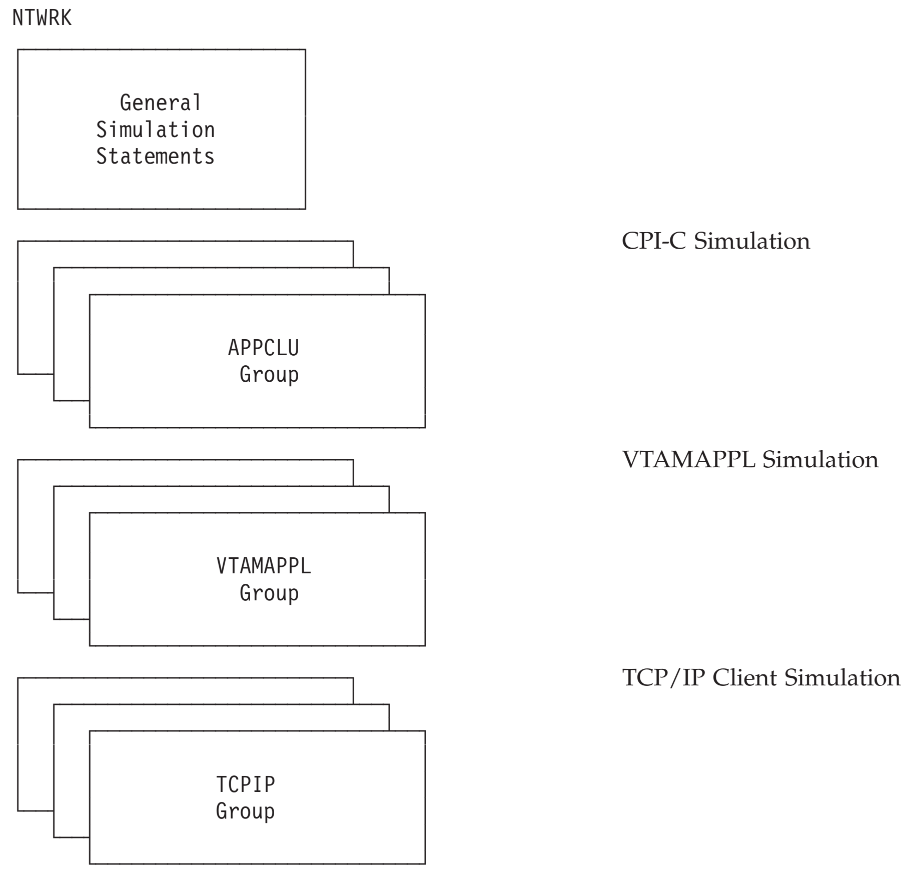


A TCP/IP Client network consists of one or more TCPIP groups. See "TCPIP group" on page 12 for information about coding this group.

```
[FE] ...
```

You can code one or more Future Event (FE) statements in a TCPIP Client simulation. See "FE - future event statement" on page 24 for more information about this statement.

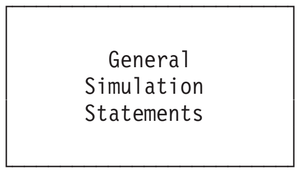
Combined network



Definition of statement groups

This section describes the statements used to code the boxes shown in the previous section. In this section, brackets mean that the statement or group of statements are optional. The ellipses (...) means that the statement or group of statements may be coded more than once.

General simulation statements



```
[NTWRKLOG]
[MSGDISK]
[RATE] ...
[UTBL] ...
[SIDEINFO
   SIDEENT ...
 SIDEEND]
[RN] ...
[UDIST] ...
[INCLUDE] ...
[IF] ...
```

```
PATH ...
```

At least one PATH statement is required in the network.

```
[DIST] ...
[UTI] ...
[FILE] ...
```

APPCLU group

```
┌────────────────────────────┐
│                            │
│           APPCLU           │
│            Group           │
│                            │
└────────────────────────────┘
APPCLU
  TP ...
```

The APPCLU statement defines the interface WSim will use when simulating CPI-C transaction programs.

The TP statement defines a transaction program to be simulated using the WSim/VTAM application program interface.

VTAMAPPL group

```
┌────────────────────────────┐
│                            │
│          VTAMAPPL          │
│            Group           │
│                            │
└────────────────────────────┘
VTAMAPPL
  LU ...
```

The VTAMAPPL statement defines the interface WSim will use when executing as a VTAM application program.

The VTAMAPPL LU statement defines a half-session to be simulated using the WSim/VTAM application program interface.

TCPIP group

```
┌────────────────────────────┐
│                            │
│          TCPIP             │
│          Group             │
│                            │
└────────────────────────────┘
TCPIP
  DEV ...
```

The TCPIP statement is required in this group. At least one DEV statement is required under the TCPIP statement.

Sequence of valid network configuration statements

Table 3 shows the statement types that are valid in a configuration definition. The statements are listed in the order in which they should be coded. Each statement is listed with the statements that can follow it, to help you determine what statements can be repeated or left out, depending on the configuration you are trying to simulate. Superscripts are used to distinguish between statements that can have different functions, such as the PATH statement, which may define the order of message generation decks to be used by a device.

Table 3. Sequence of valid network configuration statements

Statement	Can be followed by	Comments
NTWRK	NTWRKLOG MSGDISK RATE UTBL SIDEINFO RN UDIST INCLUDE IF PATH^1	NTWRK is required and must be the first statement of a network.
NTWRKLOG	MSGDISK RATE UTBL SIDEINFO RN UDIST INCLUDE IF PATH^1	The NTWRKLOG statement is optional.
MSGDISK	RATE UTBL SIDEINFO RN UDIST INCLUDE IF PATH^1	The MSGDISK statement is optional.
RATE	RATE UTBL SIDEINFO RN UDIST INCLUDE IF PATH^1	The RATE statement is required only when using rate table delays.
UTBL	UTBL SIDEINFO RN UDIST INCLUDE IF PATH^1	The UTBL statement is optional.
SIDEINFO	SIDEENT	The SIDEINFO statement is optional.

Table 3. Sequence of valid network configuration statements (continued)

Statement	**Can be followed by**	**Comments**
SIDEENT	SIDEENT SIDEEND	At least one SIDEENT statement is required after the SIDEINFO statement.
SIDEEND	RN UDIST INCLUDE IF PATH[1]	The SIDEEND statement is required following the last SIDEENT statement.
RN	RN UDIST INCLUDE IF PATH[1]	The RN statement is optional.
UDIST	UDIST INCLUDE IF PATH[1]	The UDIST statement is optional.
INCLUDE	INCLUDE IF PATH[1]	The INCLUDE statement is optional. It is primarily used to define required but unreferenced message generation decks.
IF	IF PATH[1]	The IF statement is optional.
PATH[1]	PATH[1] DIST UTI FILE APPCLU VTAMAPPL TCPIP	At least one PATH statement is required for all simulation types.
DIST	DIST UTI FILE APPCLU VTAMAPPL TCPIP	The DIST statement is optional.
UTI	UTI FILE APPCLU VTAMAPPL TCPIP	The UTI statement is optional.
FILE	FILE APPCLU VTAMAPPL TCPIP	The FILE statement is optional.
APPCLU	TP	The APPCLU statement is optional.
TP	TP APPCLU VTAMAPPL TCPIP FE	At least one TP statement is required after an APPCLU statement.
VTAMAPPL	LU	The VTAMAPPL statement is optional.
LU	LU VTAMAPPL TCPIP FE	At least one LU statement is required after a VTAMAPPL statement.

Table 3. Sequence of valid network configuration statements (continued)

Statement	Can be followed by	Comments
TCPIP	DEV	The TCPIP statement is optional.
DEV	DEV TCPIP FE	At least one DEV statement is required after a TCPIP statement.
FE	FE	The FE statement is optional.

Note:

1. Message deck selection order definition.

Chapter 3. Defining a network configuration

This chapter describes the scripting language statements that define a network configuration.

Summary of operands for configuration statements

The table in this section lists the operands that you can code for network configuration statements.

For example, the STCPPORT operand is valid for both the NTWRK and the TCPIP statements. The description for the STCPPORT operand is found under the description of the TCPIP statement.

Note: This table does not indicate when to use an operand. Refer to the operand description for information about its use.

Table 4. WSim network configuration operands

Operand	Appears on	Can also be coded on
ALTCSET	LU (VTAMAPPL simulation) DEV (TCP/IP Client simulation)	NTWRK VTAMAPPL (VTAMAPPL simulation) TCPIP (TCP/IP Client simulation)
APLCSID	LU (VTAMAPPL simulation) DEV (TCP/IP Client simulation)	NTWRK VTAMAPPL (VTAMAPPL simulation) TCPIP (TCP/IP Client simulation)
APPLID	VTAMAPPL (VTAMAPPL simulation) APPCLU (CPI-C simulation)	-
ASSOC	DEV (TCP/IP Client simulation)	TCPIP (TCP/IP Client simulation)
ATRABORT	LU (VTAMAPPL simulation) DEV (TCP/IP Client simulation)	NTWRK VTAMAPPL (VTAMAPPL simulation) TCPIP (TCP/IP Client simulation)
ATRDECK	LU (VTAMAPPL simulation) DEV (TCP/IP Client simulation)	NTWRK VTAMAPPL (VTAMAPPL simulation) TCPIP (TCP/IP Client simulation)
BASECSID	LU (VTAMAPPL simulation) DEV (TCP/IP Client simulation)	NTWRK VTAMAPPL (VTAMAPPL simulation) TCPIP (TCP/IP Client simulation)
BUFSIZE	APPCLU (CPI-C simulation) VTAMAPPL (VTAMAPPL simulation) TCPIP (TCP/IP Client simulation)	NTWRK
CCSIZE	LU (VTAMAPPL simulation) DEV (TCP/IP Client simulation)	NTWRK VTAMAPPL (VTAMAPPL simulation) TCPIP (TCP/IP Client simulation)
CHAINING	LU (VTAMAPPL simulation)	NTWRK VTAMAPPL (VTAMAPPL simulation)
CNOS	APPCLU (CPI-C simulation)	-
CNTRSEED	NTWRK	-
COLOR	LU (VTAMAPPL simulation) DEV (TCP/IP Client simulation)	NTWRK VTAMAPPL (VTAMAPPL simulation) TCPIP (TCP/IP Client simulation)

Table 4. WSim network configuration operands (continued)

Operand	Appears on	Can also be coded on
CONRATE	NTWRK	-
CPITRACE	TP (CPI-C simulation)	NTWRK APPCLU (CPI-C simulation)
CRDATALN	LU (VTAMAPPL simulation) DEV (TCP/IP Client simulation)	NTWRK VTAMAPPL (VTAMAPPL simulation) TCPIP (TCP/IP Client simulation)
DBCS	LU (VTAMAPPL simulation) DEV (TCP/IP Client simulation)	NTWRK VTAMAPPL (VTAMAPPL simulation) TCPIP (TCP/IP Client simulation)
DBCSCSID	LU (VTAMAPPL simulation) DEV (TCP/IP Client simulation)	NTWRK VTAMAPPL (VTAMAPPL simulation) TCPIP (TCP/IP Client simulation)
DELAY	TP (CPI-C simulation) LU (VTAMAPPL simulation) DEV (TCP/IP Client simulation)	NTWRK APPCLU (CPI-C simulation) VTAMAPPL (VTAMAPPL simulation) TCPIP (TCP/IP Client simulation)
DELYSEED	NTWRK	-
DISPLAY	LU (VTAMAPPL simulation) DEV (TCP/IP Client simulation)	NTWRK VTAMAPPL (VTAMAPPL simulation) TCPIP (TCP/IP Client simulation)
DLOGMOD	LU (VTAMAPPL simulation)	NTWRK VTAMAPPL (VTAMAPPL simulation)
EMTRATE	NTWRK	-
ENCR	LU (VTAMAPPL simulation)	NTWRK VTAMAPPL (VTAMAPPL simulation)
EXIT	NTWRK	-
EXTFUN	LU (VTAMAPPL simulation) DEV (TCP/IP Client simulation)	NTWRK VTAMAPPL (VTAMAPPL simulation) TCPIP (TCP/IP Client simulation)
FLDOUTLN	LU (VTAMAPPL simulation) DEV (TCP/IP Client simulation)	NTWRK VTAMAPPL (VTAMAPPL simulation) TCPIP (TCP/IP Client simulation)
FLDVALID	LU (VTAMAPPL simulation) DEV (TCP/IP Client simulation)	NTWRK VTAMAPPL (VTAMAPPL simulation) TCPIP (TCP/IP Client simulation)
FRSTTXT	TP (CPI-C simulation) LU (VTAMAPPL simulation) DEV (TCP/IP Client simulation)	NTWRK APPCLU (CPI-C simulation) VTAMAPPL (VTAMAPPL simulation) TCPIP (TCP/IP Client simulation)
FTPPORT	TCPIP	NTWRK
FUNCTS	DEV (TCP/IP Client simulation)	TCPIP (TCP/IP Client simulation)
HEAD	NTWRK	-
HIGHLITE	LU (VTAMAPPL simulation) DEV (TCP/IP Client simulation)	NTWRK VTAMAPPL (VTAMAPPL simulation) TCPIP (TCP/IP Client simulation)
INEXIT	NTWRK	-
INFOEXIT	NTWRK	-
INHBTMSG	NTWRK	-

Table 4. WSim network configuration operands (continued)

Operand	Appears on	Can also be coded on
INIT	LU (VTAMAPPL simulation)	NTWRK VTAMAPPL (VTAMAPPL simulation)
INSTANCE	TP (CPI-C simulation)	NTWRK APPCLU (CPI-C simulation)
INXEXPND	NTWRK	-
ITIME	NTWRK	-
IUTI	TP (CPI-C simulation) LU (VTAMAPPL simulation) DEV (TCP/IP Client simulation)	NTWRK APPCLU (CPI-C simulation) VTAMAPPL (VTAMAPPL simulation) TCPIP (TCP/IP Client simulation)
LOCLPORT	DEV (TCP/IP Client simulation)	-
LOGDSPLY	LU (VTAMAPPL simulation) DEV (TCP/IP Client simulation)	NTWRK VTAMAPPL (VTAMAPPL simulation) TCPIP (TCP/IP Client simulation)
LUTYPE	LU (VTAMAPPL simulation)	NTWRK VTAMAPPL (VTAMAPPL simulation)
MAXCALL	TP (CPI-C simulation) LU (VTAMAPPL simulation) DEV (TCP/IP Client simulation)	NTWRK APPCLU (CPI-C simulation) VTAMAPPL (VTAMAPPL simulation) TCPIP (TCP/IP Client simulation)
MAXNOPTN	LU (VTAMAPPL simulation) DEV (TCP/IP Client simulation)	NTWRK VTAMAPPL (VTAMAPPL simulation) TCPIP (TCP/IP Client simulation)
MAXPTNSZ	LU (VTAMAPPL simulation) DEV (TCP/IP Client simulation)	NTWRK VTAMAPPL (VTAMAPPL simulation) TCPIP (TCP/IP Client simulation)
MAXSESS	LU (VTAMAPPL simulation)	NTWRK VTAMAPPL (VTAMAPPL simulation)
MLEN	APPCLU (CPI-C simulation) VTAMAPPL (VTAMAPPL simulation) TCPIP (TCP/IP Client simulation)	NTWRK
MLOG	APPCLU (CPI-C simulation) VTAMAPPL (VTAMAPPL simulation) TCPIP (TCP/IP Client simulation)	NTWRK
MSGTRACE	TP (CPI-C simulation) LU (VTAMAPPL simulation) DEV (TCP/IP Client simulation)	NTWRK APPCLU (CPI-C simulation) VTAMAPPL (VTAMAPPL simulation) TCPIP (TCP/IP Client simulation)
NAMEHASH	NTWRK	-
NCTLEXIT	NTWRK	-
NETEXIT	NTWRK	-
NETUSER	NTWRK	-
OPTIONS	NTWRK	-
OUTEXIT	NTWRK	-
PASSWD	VTAMAPPL (VTAMAPPL simulation) APPCLU (CPI-C simulation)	-

Table 4. WSim network configuration operands (continued)

Operand	Appears on	Can also be coded on
PATH	TP (CPI-C simulation) LU (VTAMAPPL simulation) DEV (TCP/IP Client simulation)	NTWRK APPCLU (CPI-C simulation) VTAMAPPL (VTAMAPPL simulation) TCPIP (TCP/IP Client simulation)
PATHSEED	NTWRK	-
PORT	DEV (TCPIP simulation)	-
PROTMSG	LU (VTAMAPPL simulation) DEV (TCP/IP Client simulation)	NTWRK VTAMAPPL (VTAMAPPL simulation) TCPIP (TCP/IP Client simulation)
PRTSPD	LU (VTAMAPPL simulation) DEV (TCP/IP simulation)	NTWRK VTAMAPPL (VTAMAPPL simulation) TCPIP (TCP/IP simulation)
PS	LU (VTAMAPPL simulation) DEV (TCP/IP Client simulation)	NTWRK VTAMAPPL (VTAMAPPL simulation) TCPIP (TCP/IP Client simulation)
QUIESCE	TP (CPI-C simulation) LU (VTAMAPPL simulation) DEV (TCP/IP Client simulation)	NTWRK APPCLU (CPI-C simulation) VTAMAPPL (VTAMAPPL simulation) TCPIP (TCP/IP Client simulation)
REPORT	NTWRK	-
RESOURCE	LU (VTAMAPPL simulation) DEV (TCP/IP Client simulation)	NTWRK VTAMAPPL (VTAMAPPL simulation) TCPIP (TCP/IP Client simulation)
RSTATS	LU (VTAMAPPL simulation) DEV (TCP/IP Client simulation)	NTWRK VTAMAPPL (VTAMAPPL simulation) TCPIP (TCP/IP Client simulation)
RTR	LU (VTAMAPPL simulation)	NTWRK VTAMAPPL (VTAMAPPL simulation)
SAVEAREA	LU (VTAMAPPL simulation) DEV (TCP/IP Client simulation)	NTWRK VTAMAPPL (VTAMAPPL simulation) TCPIP (TCP/IP Client simulation)
SCAN	NTWRK	-
SEQ 222	TP (CPI-C simulation) LU (VTAMAPPL simulation) DEV (TCP/IP Client simulation) NTWRK (network definition)	APPCLU (CPI-C simulation) VTAMAPPL (VTAMAPPL simulation)
SERVADDR	DEV (TCP/IP Client simulation)	NTWRK TCPIP (TCP/IP Client simulation)
SIDEINFO	APPCLU (CPI-C simulation)	-
STCPHCLR	DEV (TCP/IP Client simulation)	NTWRK TCPIP (TCP/IP Client simulation)
STCPHCLX	DEV (TCP/IP Client simulation)	NTWRK TCPIP (TCP/IP Client simulation)
STCPPORT	TCPIP (TCP/IP client simulation)	NTWRK
STCPROLE	DEV (TCP/IP Client simulation)	-
STIME	NTWRK	-

Table 4. WSim network configuration operands (continued)

Operand	Appears on	Can also be coded on
STLTRACE	TP (CPI-C simulation) LU (VTAMAPPL simulation) DEV (TCP/IP Client simulation)	NTWRK APPCLU (CPI-C simulation) VTAMAPPL (VTAMAPPL simulation) TCPIP (TCP/IP Client simulation)
SUDPPORT	TCPIP (TCP/IP Client simulation)	NTWRK
TCPNAME	TCPIP (TCP/IP Client simulation)	NTWRK
TEXTSEED	NTWRK	-
THKTIME	LU (VTAMAPPL simulation) DEV (TCP/IP Client simulation)	NTWRK VTAMAPPL (VTAMAPPL simulation) TCPIP (TCP/IP Client simulation)
THROTTLE	LU (VTAMAPPL simulation)	NTWRK VTAMAPPL (VTAMAPPL simulation)
TNPORT	TCPIP	NTWRK
TPNAME	TP (CPI-C simulation)	-
TPREPEAT	TP (CPI-C simulation)	NTWRK APPCLU (CPI-C simulation)
TPSTATS	TP (CPI-C simulation)	NTWRK APPCLU (CPI-C simulation)
TPSTIME	TP (CPI-C simulation)	NTWRK APPCLU (CPI-C simulation)
TPTYPE	TP (CPI-C simulation)	NTWRK APPCLU (CPI-C simulation)
TYPE	DEV (TCPIP simulation)	NTWRK (Terminal simulation) TCPIP (TCPIP simulation)
UASIZE	LU (VTAMAPPL simulation) DEV (TCP/IP Client simulation)	NTWRK VTAMAPPL (VTAMAPPL simulation) TCPIP (TCP/IP Client simulation)
UCD	TP (CPI-C simulation)	NTWRK APPCLU (CPI-C simulation)
UCMDEXIT	NTWRK	-
UOM	LU (VTAMAPPL simulation) DEV (TCP/IP Client simulation)	NTWRK VTAMAPPL (VTAMAPPL simulation) TCPIP (TCP/IP Client simulation)
USERAREA	TP (CPI-C simulation) LU (VTAMAPPL simulation) DEV (TCP/IP Client simulation)	NTWRK APPCLU (CPI-C simulation) VTAMAPPL (VTAMAPPL simulation) TCPIP (TCP/IP Client simulation)
UTBLSEED	NTWRK	-
UTI	NTWRK	-
UXOCEXIT	NTWRK	-

Chapter 4. Defining general simulation statements

This chapter describes the general network configuration definition statements. These statements are listed in alphabetical order. For more information about the order in which you should place the statements in a network configuration deck, refer to Table 3 on page 13.

When coding the configuration definition statement for a network, you can code the common operands for the network resources at the highest level statements and override them on lower level statements, if necessary.

Operands on the NTWRK statement are grouped according to function and appear under group headings. The inclusion of an operand within a group indicates that you should use that operand only when defining resources that belong to that group.

Note: You can select operands from more than one functional group to define a particular resource. For example, operands from the SNA SIMULATION OPERANDS and 3270 SIMULATION OPERANDS groups can appear on a single LU statement.

DIST - PATH distribution statement

```
name DIST weight[,...]
```

Function

The DIST statement defines a probability distribution to be used when choosing entries from a PATH statement. The *name* field must match the *name* field of the PATH statement with which the DIST statement is associated. This statement is optional.

Where

name

Function: Specifies the name to be used to identify this distribution.

Note: This field is used to locate the PATH statement with which this distribution corresponds.

Format: From one to eight alphanumeric characters.

Default: None. This field is required.

*weight***[,...]**

Function: The number specified assigns a relative weight to the corresponding path entry. Each *weight* represents a fractional value of the total weights specified on this DIST statement. This fractional value is determined by dividing the weight specified by the total of all the weights. The probability that a particular path entry will be chosen for message generation on any given selection is this fractional value. For example:

```
1 PATH A,B,C
1 DIST 50,30,20
```

WSim adds all the weights (50 + 30 + 20 = 100), generates a random number between 1 and the sum (100), and chooses a deck that corresponds to that number. Here, 1 through 50 represents deck A, 51 through 80 represents deck B, and 81 through 100 represents deck C. If WSim generates the number 72, it would choose deck B.

Format: A series of integers from 0 to 65535 separated by commas.

Note: The number of weights entered must be the same as the number of entries on the corresponding PATH statement. The sum of the weights must be greater than zero and must not exceed 65535.

Default: None. At least one weight must be entered.

FE - future event statement

```
[name] FE COMMAND=(command)
         {,EVENT=event}
         {,TIME=ssssssss}
```

Function

The FE statement defines an operator command to be executed at a specified time after the network simulation has begun or when a specified EVENT is signaled. This statement is optional.

Where

name
Function: Specifies the name to be used for this statement (for user information only).

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

COMMAND=(*command***)**
Function: Specifies the operator command to be executed.

Format: The *command* is a character string similar to the data entered with an OPCMND statement (see “OPCMND - operator command statement” on page 173), except that data field options are not allowed.

Default: None. This operand is required.

EVENT=*event*
Function: Specifies the name of an event which, when signaled, will cause the command specified by this statement to be processed.

FE statements associated with EVENTs will still be active after the network is stopped in case you want to signal the EVENT name and cause the FE to be executed.

Format: A 1- to 8-character name.

Default: None.

TIME=*ssssssss*

Function: Specifies the time into the run that the operator command is to be processed.

FE statements with the TIME operand will be dequeued and not executed if the network is stopped before the time is reached.

Format: The value of *ssssssss* is from one to eight digits specifying the number of seconds. The range for *ssssssss* is from 1 to 21474836.

Default: None.

Note: The EVENT and TIME operands cannot appear on the same FE statement. Code either the EVENT or the TIME operand, but not both.

A future event will only be executed once. When the time expires or the event is signaled, the event will not be executed again unless the network is stopped and restarted.

FILE - FTP FILE definition statement

```
name FILE [DATA=utblname]
          [,NUMREC=integer]
          [,TYPE={E|A}]
          [,RECFM={V|F}]
          [,RECLEN=integer]
          [,MINLEN=integer]
```

Function

The FILE statement is used in a network definition to define the data to be sent for a simulated file being transferred from WSim via the FTP protocol. This statement is optional.

Where

name

Function: Provides a name to use to reference this FILE statement on PUT or APPEND statements in the script. *name* must be unique among the FILE statements within the network definition.

Format: From one to eight alphameric characters.

Default: None. This field is required.

DATA=*utblname*

Function: Identifies a user table (UTBL or MSGUTBL) from which the data for this file is obtained. Each entry in the user table represents a record in the file. Either DATA or NUMREC must be specified. Both may be specified. See further discussion under NUMREC.

Format: A one- to three-digit number identifying a UTBL statement in the current network definition or a one- to eight-character name identifying an MSGUTBL member.

Default: None. This operand is optional.

NUMREC=*integer*

Function: Identifies the number of records to be sent as part of this file transfer. If DATA is also coded, the number of records specified is obtained from the user table named by that operand. Entries in the user table are transmitted in order until the specified number has been sent. If NUMREC is greater than the number of entries in the table, WSim will cycle through the user table until the number of records specified has been transmitted. This allows for extensive pattern data as well as real file data to be represented in the user tables.

If DATA is not specified, WSim will automatically generate records that contain repeated strings of the alphabetic and numeric characters.

Format: A decimal number from 1 to 2147483647.

Default: Number of entries in the user table if DATA is also specified. None if DATA is not specified. Either DATA or NUMREC must be coded.

TYPE={E|A}

Function: Specifies the type of data contained in the file represented by this FILE statement. Transfer type is the code used in transferring the data represented by the FILE statement to a server and is controlled by FTP commands issued from your script. TYPE specifies whether the data referenced by this FILE statement should be treated as EBCDIC (E) or ASCII (A) data. This specification is used to determine whether WSim should translate the file data prior to transmission.

For the TYPE operand, you can specify the following values:

A indicates that the DATA specified for this FILE statement should be treated as ASCII data.

E indicates that the DATA specified for this FILE statement should be treated as EBCDIC data.

If the transfer type and file type agree, or if the transfer type is IMAGE, WSim does not perform any translation. This operand is only meaningful if DATA is also specified.

Format: E or A as shown.

Default: E

RECFM={V|F}

Function: Specifies whether the records in this simulated file are to be considered Variable (V) or Fixed (F) length records.

For the RECFM operand, you can specify one of the following values:

V Specifies records as variable length. If DATA is also specified, the length of each user table entry will determine the individual record lengths up to the length specified by the RECLEN operand, if any, or 65535. If DATA is not specified, the records will have random lengths between the RECLEN and MINLEN specifications (see below).

F Specifies records as fixed length. If DATA is also specified, each entry from the user table named by the DATA operand will be truncated or padded with blank characters, as necessary, to the length specified by the RECLEN operand. If DATA is not specified, the automatically generated records will each have the length specified by the RECLEN operand.

Format: V or F as shown.

Default: V

RECLEN=*integer*

Function: Specifies the length (or maximum length) of each record that is a part of this FILE. If RECFM=F is specified, each record is padded or truncated to this length. If RECFM=V is specified, each record is limited to a maximum of this length.

Format: A decimal number from 1 to 65535.

Default: None. This operand is required if RECFM=F or if DATA is not specified. It is optional if RECFM=V and DATA are specified.

MINLEN=*integer*

Function: Specifies the minimum length record to generate when RECFM=V and DATA is not specified. Not allowed if RECFM=F or if DATA is specified.

Format: A decimal number from 1 to the value specified by RECLEN.

Default: When applicable, the value specified by RECLEN.

IF - network-level logic test statement

```
[name] IF {CURSOR=(row,col)}
          {EVENT=event}
          {LOC=location}
          {LOCTEXT={cntr|(data)|integer}}
          [,AREA=area]
          [,COND={EQ|GE|GT|LE|LT|NE}]
          [,DATASAVE=(area,loc,leng)]
          [,DELAY=CANCEL]
          [,ELSE=action]
          [,LENG=value]
          [,LOCLENG=value]
          [,LOG=(data)]
          [,RESP=NO]
          [,SCAN={YES|value}]
          [,SCANCNTR=cntr]
          [,SNASCOPE={ALL|LOC|REQ|RSP}]
          [,TEXT={RESP|cntr|(data)|'xx'|integer}]
          [,THEN=action]
          [,TYPE=type]
          [,UTBL=name]
          [,UTBLCNTR=cntr]
          [,WHEN={IN|OUT}]
```

Function

The network level IF statements specify comparisons to be made on data sent or received by WSim or on terminal or device counter values, switch settings, cursor positions, or data areas. This statement performs the following functions:

- Test for event completion
- Set switches
- Override normal SNA responses
- Cancel current delays
- Save and log data
- Alter message generation paths based on comparison results.

The network level IF statement is optional, but you can code it up to 256 times. If used, the IF statement is active during the entire network simulation. The IF statements are evaluated in the order specified prior to any of the active message generation level IF statements.

Notes:

- Network-level IF statements are not evaluated for CPI-C simulations. If they are specified, WSim ignores them when simulating CPI-C transaction programs. For combined networks, they are evaluated for all resources except CPI-C transaction programs.
- See Chapter 14, "Conditions logic test not evaluated," on page 217 for more information on conditions under which a logic test is not evaluated.

Where

name

Function: Specifies the name to be used for this statement.

Format: From one to eight characters.

Default: None. This field is optional.

Note: The first three characters of this name, if coded, appear in INFO messages on loglists.

CURSOR=(*row*,*col***)**

Function: Specifies the cursor position to be compared with the current cursor position. It references the usable area screen as the operator would see it.

Note: The logic test will not be evaluated if the device is not a display device.

Format: Two values, each of which can be either an integer between 1 and 255 or a counter specification whose value is within this range specifying the row and column positions, respectively.

Default: None. You must code either the CURSOR, EVENT, LOC, or LOCTEXT operand.

Note: If you code the CURSOR operand, do not code the AREA, COND, EVENT, LENG, LOC, LOCLENG, LOCTEXT, SCAN, SCANCNTR, TEXT, UTBL, and UTBLCNTR operands.

EVENT=*event*

Function: Specifies the name of a wait or post event which is to be tested.

Format: For the EVENT operand, you can code one of the following options:

name	Specifies the name of the event to be tested for completion, where *name* is one to eight alphanumeric characters.
N±*value*	Specifies the event name to be referenced at an offset from the start of the network user area (+*value*) or back from the end of the network user area (-*value*).
U±*value*	Specifies the event name to be referenced at an offset from the start of the device user area (+*value*) or back from the end of the device user area (-*value*).
N*s+value*	Specifies an event name to be referenced at an offset from the start of the network save area.
s+value	Specifies an event name to be referenced at an offset from the start of the device save area.

Where:

value	Can be any integer from 0 to 32766 or a counter specification whose value is within this range. Zero is the offset to the first byte of the field for positive offsets (+*value*) and the offset to the last byte of the field for negative offsets (-*value*).
s	Is a savearea number from 1 to 4095.
name	Can be from one to eight alphanumeric characters or a user area or save area.

Default: None. You must code either the CURSOR, EVENT, LOC, or LOCTEXT operand.

Note: If you code the EVENT operand, do not code the AREA, COND, CURSOR, LENG, LOC, LOCLENG, LOCTEXT, SCAN, SCANCNTR, TEXT, UTBL, and UTBLCNTR operands.

LOC=*location*

Function: Specifies the starting location of the data that is to be compared.

Note: If you code the LOC operand, do not code the EVENT, CURSOR, or LOCTEXT operands.

Format: For the LOC operand, you can code one of the following options:

B±*value*	N*s*+*value*	TSW*n*|TSW*m*|...	TSEQ
C±*value*	*s*+*value*	SW*n*	DSEQ
D+*value*	(*row*,*col*)	SW*n*&SW*m*&..	NC*n*
TH+*value*	NSW*n*	SW*n*|SW*m*|...	LC*n*
RH+*value*	NSW*n*&NSW*m*&..	NSW*n*&TSW*m*&SW*n*&..	TC*n*
RU+*value*	NSW*n*|NSW*m*|...	NSW*n*|TSW*m*|SW*n*|...	DC*n*
N±*value*	TSW*n*	NSEQ	
U±*value*	TSW*n*&TSW*m*&..	LSEQ	

value can be an integer from 0 to 32766 or a counter specification whose value is within this range. Zero is the offset to the first byte of the field for positive offsets (+*value*) and the offset to the last byte of the field for negative offsets (-*value*).

Option	Description
B±*value*	For +*value*, begin testing an offset from the start of data in the device buffer. (For display devices, the device buffer is the screen image buffer.) For non-display devices and -*value*, begin testing at an offset back from the end of the data in the device buffer. For display devices and -*value*, begin testing at an offset back from the end of the screen image buffer.
C±*value*	Begin testing at an offset from the cursor for +*value*, or at an offset back from the cursor for -*value*. Normally, use this location only with display devices.
D+*value*	Begin testing at an offset from the start of the incoming or outgoing data stream. This includes the transmission header and the request header.
TH+*value*	Begin testing at an offset from the start of the transmission header.
RH+*value*	Begin testing at an offset from the start of the request header, if present.
RU+*value*	Begin testing at an offset from the start of the request unit.
N±*value*	Begin testing at an offset from the start (+*value*) or back from the end (-*value*) of the network user area defined by the NETUSER operand.

U±*value*	Begin testing at an offset from the start (+*value*) or back from the end (-*value*) of the device user area defined by the USERAREA operand.
N*s*+*value*	Begin testing at an offset from the start of the network save area specified by *s*, where *s* is an integer from 1 to 4095.
s+*value*	Begin testing at an offset from the start of the device save area specified by *s*, where *s* is an integer from 1 to 4095.
(*row,col*)	Indicates that the test is to be made at the specified row and column of the screen image of a display device, where *row* and *col* may each be an integer from 1 to 255 or a counter specification whose value is within this range. If specified for a non-display device type, the test will not be evaluated.

You can use any available counter or switch for a comparison. You can test up to 4095 switches where *n* and *m* represent switch numbers and can be from 1 to 4095.

Note: You can also specify a combination of network, terminal and device level switches (for example, TSW5|SW3|NSW7|NSW28). However, you cannot mix the & and | operators in the same LOC operand specification. Also, when one of the counter operands is coded, the corresponding value of the TEXT operand must be specified as numeric data or another counter.

Default: None. You must code either the CURSOR, EVENT, LOC, or LOCTEXT operand.

Notes:

- If, when a logic test is to be evaluated, the specified data location is not valid (for example, the location is outside the buffer or user area or not within the data transferred), the logic test is not evaluated and no action is taken unless the operand LOCLENG is coded.
- When multiple partitions are defined for a 3270 device, buffer or cursor offsets (B+, B-, C+, C-) will reference the data in the presentation space of the currently active partition. The combination (*row,col*) value will reference the display as you would see it, which could include data from more than one partition. The logic test will be performed against the presentation space data of the partition that owns the area of the display referenced by the (*row,col*) specification.
- For VTAMAPPL LUs, TSW, TSEQ, LSEQ, TC*n*, and LC*n* will reference a single set of switches and counters allocated to each VTAMAPPL.
- See Chapter 10, "Data locations," on page 209 for device-specific information concerning this operand. See Chapter 12, "Counters and switches," on page 213 for valid counter and switch specifications.

LOCTEXT={*cntr*|(*data*)|*integer*}

Function: Specifies the value that is to be used in the comparison.

Note: If you code the LOCTEXT operand, do not code the AREA, CURSOR, EVENT, LENG, LOC, or LOCLENG operands.

Format: For the LOCTEXT operand, you can enter one of the following values:

cntr	The counter to be used in the comparison. The valid values for *cntr* are NSEQ, LSEQ, TSEQ, DSEQ, NC*n*, LC*n*, TC*n*, and DC*n*, where *n* is an integer from 1 to 4095. These counters are explained under the LOC

operand on this statement. The LOCTEXT operand can specify a counter value only if the TEXT operand specifies a counter or integer value.

(*data***)**	The data coded within the text delimiter specified on the MSGTXT statement is to be used as the comparison data. The data field options can be used to specify the data (see Chapter 9, “Data field options,” on page 199). When comparing for specified data, enter hexadecimal data within the text delimiters by enclosing the digits within single quotes. Two digits compose one hexadecimal character. For example, LOCTEXT=(ABC) will generate a comparison for the three characters ABC. LOCTEXT=('AB'CD) is a comparison for three bytes including one hexadecimal character of AB and two EBCDIC characters of CD. A maximum of 32767 characters will be used for comparison. To enter a single quote, text delimiter (TXTDLM), or data field option control character (CONCHAR) as data, enter two of the characters. You can also continue the data on the next statement.
integer	A 1- to 10-digit integer ranging from 0 to 2147483647 is to be used for the comparison. This format is valid when the TEXT operand specifies a counter or integer value.

Default: None. You must code either the CURSOR, EVENT, LOC, or LOCTEXT operand.

Note: When LOCTEXT is coded, the THEN or ELSE action on an IF statement is always executed as long as the IF meets the criteria set by the SNASCOPE, TYPE, and WHEN operands. The string data comparison allows for unequal or null strings, with the shorter string being padded with blanks, unless SCAN is also coded. In this case, the LOCTEXT data is scanned and substrings within it equal to the length of the TEXT data are compared to the TEXT data.

AREA={*area*}

Function: Specifies a user area or save area location that contains the text value for which the comparison is to be made.

Format: For the AREA operand, you can code one of the following options. *value* can be any integer from 0 to 32766 or a counter specification whose value is within this range. Zero is the offset to the first byte of the field for positive offsets (+*value*) and the offset to the last byte of the field for negative offsets (-*value*). For save area references, *s* can be an integer from 1 to 4095.

N±*value*	The text is contained in the network user area where *value* is the offset from the beginning of the network user area to the text or from the end of the network user area back to the text.
U±*value*	The text is contained in the device user area where *value* is the offset from the beginning of the device user area to the text or from the end of the device user area back to the text.
N*s*+*value*	The text is contained in a network save area, where *s* is the number of the network save area and *value* is the offset from the beginning of the network save area to the text.
s+*value*	The text is contained in a device save area, where *s* is the number of the device save area and *value* is the offset from the beginning of the device save area to the text.

Default: None. The AREA, TEXT, or UTBL operand is required, except when you specify the LOC operand for switch testing or specify the EVENT operand. If you code the AREA operand, the LENG operand may also be coded. This operand cannot be coded if the LOC operand specifies a sequence or index counter, or if the TEXT or LOCTEXT operand is coded.

COND={EQ|GE|GT|LE|LT|NE}

Function: Specifies the condition for which the comparison is to be made. The data field identified by the LOC or LOCTEXT operand is compared to the data specified in the TEXT or AREA operands, and the condition is set. If the condition specified by the COND operand is met, the THEN action is taken. If the condition specified by the COND operand is not met, the ELSE action is taken.

Note: This option is not valid for a logic test that tests switches, performs a test under mask, tests an event, or tests a cursor position.

Format: For the COND operand, you can code one of the following options:

EQ	The two fields are equal.
GE	The LOC data is greater than or equal to the TEXT or AREA data or the LOCTEXT data is greater than or equal to the TEXT data.
GT	The LOC data is greater than the TEXT or AREA data or the LOCTEXT data is greater than the TEXT data.
LE	The LOC data is less than or equal to the TEXT or AREA data or the LOCTEXT data is less than or equal to the TEXT data.
LT	The LOC data is less than the TEXT or AREA data or the LOCTEXT data is less than the TEXT data.
NE	The two fields are not equal.

Default: EQ

DATASAVE=(*area*,*loc*,*leng*)

Function: Specifies data to be saved in a save area when the logic test is made, and the THEN action is taken. Each time data is saved in a save area, the length of that data is also saved to be used when the data is recalled.

If the data to be saved is longer than the save area, the data will be truncated. If the data is shorter than what is specified by the *leng* parameter, only the available data will be saved. If the specified save area is not defined in the network definition, no data will be saved, and an informational message will be written to the log data set.

Note: This operand is valid only if the THEN operand is coded on the same IF statement.

Format: For the DATASAVE operand, code the following three values (enclosed in parentheses and separated by commas):

area	Specifies which of the save areas is to be used for data retention, where *area* is either a device save area *s*, or a network save area N*s*, where *s* is an integer from 1 to 4095.
loc	Specifies the location of the data to be saved. You can enter B+*value*, C+*value*, D+*value*, TH+*value*, RH+*value*, or RU+*value*, where *value* can be an integer from 0 to 32766 or a counter specification whose value is within this range. B, C, D, TH, RH, and RU are the same as previously defined under the LOC operand.

leng Specifies the amount of data to be saved in bytes, where *leng* is an integer from 1 to 32767 or a counter specification whose value is within this range.

Default: None. This operand is optional.

DELAY=CANCEL

Function: Specifies that the current active delay is to be canceled.

Note: This operand is valid only if a THEN operand is also specified on the same IF statement. The delay is canceled only when the THEN action specified on the IF statement is taken.

Format: CANCEL

Default: None. This operand is optional.

ELSE=*action*

Function: Specifies the action to be taken if the specified condition was not met, the tested switches were off, or the tested event was not complete.

Note: If the ELSE operand is omitted and the condition is not met, no action is taken (all indicators and message generation paths are left as they were before the IF statement was encountered).

Format: You can code one of the following options:

B*name-label*	CONT	NSW(ON)	SW(ON)
B*name*	WAIT	NSW(OFF)	SW(OFF)
B-*label*	DLYCNCL	NSW*n*(ON)	SW*n*(ON)
C*name-label*	QUIESCE	NSW*n*(OFF)	SW*n*(OFF)
C*name*	RELEASE		
C-*label*		TSW(ON)	WAIT(*event*)
E*name-label*	RETURN	TSW(OFF)	POST(*event*)
E*name*	IGNORE	TSW*n*(ON)	RESET(*event*)
E-*label*	ABORT	TSW*n*(OFF)	SIGNAL(*event*)
			QSIGNAL(*event*)
VERIFY[-(*data*)]			

Note: For a description of these actions, see the THEN operand.

Default: None. You must code either the THEN or ELSE operand.

LENG=*value*

Function: Specifies the length of the text in the user area or save area specified by the AREA operand.

Format: *value* can be an integer from 1 to 32767 or a counter specification whose value is within this range.

Default: The amount of data remaining in the area starting from the offset specification. This operand is not allowed if you code the LOCTEXT or TEXT operand.

LOCLENG=*value*

Function: Specifies a length to be associated with the LOC operand data.

Note: When LOCLENG is coded, the THEN or ELSE action on an IF statement is always executed as long as the IF meets the criteria set by the SNASCOPE, TYPE, and WHEN operands. The string data comparison allows for unequal or null strings with the shorter string being padded with blanks. The LOCLENG operand cannot be coded with the CURSOR, EVENT, LOCTEXT, SCAN, or SCANCNTR operands or used with a test under mask condition.

Format: For the LOCLENG operand, you can code one of the following values:

*	Specifies that the length of the LOC operand data is all the data available in the specified area.
integer	Specifies that the length of the LOC operand data is the integer value (1-32767) specified.
cntr	Specifies that the length of the LOC data is the counter specification value (0-32767) specified.

Default: None.

LOG=(*data*)

Function: Specifies the data to be written in a LOG record to the log data set when the test is made.

Note: This operand is valid only if a THEN operand is specified.

Format: 1 to 50 bytes of EBCDIC data enclosed within the parentheses. To enter a single quote or a parenthesis as data, enter two of the characters. You cannot continue the data to another statement. Also, you cannot code data field options for this operand.

Default: None.

RESP=NO

Function: Specifies that if the THEN action is taken for this IF statement, WSim will not generate an automatic SNA response for this message. Instead, WSim will set up the TH and RH for the normal response and go to message generation to get the response data from the message generation deck. The largest response that can be built is 256 bytes long. A response will always be sent after returning from message generation.

Note: This operand is valid only if a THEN operand is also specified on the same IF statement.

Format: NO

Default: None. This operand is optional. If not coded, WSim automatically builds the SNA response.

Note: This operand is ignored for non-SNA terminals. However, the THEN action will be performed for all terminal types. Therefore, code this operand only on logic tests evaluated for SNA terminals and devices.

SCAN={YES|*value*}

Function: Specifies whether the data is to be scanned sequentially for the data specified in the AREA, TEXT, or UTBL operand. When scanning is specified, the data is searched starting at the location specified in the LOC operand or at the beginning of the text specified by the LOCTEXT operand. The data is scanned and compared with the character string as specified by the AREA or TEXT operand for LOC or the TEXT operand for LOCTEXT. If data that meets the comparison condition is found before the specified number of positions have been scanned, the THEN action is taken. Otherwise, the ELSE action is taken.

Note: Due to possible performance degradation, use this option with care.

Format: For the SCAN operand, you can code one of the following values:

YES Specifies that scanning continues until the condition is met or the end of the data is reached.

value Specifies that scanning continues until the condition is met, the number of positions specified by *value* has been scanned, or the end of data is reached. *value* can be any integer from 1 to 32767 or a counter specification whose value is within this range.

Default: None. If this operand is omitted, no scanning is performed. The LOCLENG operand cannot be coded with the SCAN operand.

SCANCNTR=*cntr*

Function: Specifies a counter to be set to the offset of text data that caused the logic test condition to be met. If text is being compared and the IF condition is met, the specified counter is assigned the value of the offset into the save area, user area, buffer, or data stream which satisfies the condition if the LOC operand was coded or the value of the offset into the LOCTEXT data if the LOCTEXT operand was coded. If the IF condition was not met, the value of the counter is unchanged.

Note: If you code this operand, do not code the CURSOR, EVENT, or LOCLENG operands. Also, do not code the SCANCNTR operand if the LOC operand specifies a switch or counter to be tested or the LOCTEXT operand specifies an integer or counter to be tested. If SCANCNTR and UTBLCNTR are both coded and the same counter is specified on both operands, the SCANCNTR operand will take precedence if a match is found.

Format: The value coded for *cntr* can be any of the counter specifications as defined by the TEXT operand.

Default: None. This operand is optional.

SNASCOPE={ALL|LOC|REQ|RSP}

Function: Specifies which SNA flows to test for the data specified in the AREA, TEXT, or UTBL operand.

Format: For the SNASCOPE operand, you can enter one of the following values:

ALL Specifies that logic testing is to performed on both SNA request and response flows.

LOC Specifies that logic testing is to be performed based on the LOC or LOCTEXT operand specification. See Chapter 10, “Data locations,” on page 209 for more information.

REQ Specifies that logic testing is to be performed on the SNA request flows only.

RSP Specifies that logic testing is to be performed on the SNA response flows only.

Default: LOC

Note: This operand is ignored for non-SNA devices.

TEXT={RESP|*cntr***|(***data***)|'***xx***'|***integer***}**

Function: Specifies the text value for which the test is to be made.

Format: For the TEXT operand, you can code one of the following options:

RESP The data to be used in the comparison was specified using the RESP operand on the previous TEXT statement. If no RESP was coded on the previous TEXT statement, a null value is used for the logic test.

cntr The value of a counter is to be used in the comparison. The valid values for *cntr* are NSEQ, LSEQ, TSEQ, DSEQ, NC*n*, LC*n*, TC*n*, and DC*n*, where *n* is an integer from 1 to 4095. These values are explained under the LOC operand on this statement. The TEXT operand can specify a counter value only if the LOC operand specifies a counter value or the LOCTEXT operand specifies a counter or integer value.

(*data***)** The data coded within the parentheses is to be used as the comparison data. When comparing with specified data, you can enter hexadecimal data within the parentheses by enclosing the digits within single quotes.

Two digits compose one hexadecimal character. For example, TEXT=(ABC) will generate a comparison for the three characters ABC. TEXT=('AB'CD) is a comparison for three bytes including one hexadecimal character of AB and two EBCDIC characters of CD. A maximum of 32767 characters will be used for comparison. To enter a single quote, parenthesis, or dollar sign ($) as data, enter two of the characters. You can also continue the data on the next statement.

For information on the data field options that you can use, see Chapter 9, “Data field options,” on page 199.

'*xx***'** A test under mask on a byte of the data is executed using the mask specified by two hexadecimal digits within single quotes. The bits of the mask correspond one for one with the bits of the byte of data. A mask indicates that the corresponding bits in the byte of data are tested. If all bits tested are set to one, the THEN action is taken. Otherwise, the ELSE action is taken.

integer A 1- to 10-digit integer ranging from 0 to 2147483647 is to be used in the comparison. This format is valid when the LOC operand specifies a counter or the LOCTEXT operand specifies a counter or integer value. The value will be compared to this text numeric value.

Default: None. Either the TEXT, UTBL, or AREA operand is required for all tests, except when you code the LOC operand for switch testing or code the EVENT operand. You cannot code the TEXT operand if the LOC operand specifies switch testing, or you code the AREA or LENG operand.

THEN=*action*

Function: Specifies the action to be taken if the specified condition was met, the switches tested were on, or the tested event was complete.

Note: When the THEN operand is omitted and the test condition is met, no action is taken (all indicators and message generation paths are left as they were before the IF statement was encountered).

Format: For the THEN operand, you can code one of the following options:

B*name-label*	CONT	NSW(ON)	SW(ON)
B*name*	WAIT	NSW(OFF)	SW(OFF)
	DLYCNCL	NSW*n*(ON)	SW*n*(ON)
C*name-label*	QUIESCE	NSW*n*(OFF)	SW*n*(OFF)
C*name*	RELEASE		
		TSW(ON)	WAIT(*event*)
E*name-label*	RETURN	TSW(OFF)	POST(*event*)

E*name*	IGNORE	TSW*n*(ON)	RESET(*event*)
	ABORT	TSW*n*(OFF)	SIGNAL(*event*)
			QSIGNAL(*event*)
VERIFY[-(*data*)]			

The following list describes each of the options.

Option	Description
ABORT	Causes the current message generation deck to be stopped, any active logic tests to be deactivated, and the next message generation deck to be selected according to normal PATH selection rules.
B	Indicates a branch to another location within the message generation decks and resets the WAIT indicator. *name* specifies the name of the message generation deck that is the branch target. *name-label* specifies a label in the named message generation deck.
C	Indicates a call to another location within the message generation decks and resets the WAIT indicator. *name* specifies the name of the message generation deck that is the call target. *name-label* specifies a label in the named message generation deck. Call differs from branch in that a return pointer is saved to allow message generation to return to the point of the call.
CONT	Specifies that message generation is to continue in the current message generation deck.
DLYCNCL	Cancels any active or pending intermessage delay.
E	Specifies an immediate execution of the statements beginning at the specified message generation deck location. *name* specifies the name of the message generation deck that is the execute target. *name-label* specifies a label in the named message generation deck. The statement types that can be executed with this action are BRANCH, CALC, CANCEL, DATASAVE, DEACT, EVENT, IF (other than WHEN=IMMED), LABEL, LOG, MONITOR, MSGTXT, ON, OPCMND, SET, SETSW, SETUTI, WTO, and WTOABRHD. Execution stops when any other statement type is encountered. This action is separate from, and does not affect, the normal message generation flow and does not reset the WAIT indicator. It takes place before the evaluation of any subsequent IF statements. It does not affect subsequent actions, and can itself be taken, regardless of whether or not any other action has already been taken.
IGNORE	Specifies that no action is to take place. In addition, no other actions of the same class (such as CONT, RETURN, and WAIT) will take place for the message being tested, even if a subsequent logic test condition is met.
NSW(ON)	Sets all network switches on up to the maximum number referenced.
NSW(OFF)	Clears all network switches up to the maximum number referenced.
NSW*n*(ON)	Sets on the indicated network switch, where *n* is an integer from 1 to 4095.
NSW*n*(OFF)	Clears the indicated network switch, where *n* is an integer from 1 to 4095.
POST(*event*)	Specifies that the named *event* is to be posted.
QSIGNAL(*event*)	Specifies that the named *event* is to be signaled, but only for the device which issued the QSIGNAL.

Option	Description
QUIESCE	Prohibits message generation until a release operation is performed. A quiesced device can receive messages and respond negatively to polls while not generating any data messages.
RELEASE	Specifies that a quiesced device is to proceed in message generation.
RESET(*event*)	Specifies that the named *event* is no longer to be considered posted.
RETURN	Specifies a return to message generation after the point of the last call. If no CALL statements have been issued, a message trace (MTRC) record is written to the log data set, and the action is ignored.
SIGNAL(*event*)	Specifies that the named *event* is to be signaled.
SW(ON)	Sets all device switches on up to the maximum number referenced.
SW(OFF)	Clears all device switches up to the maximum number referenced.
SW*n***(ON)**	Sets on the indicated switch for the device, where *n* is an integer from 1 to 4095.
SW*n*(OFF)	Clears the indicated switch for the device, where *n* is an integer from 1 to 4095.
TSW(ON)	Sets all terminal switches on up to the maximum number referenced.
TSW(OFF)	Clears all terminal switches up to the maximum number referenced.
TSW*n***(ON)**	Sets on the indicated switch for the terminal, where *n* is an integer from 1 to 4095.
TSW*n***(OFF)**	Clears the indicated switch for the terminal, where *n* is an integer from 1 to 4095.
VERIFY[-(*data***)]**	Causes a VRFY log record to be logged to the log data set for this network. If *data* is coded, it will be included in the VRFY log record. The value for *data* can be one or more characters enclosed within parentheses. Although *data* may be longer than 50 characters, no more than 50 characters will be included in the VRFY record. To enter a single quote or parenthesis as data, enter two of the characters. You can continue the data and include any data field options. The Loglist Utility (refer to , SC31-8947) can format these VRFY log records into Verification Reports.
WAIT	Prohibits message generation.

Default: None. You must code either the THEN or ELSE operand.

Notes:

- When the THEN operand is omitted, the DATASAVE, DELAY, LOG, and RESP operands are not allowed on the IF statement.
- When a BRANCH, CALL, RETURN, QUIESCE, RELEASE, or CONT action is taken, the WAIT condition is reset to OFF.
- Resetting the WAIT condition with a CONT, BRANCH, CALL, QUIESCE, or RELEASE action does not reset the *event wait* condition.
- See , SC31-8945 for more information on actions executed when multiple IF statements are coded.
- The event name specified in WAIT(*event*), POST(*event*), SIGNAL(*event*), QSIGNAL(*event*), and RESET(*event*) can either be explicitly coded as a name (up to eight alphanumeric characters), or it can reference a save area or user area for the name. These are specified as:
  – N±*value*

- N*s+value*
- U±*value*
- *s+value*

Reference the EVENT statement or EVENT operand for more information on these operand values.

TYPE=*type*

Function: Specifies the type of terminal for which this IF statement is to be evaluated.

Note: Chapter 11, "Terminal, device, and logical unit types," on page 211 contains definitions of the terminal and device types.

Format: For the TYPE operand, you can code one of the following terminal types:

LU0	LU1	LU2	LU3	LU4
LU6	LU62	FTP	LU7	TN3270
STCP	SUDP	TNNVT	TN3270E	TN3270P
TN5250				

UTBL=*name*

Function: Specifies the number of the user table, as defined by a UTBL statement or name coded on the MSGUTBL statement, containing the entries to be compared with the data defined by the LOC or LOCTEXT operand.

Format: An integer from 0 to 255 or from one to eight alphanumeric characters with the first character being alphabetic.

Default: None. This operand is optional.

Note: If you code the UTBL operand, do not code the AREA, LENG, EVENT, and TEXT operands. Also, you cannot code the UTBL operand if the LOC operand specifies a switch or counter to be tested or the LOCTEXT operand specifies a counter or integer value to be tested.

UTBLCNTR=*cntr*

Function: Specifies a counter to be set to the index of the user table entry that caused the logic test condition to be met. If the logic test condition was not met, the value of the counter is unchanged.

Note: If you code the UTBLCNTR operand, do not code the AREA, EVENT, LENG, and TEXT operands. Also, you cannot code the UTBLCNTR operand if the LOC operand specifies a switch or counter to be tested or the LOCTEXT operand specifies a counter or integer value to be tested. If SCANCNTR and UTBLCNTR are both coded and the same counter is specified on both operands, the SCANCNTR operand will take precedence if a match is found.

Format: The value of *cntr* can be any of the counter specifications as defined by the TEXT operand.

Default: None. This operand is optional.

Note: The index of the first entry in a user table is zero.

WHEN={IN|OUT}

Function: Specifies when the logic test is to be evaluated.

Format: For the WHEN operand, you can code one of the following values:

IN Specifies that the logic test will be evaluated when WSim receives the data.

OUT Specifies that the logic test will be evaluated when WSim transmits the data.

Default: IN

INCLUDE - message text definition statement

```
[name] INCLUDE deck[,...]
```

Function

The INCLUDE statement specifies message generation decks that are to be used by the network during the simulation run. You can code the INCLUDE statement multiple times. This statement is optional.

Note: The INCLUDE statement is only required for the purpose of naming message generation decks not referenced elsewhere in the network definition. For example, a message generation deck that is not initially used, but needs to be available for use in an A (Alter) operator command, must be named with an INCLUDE statement.

Where

name

Function: Specifies the symbolic name of this statement (for user information only).

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

deck[,...]

Function: Specifies the name of a message generation deck to be included for message generation.

Format: A 1- to 8-character name that conforms to standard JCL naming conventions.

Default: None. You must specify at least one message generation deck.

Note: You can code all of the referenced message generation decks on a single INCLUDE statement or divide them between multiple statements.

MSGDISK - control block paging data set definition statement

```
[name] MSGDISK {DDNAME=ddname}
               {SPACE=blocks[,UNIT={unit|SYSDA}]}
               [,WORKSET=worknum]
```

Function

The MSGDISK statement defines the work data set for the paging of control blocks representing message generation decks, User Tables, and IF statements. It also specifies the amount of virtual storage to be used as a working set for this paging. The MSGDISK statement is optional.

Where

name

Function: Specifies the symbolic name for this statement (for user information only).

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

DDNAME=*ddname*

Function: Specifies the name on a DD statement included in the execution JCL, which defines the work data set for the paging of control block data.

Format: A 1- to 8-character name that conforms to standard JCL DD statement naming conventions. The value of *ddname* should be unique for each concurrently initialized network.

Default: None. You must code either the DDNAME operand or the SPACE operand.

Note: For OS/VS, the DD statement named by *ddname* should have the following format:

```
//ddname DD SPACE=(4096,blocks),UNIT=unit
```

where:

blocks	Specifies the number of 4096-byte blocks for which space is to be allocated. It can be an integer from 1 to 16777215. The number of blocks required can be obtained from message ITP659I, which was produced by an earlier preprocessor run for the network.
unit	Specifies the unit name to be used in allocating the work data set.

SPACE=*blocks***[,UNIT={***unit***|SYSDA}]**

Function: Specifies the amount of space to be allocated for the work data set, and optionally specifies the unit to be used in allocating the work data set.

Format: The values for *blocks* and *unit* are the same as described for the DDNAME operand.

Default: None. You must code either the DDNAME operand or the SPACE operand.

Note: If you code the SPACE operand, the work data set will be dynamically allocated, and the UNIT operand is optional with a default value of SYSDA.

The SPACE and UNIT operands are valid only for an OS/VS2 MVS system. However, the preprocessor will not flag them as errors if it is executing on any other system. WSim will flag them as errors if the network is initialized on another system.

Note: You must code the SPACE operand instead of the DDNAME operand when using the WSim/ISPF Interface.

WORKSET=*worknum*

Function: Specifies the number of 4096-byte blocks to be kept in virtual storage.

Format: An integer from 2 to 4095.

Default: The number of blocks required to hold the first 4096-byte block of each message generation deck plus one.

Note: You can code the DDNAME operand or the SPACE operand, but not both.

No work data set allocation or writing of blocks to disk will be done during a preprocessor run.

NTWRK - network definition statement

```
                 Network Control Operands
      name NTWRK [,CNTRS={integer|3}]
                 [,CNTRSEED={integer|7935629}]
                 [,CONRATE={YES|NO}]
                 [,DELYSEED={integer|9104901}]
                 [,EMTRATE=(rate,interval)]
                 [,EXIT=member]
                 [,HEAD={'chars'|'WSim INTERVAL REPORT'}]
                 [,INEXIT=member]
                 [,INFOEXIT=member]
                 [,INHBTMSG=(integer,integer-integer,WTO,ACT)]
                 [,INXEXPND={YES|NO}]
                 [,ITIME={integer|2}]
                 [,NAMEHASH={integer|10}]
                 [,NCTLEXIT=member]
                 [,NETEXIT=member]
                 [,NETUSER={integer|0}]
                 [,OPTIONS=(option1[,...])]                 .
                 [,OUTEXIT=member]
                 [,PATHSEED={integer|1532001}]
                 [,REPORT={FULL|LINE|RATE|NONE}]
222              [,SCAN=(x,y[,z[,sname]])]
222              [,SEQ=(integer|0)]
                 [,STIME={integer|0}]
                 [,TEXTSEED={integer|3841995}]
                 [,UCMDEXIT=member]
                 [,UTBLSEED={integer|5736539}]
                 [,UTI={integer|0}]
                 [,UXOCEXIT=member]
```

Function

The NTWRK statement is required. It performs the following functions:

- Provides a name for controlling the network with operator commands
- Specifies characteristics that apply to the network as a whole
- Specifies operands that carry default values to lower level statements.

See Table 4 on page 17 for operands that can be coded on this statement to provide defaults for lower-level statements and where these operands are defined.

Where

name

Function: Specifies the symbolic name used to identify the network on reports or for operator commands. This is also the member name in the INITDD data set.

Format: A 1- to 8-character name that conforms to standard JCL member name conventions. The first character of *name* must not be a numeric character.

Default: None. This field is required.

CNTRS={*integer*|3}

Function: Specifies the largest counter allocated for each line, terminal, and device, as well as the largest counter allocated for the entire network, when user exits are specified. If user exits are not specified, this operand is ignored and the number of counters allocated will be equal to the largest counter referenced in the network. If user exits are specified, the number of counters allocated will be equal to either the value specified on the CNTRS= operand or the number of counters referenced in the network, whichever is larger.

If user exits are used, the value returned via the interface routine, word 2 request X'55' will represent the number of counters allocated, which may or may not be the value coded on the CNTRS= operand.

Note: The largest counter allocated will be 3.

Format: An integer from 3 to 4095.

Default: 3

CNTRSEED={*integer*|7935629}

Function: Specifies the seed value to be used when generating random numbers for the SET statement.

Format: An integer from 1 to 2147483647.

Note: The default value was selected to provide a uniform random distribution. Care should be exercised in selecting other values.

Default: 7935629

CONRATE={YES|NO}

Function: Specifies whether or not the interval report message rates are to be printed at the operator console each time an interval report is printed for the network.

Format: YES or NO.

Default: NO

DELYSEED={*integer*|9104901}

Function: Specifies the seed value to be used when generating numbers for calculating average delays and selecting delay values from rate tables.

Format: An integer from 1 to 2147483647.

Note: The default value was selected to provide a uniform random distribution. Care should be exercised in selecting other values.

Default: 9104901

EMTRATE=(*rate*,*interval*)

Function: Specifies a message transfer rate (messages transmitted by WSim) and an adjustment interval through which WSim automatically adjusts the UTI to maintain the desired rate for an entire network. Refer to , SC31-8945 for a discussion of automatic UTI adjustment.

Format: For the EMTRATE operand, you can code the following values:

rate Specifies the number of messages per minute to be transmitted by the network, where *rate* is an integer from 0 to 65535.

interval
Specifies the duration, in seconds, of the interval at which the UTI is to be adjusted to affect the desired rate, where *interval* is an integer from 0 to 65535.

Default: None. This operand is optional.

EXIT=*member*

Function: Specifies the member (network-level user exit load module) in the load library, or a library concatenated to it, which is to be loaded with the network and is to gain control each time a message is transmitted or received at a terminal or device.

Note: If you code the EXIT operand, you cannot code the NETEXIT, INEXIT, OUTEXIT, or NCTLEXIT operands. Refer to , SC31-8950 for more information on user exit facilities.

Format: A 1- to 24-character name that conforms to standard JCL member naming conventions.

Default: None. This operand is optional.

HEAD={'*chars***'|'WSim INTERVAL REPORT'}**

Function: Specifies the heading which is printed on each interval report for the network.

Format: 1 to 24 characters enclosed in single quotes.

Default: "WSim INTERVAL REPORT"

INEXIT=*member*

Function: Specifies the member (input user exit load module) in the load library, or a library concatenated to it, which is to be loaded with the network and is to gain control each time a message is received at a terminal or device. Refer to , SC31-8950 for more information on user exit facilities.

Format: A 1- to 8-character name that conforms to standard JCL member naming conventions.

Default: None. This operand is optional.

INFOEXIT=*member*

Function: Specifies the member (informational user exit load module) in the load library, or a library concatenated to it, which is to be loaded with the network and is to gain control each time a WSim-generated informational message is logged on the log data set.

Note: Refer to , SC31-8950 for more information on user exit facilities.

Format: A 1- to 8-character name which conforms to standard JCL member naming conventions.

Default: None. This operand is optional.

INHBTMSG=(*integer*,*integer-integer*,WTO,ACT)

Function: Specifies the messages (ITP*xxx*I) that are not to be displayed on the console and not written to the log data set.

Format: For the INHBTMSG operand, you can code one or more of the following values (separated by commas):

integer Specifies a particular message number to be inhibited. For a list of messages that cannot be inhibited, see the following note:

Note: The following messages cannot be inhibited:

WSim Console Messages (1-399)

001	066-067	098-099	180-188
003-005	069-073	101-106	190-201
007-008	075	108-112	207-224
011	078-088	114-120	
017-021	090	164-167	
023-028	093	169-173	
030-062	095	176	

WSim Log Data Messages (401-499)

401-402	419-420	459-466
406	433-440	471-472
408-409	442	477
411-413	457	491
415-417		

integer-integer
Specifies a range of messages to be inhibited. Even if a particular message within the range cannot be inhibited, no error checking will be done.

WTO Specifies that the Write-to-Operator statements are to be inhibited (ITP113I, ITP137I).

ACT Specifies that the activation messages are to be inhibited (ITP089I, ITP091I, ITP092I, ITP094I, ITP107I, ITP174I, ITP175I).

Refer to , SC31-8951 for descriptions of these messages.

Default: None. This operand is optional.

INXEXPND={YES|NO}

Function: Specifies whether data expansion is allowed when an input user exit is called. If you specify INXEXPND=YES, input data is moved to an area that allows data expansion up to the length specified by the BUFSIZE operand before the input user exit is called. The input exit can then adjust the length and contents of the data up to that size. The size of the buffer available is passed as one of the parameters to the input exit. Refer to , SC31-8950 for more information about input user exits.

Format: YES or NO.

Default: NO

ITIME={*integer*|2}

Function: Specifies the time in minutes between network interval reports.

Format: An integer from 1 to 240 (4 hours).

Default: 2

NAMEHASH={*integer***|10}**

Function: This operand is used to determine the size of the table used in locating a particular name within the network. For very large networks, increasing the value of this operand may improve performance in cases where WSim must locate a particular resource, such as in processing operator commands. The number of entries in the table is two raised to the value coded for NAMEHASH.

Format: An integer from 2 to 18. The recommended size of the table is 30% more entries than the number of names in the network.

Default: 10. This provides for 1024 entries in the table.

NCTLEXIT=*member*

Function: Specifies the member (network control user exit load module) in the load library, or a library concatenated to it, which is to be loaded with the network and is to gain control each time a network is initialized, reset, or cancelled. Refer to , SC31-8950 for more information on user exit facilities.

Format: A 1- to 8-character name that conforms to standard JCL member naming conventions.

Default: None. This operand is optional.

NETEXIT=*member*

Function: Specifies the member (network-level user exit load module) in the load library, or a library concatenated to it, that is to be loaded with the network and is to gain control in all situations that INEXIT, OUTEXIT, NCTLEXIT, and UCMDEXIT are invoked unless overridden for that situation by one of the more specific exit operands.

Note: Code either the EXIT operand or the NETEXIT operand, but not both. Refer to , SC31-8950 for more information on user exit facilities.

Format: A 1- to 8-character name that conforms to standard JCL member naming conventions.

Default: None. This operand is optional.

NETUSER={*integer***|0}**

Function: Defines an area of storage that can be used as a work area by user exits or any device or logical unit in the network that is capable of message generation.

Format: An integer from 0 to 32767.

Default: 0

Note: If *integer* is not a multiple of eight bytes, this value is rounded up to the next multiple of eight bytes.

OPTIONS=(*option***[,...])**

Function: Specifies various options that are to be used in this network.

Format: For the OPTIONS operand, you can code one or more of the following keywords (separated by commas):

CONRATE

Specifies that the interval report message rates are to be printed at the operator console each time an interval report is printed for the network. Specifying this option is the same as coding the CONRATE=YES operand on the NTWRK statement.

DEBUG
Specifies that the network is being executed in debug mode, which causes the following information to be written to the log data set:
- CPI-C trace data at the VTAM APPC API level.
- TCP/IP trace data at the negotiations level.

MONCMND
Specifies that the operator commands executed from the message generation deck will be monitored on the console as they are processed.

Default: If you do not code this operand, the DEBUG and MONCMND options will not be active, and the CONRATE option will default to the CONRATE operand setting or to its default of CONRATE=NO. If you specify both the CONRATE operand and the CONRATE option, the setting processed last will take effect.

OUTEXIT=*member*
Function: Specifies the member (output user exit load module) in the load library, or a library concatenated to it, which is to be loaded with the network and is to gain control each time a message is transmitted at a terminal or device. Refer to , SC31-8950 for more information on user exit facilities.

Format: A 1- to 8-character name that conforms to standard JCL member naming conventions.

Default: None. This operand is optional.

PATHSEED={*integer***|1532001}**
Function: Specifies the seed value to be used when generating random numbers for selecting PATH entries according to the distribution specified on a DIST statement.

Format: An integer from 1 to 2147483647.

Note: The default value was selected to provide a uniform random distribution. Care should be exercised in selecting other values.

Default: 1532001

REPORT={FULL|LINE|RATE|NONE}
Function: Specifies the type of interval report to be printed for this network.

Note: This operand has no effect on the information printed on the end report when the network is canceled.

Format: For the REPORT operand, you can code one of the following values:

FULL Specifies that the entire report, including terminal and device statistics, line totals, cumulative totals, and rates, will be printed.

LINE Specifies that the report will include only line totals, cumulative totals, and message rates.

RATE Specifies that the report will include only the network totals and message rates.

NONE
Specifies that no interval report is to be printed.

Default: FULL

SCAN=(*x*,*y*[,*z*[,*sname*]])

Function: Specifies the inclusion of the Scan/Display/Recovery option for the network. For more information on these functions, refer to , SC31-8945.

Format: For the SCAN operand, you can code the following values:

- *x* Specifies the interval in minutes between the inactive terminal reports, where *x* is an integer from 0 to 255. If you code 0 for this value, no reports will be generated.
- *y* Specifies the time of inactivity in minutes before a terminal is listed as inactive, where *y* is an integer from 0 to 255. If you code 0 for this value, the terminal will not become inactive, and no inactivity report will be generated.
- *z* Specifies the number of minutes to delay after detecting a terminal is inactive before invoking automatic terminal recovery. If you omit this value, automatic terminal recovery is not invoked. If you code 0 for this value, automatic terminal recovery is invoked as soon as the terminal is detected as inactive.
- *sname* Specifies a message generation deck to be used as a substitute message generation deck for those decks that cause terminals to enter automatic terminal recovery. If you code this operand and a terminal enters automatic terminal recovery, a delete path entry function will automatically be performed. The message generation deck named in the current path entry will be deleted, and *sname* will be substituted for it. The path is changed for all terminals that reference it. If you do not code *sname*, or if *sname* has itself been deleted, no automatic delete will be performed. Refer to , SC31-8948 for more information on the delete and reinstate function of the A (Alter) operator command.

Default: None. This operand is optional.

Note: You can change the values for *x*, *y*, and *z* by using the A (Alter) operator command.

222 **SEQ=(*integer*|0)**
222 **Function:** Specifies the initial value for DSEQ, the device counter for the
222 network.

STIME={*integer*|0}

Function: Specifies the startup delay in seconds for the VTAM applications, APPC LUs, and TCP/IP connections. For example, if STIME=4, the first such resource will start at 4 seconds, the second at 8, and the third at 12. The timing starts when you enter the START command for the network.

Format: A 1- to 3-digit integer from 0 to 999.

Default: 0. All such resources will be started when the network is started.

TEXTSEED={*integer*|3841995}

Function: Specifies the seed value to be used when generating random numbers to be included in statement data with the RNUM special option.

Format: An integer from 1 to 2147483647.

Note: The default value was selected to provide a uniform random distribution. Care should be exercised in selecting other values.

Default: 3841995

UCMDEXIT=*member*

Function: Specifies the member (operator command user exit load module) in the load library, or a library concatenated to it, which is to be loaded with the network and is to gain control each time the operator enters a $ (User Exit) operator command. Refer to , SC31-8950 for more information on user exit facilities.

Format: A 1- to 8-character name that conforms to standard JCL member naming conventions.

Default: None. This operand is optional.

UTBLSEED={*integer***|5736539}**

Function: Specifies the seed value to be used when generating random numbers for selecting entries from a user table (UTBL), either randomly or according to the distribution specified on a UDIST statement.

Format: An integer from 1 to 2147483647.

Note: The default value was selected to provide a uniform random distribution. Care should be exercised in selecting other values.

Default: 5736539

UTI={*integer***|0}**

Function: Specifies the network level user time interval (UTI) to be used in computing intermessage delays or think times for the simulated devices.

Format: One to five digits specifying the number of 0.01 second intervals, where *integer* has a maximum of 65535.

Default: 0 (no delay)

Note: This value can be overridden by specifying IUTI on a lower level device. See , SC31-8945 for more information.

UXOCEXIT=*member*

Function: Specifies the member (user exit operator command and user exit load module) in the load library, or a library concatenated to it, that is to be loaded with the network and is to gain control each time an operator command that was issued from the User Exit Interface Routine using the function that requests notification of the end of command execution ends. Refer to , SC31-8950 for more information on user exit facilities.

Format: A 1- to 8-character name that conforms to standard JCL member naming conventions.

Default: None. This operand is optional.

NTWRKLOG - network log data set statement

```
[name] NTWRKLOG {DDNAME=ddname}
                [,NCP={integer|5}]
```

Function

The NTWRKLOG statement specifies a separate log data set for the network. This statement is optional.

Note: Only three NTWRKLOG statements are supported when you use the WSim/ISPF Interface.

Where

name

Function: Specifies the name for this statement.

Format: A 1- to 8-character name.

Default: None. This field is optional.

DDNAME=*ddname*

Function: Specifies the name of a DD statement in the execution JCL that defines the log data set for the network.

Format: A 1- to 8-character name that conforms to standard JCL naming conventions.

Default: None. You must code the DDNAME operand.

NCP={*integer***|5}**

Function: Specifies the number of buffers used to write to the log data set. The length of each buffer is determined by BLKSIZE.

Format: An integer from 2 to 255.

Default: 5

Note:
In OS systems, NCP can be specified on the DD statement, and if so, it will override the value specified on the NTWRKLOG statement. In general, the precedence order is as follows:

1. DD statement DCB suboperand NCP (for OS systems)
2. NTWRKLOG NCP operand
3. NCP execution parameter
4. 5 (default).

Note: If a network does not specify an NTWRKLOG data set to be used, all data for that network will be logged to a general log data set. If it is not possible to associate some data with a specific network, the data will be logged to the general log data set.

PATH - message generation sequence statement

```
name PATH deck[,...]
        [,CYCLIC={YES|NO}]
```

Function

The PATH statement is required. It specifies the sequence in which the message generation decks are to be referenced during the simulation run.

Where

name

Function: Specifies the name to be used in a configuration definition statement PATH operand to reference this PATH statement.

Format: From one to eight alphanumeric characters.

Default: None. This field is required.

*deck***[,...]**

Function: Specifies the name of a message generation deck to be referenced on this path.

Format: A 1- to 8-character name that conforms to standard JCL naming conventions.

Default: None. You must code at least one message generation deck.

Note: When a terminal references a PATH statement with CYCLIC=NO, the message generation decks will be used in the sequence specified on the PATH statement, unless there is a DIST statement corresponding to this PATH statement. When the final message generation deck selected by the PATH statements is completely processed, the processing begins again with a message generation deck selection from the first PATH statement and the cycle is repeated.

CYCLIC={YES|NO}

Function: Specifies whether an individual path is cyclic. For a PATH with CYCLIC=YES, the first terminal or device to select that path will select the first entry, the second will select the second entry, and the third will select the third entry. When the last entry has been selected, the next terminal to select the path will select the first entry.

Format: YES or NO.

Default: NO

Note: You can code this operand either before all message generation deck names or after all message generation deck names, but you cannot code it between message generation deck names. If you code a DIST statement that corresponds to a cyclic PATH statement, the DIST statement will be ignored in path selection.

RATE - rate table statement

```
integer RATE member
```

Function

The RATE statement specifies a rate table member in the partitioned data set described by the RATEDD DD statement in the execution JCL. It also specifies an integer name which can be referenced by delay operands on other statements.

The RATE statement is optional, but it can be coded more than once.

Where

integer

Function: Specifies the number used by the DELAY operand to reference this member.

Format: An integer from 0 to 255.

Default: None. This number is required.

member

Function: Specifies the member that WSim will load as the rate table.

Format: A 1- to 8-character name that conforms to standard JCL member naming conventions.

Default: None. This operand is required.

RN - random number statement

```
integer RN [HIGH={integer|100}]
           [,LOW={integer|0}]
```

Function

The RN statement defines the lower and upper limits for generation of a random number to be inserted into a data field or used as a delay value. It is referenced by the RNUM data field option or an R delay value.

The RN statement is optional, but it can be coded more than once.

Where

integer

Function: Specifies the number to be used to reference this statement in message generation.

Format: An integer from 0 to 255.

Default: None. This number is required.

HIGH={*integer*|100}

Function: Specifies the upper limit of the random number to be generated.

Note: The upper limit must be greater than the LOW specification.

Format: An integer from 1 to 2147483647.

Default: 100

LOW={*integer*|0}

Function: Specifies the lower limit of the random number to be generated.

Note: The lower limit must be less than the HIGH specification.

Format: An integer from 0 to 2147483646.

Default: 0

Note: The HIGH and LOW values are included within the range of random numbers.

SIDEEND - side information table end statement

```
[name] SIDEEND
```

Function

The SIDEEND statement defines the end of a CPI-C side information table.

Where

name

Function: Specifies the symbolic name of this statement (for user information only).

Format: 1 to 8 alphanumeric characters.

Default: None. This field is optional.

SIDEENT - side information table entry statement

```
[name] SIDEENT [DESTNAME=name]
               [,LUNAME=name]
               [,MODENAME=name]
               [,TPNAME={name|'name'|(name)|('name')}]
```

Function

The SIDEENT statement defines and names one symbolic destination name in a CPI-C side information table.

Where

name

Function: Specifies the symbolic name of this statement (for user information only).

Format: 1 to 8 alphanumeric characters.

Default: None. This field is optional.

DESTNAME=*name*

Function: Specifies the CPI-C symbolic destination name that is being defined.

Format: A 1- to 8-character name from character set 01134[1].

Default: None.

Note: The DESTNAME must be unique within the table.

LUNAME=*name*

Function: Specifies the LU name to be associated with the symbolic destination.

1. For the definition of character sets 01134 and 00640, refer to *SAA Common Programming Interface Communications Reference*.

Format: A 1- to 17-character name from character set 01134[1]. A fully-qualified name is specified as a network ID (up to 8 characters), followed by a period, followed by an LU name (up to 8 characters).

Default: The DESTNAME value.

MODENAME=*name*

Function: Specifies the mode name to be associated with the symbolic destination.

Format: A 1- to 8-character name of a VTAM logon mode table (MODETAB) entry. The mode name must be from character set 01134[1].

Default: #INTER

TPNAME={*name***|'***name***'|(***name***)|('name')}**

Function: Specifies the name of a CPI-C transaction program that is to be associated with the symbolic destination.

Format: A 1- to 64-character name specified as a string of up to 64 characters or 128 hexadecimal digits. The TP name must be from character set 00640[1]. A quoted name indicates that it is a hexadecimal string.

The name can be continued across multiple lines by placing a comma after the last digit on a line and continuing with the next digit anywhere beyond column 1 on the next line. If the name is continued across multiple lines, it must be enclosed in parentheses.

Default: The DESTNAME value.

SIDEINFO - side information table begin statement

```
[name] SIDEINFO
```

Function

The SIDEINFO statement defines the beginning of a CPI-C side information table. This statement is optional.

The symbolic destination names defined in this table are known globally in the network. The definition can be overridden at the LU level by using the SIDEINFO operand on the APPCLU statement.

Where

name

Function: Specifies the symbolic name of this statement (for user information only).

Format: 1 to 8 alphanumeric characters.

Default: None. This field is optional.

UDIST - UTBL distribution statement

```
integer UDIST weight[,...]
```

Function

The UDIST statement defines a probability distribution to be used in choosing entries from a UTBL statement. The UDIST statement is referenced by the UTBL data field option.

The UDIST statement is optional but if used, it can be coded up to a maximum of 256 times.

Where

integer
: **Function:** Specifies the number to be used to identify this distribution in the TEXT statement special option field.

  Format: An integer from 0 to 255.

  Default: None. This field is required.

weight[,...]
: **Function:** The number specified assigns a relative weight to the corresponding UTBL entry. Each *weight* represents a fractional value of the total weights specified on this UDIST statement. This fractional value is determined by dividing the specified weight by the total of all weights. The probability that a particular UTBL entry will be chosen for message generation on any given selection is this fractional value.

  Format: A series of integers from 0 to 9999 separated by commas.

  You can code up to a maximum of 2000 weights. The number of weights entered must be less than or equal to the number of entries on the corresponding UTBL statement referenced in the TEXT statement special option field. The sum of the weights must be greater than zero and must not exceed 9999.

  Default: None. You must code at least one weight.

UTBL - user data table statement

```
integer UTBL {(entry)[,...]}
             {member}
```

Function

The UTBL statement builds a table of user data entries that can be inserted in data fields. The UTBL statement is optional, but if used, can be coded more than once. MSGUTBLs can be referenced directly using the name field value coded on an MSGUTBL statement.

Where

integer
: **Function:** Specifies the number used to identify this table. This number is referenced by the UTBL data field option.

  Format: An integer from 0 to 255.

Default: None. This number is required.

(*entry***)[,...]**
Function: Specifies the actual text to be inserted in the message being composed.

Format: Any amount of data enclosed in parentheses. However, it may be truncated during message generation. Enter hexadecimal data by enclosing the digits in single quotes. To enter single quotes or parentheses in the data, enter two of the characters. If an entry contains more data than can be specified on one record, it can be continued by coding data through column 71 and continuing in column 2 of the next record or by using the "+" continuation character.

You can code up to a maximum of 2147483647 entries for each UTBL statement. The length of the entries in the table will vary with the length of the data entered. When entering hexadecimal data into the table, make sure that line control characters are not included in the data.

Default: None. At least one entry or a member must be specified.

member
Function: Specifies the name of an MSGUTBL statement from which the user data table entries will be obtained.

Note: The MSGUTBL statements are stored as separate members of the data set named by the MSGDD DD statement.

Format: A name from one to eight alphanumeric characters.

Default: None. You must code either a member or a list of user data entries.

UTI - user time interval statement

```
name UTI integer
```

Function

The UTI statement defines an alternate UTI for resources in the network. This UTI can be referenced by specific devices with the IUTI operand or from within message generation decks. Defining more than one UTI allows you to have devices operating at different speeds within the same network.

Where

name
Function: Specifies a name to be used to reference this UTI.

Format: From one to eight alphanumeric characters.

Default: None. This field is required. The character string 'NTWRKUTI' is reserved. It is used to reference the network level UTI value.

integer
Function: Specifies user time interval (UTI) to be used in computing intermessage delays or think times. *integer* specifies the number of 0.01 second intervals.

Format: An integer from 0 to 65535.

Default: None.

Chapter 5. Defining CPI-C simulation statements

This chapter describes Common Programming Interface Communications (CPI-C) network configuration definition statements. These statements are listed in alphabetical order. For more information about the order in which you need to code these statements, see Table 3 on page 13.

When you code the network configuration definition statements, you can code the common operands for the network resources at the highest level statements and override them on lower level statements, if necessary.

Summary of operands

Table 5 lists the operands you can use in a CPI-C simulation, the statements where they appear, and other statements wherethey can be coded.

Table 5. CPI-C simulation operands

Operand	Appears on	Can also be coded on
APPLID	APPCLU	-
BUFSIZE	APPCLU	NTWRK
CNOS	APPCLU	-
CPITRACE	TP	APPCLU, NTWRK
DELAY	TP	APPLCU, NTWRK
FRSTTXT	TP	APPCLU, NTWRK
INSTANCE	TP	APPCLU, NTWRK
IUTI	TP	APPCLU, NTWRK
MAXCALL	TP	APPCLU, NTWRK
MLEN	APPCLU	NTWRK
MLOG	APPCLU	NTWRK
MSGTRACE	TP	APPCLU, NTWRK
QUIESCE	TP	APPCLU, NTWRK
PASSWD	APPCLU	-
PATH	TP	APPCLU, NTWRK
SEQ	TP	APPCLU, NTWRK
SIDEINFO	APPCLU	-
STLTRACE	TP	APPCLU, NTWRK
TPNAME	TP	-
TPREPEAT	TP	APPCLU, NTWRK
TPSTATS	TP	APPCLU, NTWRK
TPSTIME	TP	APPCLU, NTWRK
TPTYPE	TP	APPCLU, NTWRK
UCD	TP	APPCLU, NTWRK
USERAREA	TP	APPCLU, NTWRK

APPCLU - APPCLU statement

```
[name] APPCLU [BUFSIZE={integer|32767}]
              [,MLEN=integer]
              [,MLOG={YES|NO}]

              APPCLU Operands
              [,APPLID=name]
              [,CNOS=cnos data]
              [,PASSWD=password]
              [,SIDEINFO=side info data]
```

Function

The APPCLU statement defines a VTAM application program symbolic name and the password associated with that name. WSim uses this VTAM application program to simulate a logical unit on which CPI-C transaction programs run.All transaction programs defined on this logical unit use the CPI-C API to perform program-to-program communications using LU 6.2 protocols. [2] If VTAM application programs are also defined in the network, all APPCLU definitions must precede the VTAMAPPL definitions.

See Table 5 on page 59 for operands that can be coded on this statement to provide defaults for lower-level statements and where these operands are defined.

Where

name

Function: Specifies the symbolic name used to reference the resource on printed reports, with the data field options in the scripting statements, and with operator commands.

Format: From one to eight alphanumeric characters.

Default: APPLID operand value.

Note: You must code either the APPCLU *name* field or APPLID operand value.

APPLID=*name*

Function: Specifies the VTAM application program symbolic name. This name must match an entry in VTAM's configuration tables (VTAMLST) created using a VTAM APPL definition statement. The name specified is the name of the APPL statement or ACBNAME operand value coded on an APPL statement. The VTAM APPL definition in the VTAMLST must specify APPC=YES.

The APPLID name must be unique within all APPC LUs defined in the simulation.

Format: From one to eight alphanumeric characters.

Default: The APPCLU name field value.

Note: You must code either the APPCLU *name* field or APPLID operand value.

BUFSIZE={*integer***|32767}**

Function: Specifies the maximum size of the buffers which the TPs defined on

2. LU 6.2 is also called Advanced Program-to-Program Communication (APPC).

this APPC LU will use to send and receive data. Buffers that are generated dynamically by WSim will be the size specified by this operand. Buffers that are defined in scripts must be equal to or less than the size specified by this operand.

Format: An integer from 100 to 32767.

Default: 32767

CNOS=*cnos data*

```
CNOS=((LUNAME=name
     [,MODENAME=name]
     [,SESSIONS={integer|2}]
     [,CWL={integer|1}]
     [,CWP={integer|1}])
                 .
                 .
                 .
     [,(LUNAME=name
     [,MODENAME=name]
     [,SESSIONS={integer|2}]
     [,CWL={integer|1}]
     [,CWP={integer|1}])])
```

Function: Specifies the partner LU name and mode name, and their associated session limits. For any LU pairs for which no CNOS specification has been made, WSim manages sessions as required for the simulation run.

Format: For the CNOS operand, you can specify the following:

CWL={*integer* | 1}
: **Function:** Specifies the minimum number of contention winner sessions for the local LU.

  Format: An integer from 0 to 32767. The total of the contention winners for the local and partner LUs must be less than or equal to the session limit.

  Default: The value specified on the APPL statement that defines this APPC LU in the VTAMLST. If no value is specified on the APPL statement, the default is 1.

CWP={*integer* | 1}
: **Function:** Specifies the minimum number of contention winner sessions for the partner LU.

  Format: An integer from 0 to 32767. The total of the contention winners for the local and partner LUs must be less than or equal to the session limit.

  Default: The value specified on the APPL statement that defines this APPC LU in the VTAMLST. If no value is specified on the APPL statement, the default is 1.

LUNAME=*name*
: **Function:** Specifies the partner LU name.

  Format: A 1- to 17-character name from character set 01134[3].A fully-qualified name is specified as a network ID (up to 8 characters),

3. For the definition of character sets 01134 and 00640, refer to *SAA Common Programming Interface Communications Reference*.

followed by a period, followed by an LU name (up to 8 characters). The combination of LUNAME and MODENAME must be unique within the CNOS operand.

Default: None. This field is required.

MODENAME=*name*

Function: Specifies the mode to which the session limit applies.

Format: A 1- to 8-character name of a VTAM logon mode table (MODETAB) entry. The mode name must be from character set 01134[3]. The combination of LUNAME and MODENAME must be unique within the CNOS operand.

Default: If this parameter is omitted, the session limits will apply to all modes used by the pair of LUs.

SESSIONS={*integer* **| 2}**

Function: Specifies the session limit between the local LU and the partner LU for the mode specified.

Format: An integer from 1 to 32767.

Default: The value specified on the APPL statement that defines this APPC LU in the VTAMLST. If no value is specified on the APPL statement, the default is 2.

MLEN=*integer* **Function:** Specifies the maximum number of data characters to be written to the log data set for each data transfer.

Format: An integer from 100 to 32767.

Note: If this value is larger than the data transferred, the entire transmission is logged. If the specification is smaller than the amount of data transferred, only the specified length is logged.

Default: If you omit this operand, the entire transmission will be logged.
MLOG={**YES** | NO} **Function:** Specifies whether this LU will use the message logging function.

Format: YES or NO.

Default: YES PASSWD=*password***Function:** Specifies the password associated with the VTAM application program symbolic name. This password must match the PRTCT operand value coded on the corresponding APPL statement in the VTAMLST.

Format: From one to eight alphanumeric characters.

Default: APPLID operand value.

SIDEINFO=*side info data*

```
SIDEINFO=((DESTNAME=name
        [,LUNAME=name]
        [,MODENAME=name]
        [,TPNAME={name|'name'|(name)|('name')}])
                   .
                   .
                   .
        [,(DESTNAME=name
        [,LUNAME=name]
        [,MODENAME=name]
        [,TPNAME={name|'name'|(name)|('name')}]))
```

Function: Specifies a side information table for this APPC LU. Symbolic destination names that are defined at the LU level override any definitions for the same symbolic destination names specified at the network level via the SIDEINFO network definition statement.

Format: For the SIDEINFO operand, you can specify the following parameters:

DESTNAME=*name*
: **Function:** Specifies the symbolic destination name.

  Format: A 1- to 8-character name from character set 01134[4]. The DESTNAME must be unique within the SIDEINFO operand.

  Default: None. This operand is required.

LUNAME=*name*
: **Function:** Specifies the LU name to be associated with the symbolic destination. A fully-qualified name is specified as a network ID (up to 8 characters), followed by a period, followed by an LU name (up to 8 characters).

  Format: A 1- to 17-character name from character set 01134[4].

  Default: The DESTNAME value.

MODENAME=*name*
: **Function:** Specifies the mode name to be associated with the symbolic destination name.

  Format: A 1- to 8-character name of a VTAM logon mode table (MODETAB) entry. The mode name must be from the character set 01134[4].

  Default: #INTER

TPNAME={*name***|(***name***)|'***name***'|('***name***')}**
: **Function:** Specifies the transaction program name to be associated with the symbolic destination name.

  Format: A 1- to 64-character name specified as a string of up to 64 characters or 128 hexadecimal digits. A quoted name indicates that it is a hexadecimal string. The TP name must be from character set 00640[4].

  The name can be continued across multiple lines by placing a comma after the last digit on a line and continuing with the next digit anywhere beyond column 1 on the next line. If the name is continued across multiple lines, it must be enclosed in parentheses.

  Default: The DESTNAME value.

4. For the definition of character sets 01134 and 00640, refer to *SAA Common Programming Interface Communications Reference*.

TP - CPI-C transaction program definition statement

```
[name] TP [CPITRACE={MSG|VERB|VERBEND|NONE}]
          [,DELAY={A(integer)}
                  {F(integer)}
                  {R(integer1[,integer2])}
                  {T(integer)}
                  {F(1)}]
          [,FRSTTXT=deck]
          [,INSTANCE={(initial,maximum)|(1,1)}]
          [,IUTI=uti]
          [,MAXCALL={integer|5}]
          [,MSGTRACE={YES|NO}]
          [,PATH=(name,...)]
          [,QUIESCE={YES|NO}]
          [,SEQ={integer|0}]
          [,STLTRACE={YES|NO}]
          [,TPNAME={name|'name'|(name)|('name')}]
          [,TPREPEAT={YES|NO|integer}]
          [,TPSTATS={YES|NO}]
          [,TPSTIME={integer|0}]
          [,TPTYPE={CLIENT|SERVER}]
          [,UCD={YES|BOTH|NO}]
          [,USERAREA={integer|0}]
```

Function

The TP statement defines a CPI-C transaction program (TP). One or more TP statements follow the APPCLU statement and define all TPs for that LU. For each TP that is resident on a given LU, a TP statement must be specified following the APPCLU statement that defines the LU. At least one TP statement must be specified after each APPCLU. The same TP name may be specified after multiple APPCLU statements (the associated path list does not need to be the same).

Note: All of the TP statement operands except TPNAME may be coded on the APPCLU statement or the NTWRK statement.

Where

name

Function: Specifies the symbolic name used to reference the resource on printed reports, with the data field options in the scripting statements, and with operator commands.

Note: To avoid confusion when running WSim or the log data set analysis programs, all resources in a network should have unique names.

Format: From one to eight alphanumeric characters.

Default: The TPNAME operand value.

Note: You must code either the TP *name* field or TPNAME operand value.

CPITRACE={MSG|VERB|VERBEND|NONE}

Function: Specifies the level of CPI-C tracing that is written to the log data set for formatting by the loglist program.

Format: For the CPITRACE operand, you can specify one of the following values:

MSG Specifies that WSim is to log only trace messages indicating the issuance and completion of CPI-C verbs. This option enables tracing of CPI-C verb flows, states, and return codes in a very concise manner. The messages are logged in a compressed form that requires minimal space in the log dataset.

VERB Specifies that WSim is to log the complete CPI-C verbs when they are issued and when they complete. This option enables tracing of CPI-C verb flows, parameters, states and return codes in a more expansive manner than the MSG option. This option requires more space in the log dataset than the MSG option.

VERBEND
Specifies that WSim is to log CPI-C verbs only when they complete. This option provides the same type of trace as the VERB option, except that only verb completions are logged.

NONE
Specifies that no CPI-C trace information is to be logged.

Note: If CPITRACE=NONE is specified, all CPI-C message logging is inhibited, including error message logging.

Default: NONE.

DELAY={A(*integer*)|F(*integer*)|R(*integer1*[,*integer2*])|T(*integer*)|F(1)}
Function: Specifies the value to be multiplied by the active UTI to define the delay between issuance of CPI-C verbs that are delimiters.

Note: For CPI-C simulations, the calculation of delay begins immediately after the current verb has been generated. This is equivalent to THKTIME=IMMED in a VTAMAPPL simulation.

Format: For the DELAY operand, you can specify one of the following values:

A(*integer***)**
Indicates the delay is to be chosen randomly from the range 0 to 2 times the integer, where *integer* is an integer from 0 to 1073741823. The average delay will be *integer*.

F(*integer***)**
Indicates the delay value is fixed at the value specified by the integer, where *integer* is an integer from 0 to 2147483647.

R(*integer***)**
Indicates that the delay is to be chosen randomly from the range specified by an RN statement, where *integer* specifies the name field on an RN statement and is an integer from 0 to 255. For more information on the RN statement, see "RN - random number statement" on page 52.

R(*integer1***,***integer2***)**
Indicates that the delay is to be chosen randomly in the range of low (*integer1*) to high (*integer2*) where *integer1* is an integer from 0 to 2147483646 and *integer2* is an integer from 1 to 2147483647. The value coded for *integer1* must be less than the value coded for *integer2*.

T(*integer***)**
Indicates that the delay is to be chosen randomly from the rate table specified by a RATE statement, where *integer* specifies the name field on a RATE statement and is an integer from 0 to 255. For more

information on the RATE statement, see the Rate Table Statement on page "RATE - rate table statement" on page 51.

Notes:

- If you code only one integer for a value, the parentheses are optional. For example, A(5) can also be A5.
- This operand sets the default delay value. Any DELAY statement coded in the message generation deck overrides this value.

Default: F(1)

FRSTTXT=*deck*

Function: Defines the first message generation deck to be used when the transaction program is started.

Format: A 1- to 8-character name specifying one of the message generation decks.

Default: None. If you omit this operand, the transaction program will begin with the first message generation deck specified on the PATH operand for the TP. If no PATH operand is coded or defaulted for this TP, the first path in the network will be chosen.

INSTANCE={(*initial***,***maximum***)|(<u>1,1</u>)}**

Function: Specifies the number of instances of the transaction program that are to be activated when the network is started, and the maximum number of concurrent instances that are supported.

Format: For *initial*, the format is an integer from 0 to 32767. For *maximum*, the format is an integer from 1 to 32767.

Note:

- If *initial* has a value of 0, a TP instance will be activated only when an attach request is received for the TP. This value is typically used for a TP that is simulating a server in a client/server environment. If 0 is specified as the initial value for a client TP, the TP will never be activated because WSim will not allow a client to accept an attach request.
- WSim never allows more than the maximum number of concurrent instances to be executing simultaneously. Therefore, if the maximum value is less than the initial value, the maximum concurrent instances will be activated when the network is started. The remainder of the initial instances will be activated as other instances terminate.

Default: For both initial and maximum the default is 1.

IUTI=*uti*

Function: Specifies the name of a user time interval (UTI) which is used in calculating all delays for this TP, unless overridden by a message generation statement. *uti* must reference a UTI statement defined within the network configuration statements. For more information on the UTI statement, see the User Time Interval Statement on page "UTI - user time interval statement" on page 56.

Format: A one to eight alphanumeric character name.

Default: None.

MAXCALL={*integer***|<u>5</u>}**

Function: Specifies the maximum number of outstanding message generation deck calls for the transaction program.

Format: An integer from 0 to 255.

Default: 5

MSGTRACE={YES|NO}

Function: Specifies whether message generation trace records for this resource should be written to the log data set.

Format: YES or NO.

Default: NO

PATH=(*name*,...)

Function: Specifies the PATH statements for message generation deck selection to be referenced by this transaction program in controlling message generation.

Format: A list of 1- to 8-character alphanumeric names separated by commas and enclosed in parentheses.

Default: None. If you omit this operand, the transaction program will execute all PATH statements in the order in which they are coded in the NTWRK statement.

QUIESCE={YES|NO}

Function: Specifies whether the transaction program will be automatically marked quiesced during network initialization. No messages will be generated for this TP until it is released by an operator command.

Format: YES or NO.

Default: NO

Note: If you do not want the statements to be executed when the first deck enters message generation, put a STOP statement in the first deck entered to get the start-up delay.

SEQ={*integer*|0}

Function: Specifies the initial value for the device sequence counter at network initialization or after a network reset.

Format: An integer from 0 to 2147483647.

Default: 0

STLTRACE={YES|NO}

Function: Specifies whether STL trace records for this resource should be written to the log data set.

Format: YES or NO.

Default: NO

TPNAME={*name*|'*name*'|(*name*)|('*name*')}

Function: Specifies the name of the CPI-C transaction program that is to be simulated. A TP name may be specified once as a server TP and again as a client TP for a given APPCLU.

Format: A 1- to 64-character name specified as a string of up to 64 characters or 128 hexadecimal digits. A quoted *name* indicates that it is a hexadecimal string. The TP name must be from character set 00640[5].

5. For the definition of character sets 01134 and 00640, refer to *SAA Common Programming Interface Communications Reference*.

The name can be continued across multiple lines by placing a comma after the last digit on a line and continuing with the next digit anywhere beyond column 1 on the next line. If the name is continued across multiple lines, it must be enclosed in parentheses.

Default: The *name* field value for the TP statement.

Note: You must code either the *name* field for the TP statement, or the TPNAME operand value.

TPREPEAT={YES|NO|*integer*}

Function: Specifies whether message generation should repeat the paths defined for the transaction program, or should end for the TP when the end of the path sequence is reached.

Format: For the TPREPEAT operand, you can specify one of the following values:

YES Specifies that at the end of the path sequence defined for the TP, message generation should continue with the first deck in the first path.

NO Specifies that message generation should end for the TP when the end of the path sequence is reached. This is comparable to the TP terminating when it reaches the end of the code path.

integer
Specifies the number of times the path sequence for the TP should be executed. Message generation will end after the path sequence has been repeated the specified number of times. The value can be from 1 to 32767.

Default: NO

Note: To simulate the way a typical TP would behave, specify NO or take the default. Specifying YES or an integer allows use of the DIST network definition statement, or the CYCLIC operand of the PATH statement, to control message generation. You can use this technique if you want a TP to be repeatedly executed, perhaps on a delay basis, and you want to select one of multiple message decks to represent the TP each time it is invoked.

TPSTATS={YES|NO}

Function: Specifies whether statistics about messages sent and received are to be kept for each individual TP instance.

Format: YES or NO.

Default: NO

Note: If TPSTATS=NO is specified or the default is taken, individual statistics are kept only for the first instance of each TP. If multiple TP instances are being simulated, coding TPSTATS=YES will use significantly more run-time storage. If storage is constrained and statistics are not required for individual TP instances, code TPSTATS=NO (or accept the default value of NO). Coding TPSTATS=NO will also significantly improve the performance of simulations involving large numbers of TP instances.

Note: If a query is issued against a TP that has TPSTATS set to NO, a record of the instance will be found only if it is still active or it is the first instance.

TPSTIME={*integer*|0}

Function: Specifies a stagger time to be used in initiating multiple TP instances at network startup. The value represents the number of .01 second intervals that should elapse between the start of each TP instance. Using a small stagger time in conjunction with specifying TPSTATS=NO can significantly improve the storage utilization of networks involving multiple TP instances. If a stagger time is not used, all storage requirements for all client TP instances must be available at network start time. If a stagger time is used, only a small amount of the total client storage requirements may be needed at any point of time.

Format: An integer from 0 to 65535.

Default: 0

TPTYPE={CLIENT|SERVER}

Function: Specifies whether the TP is a client or a server. A server TP is one that accepts incoming conversations. A TP may be a client TP on one conversation and a server TP on another. If a TP can accept an incoming conversation, it must be specified as a server TP, even though on some conversations it may be a client. WSim does not allow a client TP to accept incoming conversations. The same TP name may be specified as both a client and server on the same APPCLU. The path specified for the server will be allowed to accept incoming conversations, and the path specified for the client will not.

Format: CLIENT or SERVER.

Default: CLIENT

UCD={YES|BOTH|NO}

Function: Specifies whether the TP is to recognize user control data and treat it as if it were application data. Some early LU 6.2 transaction programs tag application data with the user control data GDS ID. CPI-C does not support user control data. However, this operand allows the CPI-C implementation to be compatible with many of these early LU 6.2 applications.

Format: For the UCD operand, you can specify one of the following values:

YES Specifies that user control data is to be treated as application data. When WSim is sending, the data will be tagged with the user control data GDS ID. When WSim is receiving, only data tagged with the user control data GDS ID will be recognized as application data. Data received that is tagged with the application data GDS ID is ignored.

BOTH Specifies that both user control data and application data are to be treated as application data. When WSim is sending, the data will be tagged with the application data GDS ID. When WSim is receiving, data tagged with either the user control data or the application data GDS IDs will be recognized as application data.

NO Specifies that user control data should not be treated as application data. Only data tagged with the application data GDS ID will be recognized as application data. When WSim is sending, the data will be tagged with the application data GDS ID. When WSim is receiving, only data tagged with the application data GDS IDs will be recognized as application data. Data received that is tagged with the user control data GDS ID is ignored.

Default: NO

Note: If the transaction programs for both the client and the server side of a conversation are being simulated by WSim, the UCD parameter must be consistent between the two TP definitions. For example, it is not legitimate to specify UCD=NO for the client TP and UCD=YES for the server TP. It would, however, be acceptable to specify UCD=NO for the client and UCD=BOTH for the server.

USERAREA={*integer*|0}

Function: Defines an area of storage for this transaction program to be used for a scratch pad or user exit work area.

Format: An integer from 0 to 32767.

Note: If *integer* is not a multiple of eight bytes, this value is rounded up to the next multiple of eight bytes.

Default: 0

Chapter 6. Defining VTAMAPPL simulation statements

The following sections describe VTAMAPPL network configuration definition statements. These statements are listed in alphabetical order. For more information about the order in which you need to code these statements, see Table 3 on page 13.

When you code the configuration definition statements, you can code the common operands for the network resources at the highest level statements and override them on lower level statements, if necessary.

Operands on the VTAMAPPL and LU statements are grouped according to function and appear under group headings. You should use an operand within a group when you define resources that belong to that group.

Note: You can select operands from more than one functional group to define a particular resource.

Summary of operands

Table 6 lists the operands you can use in a VTAMAPPL simulation, where these operands are defined, and where these operands can be coded.

Table 6. VTAMAPPL simulation operands

CHAINING	LU	NTWRK, VTAMAPPL
COLOR	LU	NTWRK, VTAMAPPL
CRDATALN	LU	NTWRK, VTAMAPPL
DBCS	LU	NTWRK, VTAMAPPL
DBCSCSID	LU	NTWRK, VTAMAPPL
DELAY	LU	NTWRK, VTAMAPPL
DISPLAY	LU	NTWRK, VTAMAPPL
DLOGMOD	LU	NTWRK, VTAMAPPL
ENCR	LU	NTWRK, VTAMAPPL
Operand	Appears on	Can also be coded on
ALTCSET	LU	NTWRK, VTAMAPPL
APLCSID	LU	NTWRK, VTAMAPPL
APPLID	VTAMAPPL	-
ATRABORT	LU	NTWRK, VTAMAPPL
ATRDECK	LU	NTWRK, VTAMAPPL
BASECSID	LU	NTWRK, VTAMAPPL
BUFSIZE	VTAMAPPL	NTWRK
CCSIZE	LU	NTWRK, VTAMAPPL
EXTFUN	LU	NTWRK, VTAMAPPL
FLDOUTLN	LU	NTWRK, VTAMAPPL
FLDVALID	LU	NTWRK, VTAMAPPL

Table 6. VTAMAPPL simulation operands (continued)

FRSTTXT	LU	NTWRK, VTAMAPPL
HIGHLITE	LU	NTWRK, VTAMAPPL
INIT	LU	NTWRK, VTAMAPPL
IUTI	LU	NTWRK, VTAMAPPL
LOGDSPLY	LU	NTWRK, VTAMAPPL
MAXCALL	LU	NTWRK, VTAMAPPL
MAXNOPTN	LU	NTWRK, VTAMAPPL
MAXPTNSZ	LU	NTWRK, VTAMAPPL
MAXSESS	LU	NTWRK, VTAMAPPL
MLEN	VTAMAPPL	NTWRK
MLOG	VTAMAPPL	NTWRK
MSGTRACE	LU	NTWRK, VTAMAPPL
PASSWD	VTAMAPPL	-
PATH	LU	NTWRK, VTAMAPPL
PROTMSG	LU	NTWRK, VTAMAPPL
PRTSPD	LU	NTWRK, VTAMAPPL
PS	LU	NTWRK, VTAMAPPL
QUIESCE	LU	NTWRK, VTAMAPPL
RESOURCE	LU	NTWRK, VTAMAPPL
RSTATS	LU	NTWRK, VTAMAPPL
RTR	LU	NTWRK, VTAMAPPL
SAVEAREA	LU	NTWRK, VTAMAPPL
SEQ	LU	NTWRK, VTAMAPPL
STLTRACE	LU	NTWRK, VTAMAPPL
THKTIME	LU	NTWRK, VTAMAPPL
THROTTLE	LU	NTWRK, VTAMAPPL
UASIZE	LU	NTWRK, VTAMAPPL
UOM	LU	NTWRK, VTAMAPPL
USERAREA	LU	NTWRK, VTAMAPPL

LU - VTAMAPPL logical unit definition statement

```
[name] LU [ATRABORT={DECK|PATH|NONE}]
          [,ATRDECK=rname]
          [,CRDATALN={integer|20}]
          [,DELAY={A(integer)}
                  {F(integer)}
                  {R(integer1[,integer2])}
                  {T(integer)}
                  {F(1)}]
          [,FRSTTXT=deck]
          [,IUTI=uti]
          [,MAXCALL={integer|5}]
          [,MSGTRACE={YES|NO}]
          [,PATH=(name,...)]
          [,PRTSPD=integer]
          [,QUIESCE={YES|NO}]
          [,RSTATS={YES|NO}]
          [,SAVEAREA=(num,size)]
          [,SEQ={integer|0}]
          [,STLTRACE={YES|NO}]
          [,THKTIME={IMMED|UNLOCK}]
          [,USERAREA={integer|0}]

          SNA Simulation Operands
          [,CHAINING={AUTO|MAN}]
          [,DLOGMOD=name]
          [,ENCR={NONE|OPT|REQD|SEL}]
          [,INIT={PRI|SEC}]
          [,LUTYPE={LU0|LU1|LU2|LU3|LU4|LU6|LU7}]
          [,MAXSESS={(pri,sec)|(0,1)}]
          [,RESOURCE=name]
          [,RTR={YES|NO}]
          [,THROTTLE={n|1}]

          Display Simulation Operands
          [,DISPLAY=(a,b[,c,d])]
          [,LOGDSPLY={BEGIN|BOTH|END|NONE}]
          [,PROTMSG={YES|NO}]
          3270 Simulation Operands
          [,ALTCSET={APL|NONE}]
          [,APLCSID=(integer1,integer2|963,310)]
          [,BASECSID=(integer1,integer2|697,37)]
          [,CCSIZE=(x,y)]
          [,COLOR={GREEN|MULTI|ORANGE}]
          [,DBCS={YES|NO}]
          [,DBCSCSID=(integer1,integer2|370,300)]
          [,EXTFUN={YES|NO}]
          [,FLDOUTLN={YES|NO}]
          [,FLDVALID={YES|NO}]
          [,HIGHLITE={YES|NO}]
          [,MAXNOPTN={integer|0}]
          [,MAXPTNSZ=integer]
          [,PS={({S|T},...)|NONE}]
          [,UASIZE=(w,h)]
          [,UOM={INCH|MM}]
```

Function

The LU statement must be coded at least once after each VTAMAPPL statement. The LU statement defines one or more logical unit half-sessions to be simulated using the WSim/VTAM application program interface, and it defines the type of logical unit simulation to be performed for the SNA half-sessions. You can use this statement to override the operand defaults from higher level statements.

Where

name

Function: Specifies the symbolic name used to reference the resource on printed reports, with the data field options in the scripting statements, and with operator commands.

Note: To avoid confusion when running WSim or the log data set analysis programs, all resources in a network should have unique names.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

APLCSID=(*integer1*,*integer2*|963,310)

Function: Specifies the 3270 character set ID for the alternate APL character set when you also specify ALTCSET=APL. *integer1* specifies the character set number. *integer2* specifies the code page number.

Format: *integer1* and *integer2* must be in the range 0–65535.

Default: (963,310)

ALTCSET={APL|NONE}

Function: Specifies whether the 3270 APL/TEXT character set is to be supported.

Note: This operand is valid for secondary LU2 and LU3 only.

Format: For the ALTCSET operand, you can code one of the following values:

APL Specifies that the 3270 APL/TEXT character set will be supported. The APL/TEXT character set uses the graphic escape character (X'08') followed by the character code point sequences.

NONE
Specifies that the 3270 APL/TEXT character set will not be supported.

Default: NONE

ATRABORT={DECK|PATH|NONE}

Function: Specifies whether the current message generation deck or current path will be aborted when the device enters automatic terminal recovery.

Format: For the ATRABORT operand, you can code one of the following values:

DECK Specifies that the current message generation path entry is to be aborted upon entry to automatic terminal recovery.

PATH Specifies that the current path is to be aborted upon entry to automatic terminal recovery.

NONE
Specifies that the current path entry is not to be aborted. Message generation will automatically call the ATRDECK, if it exists, and then return to the place in the message text deck where the device entered automatic terminal recovery.

Note: If ATRABORT=NONE is specified and an ATRDECK exists, there must be an available message generation deck call level for the ATRDECK to be executed.

Default: DECK

ATRDECK=*rname*

Function: Specifies the name of a message generation deck to be invoked automatically as the recovery message generation deck when automatic terminal recovery is entered. Refer to , SC31-8945 for a discussion of automatic terminal recovery.

Format: A 1- to 8-character name conforming to standard JCL naming conventions.

Default: None.

BASECSID=(*integer1***,***integer2***|697,37)**

Function: Specifies the 3270 character set ID for the base character set. *integer1* specifies the character set number. *integer2* specifies the code page number.

Format: *integer1* and *integer2* must be in the range 0–65535.

Default: (697,37)

CCSIZE=(*x***,***y***)**

Function: Specifies the display character cell size for a 3270 device that has extended function support.

Format: For the CCSIZE operand, you can code the following values:

x	Specifies the number of units in the width of the character, where *x* is an integer from 0 to 255.
y	Specifies the number of units in the height of the character, where *y* is an integer from 0 to 255.

The width and height of a fixed character cell must be nonzero values. Variable character cells must be defined as (0,0). CCSIZE=(*x*,0) or CCSIZE=(0,*y*) are not valid combinations and will be flagged as errors.

When the character cell is defined as variable, the character cell size will be determined by the device. UASIZE must be coded if CCSIZE=(0,0).

Default: The following default values are assumed if this operand is not specified:

(9,16)	For device with display sizes of less than 3440 bytes
(9,12)	For device with display sizes of 3440 or more bytes
(10,8)	For a 3270 printer.

CHAINING={AUTO|MAN}

Function: Specifies whether automatic SNA chaining will be performed on the text data inserted into the terminal buffer during message generation.

Format: For the CHAINING operand, you can code one of the following values:

AUTO

Indicates that automatic chaining of the text data will be performed. Specify CHAINING=AUTO to utilize a BUFSIZE operand value greater than 32000 on the preceding VTAMAPPL statement.

MAN Indicates that chaining of the text data will be performed by the message generation deck.

Default: MAN

COLOR={GREEN|MULTI|ORANGE}

Function: Specifies whether seven color display or four color printer support will be provided for a 3270 device that has extended function support.

Format: For the COLOR operand you can code one of the following values:

GREEN
Specifies no color support, such as the 3278.

MULTI
Specifies multiple color support, (seven color display or four color printer).

ORANGE
Specifies color support but displays only in orange.

Default: GREEN

CRDATALN={*integer*|20}

Function: Specifies the length of the data to be reported in Last Message Transmitted and Last Message Received in an inactivity report, a response to the Query operator command, or when the device is in console recovery.

Format: An integer from 20 to 255.

Default: 20

DBCS={YES|NO}

Function: Specifies whether DBCS is supported for the simulated 3270 display.

Format: YES or NO.

Default: NO

Note: When you specify DBCS=YES, the following occurs:

- The DBCS-ASIA and character sets (with DBCS) query reply structure fields are inserted into the 3270 query reply when it is generated.
- Partitioning and field validation are not supported.

DBCSCSID=(*integer1*,*integer2*|370,300)

Function: Specifies the 3270 character set ID for the DBCS character set when you also code DBCS=YES. *integer1* specifies the character set number. *integer2* specifies the code page number.

Format: *integer1* and *integer2* must be in the range 0–65535.

Default: (370,300)

DELAY={A(*integer*)|F(*integer*)|R(*integer1*[,*integer2*])|T(*integer*)|F(1)}

Function: Specifies the value to be multiplied by the active UTI to define the delay between sending of messages.

Format: For the DELAY operand, you can code one of the following values:

A(*integer*)
Indicates the delay is to be chosen randomly from the range 0 to 2 times the integer, where *integer* is an integer from 0 to 1073741823. The average delay will be *integer*.

F(*integer***)**
: Indicates the delay value is fixed at the value specified by the integer, where *integer* is an integer from 0 to 2147483647.

R(*integer***)**
: Indicates that the delay is to be chosen randomly from the range specified by an RN statement, where *integer* specifies the name field on an RN statement and is an integer from 0 to 255.

R(*integer1***,***integer2***)**
: Indicates that the delay is to be chosen randomly in the range of low (*integer1*) to high (*integer2*) where *integer1* is an integer from 0 to 2147483646 and *integer2* is an integer from 1 to 2147483647. The value coded for *integer1* must be less than the value coded for *integer2*.

T(*integer***)**
: Indicates that the delay is to be chosen randomly from the rate table specified by a RATE statement, where *integer* specifies the name field on a RATE statement and is an integer from 0 to 255.

Notes:

- If you code only one integer for a value, the parentheses are optional. For example, A(5) can also be A5.
- This operand sets the default delay value. Any DELAY statement coded in the message generation deck overrides this value.

Default: F(1)

DISPLAY=(*a***,***b***[,***c***,***d***])**

Function: Specifies the default and alternate screen sizes for displays. The default size for 3270 devices will be set at start time and changed to the alternate size when WSim receives an ERASE WRITE ALTERNATE command.

Format: 2 or 4 numbers between 1 and 255 specifying the number of rows and columns for the default and alternate screen sizes.

Note: For 3270 LUs, the display size (for displays) or buffer size (for printers) as determined by the products (*a* x *b*) or (*c* x *d*) cannot be greater than 4096, if EXTFUN=NO, or 16384, if EXTFUN=YES.

Default: The DISPLAY operand can have the following defaults:

(24,80,24,80)
: For LU2 and LU3 types.

(24,80) For LU7 types. If you specify an alternative display size for LU7, it is not used.

Note: If you specify only *a* and *b*, *c* defaults to *a*, and *d* defaults to *b*.

DLOGMOD=*name*

Function: Specifies the name of a VTAM logon mode table (MODETAB) entry for an LU.

Format: A 1- to 8-character name that matches the LOGMODE operand name of an entry in the VTAM logon mode table for this LU.

Default: None.

ENCR={NONE|OPT|REQD|SEL}

Function: Specifies the level of cryptography that is to be established for the device.

Format: For the ENCR operand, you can code one of the following values:

NONE
Specifies that no data encryption is to occur.

OPT Specifies that data encryption is optional depending on a parameter in the BIND command.

REQD Specifies that data encryption is always required for a session involving the device.

SEL Specifies that data encryption is selective on an RU basis, as defined by a bit in the RH.

Default: NONE

EXTFUN={YES|NO}

Function: Specifies whether extended function support is to be provided for a 3270 device.

Note: Extended function support can only be provided for the logical unit types LU2 and LU3.

Format: For the EXTFUN operand, you can enter one of the following values:

YES Specifies that extended function support will be provided to include the following:

- The Write Structured Field (WSF) command will be accepted for a read partition query structured field. A query response will be generated to reflect the device resources.
- 12- or 14-bit buffer addresses will be accepted from the application. 14-bit addresses will be generated only if the current display buffer size is greater than 4096 bytes.
- The display device can be defined with display sizes of up to 16384 bytes.
- The Data Analysis/APL character set will not be supported.

NO Specifies that no extended function support will be supplied. The display device may have display sizes of up to 4096 bytes, and the Data Analysis/APL character set will be supported.

Default: YES

FLDOUTLN={YES|NO}

Function: Specifies whether field outlining is supported for the simulated 3270 display.

Format: YES or NO.

Default: NO

Note: When you specify FLDOUTLN=YES, the Field Outlining query reply structured field is inserted into the 3270 query reply when it is generated.

FLDVALID={YES|NO}

Function: Specifies whether field validation support is to be provided for an LU2 device that has extended function support.

Note: This operand is valid for type LU2 only.

Format: YES or NO.

Default: NO

FRSTTXT=*deck*

Function: Defines the first message generation deck to be used when the network is started.

Format: A 1- to 8-character name specifying one of the message generation decks.

Default: None. If you omit this operand, the devices will begin with the first message generation deck specified on the PATH operand for the device. If no PATH operand is coded or defaulted for this device, the first PATH in the network will be chosen.

HIGHLITE={YES|NO}

Function: Specifies whether highlighting support is to be provided for a 3270 LU2 that has extended function support.

Format: YES or NO.

Default: NO

INIT={PRI|SEC}

Function: Specifies which logical unit will initiate the first SNA session.

Format: PRI or SEC.

Note: For secondary LU half-sessions, if INIT=SEC and the RESOURCE operand are coded, WSim will automatically generate an INITIATE-SELF command. If the RESOURCE operand is not coded with INIT=SEC, you must provide the INITIATE-SELF command with the CMND statement.

Do not code both the RESOURCE statement and the CMND statement or duplicate INIT-SELFs will be generated.

For primary LU half-sessions, if INIT=PRI and the RESOURCE operand are coded, WSim will automatically generate an INITIATE-SELF command. If the RESOURCE operand is not coded with INIT=PRI, you must provide the INITIATE-SELF command with the CMND statement.

Default: PRI. The primary logical unit will initiate the first session.

IUTI=*uti*

Function: Specifies the name of a UTI which is used in calculating all delays for this device, unless overridden by a message generation statement. *uti* must reference a UTI statement defined within the network configuration statements.

Format: From one to eight alphanumeric characters.

Default: None.

LOGDSPLY={BEGIN|BOTH|END|NONE}

Function: Specifies whether 3270 display buffers, printer buffers, and 5250 display buffers are to be automatically written to the log data set for formatting by the loglist program.

Note: This option is meaningful for all simulated devices in the IBM 3270 and IBM 5250 Information Display Systems. It is meaningful for the following LU types: LU2, LU3, and LU7. It is ignored for all others.

Format: For the LOGDSPLY operand, you can code one of the following values:

BEGIN
Specifies that the display buffers are to be logged only at the beginning of message generation.

BOTH Specifies that the display buffers are to be logged at the beginning and end of message generation.

END Specifies that the display buffers are to be logged only at the end of message generation.

NONE
Specifies that no display buffers are to be logged.

Default: NONE

`LUTYPE={LU0|LU1|LU2|LU3|LU4|LU6|LU7}`

Function: Specifies the type of logical unit to be simulated for the SNA half-session.

Note: WSim will process unformatted function management data (FMD) in a request/response unit (RU) only for secondary LU2, LU3, and LU7 types of logical units.

Format: One of the following types must be entered:

LU0, LU1, LU2, LU3, LU4, LU6, LU7

Default: LU0

`MAXCALL={`*`integer`*`|5}`

Function: Specifies the maximum number of outstanding message generation deck calls for the device.

Format: An integer from 0 to 255.

Default: 5

`MAXNOPTN={`*`integer`*`|0}`

Function: Specifies the maximum number of display partitions that can be defined concurrently for an LU2 device that has extended function support.

Format: An integer from 0 to 16.

Default: 0

`MAXPTNSZ=`*`integer`*

Function: Specifies the maximum size of a partition in bytes.

Note: This operand is valid only when the MAXNOPTN operand value is greater than zero. Scrollable partitions support is assumed when the MAXPTNSZ value is greater than the DISPLAY operand value.

Format: An integer from 1 to 32767.

Default: The number of bytes specified by the DISPLAY operand.

`MAXSESS={(`*`pri,sec`*`)|(0,1)}`

Function: Specifies the maximum number of concurrent half-sessions to be allowed with this logical unit.

Format: For the MAXSESS operand, you can code the following values (separated by a comma and enclosed in parentheses):

pri Specifies the maximum number of primary half-sessions permitted, where *pri* is a number from 0 to 255.

sec Specifies the maximum number of secondary half-sessions permitted, where *sec* is a number from 0 to 255.

Default: (0,1)

Note: Multiple half-sessions that are active concurrently will operate independently. You must code at least one half-session for *pri* or *sec*. MAXSESS=(0,0) is invalid.

MSGTRACE={YES|NO}
Function: Specifies whether message generation trace records for this resource should be written to the log data set.

Format: YES or NO.

Default: NO

PATH=(*name*,...)
Function: Specifies the PATH statements for message generation deck selection to be referenced by this device in controlling message generation.

Format: A list of alphanumeric names separated by commas and enclosed in parentheses.

Default: None. If you omit this operand, the device will execute all PATH statements in the order in which they are coded in the NTWRK statement.

PROTMSG={YES|NO}
Function: Specifies whether the field protected message ITP403I is to be written to the log data set.

Note: You can also use the INHBTMSG operand on the NTWRK statement to inhibit the printing of this message.

Format: YES or NO.

Default: YES

PRTSPD=*integer*
Function: Specifies the speed at which the device being simulated will print the data received. The device buffer is assumed to be busy for the time taken to print the data. SNA definite responses will be sent after the data has been printed.

Format: An integer from 0 to 32767 that specifies the print speed for the device in characters per second. PRTSPD=0 means "immediate completion of print operation".

Note: This option does not take into account the extra time required for color printers.

Default: 0.

PS={({S|T},...)|NONE}
Function: Specifies the number and types of Programmed Symbols character sets for a 3270 device that has extended function support.

Format: For the PS operand, you can code one of the following values:

S
Specifies a single plane Programmed Symbols character set.

T
Specifies a triple plane Programmed Symbols character set.

NONE
Specifies that no Programmed Symbols character sets will be defined.

If you specify DBCS=NO, you can code up to six occurrences of the letters S and T with the letters separated by commas and the entire operand value enclosed in parentheses. If you specify DBCS=YES, you can code up to five occurrences of the letters S and T with the letters separated by commas and the entire operand value enclosed in parentheses. For example: PS=(S,S,T,T,S,T) is coded for a 3279-S3G.

Default: NONE

QUIESCE={YES|NO}
Function: Specifies that the device will be automatically marked quiesced during network initialization. No messages will be generated for that device until it is released by an operator command or logic test.

Format: YES or NO.

Default: NO

Note: If you do not want the statements to be executed when the first deck enters message generation, put a STOP statement in the first deck entered to get the start-up delay.

RESOURCE=*name*
Function: Specifies the SNA network name of the logical unit with which a session will be initiated or terminated. If you code the RESOURCE operand for this device, WSim will automatically send the INITIATE-SELF command. When the session is terminated, it will again send the INITIATE-SELF command.

Note: For secondary LU half-sessions, if RESOURCE and INIT=SEC are coded, WSim will automatically send the INITIATE-SELF command. When the session is terminated, it will again send the INITIATE-SELF command.

For primary LU half-sessions, if RESOURCE and INIT=PRI are coded, WSim will automatically send the INITIATE-SELF command. When the session is terminated, it will again send the INITIATE-SELF command.

The RESOURCE operand will be ignored for all other non-initiating LU half-sessions, that is, secondary half-session with INIT=PRI or primary half-session with INIT=SEC.

You can use the CMND COMMAND=INIT instead of RESOURCE if only one pass through the decks is required or to control when the INITIATE-SELF command should be sent. To use the CMND statement to send the INITIATE-SELF command, do not code the RESOURCE operand and make sure that the CMND statement is the first statement in the first deck referenced. Do not code both the RESOURCE statement and the CMND statement or duplicate INIT-SELFs will be generated. Refer to , SC31-8945 for more information on initiating sessions.

Format: A 1- to 8-character name that conforms to SNA network name specifications.

Default: None.

RSTATS={YES|NO}
Function: Specifies whether or not online response time statistics will be accumulated for this device and reported when the RSTATS (W) operator command is issued for the device.

Format: For the RSTATS operand, you can code one of the following values:

YES Specifies that online response statistics will be accumulated and reported when the RSTATS (W) operator command is issued for the device.

NO Specifies that no online response statistics will be kept for this device, and no report will be displayed for the device when the RSTATS (W) operator command is issued.

Note: If RSTATS=NO is specified in the network definition, you cannot start it at a later time (with the Alter (A) operator command) because of the storage allocation necessary at initialization time.

Default: NO

RTR={YES|NO}

Function: Specifies whether or not the SNA command Ready-to-Receive will be sent by a secondary logical unit in a bracket contention situation.

Format: For the RTR operand, you can code one of the following values:

YES Specifies that the RTR will be sent.

NO Specifies that the RTR will not be sent.

Default: NO

SAVEAREA=(*num*,*size*)

Function: Specifies the number and size of the static save areas to be pre-allocated for input data save and recall.

Format: For the SAVEAREA operand, you can code the following values:

num Specifies the number of save areas to be allocated for this device, where *num* is an integer from 1 to 4095.

size Specifies the size for each save area in bytes, where *size* is an integer from 1 to 32767.

Default: None. This operand is not required. Save areas not defined by this operand will be allocated dynamically for this device.

SEQ={*integer*|0}

Function: Specifies the initial value for the device sequence counter at network initialization or after a network reset.

Format: An integer from 0 to 2147483647.

Default: 0

STLTRACE={YES|NO}

Function: Specifies whether STL trace records for this resource should be written to the log data set.

Format: YES or NO.

Default: NO

THKTIME={IMMED|UNLOCK}

Function: Specifies when the message delay is to be started. The delay will be started after all SNA responses have been received, all WAIT conditions have been satisfied, and a keyboard restore message has been received for a display.

Format: For the THKTIME operand, you can code one of the following values:

IMMED
Specifies that the calculated delay for the next message starts immediately after the current message has been generated by WSim.

UNLOCK
Specifies that calculation of the delay for the next message starts when WSim is able to generate another message.

Note: For non-display logical units running in exception response only mode and not setting the WAIT indicator, use of the UNLOCK option will cause messages to be sent from WSim with no intermessage delays.

Exception Response Only	Unlock Applicable
Yes	No
No	Yes

Default: IMMED

THROTTLE={*n*|1}
Function: Specifies the number of messages that this LU or DEV can have outstanding (waiting to be transmitted or being transmitted) at any one time.

Format: An integer between 1 and 255.

Default: 1

UASIZE=(*w*,*h*)
Function: Specifies the size of the display screen in PELs (picture elements).

Format: 2 numbers between 1 and 999 specifying the width (*w*) and height (*h*) of the screen in PELs.

Default: None. This operand is required if CCSIZE=(0,0) is specified. It will be ignored if CCSIZE=(*x*,*y*) is specified.

UOM={INCH|MM}
Function: Specifies the unit of measurement for the distance between PELs (picture elements) on the screen of a display.

Format: For the UOM operand, you can code one of the following values:

INCH Specifies that the distance is to be measured in inches.

MM Specifies that the distance is to be measured in millimeters.

Default: INCH

USERAREA={*integer*|0}
Function: Defines an area of storage for this device to be used for a scratch pad or user exit workarea.

Format: An integer from 0 to 32767.

Note: If *integer* is not a multiple of eight bytes, this value is rounded up to the next multiple of eight bytes.

Default: 0

VTAMAPPL - VTAMAPPL statement

```
[name] VTAMAPPL [BUFSIZE={integer|265}]
                [,MLEN=integer]
                [,MLOG={YES|NO}]

                VTAMAPPL Operands
                [,APPLID=name]
                [,PASSWD=password]
```

Function

The VTAMAPPL statement defines the VTAM application program symbolic name and the password associated with the VTAM application program symbolic name. The VTAMAPPL statement is the first statement in the definition of a WSim/VTAM application program interface.

See Table 6 on page 71 for operands that can be coded on this statement to provide defaults for lower-level statements and where these operands are defined.

Where

name

Function: Specifies the symbolic name used to identify the WSim/VTAM application.

Format: From one to eight alphanumeric characters.

Default: APPLID operand value.

APPLID=*name*

Function: Specifies the VTAM application program symbolic name. This name must match an entry in VTAM's configuration tables (VTAMLST) created using a VTAM APPL definition statement. The name specified is the name of the APPL statement or ACBNAME operand value coded on an APPL statement.

Format: From one to eight alphanumeric characters.

Default: The VTAMAPPL name field value.

Note: You must code either the VTAMAPPL name field or APPLID operand value.

BUFSIZE={*integer***|265}**

Function: Specifies the size of the buffer in which messages will be built for and received by the LUs associated with this application program.

Format: An integer from 100 to 32767. If *integer* is less than 265, BUFSIZE=265 is used.

Note: This value should be at least as great as the length of the longest PIU (RU plus the length of the SNA FID 2 transmission header (TH) and request header (RH)) that can be generated or received by this application. Any received message longer than the value of the BUFSIZE operand will be truncated.

Default: 265

MLEN=*integer*

Function: Specifies the maximum number of data characters to be written to the log data set for each data transfer.

Format: An integer from 100 to 32767.

Note: If this value is larger than the data transferred, the entire transmission is logged. If the specification is smaller than the amount of data transferred, only the specified length is logged.

Default: If you omit this operand, the entire transmission will be logged.

MLOG={YES|NO}

Function: Specifies whether or not this VTAMAPPL will use the message logging function.

Format: YES or NO.

Default: YES

PASSWD=*password*

Function: Specifies the password associated with the VTAM application program symbolic name. This password must match the PRTCT operand value coded on the corresponding APPL statement in the VTAMLST.

Format: From one to eight alphanumeric characters.

Default: APPLID operand value.

Chapter 7. Defining TCP/IP client simulation statements

This chapter describes TCP/IP network configuration definition statements. TCP/IP networks may include simulated Telnet 3270, 3270E, 5250, or NVT clients, simulated File Transfer Protocol (FTP) clients, simulated Simple TCP or UDP clients, or any combination of these clients.

These statements are listed in alphabetical order. For more information about what order you need to code these statements, see Table 3 on page 13. For FTP simulation, see also the FILE statement in Chapter 4, "Defining general simulation statements," on page 23.

When you code the network configuration definition statements, you can code the common operands for the network resources at the highest level statements and override them on lower level statements, if necessary.

Operands on the DEV statement are grouped according to function and appear under group headings. You should use an operand within a group when you define resources that belong to that group.

Note: You can select operands from more than one functional group to define a particular resource.

Summary of operands

Table 7 lists the operands you can use in a TCP/IP simulation, where these operands are defined, and where these operands can be coded.

Table 7. TCP/IP simulation operands

Operand	Appears on	Can also be coded on
ALTCSET	DEV	NTWRK, TCPIP
APLCSID	DEV	NTWRK, TCPIP
ASSOC	DEV	TCPIP
ATRABORT	DEV	NTWRK, TCPIP
ATRDECK	DEV	NTWRK, TCPIP
BASECSID	DEV	NTWRK, TCPIP
BUFSIZE	TCPIP	NTWRK
CCSIZE	DEV	NTWRK, TCPIP
COLOR	DEV	NTWRK, TCPIP
CRDATALN	DEV	NTWRK, TCPIP
DBCS	DEV	NTWRK, TCPIP
DBCSCSID	DEV	NTWRK, TCPIP
DELAY	DEV	NTWRK, TCPIP
DISPLAY	DEV	NTWRK, TCPIP
EXTFUN	DEV	NTWRK, TCPIP
FLDOUTLN	DEV	NTWRK, TCPIP
FLDVALID	DEV	NTWRK, TCPIP

Table 7. TCP/IP simulation operands (continued)

Operand	**Appears on**	**Can also be coded on**
FRSTTXT	DEV	NTWRK, TCPIP
FTPPORT	TCPIP	NTWRK
FUNCTS	DEV	TCPIP
HIGHLITE	DEV	NTWRK, TCPIP
IUTI	DEV	NTWRK, TCPIP
LOCLPORT	DEV	-
LOGDSPLY	DEV	NTWRK, TCPIP
MAXCALL	DEV	NTWRK, TCPIP
MAXNOPTN	DEV	NTWRK, TCPIP
MAXPTNSZ	DEV	NTWRK, TCPIP
MLEN	TCPIP	NTWRK
MLOG	TCPIP	NTWRK
MSGTRACE	DEV	NTWRK, TCPIP
PATH	DEV	NTWRK, TCPIP
PORT	DEV	-
PROTMSG	DEV	NTWRK, TCPIP
PS	DEV	NTWRK, TCPIP
QUIESCE	DEV	NTWRK, TCPIP
RESOURCE	DEV	TCPIP
RSTATS	DEV	NTWRK, TCPIP
SAVEAREA	DEV	NTWRK, TCPIP
SEQ	DEV	NTWRK, TCPIP
SERVADDR	DEV	NTWRK, TCPIP
STCPHCLR	DEV	NTWRK, TCPIP
STCPHCLX	DEV	NTWRK, TCPIP
STCPPORT	TCPIP	NTWRK
STCPROLE	DEV	-
STLTRACE	DEV	NTWRK, TCPIP
SUDPPORT	TCPIP	NTWRK
TCPNAME	TCPIP	NTWRK
THKTIME	DEV	NTWRK, TCPIP
TNPORT	TCPIP	NTWRK
TYPE	DEV	TCPIP
UASIZE	DEV	NTWRK, TCPIP
UOM	DEV	NTWRK, TCPIP
USERAREA	DEV	NTWRK, TCPIP

DEV - TCP/IP device definition statement

```
[name] DEV [ASSOC={YES|NO}]
           [,ATRABORT={DECK|PATH|NONE}]
           [,ATRDECK=rname]
           [,CRDATALN={integer|20}]
           [,DELAY={A(integer)}
                   {F(integer)}
                   {R(integer1[,integer2])}
                   {T(integer)}
                   {F(1)}]
           [,FRSTTXT=deck]
           [,FUNCTS=(integer,...)]
           [,IUTI=uti]
           [,MAXCALL={integer|5}]
           [,LOCLPORT=integer]
           [,MSGTRACE={YES|NO}]
           [,PATH=(name{*integer,...)]            111
           [,PORT=integer]
           [,PRTSPD=integer]
           [,QUIESCE={YES|NO}]
           [,RESOURCE=name]
           [,RSTATS={YES|NO}]
           [,SAVEAREA=(num,size)]
           [,SEQ={integer|0}]
           [,SERVADDR=addr]
           [,STCPHCLR={YES|NO}]
           [,STCPHCLX={YES|NO}]
           [,STCPROLE={client|server}]
           [,STLTRACE={YES|NO}]
           [,THKTIME={IMMED|UNLOCK}]
           [,TYPE={TN3270|FTP|STCP|SUDP|TN3270E|TN3270P|TN5250|TNNVT}]
           [,USERAREA={integer|0}]

           Display Simulation Operands
           [,DISPLAY=(a,b[,c,d])]
           [,LOGDSPLY={BEGIN|BOTH|END|NONE}]
           [,PROTMSG={YES|NO}]

           3270 Simulation Operands
           [,ALTCSET={APL|NONE}]
           [,APLCSID=(integer1,integer2|963,310)]
           [,BASECSID=(integer1,integer2|697,37)]
           [,CCSIZE=(x,y)]
           [,COLOR={GREEN|MULTI|ORANGE}]
           [,DBCS={YES|NO}]
           [,DBCSCSID=(integer1,integer2|370,300)]
           [,EXTFUN={YES|NO}]
           [,FLDOUTLN={YES|NO}]
           [,FLDVALID={YES|NO}]
           [,HIGHLITE={YES|NO}]
           [,MAXNOPTN={integer|0}]
           [,MAXPTNSZ=integer]
           [,PS={({S|T},...)|NONE}]
           [,UASIZE=(w,h)]
           [,UOM={INCH|MM}]
```

Function

The DEV statement specifies the simulated TCP/IP client. This statement overrides the operand defaults specified on the TCPIP statement for this device only.

Where

name

Function: Specifies the symbolic name used to reference the resource on printed reports, with the data field options in the scripting statements, and with operator commands.

Note: To avoid confusion when running WSim or the log data set analysis programs, make sure that all resources in a network have unique names.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

ASSOC={YES|NO}

Function: Specifies whether the ASSOCIATE command is to be used when requesting a DEVICE-TYPE that represents a printer. Only valid when TYPE=TN3270P.

Format: YES or NO.

Default: NO

ALTCSET={APL|NONE}

Function: Specifies whether the 3270 APL/TEXT character set is to be supported.

Format: For the ALTCSET operand, you can code one of the following values:

APL Specifies that the 3270 APL/TEXT character set will be supported. The APL/TEXT character set uses the graphic escape character (X'08') followed by the character code point sequences.

NONE
Specifies that the 3270 APL/TEXT character set will not be supported.

Default: NONE

APLCSID=(*integer1*,*integer2*|963,310)

Function: Specifies the 3270 character set ID for the alternate APL character set when you also specify ALTCSET=APL. *integer1* specifies the character set number. *integer2* specifies the code page number.

Format: *integer1* and *integer2* must be in the range 0–65535.

Default: (963,310)

ATRABORT={DECK|PATH|NONE}

Function: Specifies whether the current message generation deck or current path will be aborted when the device enters automatic terminal recovery.

Format: For the ATRABORT operand, you can code one of the following values:

DECK Specifies that the current message generation path entry is to be aborted upon entry to automatic terminal recovery.

PATH Specifies that the current path is to be aborted upon entry to automatic terminal recovery.

NONE
Specifies that the current path entry is not to be aborted. Message generation will automatically call the ATRDECK, if it exists, and then return to the place in the message generation deck where the device entered automatic terminal recovery.

Note: If ATRABORT=NONE is specified and an ATRDECK exists, there must be an available message generation deck call level in order for the ATRDECK to be executed.

Default: DECK

ATRDECK=*rname*

Function: Specifies the name of a message generation deck to be invoked automatically as the recovery message generation deck when automatic terminal recovery is entered. Refer to , SC31-8945 for a discussion of automatic terminal recovery.

Format: A 1- to 8-character name that conforms to standard JCL naming conventions.

Default: None.

BASECSID=(*integer1***,***integer2***|697,37)**

Function: Specifies the 3270 character set ID for the base character set. *integer1* specifies the character set number. *integer2* specifies the code page number.

Format: *integer1* and *integer2* must be in the range 0–65535.

Default: (697,37)

CCSIZE=(*x***,***y***)**

Function: Specifies the display character cell size for a 3270 device that has extended function support.

Format: For the CCSIZE operand, you can code the following values:

x Specifies the number of units in the width of the character, where *x* is an integer from 0 to 255.

y Specifies the number of units in the height of the character, where *y* is an integer from 0 to 255.

The width and height of a fixed character cell must be nonzero values. Variable character cells must be defined as (0,0). CCSIZE=(*x*,0) or CCSIZE=(0,*y*) are not valid combinations and will be flagged as errors.

When the character cell is defined as variable, the character cell size will be determined by the device. UASIZE must be coded if CCSIZE=(0,0).

Default: The following default values are assumed if this operand is not specified:

(9,16) For devices with display sizes of less than 3440 bytes

(9,12) For devices with display sizes of 3440 or more bytes

COLOR={GREEN|MULTI|ORANGE}

Function: Specifies whether seven color display or four color printer support will be provided for a 3270 device that has extended function support.

Format: For the COLOR operand, you can code one of the following values:

GREEN
Specifies no color support, such as the 3278.

MULTI
Specifies multiple color support, such as a seven color display or four color printer.

ORANGE
Specifies color support but displays only in orange.

Default: GREEN

CRDATALN={*integer*|20}

Function: Specifies the length of the data to be reported in the Last Message Transmitted and Last Message Received fields for an inactivity report, a response to the Q (Query) operator command, or when the device is in console recovery.

Format: An integer from 20 to 255.

Default: 20

DBCS={YES|NO}

Function: Specifies whether DBCS is supported for the simulated 3270 display.

Format: YES or NO.

Default: NO

Note: When you specify DBCS=YES, the following occurs:

- The DBCS-ASIA and character sets (with DBCS) query reply structure fields are inserted into the 3270 query reply when it is generated.
- Partitioning and field validation are not supported.

DBCSCSID=(*integer1*,*integer2*|370,300)

Function: Specifies the 3270 character set ID for the DBCS character set when you also code DBCS=YES. *integer1* specifies the character set number. *integer2* specifies the code page number.

Format: *integer1* and *integer2* must be in the range 0–65535.

Default: (370,300)

DELAY={A(*integer*)|F(*integer*)|R(*integer1*[,*integer2*])|T(*integer*)|F(1)}

Function: Specifies the value to be multiplied by the active UTI to define the delay that will occur between the sending of messages.

Format: For the DELAY operand, you can code one of the following values:

A(*integer*)
: Indicates the delay is to be chosen randomly from the range 0 to 2 times the integer, where *integer* is an integer from 0 to 1073741823. The average delay will be *integer*.

F(*integer*)
: Indicates the delay value is fixed at the value specified by the integer, where *integer* is an integer from 0 to 2147483647.

R(*integer*)
: Indicates that the delay is to be chosen randomly from the range specified by an RN statement, where *integer* is an integer from 0 to 255 which corresponds to the name of the referenced RN statement.

R(*integer1*,*integer2*)
: Indicates that the delay is to be chosen randomly in the range of low (*integer1*) to high (*integer2*), where *integer1* is an integer from 0 to 2147483646 and *integer2* is an integer from 1 to 2147483647. The value coded for *integer1* must be less than the value coded for *integer2*.

T(*integer*)
: Indicates that the delay is to be chosen randomly from the rate table specified by a RATE statement, where *integer* specifies the name field on a RATE statement and is an integer from 0 to 255.

Notes:

- If you code only one integer for a value, the parentheses are optional. For example, A(5) can also be A5.
- This operand sets the default delay value. Any DELAY statement coded in the message generation deck overrides this value.

Default: F(1)

DISPLAY=(*a*,*b*[,*c*,*d*])

Function: Specifies the default and alternate screen sizes for displays. The default size for 3270 devices will be set at start time and changed to the alternate size when WSim receives an ERASE WRITE ALTERNATE command.

Format: Two or four numbers between 1 and 255 specifying the number of rows and columns for the default and alternate screen sizes.

Default: (24,80,24,80)

Note: If you specify only *a* and *b*, *c* defaults to *a*, and *d* defaults to *b*.

EXTFUN={YES|NO}

Function: Specifies whether extended function support is to be provided for a 3270 device.

Format: For the EXTFUN operand, you can code one of the following values:

YES Specifies that extended function support will be provided to include the following:

- The Write Structured Field command (WSF) will be accepted for a read partition query structured field. A query response will be generated to reflect the device resource.
- 12- or 14-bit buffer addresses will be accepted from the application. 14-bit addresses will be generated only if the current display buffer size is greater than 4096 bytes.
- The display device can be defined with display sizes of up to 16384 bytes.
- The Data Analysis/APL character set will not be supported.

NO Specifies that no extended function support will be supplied. The display device can have display sizes of up to 4096 bytes, and the Data Analysis/APL character set will be supported.

Default: YES

Refer to , SC31-8945 for more information on defining 3270 device types and the support options available.

FLDOUTLN={YES|NO}

Function: Specifies whether field outlining is supported for the simulated 3270 display.

Format: YES or NO.

Default: NO

Note: When you specify FLDOUTLN=YES, the Field Outlining query reply structured field is inserted into the 3270 query reply when it is generated.

FLDVALID={YES|NO}

Function: Specifies whether field validation support is to be provided for an 8775 device that has extended function support.

Format: YES or NO.

Default: NO

FRSTTXT=*deck*

Function: Defines the first message generation deck to be used when the network is started.

Format: A 1- to 8-character name specifying one of the message generation decks.

Default: None. If you omit this operand, the devices will begin with the first deck specified on the PATH operand for the device. If no PATH operand is coded or defaulted for this device, the first PATH in the network will be chosen.

FUNCTS=(*integer***,...}**

Function: Specifies the list of 3270 options supported for the specific FUNCTIONS REQUEST command that the sender would like to see supported on this session.

Format: Integers between 0 and 4 or a single integer value of 5.

0 Bind image, allows the server to send the SNA Bind image and Unbind notification to the client.

1 Data stream control, for TYPE=TN3270P only. Allows the use of standard 3270 data stream. This corresponds to LU type 3 SNA sessions.

2 Responses, provides support for positive and negative response handling. Allows the server to reflect to the client any and all definite, exception, and no response requests sent by the host application.

3 SNA character stream control codes for printer sessions only. Allows the use of the SNA Character Stream (SCS) and SCS control codes on the session. SCS is used with LU type 1 SNA sessions.

Note: WSim can receive LU type 1 data but nothing is done except logging and logic testing. There is no printer speed delay generated. No module is called to check the data or set a delay.

4 Sysreq, allows the client and server to emulate some (or all, depending on the server) of the functions of the SYSREQ key in an SNA environment.

5 Null list, no options are supported.

Default: 0,1,2,3,4

HIGHLITE={YES|NO}

Function: Specifies whether highlighting support is to be provided for a 3270 that has extended function support.

Format: YES or NO.

Default: NO

IUTI=*uti*

Function: Specifies the name of a UTI which is used in calculating all delays for this device, unless overridden by a message generation statement. *uti* must reference a UTI statement defined within the network configuration statements.

Format: From one to eight alphanumeric characters.

Default: None.

`LOCLPORT=`*`integer`*

Function: Specifies the local port number to be used by a Simple TCP or Simple UDP device. When a local port number is specified for STCP or SUDP devices, WSim will obtain the socket and BIND that socket to the specified local port before any data is transmitted or received on that socket.

Format: An integer from 1 to 65535 representing the local port number to be used.

Note: This operand can only be specified on a DEV statement associated with a TCPIP statement.

Default: None. This field is optional.

`LOGDSPLY={BEGIN|BOTH|END|NONE}`

Function: Specifies whether 3270 display buffers are automatically written to the log data set for formatting by the loglist program.

Format: For the LOGDSPLY operand, you can code one of the following values:

BEGIN
Specifies that the display buffers are to be logged only at the beginning of message generation.

BOTH Specifies that the display buffers are to be written to the log data set at the beginning and end of message generation.

END Specifies that the display buffers are to be logged only at the end of message generation.

NONE
Specifies that no display buffers are to be logged.

Default: NONE

`MAXCALL={`*`integer`*`|5}`

Function: Specifies the maximum number of outstanding message generation deck calls for the device.

Format: An integer from 0 to 255.

Default: 5

`MAXNOPTN={`*`integer`*`|0}`

Function: Specifies the maximum number of display partitions that can be defined concurrently for an LU2 device that has extended function support.

Format: An integer from 0 to 16.

Default: 0

`MAXPTNSZ=`*`integer`*

Function: Specifies the maximum size of a partition in bytes.

Note: This operand is valid only when the MAXNOPTN operand value is greater than zero. Scrollable partitions support is assumed when the MAXPTNSZ value is greater than the DISPLAY operand value.

Format: An integer from 1 to 32767.

Default: The number of bytes specified by the DISPLAY operand.

MSGTRACE={YES|NO}

Function: Specifies whether message generation trace records for this resource should be written to the log data set.

Format: YES or NO.

Default: NO

111 **PATH=(***name*{**integer*,...**)**

Function: Specifies the PATH statements for message generation deck selection to be referenced by this device in controlling message generation.

111 **Format:** A list of alphanumeric names, optionally followed by an asterisk and
111 iteration count, separated by a comma, and enclosed in parentheses.

Default: None. If you omit this operand, the device will execute all PATH statements in the order in which they are coded in the network configuration definition.

111 ***Integer:*** Specifies the number of iterations to be run for the path that is
111 specified by *name*.

PORT=*integer*

Function: Specifies the TCP/IP port to be used when establishing a connection for the protocol used by this simulated device.

Format: The decimal representation of the port.

Default: The well-known port for the protocol being simulated for Telnet 3270, 3270E, 5250, and NVT and for FTP, or 256 for STCP and SUDP.

Table 8. Defaults for PORT

Protocol (from TYPE= operand)	TCPIP operand that supplies default:	Default port (used if not specified on TCPIP):
Telnet 3270, 3270E, 5250, or NVT	TNPORT	23
FTP	FTPPORT	21
STCP	STCPPORT	256
SUDP	SUDPPORT	256

Note: For FTP, both the port specified and the next lower numbered port are normally used as part of the protocol.

PROTMSG={YES|NO}

Function: Specifies whether the field-protected message ITP403I is to be written to the log data set.

Note: You can also use the INHBTMSG operand on the NTWRK statement to inhibit the printing of this message.

Format: YES or NO.

Default: YES

PRTSPD=*integer*

Function: Specifies the speed at which the device being simulated will print the data received. The device buffer is assumed to be busy for the time taken to print the data. A TN3270E response will be sent after the data has been printed.

Format: An integer from 0 to 32767 that specifies the print speed for the device in characters per second. PRTSPD=0 means "immediate completion of print operation."

Note: This option is only valid for TYPE=TN3270P and LU3 printer types. This option does not take into account the extra time required for color printers.

Default: 0

PS={({S|T},...)|NONE}

Function: Specifies the number and types of Programmed Symbols character sets for a 3270 device that has extended function support.

Format: For the PS operand, you can code one of the following values:

S Specifies a single plane Programmed Symbols character set.

T Specifies a triple plane Programmed Symbol character set.

NONE
Specifies that no Programmed Symbol character sets will be defined.

If you specify DBCS=NO, you can code up to six occurrences of the letters S and T with the letters separated by commas and the entire operand value enclosed in parentheses. If you specify DBCS=YES, you can code up to five occurrences of the letters S and T with the letters separated by commas and the entire operand value enclosed in parentheses. For example: PS=(S,S,T,T,S,T), is coded for a 3279-S3G.

Default: NONE

QUIESCE={YES|NO}

Function: Specifies that the device will be automatically marked quiesced during network initialization. No messages will be generated for that device until it is released by an operator command or logic test.

Format: YES or NO.

Default: NO

Note: If you do not want the statements to be executed when the first deck enters message generation, put a STOP statement in the first deck entered to get the start-up delay.

RESOURCE=*name*

Function: Specifies the TN3270E LU name to connect or associate with.

Format: A 1- to 8- character name.

Default: None.

RSTATS={YES|NO}

Function: Specifies whether or not online response time statistics will be accumulated for this device, and reported when the W (RSTATS Query) operator command is issued for the device.

Format: For the RSTATS operand, you can code one of the following values:

YES Specifies that online response statistics will be accumulated and reported when the W (RSTATS Query) operator command is issued for the device.

NO Specifies that no online response statistics will be kept for this device, and no report will be displayed for the device when the W (RSTATS Query) operator command is issued.

Note: If RSTATS=NO is specified in the network definition, you cannot start it at a later time (with the A (Alter) operator command) because of the storage allocation necessary at initialization time.

Default: NO

SAVEAREA=(*num*,*size*)
Function: Specifies the number and size of the static save areas to be pre-allocated for input data save and recall.

Format: For the SAVEAREA operand, you can code the following values:

num Specifies the number of save areas to be allocated for this device, where *num* is an integer from 1 to 4095.

size Specifies the size for each save area in bytes, where *size* is an integer from 1 to 32767.

Default: None. Save areas not defined by this operand will be allocated dynamically for this device.

SEQ={*integer*|0}
Function: Specifies the initial value for the device sequence counter at network initialization or after a network reset.

Format: An integer from 0 to 2147483647.

Default: 0

SERVADDR=*addr*
Function: Specifies the host address to which you want to connect in a TCP/IP simulation.

Format: A value in IP dotted decimal address format. A dotted decimal address is in the form *xxx.xxx.xxx.xxx*, where *xxx* is a number in the range 0 to 255.

Default: None. This operand is optional.

STCPHCLR={YES|NO}
Function: Specifies Half-Close Receive support for a TCP/IP STCP device.

Format: For the STCPHCLR operand, you can code one of the following values:

YES Specifies a received zero-length null message will be interpreted as a shutdown of the socket receive leg.

NO Specifies a received zero-length null message will be interpreted as a close of the socket transmit and receive legs.

Default: NO

STCPHCLX={YES|NO}
Function: Specifies Half-Close Transmit support for a TCP/IP STCP device.

Format: For the STCPHCLX operand, you can code one of the following values:

YES Specifies a transmitted zero-length null message will be translated into a socket shutdown request to close only the socket transmit leg.

NO Specifies a transmitted zero-length null message will be translated into a socket close request to close the socket transmit and receive legs.

Default: NO

STCPROLE={CLIENT|SERVER}

Function: Specifies whether the device is to act as a client or server.

Format: CLIENT or SERVER

Default: CLIENT

Note: This operand can only be specified on a DEV statement associated with a TCPIP statement.

STLTRACE={YES|NO}

Function: Specifies whether STL trace records for this resource should be written to the log data set.

Format: YES or NO.

Default: NO

THKTIME={IMMED|UNLOCK}

Function: Specifies when the message delay is to be started. The delay will be started after all WAIT conditions have been satisfied and a keyboard restore message has been received for a display device.

Format: For the THKTIME operand, you can code one of the following values:

IMMED
Specifies that the calculated delay for the next message starts immediately after the current message has been generated by WSim.

UNLOCK
Specifies that calculation of the delay for the next message starts when WSim is able to generate another message.

Default: IMMED

TYPE={TN3270|TN3270E|TN3270P|TNNVT|TN5250|FTP|STCP|SUDP}

Function: Specifies whether the device represents a Telnet 3270 or 3270E, Telnet 5250, Telnet line mode network virtual terminal, FTP, or STCP or SUDP client.

Format: For the TYPE operand, you can specify one of the following values:

TN3270
Specifies that this statement represents a Telnet 3270 client.

TN3270E
Specifies that this statement represents a Telnet 3270E terminal.

TN3270P
Specifies that this statement represents a Telnet 3270E printer.

TNNVT
Specifies that this statement represents a Telnet line mode network virtual terminal client.

TN5250
Specifies that this statement represents a Telnet 5250 terminal.

FTP Specifies that this statement represents a File Transfer Protocol client.

STCP Specifies that this statement represents a Simple TCP client.

SUDP Specifies that this statement represents a Simple UDP client.

Default: TN3270

UASIZE=(*w*,*h*)

Function: Specifies the size of the display screen in PELs (picture elements).

Format: 2 integers between 1 and 999 specifying the width (*w*) and height (*h*), respectively, of the screen in PELs.

Default: None. This operand is required if CCSIZE=(0,0) is specified. It will be ignored if CCSIZE=(*x*,*y*) is specified.

UOM={INCH|MM}

Function: Specifies the unit of measurement for the distance between PELs (picture elements) on the screen of a display.

Format: For the UOM operand, you can code one of the following values:

INCH Specifies that the distance is to be measured in inches.

MM Specifies that the distance is to be measured in millimeters.

Default: INCH

USERAREA={*integer*|0}

Function: Defines an area of storage for this device to be used for a scratch pad or user exit workarea.

Format: An integer from 0 to 32767.

Note: If *integer* is not a multiple of eight bytes, this value is rounded up to the next multiple of eight bytes.

Default: 0

TCPIP - TCP/IP connection definition statement

```
[name] TCPIP [BUFSIZE={integer|2048}]
             [,FTPPORT=integer]
             [,MLEN=integer]
             [,MLOG={YES|NO}]
             [,STCPPORT=integer]
             [,SUDPPORT=integer]
             [,TCPNAME={name|TCPIP}]
             [,TNPORT=integer]
```

Function

The TCPIP statement specifies the characteristics of a TCP/IP connection. Operands specified on the TCPIP statement override the operand defaults specified on the NTWRK statement.

See Table 7 on page 87 for operands that can be coded on this statement to provide defaults for lower-level statements and where these operands are defined.

Where

name

Function: Specifies the symbolic name used to reference the resource on printed reports, with the data field options in the scripting statements, and with operator commands.

Note: To avoid confusion when running WSim or the log data set analysis programs, all resources in a network should have unique names.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

BUFSIZE={*integer*|2048}

Function: BUFSIZE specifies the size of the buffer in which messages will be built for and received by the devices following this TCPIP statement. Received messages longer than this size will be divided by the TCP/IP product into multiple segments, each received separately. Messages to be transmitted may be truncated or broken into segments depending on the specific type of client being simulated.

- Telnet 3270, 3270E, 5250, or NVT clients will split the data to be transmitted into multiple segments of approximately the length specified by BUFSIZE.
- Commands generated by FTP clients are limited to 256 bytes regardless of the BUFSIZE specification, but BUFSIZE is used to determine the size of individual segments of file data transmitted and the size of received segments on the FTP command connection. File data received on the FTP data connection is always received in segments of a maximum size of 32000 bytes.
- STCP and SUDP clients will truncate messages to be transmitted at the size specified by BUFSIZE and will receive data in segments no longer than the size specified by BUFSIZE.

Format: An integer from 100 to 32767, although if *integer* is less than 2048, BUFSIZE=2048 is used.

Default: 2048

FTPPORT=*integer*

Function: Specifies the default port number for FTP devices.

Note: Both the port specified and the next lower numbered port are normally used as a part of the protocol.

Format: A decimal representation of the port to be used. The value can be in the range from 1 to 65535.

Default: 21

MLEN=*integer*

Function: Specifies the maximum number of data characters to be written to the log data set for each data transfer.

Format: An integer from 100 to 32767.

Note: If this value is larger than the data transferred, the entire transmission is logged. If the specification is smaller than the amount of data transferred, only the specified length is logged.

Default: If you omit this operand, the entire transmission will be logged.

MLOG={YES|NO}

Function: Specifies whether or not this device will use the message logging function.

Format: YES or NO.

Default: YES

STCPPORT=*integer*

Function: Specifies the default port number for Simple TCP clients.

Format: A decimal representation of the port to be used. The value can be in the range from 1 to 65535.

Default: 256

SUDPPORT=*integer*

Function: Specifies the default port number for Simple UDP clients.

Format: A decimal representation of the port to be used. The value can be in the range from 1 to 65535.

Default: 256

TCPNAME={*name***|**__**TCPIP**__**}**

Function: Specifies the name of the TCP/IP virtual machine or address space on the local host.

Format: From 1 to 8 alphanumeric characters.

Default: TCPIP

TNPORT=*integer*

Function: Specifies the default port number for Telnet 3270, 3270E, 5250, or NVT devices.

Format: A decimal representation of the port to be used. The value can be in the range from 1 to 65535.

Default: 23

Chapter 8. Defining the message generation deck

This chapter describes the scripting language statements that define the data to be entered at the simulated terminals and devices. It also describes other statements, such as logic test statements, that allow you to control the order of message generation.

Message generation statement categories

The message generation definition statements are divided into the following categories:

- General definition statements
- Systems Network Architecture (SNA) simulation statements
- 3270 simulation statements
- 5250 simulation statements
- CPI-C simulation statements.

General definition statements

The following message generation statements are valid for all terminal types, device types, and line disciplines:

BRANCH	Provides internal branching
CALC	Performs arithmetic operations during message generation
CALL	Provides internal branching with return
CANCEL	Cancels delayed event actions
DATASAVE	Saves data from a terminal or device buffer
DEACT	Deactivates message generation deck logic tests
DELAY	Provides think time override capability
ENDTXT	Defines the end of a message generation deck
EVENT	Performs a post, reset, or signal action on an event
EXIT	Defines a message generation deck user exit
IF[1]	Defines message generation deck logic tests
LABEL	Provides user labels for branching
LOG	Writes user information for message log data set
MSGTXT	Defines the start of a message generation deck
ON	Sets up an action to be taken when an event is signaled
OPCMND	Issues operator commands
PUSH	Places a string on a queue.
QUEUE	Adds a string to a queue.
QUIESCE	Stops message generation and quiesces the terminal
RETURN	Provides return from called subroutines
SET	Sets sequence counters
SETSW	Sets or clears user switches
SETUTI	Sets a new UTI to be active

STOP	Acts as a delimiter without setting the wait indicator
TEXT[1]	Defines data being entered at the terminals
WAIT	Forces devices to wait for response messages
WTO	Writes user information to system console.
WTOABRHD	Writes user information to the system console with abbreviated headers.

Note:

1. For CPI-C simulations, WSim ignores TEXT statements and IF statements that have WHEN=IN or WHEN=OUT specified.

SNA simulation statements

The following statements are valid for SNA terminals, including 3270 SNA, but these statements are ignored for any non-SNA terminals or CPI-C simulations:

CMND	Sends SNA commands
RESP	Overrides normal SNA responses
RH	Overrides normal request and response headers
SYSREQ	Provides SYSREQ key functions
TH	Overrides normal transmission headers.

3270 simulation statements

The following statements are valid for 3270 terminals, but primary logical units will ignore these statements:

BTAB	Simulates the Back Tab key
CHARSET	Selects the device character set
CLEAR	Clears the device buffer to nulls
CLEARPTN	Clears the active partition
COLOR	Selects the display color option
CTAB	Simulates the conditional Tab function
CURSOR	Positions the cursor
CURSRSEL	Simulates the Cursor Select key
DELETE	Simulates the Delete key
DUP	Duplicates characters
ENTER	Simulates the Enter key
EREOF	Simulates the Erase to the End Of Field key
ERIN	Simulates the Erase Input key
ERROR	Provides terminal error simulation
FM	Simulates the Field Mark key
HIGHLITE	Selects the display highlighting option
HOME	Simulates the Home key
INSERT	Simulates the Insert key
JUMP	Defines the next partition as the active partition
LCLEAR	Clears the device buffer to nulls

MONITOR	Displays the simulated 3270 display image being monitored by the display monitor facility (DMF)
NL	Simulates the New Line key
PA*n*	Simulates the Program Access keys one through three (PA1 - PA3)
PF*n*	Simulates the Program Function keys 1 through 24 (PF1 - PF24)
RESET	Simulates the Reset key
SCROLL	Scrolls the partition data within the window/viewport
SELECT	Simulates the Selector pen
STRIPE	Provides magnetic stripe input
TAB	Simulates the Tab key.

5250 simulation statements

The following statements are valid for 5250 terminals, but primary logical units will ignore these statements:

CLEAR	Clears the device buffer to nulls
CMD*n*	Simulates the Program Function keys 1 through 24 (PF1 - PF24)
CURSOR	Positions cursor
DUP	Duplicates characters
ENTER	Simulates the Enter key
ERIN	Simulates the Erase Input key
FLDADV	Simulates the Field Advance key
FLDBKSP	Simulates the Field Backspace key
FLDMINUS	Simulates the Field Minus key
FLDPLUS	Simulates the Field Plus key
HELP	Simulates the Help key
HOME	Simulates the Home key
NL	Simulates the New Line key
PRINT	Simulates the Print Request key
ROLLDOWN	Simulates the Roll Down key
ROLLUP	Simulates the Roll Up key
SELECT	Simulates the Selector pen.

CPI-C simulation statements

The following statement group is valid for Common Programming Interface Communications (CPI-C) transaction programs.

CMxxxx
Simulates CPI-C calls.

"CMxxxx" is a generalized notation used to indicate a category of statements that simulate CPI-C calls. The name of the statement begins with "CM" and is followed by two to four characters to give the four- to six-character name of the statement. Table 9 on page 120 names the statements that are used to simulate CPI-C calls.

Message generation statement descriptions

The remainder of this chapter describes the message generation deck statements.

BRANCH - branch statement

```
[name] BRANCH [LABEL=name]
              [,NAME=name]
```

Function

The BRANCH statement unconditionally changes the course of message generation by specifying a message generation deck name and a statement name to be executed next. This statement does not provide a way to return the message generation course to the statement where the branch took place.

When using the BRANCH statement, you must code either the NAME or the LABEL operand. However, you can also code both operands.

Where

name
: **Function:** Specifies a name used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

LABEL=*name*
: **Function:** Specifies the name of a statement within a message generation deck to which the branch is to be taken.

Format: From one to eight alphanumeric characters.

Default: If omitted, the branch is assumed to be for the beginning of the message generation deck specified by the NAME operand.

NAME=*name*
: **Function:** Specifies the name of the message generation deck to which the branch is to be taken.

Format: From one to eight alphanumeric characters.

Default: If omitted, the name coded for LABEL is assumed to exist within the current message generation deck.

BTAB - back tab key statement

```
[name] BTAB
```

Function

The BTAB statement simulates the Back Tab key on a display device. This statement is valid for 3270 simulation. This statement is a delimiter in some cases.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

CALC - calculate statement

```
[name] CALC LOC=location
           ,VALUE=[±]integer
           [,LENG=integer]
```

Function

The CALC statement sets, adds to, or subtracts from a value at a specific location in a save area or user area. You can use the CALC statement only on data in hexadecimal EBCDIC format.

Note: If you subtract a larger number from a smaller number, the result is always zero.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

LOC=*location*

Function: Specifies the starting location where the arithmetic operation is to be performed.

Format: For the LOC operand, you can code one of the following options. *value* can be any integer from 0 to 32766 or a counter specification whose value is within this range. Zero is the offset to the first byte of the field for the positive offsets (+*value*) and the offset to the last byte of the field for negative offsets (-*value*).

N±*value*
: Perform the arithmetic operation at an offset from the start(+*value*) or back from the end(-*value*) of the network user area.

U±*value*
: Perform the arithmetic operation at an offset from the start(+*value*) or back from the end(-*value*) of the device user area.

N*s***+***value*
: Perform the arithmetic operation at an offset from the start of the network save area where *s* is the network save area and can be any integer from 1 to 4095.

s+value
: Perform the arithmetic operation at an offset from the start of the device save area where *s* is the device save area and can be any integer from 1 to 4095.

Note: If you code **N***s+value* or *s+value*, make sure that you have previously placed data in the save area at that offset.

Default: None. This operand is required.

VALUE=[±]*integer*
: **Function:** Specifies the value to be set, added, or subtracted.

Format: For the VALUE operand, you can code one of the following values:

integer Indicates that the value is to be set at the location specified by the LOC operand, where *integer* is a number from 0 to 999999999.

±*integer*
: Indicates that the value is to be added to or subtracted from the location specified by the LOC operand, where *integer* is a number from 0 to 999999999.

Default: None. This operand is required.

LENG=*integer*
: **Function:** Specifies the amount of data to be used in the arithmetic operation in bytes.

Format: An integer from one to nine.

Default: Length of the integer specified by the VALUE operand.

CALL - call subroutine statement

```
[name] CALL [LABEL=name]
            [,NAME=name]
```

Function

The CALL statement unconditionally alters the course of message generation by specifying a message generation deck name and a label name to be executed next. Unlike the BRANCH statement, however, the CALL statement returns control to the point of the call when a RETURN or ENDTXT statement is processed.

When using the CALL statement, you must code either the NAME or the LABEL operands. However, you can also code both operands.

Where

name
: **Function:** Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

LABEL=*name*

Function: Specifies the name of a statement within a message generation deck to which the branch is to be taken.

Format: From one to eight alphanumeric characters.

Default: If omitted, the call is assumed to be for the beginning of the message generation deck specified by the NAME operand.

NAME=*name*

Function: Specifies the name of the message generation deck being called.

Format: From one to eight alphanumeric characters.

Default: If omitted, the name coded for LABEL is assumed to exist within the current message generation deck.

CANCEL - cancel event statement

```
[name] CANCEL EVENTTAG=tag
```

Function

The CANCEL statement cancels one or more event actions associated with an outstanding timer previously specified by the same simulated resource via execution of an "EVENT with TIME=" statement.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

EVENTTAG=*tag*

Function: Specifies the tag value assigned to one or more EVENT statements with the TIME= operand coded.

Format: From one to eight alphanumeric characters.

Default: None. This field is required.

CHARSET - character set select statement

```
[name] CHARSET {APL|FIELD|PSA|PSB|PSC|PSD|PSE|PSF|PSNAME=byte}
```

Function

The CHARSET statement simulates the action of a 3270 terminal key, which selects the character set for subsequent data input. This statement is valid for 3270 simulation.

If you do not use the CHARSET statement, the character set will be determined by the extended field attribute byte value. This statement is a delimiter in some cases.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

{APL|FIELD|PSA|PSB|PSC|PSD|PSE|PSF|PSNAME=*byte*}

Function: Specifies the character set to be used for subsequent data input from this terminal.

Format: For the CHARSET statement you can code one of the following operands:

APL Selects the special APL/Text character set for APL characters, which must be sent using the two-character graphic escape sequence.

FIELD Specifies that the character set is determined by the extended field attribute byte. Use the FIELD operand to return to the standard EBCDIC character set after an APL selection.

PSA Selects the first Programmed Symbol (PS) character set defined for the device.

PSB Selects the second PS defined for the device.

PSC Selects the third PS defined for the device.

PSD Selects the fourth PS defined for the device.

PSE Selects the fifth PS defined for the device.

PSF Selects the sixth PS defined for the device.

PSNAME=*byte*
Selects the Programmed Symbols character set named by the value for *byte*, where *byte* is either one EBCDIC character or two hexadecimal digits that have a value between X'40' and X'EF'.

Default: FIELD

CLEAR - clear key statement

```
[name] CLEAR
```

Function

The CLEAR statement simulates the Clear key on a display device. This statement is valid for 3270 and 5250 simulation. This statement is a delimiter in some cases.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

CLEARPTN - clear partition key statement

```
[name] CLEARPTN
```

Function

The CLEARPTN statement simulates the Clear Partition key on a 3270 display terminal. This statement is valid for 3270 simulation. This statement is a delimiter in some cases.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

CMD - program function key statement

```
[name] CMDn
```

Function

The CMD statement simulates one of the 24 Program Function keys for a 5250 display device. This statement is valid only for 5250 simulation. This statement is a delimiter in some cases.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

CMD*n*

Function: Specifies the Program Function key to simulate.

Format: A 1- or 2-digit number between 1 and 24.

Default: None. You must specify *n*.

CMND - SNA command statement

```
[name] CMND [COMMAND=cmnd]
            [,DATA=(data...)]
            [,LOG={byte|Ns+value|s+value|N±value|U±value|00}]
            [,MODE=name]
            [,PSEQACT={IGNORE|SET|TEST|TESTSET}]
            [,PSEQVAL={integer|0}]
            [,RESOURCE=name]
            [,RESP=(data...)]
            [,SENSE={sense|'00000000'}]
            [,SON={son|'00'}]
            [,SSEQACT={IGNORE|SET|TEST|TESTSET}]
            [,SSEQVAL={integer|0}]
```

Function

The CMND statement builds an SNA command to be sent by the simulated logical unit and sets the SNA headers in the message to indicate the presence of the command. You can also define your own command or data to be sent with the command. The SNA statement is valid only for SNA simulation and is ignored for CPI-C transaction programs.

You can use the CMND statement to initiate a session from a logon deck, for a secondary LU when INIT=SEC is coded but RESOURCE is not coded on the DEV or LU statement, or for a primary LU when INIT=PRI is coded but RESOURCE is not coded on the LU statement. In the case where a terminal logs off and then enters message generation to logon again, the CMND statement must be the first statement in that message generation deck.

Note: To use the CMND statement, you must code either the COMMAND or DATA operand. However, you can code both operands. This statement is a conditional delimiter.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

COMMAND=*cmnd*

Function: Specifies the command that is to be simulated.

Format: For the COMMAND operand, you can code one of the following values:

SHUTD	Shutdown
SIGNAL	Signal
STSN	Set and test sequence numbers
TERM	Terminate self, format 0
UNBIND	Unbind session.
BID	Bid

BIS	Bracket initiation stopped
CANCEL	Cancel chain
CHASE	Chase responses
CLEAR	Clear
INIT	Initiate self, format 0
LUSTAT	LU status
QUEC	Quiesce at end of chain
RELQ	Release quiesce
RSHUTD	Request shutdown
RTR	Ready to receive
SBI	Stop bracket initiation
SDT	Start data traffic

The following chart shows the valid operands that you can code with the command options listed. The COMMAND options that are not listed below can only be coded singularly on the CMND statement.

Option	Valid Associated Operands
INIT	DATA, LOG, MODE, RESOURCE, RESP
TERM	LOG, RESOURCE, RESP
STSN	LOG, PSEQACT, PSEQVAL, RESP, SSEQACT, SSEQVAL
LUSTAT	LOG, RESP, SENSE
SIGNAL	LOG, RESP, SENSE
UNBIND	SENSE, SON

Default: None. This operand is optional.

Note: You must code either the COMMAND or DATA operand. You can code both the COMMAND and DATA operands only for the INIT command. Do not code both the RESOURCE statement and the CMND statement or duplicate INIT-SELFs will be generated. VTAMAPPL LU simulations only support INIT (initiate-self, format 0).

DATA=(*data*...)

Function: If you are simulating the INIT command, the DATA operand specifies your own data to be sent with the command.

If you do not code the COMMAND operand, the DATA operand specifies your own command. However, if you send an unsupported command, you must also specify the entire request/response unit (RU). Use the TH and RH statements to specify the SNA headers for the command.

Format: You can code any amount of data in this operand, but only a logical maximum of 255 bytes will be sent. The data is enclosed by the text delimiting characters specified on the MSGTXT statement. (The default is left and right parentheses.) You can also continue this statement if the data will not fit on one line. However, if a single text delimiting character is detected in column 71, it indicates the end of the operand, and any data after column 71 is ignored.

You can also use the data field options (see Chapter 9, "Data field options," on page 199). Enter hexadecimal data enclosing the digits within single quotes. To enter a single quote, the special control character (CONCHAR), or a text delimiting character (TXTDLM) as data, use two of the characters. If you enter two text delimiting characters, they must be on the same statement (no continuation between the characters).

For a logical unit defined as LU6.2 within VTAMAPPL, the DATA= operand must be coded with user data along with COMMAND=INIT. The following is an example of user data.

```
DATA=('00'),                           Structured subfields follow
     ('0902E2D5C1E2E5C3D4C7'),         Mode Name
     ('0403000001'),                   Session Instance Identifier
     ('0C04D5C5E3C14BE4E2C5D9F0F1')    Network-Qualified PLU
```

The user data subfields are as follows:

- Length of mode name subfield is X'09'
  - Mode name is SNASVCMG, X'E2D5C1E2E5C3D4C7'
- Length of Session Instance Identifier is X'04'
  - Session Instance Identifier data is X'000001'
- Length of network-qualified subfield is X'0C'
  - Network-Qualified PLU is NETA.USER01, X'D5C5E3C14BE4E2C5D9F0F1'.

Default: None. This operand is optional.

LOG={*byte*|N*s+value*|*s+value*|N±*value*|U±*value*|00}

Function: Specifies a single byte of data to be included in the message log header for this data transmission and for all records for this device until the next CMND or TEXT statement without MORE=YES coded, is generated.

Format: For the LOG operand, you can enter one of the following options. The value for *value* can be any integer from 0 to 32766 or a counter specification whose value is within this range. Zero is the offset to the first byte of the field.

byte One byte of data. If this byte is a single EBCDIC character, the EBCDIC character is logged. If this byte is two hexadecimal digits, the two digits are logged.

N*s+value*
Specifies an offset into a network save area, where *s* is the number of the network save area and is an integer from 1 to 4095.

s+value
Specifies an offset into a device save area, where *s* is the number of the device save area and is an integer from 1 to 4095.

N±*value*
Specifies an offset into a network user area, where *+value* is the offset from the start of the user area and *-value* is the offset back from the end of the user area.

U±*value*
Specifies an offset into a device user area, where *+value* is the offset from the start of the user area and *-value* is the offset back from the end of the user area.

Note: If an area is specified and its length is longer than one byte, the first byte is used. If the area contains no data or *value* specifies an offset that is outside the area, X'00' is used for the log byte and an informational message is logged.

Default: 00

Note: The Loglist Utility will cause both representations (EBCDIC and hexadecimal) of the log character to be printed on the formatted log report under the heading User Data. The log byte is reset by the detection of a CMND or TEXT statement during message generation.

MODE=*name*

Function: Specifies the logmode table entry to be used in constructing the session BIND. This name is the same as the name specified by the LOGMODE operand on a MODEENT statement.

Note: This operand is valid only with the INIT command.

Format: For the MODE operand, you can code the following options. The value for *value* can be any integer from 0 to 32766 or a counter specification whose value is within this range. Zero is the offset to the first byte of the field for positive offsets (+*value*) and the offset to the last byte of the field for negative offsets (-*value*).

name Specifies the mode name to be used, where *name* is from one to eight alphanumeric characters.

N±*value*
Specifies the mode name to be referenced at an offset from the start (+*value*) or back from the end (-*value*) of the network user area.

U±*value*
Specifies the mode name to be referenced at an offset from the start (+*value*) or back from the end (-*value*) of the device user area.

N*s***+***value*
Specifies the mode name to be referenced at an offset from the start of the network save area, where *s* is the network save area and can be any integer from 1 to 4095.

*s***+***value*
Specifies the mode name to be referenced at an offset from the start of the device save area, where *s* is the device save area and can be any integer from 1 to 4095.

Default: None. This operand is optional.

Note: You can use N±*value*, U±*value*, N*s+value*, and *s+value* for variable mode names. The first eight bytes of data beginning at the offset (*value*) comprise the name. For the network and device user area, code the name and then pad it with blanks if the length of the name is less than eight. If the area does not exist or no data is present, the name will consist of eight blanks. Because no validity checking is performed on the name, you can use a name that cannot be expressed as EBCDIC characters. You can put the name to be referenced into the save area or user area with a DATASAVE statement.

PSEQACT={IGNORE|SET|TEST|TESTSET}

Function: Specifies the action to be executed by the set-and-test-sequence-number (STSN) receiver for the primary-to-secondary sequence number.

Note: This operand is valid only with the STSN command.

Format: For the PSEQACT operand, you can code one of the following keywords:

IGNORE
Specifies that this STSN command is to be ignored.

SET Specifies that the primary-to-secondary sequence number of the secondary end user is to be set to the PSEQVAL operand value.

TEST Specifies that the secondary end user must return its primary-to-secondary sequence number in the response RU.

TESTSET
Specifies that the primary-to-secondary sequence number of the control program (CP) manager is to be set to the PSEQVAL operand value, and the secondary end user is to compare that value against its own and respond accordingly.

Default: SET

PSEQVAL={*integer*|0}
Function: Specifies the primary-to-secondary sequence number value to be sent with the STSN.

Note: The operand is valid only with the STSN command.

Format: Any integer from 0 to 65535.

Default: 0

RESOURCE=*name*
Function: Specifies the SNA network name of the logical unit with which the session is to be established or terminated.

Note: This operand is valid only with the INIT and TERM commands.

Format: For the RESOURCE operand, *name* must conform to SNA network naming conventions. You can code one of the following options. The value for *value* can be any integer from 0 to 32766 or a counter specification whose value is within this range. Zero is the offset to the first byte of the field for positive offsets (+*value*) and the offset to the last byte of the field for negative offsets (-*value*).

name Specifies the resource name to be used, where *name* is from one to eight alphanumeric characters.

N±*value*
Specifies the resource name to be referenced at an offset from the start (+*value*) or back from the end (-*value*) of the network user area.

U±*value*
Specifies the resource name to be referenced at an offset from the start (+*value*) or back from the end (-*value*) of the device user area.

N*s***+***value*
Specifies the resource name to be referenced from the start of the network save area, where *s* is the network save area and can be any integer from 1 to 4095.

*s***+***value*
Specifies the resource name to be referenced from the start of the device save area, where *s* is the device save area and can be any integer from 1 to 4095.

Default: The RESOURCE parameter specified on the NTWRK, DEV, or LU statement.

Note: You can use N±*value*, U±*value*, N*s*+*value*, and *s*+*value* for variable resource names. The first eight bytes of data beginning at the offset (*value*) comprise the name. For the network and device user area, code the name and then pad it with blanks if the length of the name is less than eight. If the area does not exist or no data is present, the name will consist of eight blanks. Because no validity checking is performed on the name, you can use a name that cannot be expressed as EBCDIC characters. You can put the name to be referenced into the save area or user area with a DATASAVE statement.

RESP=(*data*...)

Function: Specifies the text data to be used for comparison with a message when an IF statement is encountered with the TEXT=RESP operand.

Format: At least one byte of data enclosed by the text delimiting character specified on the MSGTXT statement. (The default is left and right parentheses.) You can enter a maximum of 25 characters.

Note: You cannot continue the data enclosed in text delimiting characters. The data field options explained in Chapter 9, "Data field options," on page 199 are not valid for this operand.

Enter hexadecimal data within the text delimiting characters by enclosing the digits within single quotes. To enter a single quote, the special control character (CONCHAR), or a text delimiting character (TXTDLM) as data, enter two of the characters.

Default: None. This operand is optional.

SENSE={*sense*|'00000000'}

Function: Specifies the sense information to be sent with the command being simulated.

Note: This operand is valid only for the LUSTAT, SIGNAL, and UNBIND commands. Do not code this operand for any other commands.

Format: Eight hexadecimal digits enclosed within single quotes.

Default: '00000000'

SON={*son*|'00'}

Function: Specifies the session outage notification (SON) code to be sent with the UNBIND command.

Note: This operand is valid only for the UNBIND command. Do not code this operand for any other commands.

Format: Two hexadecimal digits enclosed within single quotes.

Default: '00'

SSEQACT={IGNORE|SET|TEST|TESTSET}

Function: Specifies the action to be executed by the STSN receiver for the secondary-to-primary sequence number.

Note: This operand is valid only with the STSN command.

Format: For the SSEQACT operand, you can enter one of the following keywords:

IGNORE
Specifies that this STSN command is to be ignored.

SET Specifies that the secondary-to-primary sequence number of the secondary end user is to be set to the SSEQVAL operand value.

TEST Specifies that the secondary end user must return its secondary-to-primary sequence number in the response RU.

TESTSET
Specifies that the secondary-to-primary sequence number of the CP manager is to be set to the SSEQVAL operand value, and the secondary end user is to compare that value against its own and respond accordingly.

Default: SET

SSEQVAL={*integer*|0}
Function: Specifies the secondary-to-primary sequence number value to be sent with the STSN.

Note: The operand is valid only with the STSN command.

Format: Any integer from 0 to 65535.

Default: 0

CMxxxx - CPI-C simulation statement group

```
[name] CMxxxx (parameters)
            [,LOG={byte|Ns+value|s+value|N±value|U±value|00}]
```

Function

"CMxxxx"is a generalized notation used to indicate a category of statements that simulate CPI-C calls or a CPI-C extension call. The name of the statement begins with "CM" and is followed by two to four characters to give the four- to six-character name of the statement. Table 9 on page 120 names the statements that are used to simulate CPI-C calls. These statements are valid only for CPI-C simulations.

Note: While CPI-C simulation statements can be coded in the scripting language, it is recommended that STL be used. In STL, the CPI-C parameters and values are predefined STL variables, making the coding of CPI-C statements much easier. For more information about individual CPI-C simulation statements, refer to Part 2, "Guide to using STL and the STL Translator," on page 219.

Some statements in this group are delimiters in some cases. For more information about which statements can be delimiters, refer to the "Understanding Delimiters" chapter of *Creating WSim Scripts*.

Where

name
Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

CM*xxxx*
Function: The name of the statement that simulates either a CPI-C Call or a CPI-C extension call.

Format: *xxxx* can be a two- to four-character identifier appended following CM. Table 9 on page 120 shows the values that *xxxx* can have.

Default: None. You must specify *xxxx*.

parameters

Function: Specifies the parameter information that relates to the particular statement. The parameters vary by statement.

Format: The input parameters that are character data must be specified as save area references (network or device) using either a direct save area reference or the RECALL data field option. The input parameters that are numeric data must be specified as counter references (network, line, term, or device).

The output parameters must be specified as either device save area references (for character data) or device counter references (for numeric data).

In the definitions below, `n` represents the save area or counter number.

CPI-C statements have the following form:

```
name  CMxxxx({Sn|$recall,Sn$|n|$recall,n$},    (character input device save area)
            {Nn|$recall,Nn$},    (character input network save area)
             DCn,                (numeric input device counter)
             TCn,                (numeric input term counter)
             LCn,                (numeric input line counter)
             NCn,                (numeric input network counter)
            {Sn|n},              (character output)
                  .
                  .
             DCn)                (numeric output)
           {[,LOG={byte|Ns+value|s+value|N±value|U±value|00}]}
```

If numeric input data needs to be unique for a specific instance of a transaction program, a device counter must be used. If the data needs to be unique to a specific transaction program, but may be shared across instances, a term counter should be used. If the data needs to be unique to a specific LU, but may be shared across transaction programs, a line counter should be used. If the data may be shared across all LUs, use a network counter.

If character input data needs to be unique for a specific instance of a transaction program, a device save area must be used. If the data may be shared across transaction programs on all LUs, a network save area should be used.

Figure 1 on page 124 gives an example of the use of save areas and counters to specify CPI-C parameters.

Default: None.

LOG={*byte*|N*s*+*value*|*s*+*value*|N±*value*|U±*value*|00}

Function: Specifies a single byte of data to be included in the message log header for this CPI-C statement and for all records for this transaction program until the next CPI-C statement is processed.

Format: For the LOG operand, you can enter one of the following options. The value for *value* can be any integer from 0 to 32766 or a counter specification whose value is within this range. Zero is the offset to the first byte of the field.

byte One byte of data. If this byte is a single EBCDIC character, the EBCDIC character is logged. If this byte is two hexadecimal digits, the two digits are logged.

N*s+value*
: Specifies an offset into a network save area, where *s* is the number of the network save area and is an integer from 1 to 4095.

s+value
: Specifies an offset into a device save area, where *s* is the number of the device save area and is an integer from 1 to 4095.

N±*value*
: Specifies an offset into a network user area, where *+value* is the offset from the start of the user area and *-value* is the offset back from the end of the user area.

U±*value*
: Specifies an offset into a device user area, where *+value* is the offset from the start of the user area and *-value* is the offset back from the end of the user area.

Note: If an area is specified and its length is longer than one byte, the first byte is used. If the area contains no data or *value* specifies an offset that is outside the area, X'00' is used for the log byte and an informational message is logged.

Default: 00

Note: The Loglist Utility will cause both representations (EBCDIC and hexadecimal) of the log character to be printed on the formatted log report under the heading User Data. The log byte is reset by the next CPI-C statement during message generation.

Table 9 shows the relationship between the CPI-C calls and the corresponding CPI-C simulation statements, and provides a prototype for the statements:

Table 9. Statement prototypes of CPI-C calls

CPI-C Call	Statement Prototype
Accept_Conversation	CMACCP (conversation_ID, return_code)
Allocate	CMALLC (conversation_ID, return_code)
Confirm	CMCFM (conversation_id, request_to_send_received, return_code)
Confirmed	CMCFMD (conversation_ID, return_code)
Deallocate	CMDEAL (conversation_ID, return_code)
Extract_Conversation_State	CMECS (conversation_ID, conversation_state, return_code)
Extract_Conversation_Type	CMECT (conversation_ID, conversation_type, return_code)
Extract_Mode_Name	CMEMN (conversation_ID, mode_name, mode_name_length, return_code)
Extract_Partner_LU_Name	CMEPLN (conversation_ID, partner_LU_name, partner_LU_name_length, return_code)
Extract_Sync_Level	CMESL (conversation_ID, sync_level, return_code)
Flush	CMFLUS (conversation_ID, return_code)
Initialize_Conversation	CMINIT (conversation_ID, sym_dest_name, return_code)
Prepare_To_Receive	CMPTR (conversation_ID, return_code)
Set_Fill	CMSF (conversation_ID, fill, return_code)
Set_FM_Header_5_Extension	CMSFM5 (conversation_ID, FMH5_extension, FMH5_extension_length, return_code)

Table 9. Statement prototypes of CPI-C calls (continued)

CPI-C Call	Statement Prototype
Set_Log_Data	CMSLD (conversation_ID, log_data, log_data_length, return_code)
Set_Mode_Name	CMSMN (conversation_ID, mode_name, mode_name_length, return_code)
Set_Partner_LU_Name	CMSPLN (conversation_ID, partner_LU_ name, partner_LU_name_length, return_code)
Set_Prepare_To_Receive_Type	CMSPTR (conversation_ID, prepare_to_receive_type, return_code)
Set_Receive_Type	CMSRT (conversation_ID, receive_type, return_code)
Receive	CMRCV (conversation_ID, receive_buffer, requested_length, data_received, received_length,status_received, request_to_send_received, return_code)
Request_To_Send	CMRTS (conversation_ID, return_code)
Send_Data	CMSEND (conversation_ID, send_buffer, send_length, request_to_send_received, return_code)
Send_Error	CMSERR (conversation_ID, request_to_send_received, return_code)
Set_Conversation_Type	CMSCT (conversation_ID, conversation_type, return_code)
Set_Deallocate_Type	CMSDT (conversation_ID, deallocate_type, return_code)
Set_Error_Direction	CMSED (conversation_ID, error_direction, return_code)
Set_Return_Control	CMSRC (conversation_ID, return_control, return_code)
Set_Send_Type	CMSST (conversation_ID, send_type, return_code)
Set_Sync_Level	CMSSL (conversation_ID, sync_level, return_code)
Set_TP_Name	CMSTPN (conversation_ID, TP_name, TP_name_length, return_code)
Test_Request_To_Send_Received	CMTRTS (conversation_ID, request_to_send_received, return_code)

Following are definitions of the individual parameters of the statements:

conversation_ID
: Specifies the identifier assigned to the conversation. When *conversation_ID* is used in CMACCP or CMINIT, the format is character output. In all other statements, the format is character input.

conversation_state
: Specifies the conversation state that is returned to the local program. The format is numeric output.

conversation_type
: Specifies the type characteristic of the conversation. When used in CMECT, the format is numeric output. When used in CMSCT, the format is numeric input.

data_received
: Specifies whether the program received data. The format is numeric output.

deallocate_type
: Specifies the type of deallocation to be performed. The format is numeric input.

error_direction
: Specifies the direction of data flow in which the program detected an error. The format is numeric input.

fill Specifies whether the program is to receive data in terms of the logical-record format of the data or independent of the logical-record format. The format is numeric input.

FMH5_extension
Specifies the information beyond the base FM Header type 5. The format is character input.

FMH5_extension_length
Specifies the total length of the FMH-5 extension. The format is numeric input.

log_data
Specifies the program-unique error information that is to be logged. The format is character input.

log_data_length
Specifies the length of the program-unique error information. The format is numeric input.

mode_name
Specifies the name of the mode that designates the properties of the session to be allocated to the conversation. When used in CMEMN, the format is character output. When used in CMSMN, the format is character input.

mode_name_length
Specifies the length of the *mode_name* parameter. When used in CMEMN, the format is numeric output. When used in CMSMN, the format is numeric input.

partner_LU_name
Specifies the name of the logical unit where the remote program is located. When used in CMEPLN, the format is character output. When used in CMSPLN, the format is character input.

partner_LU_name_length
Specifies the length of the *partner_LU_name* parameter. When used in CMEPLN, the format is numeric output. When used in CMSPLN, the format is numeric input.

prepare_to_receive_type
Specifies the type of prepare-to-receive processing to be performed for this conversation. The format is numeric input.

receive_buffer
Specifies the variable in which the program is to receive data. The format is character output.

received_length
Specifies the variable containing the amount of data the program received. The format is numeric output.

receive_type
Specifies the type of receive operation that is to be performed. The format is numeric input.

requested_length
Specifies the maximum amount of data the program is to receive. The format is numeric input.

request_to_send_received
Specifies the variable containing an indication of whether or not a request-to-send notification has been received. The format is numeric output.

return_code
Specifies the result of the statement execution. The format is numeric output.

return_control
Specifies when a program receives control back after issuing a CMALLC statement. The format is numeric input.

send_buffer
Specifies the information to be sent to the conversation partner. The format is character input.

send_length
Specifies the size of the *send_buffer* parameter and the number of bytes to be sent on the conversation. The format is numeric input.

send_type
Specifies what, if any, information is to be sent in addition to any data supplied on the CMSEND call, and whether the data is to be sent immediately or buffered. The format is numeric input.

status_received
Specifies the variable containing an indication of the conversation status. The format is numeric output.

sym_dest_name
Specifies the symbolic name which points to an entry in the side information table. The format is character input.

sync_level
Specifies the synchronization level characteristic of the conversation. When used in CMESL, the format is numeric output. When used in CMSSL, the format is numeric input.

TP_name
Specifies the name of the remote program. The format is character input.

TP_name_length
Specifies the length of the *TP_name* parameter. The format is numeric input.

Note: For detailed information regarding the CPI-C call definitions, refer to

- Part 2, "Guide to using STL and the STL Translator," on page 219 for the statement descriptions and general information relating to the statement definitions
- *Systems Application Architecture CPI Communications Reference*

Figure 1 on page 124 is an example of the use of save areas and counters to specify CPI-C parameters.

```
*************************************************
* Device save area usage:
*  1=conversation id
*  2=destination name
*  3=send buffer
*
* Device counter usage:
*  dc1=return code
*  dc2=send length
*  dc3=request-to-send received
*
* Set the symbolic destination name to "SERVER".
        DATASAVE AREA=2,TEXT=(SERVER)
*
* Initialize a conversation with "SERVER".
        CMINIT (1,2,DC1)
*
* Allocate the conversation with "SERVER".
        CMALLC (1,DC1)
*
* Setup the send buffer and length.
        DATASAVE AREA=3,TEXT=(DATA TO SEND)
        SET  DC2=LENG(3)
*
* Send the data to "SERVER".
        CMSEND (1,3,DC2,DC3,DC1)
*
* Deallocate the conversation with "SERVER".
        CMDEAL (1,DC1)
*************************************************
```

Figure 1. Use of save areas and counters to specify CPI-C parameters

COLOR - display color select statement

[*name*] COLOR {BLUE|FIELD|GREEN|PINK|RED|TURQUOISE|WHITE|YELLOW}

Function

The COLOR statement simulates the action of the 3270 device keys, which selects the color for displaying subsequent data input. This statement is valid only for 3270 simulation.

If you do not code COLOR statement, the color will be selected by the extended field attribute byte value. This statement is a delimiter in some cases.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

{BLUE|FIELD|GREEN|PINK|RED|TURQUOISE|WHITE|YELLOW}

Function: Specifies the color to be used for displaying subsequent data input from this device.

Format: BLUE, FIELD, GREEN, PINK, RED, TURQUOISE, WHITE, or YELLOW.

Note: Use the FIELD operand to select the color defined by the extended field attribute byte.

Default: FIELD

CTAB - conditional tab statement

```
[name] CTAB
```

Function

The CTAB statement conditionally tabs to the next field only if the cursor is not currently at the beginning of a field. This statement is valid for 3270 simulation. This statement is a delimiter in some cases.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

CURSOR - position cursor statement

```
[name] CURSOR [COLUMN={value|1}]
              [,DOWN=value]
              [,LEFT=value]
              [,OFFSET=value]
              [,RIGHT=value]
              [,ROW={value|1}]
              [,UP=value]
```

Function

The CURSOR statement simulates the action of the cursor positioning keys on the display device. This statement is valid for 3270 and 5250 simulation. This statement is a delimiter in some cases.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

COLUMN={*value*|1}

Function: Specifies the column position of the cursor setting.

Format: *value* can be an integer from 1 to 255 or a counter specification whose value is within this range.

Default: 1

Note: If you code the COLUMN operand, do not code the DOWN, LEFT, OFFSET, RIGHT, and UP operands.

DOWN=*value*

Function: Specifies the number of positions down for the cursor.

Format: *value* can be an integer from 1 to 255 or a counter specification whose value is within this range.

Default: None.

Note: If you code the DOWN operand, do not code the COLUMN, LEFT, OFFSET, RIGHT, ROW, and UP operands.

LEFT=*value*

Function: Specifies the number of positions to the left for the cursor.

Format: *value* can be an integer from 1 to 255 or a counter specification whose value is within this range.

Default: None.

Note: If you code the LEFT operand, do not code the COLUMN, DOWN, OFFSET, RIGHT, ROW, and UP operands.

OFFSET=*value*

Function: Specifies the location at which the cursor will be positioned as an offset from the beginning of the display buffer. For a 3270 with multiple partitions defined, this operand specifies the cursor offset from the beginning of the presentation space of the currently active partition.

Format: *value* can be an integer from 0 to 32766 or a counter specification whose value is within this range.

Default: None.

Note: If you code the OFFSET operand, do not code the COLUMN, DOWN, LEFT, RIGHT, ROW, and UP operands.

RIGHT=*value*

Function: Specifies the number of positions to the right for the cursor.

Format: *value* can be an integer from 1 to 255 or a counter specification whose value is within this range.

Default: None.

Note: If you code the RIGHT operand, do not code the COLUMN, DOWN, LEFT, OFFSET, ROW, and UP operands.

ROW={*value***|1}**

Function: Specifies the row position of the cursor setting.

Format: *integer* can be an integer from 1 to 255 or a counter specification whose value is within this range.

Default: 1

Note: If you code the ROW operand, do not code the DOWN, LEFT, OFFSET, RIGHT, and UP operands.

`UP=`*`value`*

Function: Specifies the number of positions up for the cursor.

Format: *value* can be an integer from 1 to 255 or a counter specification whose value is within this range.

Default: None.

Note: When multiple partitions are defined for a 3270, the ROW and COLUMN specifications reference the display as you would see it, which could include data from more than one partition. The partition owning the area of the display to which the cursor was positioned with the ROW and COLUMN specifications becomes the currently active partition.

CURSRSEL - cursor select key statement

Where

```
[name] CURSRSEL
```

Function

The CURSRSEL statement simulates the action of the Cursor Select key. This statement is valid for 3270 simulation. This statement is a delimiter in some cases.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

DATASAVE - save data statement

```
[name] DATASAVE [AREA={s|Ns|N±value|U±value|1}]
                [,CONVERT={YES|NO}]
                [,COUNT=value]
                [,FUNCTION={BITAND}
                           {BITOR}
                           {BITXOR}
                           {B2X}
                           {CENTER}
                           {COPIES}
                           {DELWORD}
                           {OVERLAY}
                           {REVERSE}
                           {SPACE}
                           {STRIP}
                           {STRIPL}
                           {STRIPT}
                           {SUBWORD}
                           {X2B}
                           {X2C}
                           {INSERT}
                           {DELETE}
                           {LEFT}
                           {RIGHT}
                           {DBCSADD}
                           {DBCSADJ}
                           {DBCSDEL}
                           {DBCS2SB}
                           {SB2DBCS}
                           {SB2MDBCS}
                           {TRANSLATE}]
                [,INSERT=(data)]
                [,LENG={value|100}]
                [,LOC={B±value}
                      {C±value}
                      {D+value}
                      {TH+value}
                      {RH+value}
                      {RU+value}
                      {(row,col)}
                      {*}]
                [,PAD={char|blank}]
                [,PLENG=value]
                [,POS=value]
                [,TABLEI=data]
                [,TABLEO=data]
                [,TEXT=([data])]
                [,TEXT2=([data])]
```

Function

The DATASAVE statement saves device buffer data or explicitly specified data into a save area or user area. You can also use the DATASAVE statement to clear a specified save area or user area.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

AREA={*s*|N*s*|N±*value*|U±*value*|1}
Function: Specifies whether one of the save areas or user areas is to be used to save the data.

Format: For the AREA operand, you can enter one of the following options. The value for *value* can be any integer from 0 to 32766 or a counter specification whose value is within this range. Zero is the offset to the first byte of the field for positive offsets (+*value*) and the offset to the last byte of the field for negative offsets (-*value*).

s Specifies a device save area to be used to save the data, where *s* is an integer from 1 to 4095.

N*s* Specifies a network save area to be used to save the data, where *s* is an integer from 1 to 4095.

N±*value*
Specifies a network user area to be used to save the data, where +*value* is the offset from the start of the user area and -*value* is the offset back from the end of the user area.

U±*value*
Specifies a device user area to be used to save the data, where +*value* is the offset from the start of the user area and -*value* is the offset back from the end of the user area.

Note: If *value* specifies an offset that is outside the user area, no data is saved and an informational message is written to the log data set.

Default: 1, indicating device save area number 1.

CONVERT={YES|NO}
Function: Specifies that the data to be saved will be converted from a hexadecimal format to a printable format before being placed in the save area or user area.

Format: For the CONVERT operand, you can code one of the following values:

YES Specifies that each byte of data to be saved will be converted into two bytes, which are the EBCDIC representations of each of the two hexadecimal digits in the byte being saved. For example, the hexadecimal character X'C1' would be converted to the hexadecimal characters X'C3F1'.

Note: If CONVERT=YES is coded, the amount of space required in the save area or user area will be twice the value specified by the LENG operand.

NO Specifies that the data to be saved will not be converted.

Default: NO

COUNT=*value*
Function: When FUNCTION=COPIES is coded, *value* specifies the number of copies to create.

Note: This operand is only valid when the TEXT operand is coded and when FUNCTION=COPIES is coded.

Format: An integer from 1 to 32767 or a counter specification whose value is within this range.

Default: None. This operand is required when FUNCTION=COPIES.

FUNCTION={BITAND}

{BITOR}

{BITXOR}

{B2X}

{CENTER}

{COPIES}

{DELWORD}

{OVERLAY}

{REVERSE}

{SPACE}

{STRIP}

{STRIPL}

{STRIPT}

{SUBWORD}

{X2B}

{X2C}

{INSERT}

{DELETE}

{LEFT}

{RIGHT}

{DBCSADD}

{DBCSADJ}

{DBCSDEL}

{DBCS2SB}

{SB2DBCS}

{SB2MDBCS}

{TRANSLATE}

Function: Specifies that a string manipulation function be performed on the TEXT operand.

Note: This operand is only valid when the TEXT operand is coded.

Format: You can code one of the following values for the FUNCTION operand:

BITAND
Specifies that the data coded for the TEXT and TEXT2 operands will be logically AND'ed together, bit by bit. If the PAD operand is specified, the shorter text data of TEXT and TEXT2 will be extended with the PAD character on the right before carrying out the logical operation.

BITOR
Specifies that the data coded for the TEXT and TEXT2 operands will be logically inclusive-OR'ed together, bit by bit. If the PAD operand is specified, the shorter text data of TEXT and TEXT2 will be extended with the PAD character on the right before carrying out the logical operation.

BITXOR
Specifies that the data coded for the TEXT and TEXT2 operands will be logically eXclusive-OR'ed together, bit by bit. If the PAD operand is specified, the shorter text data of TEXT and TEXT2 will be extended with the PAD character on the right before carrying out the logical operation.

B2X Specifies that the data coded for the TEXT operand will be converted to hexadecimal.

CENTER
Specifies that the data coded for the TEXT operand will be centered with the PAD character around both ends until a length of PLENG is reached.

COPIES
Specifies that the data coded for the TEXT operand will be copied and concatenated by the number specified on the COUNT operand.

DELWORD
Specifies that the data coded for the TEXT operand will be deleted starting at the word corresponding to the value coded on the POS operand for a length of PLENG words.

OVERLAY
Specifies that the data coded for the INSERT operand is to be overlaid on the data specified by the TEXT operand. If the PLENG operand is also coded, the data specified by the INSERT operand is padded or truncated to that length before it is overlaid on the TEXT data. If the POS operand is also specified, the INSERT data is overlaid starting at that position in the TEXT data. If the PAD operand is specified, the specified PAD character is used, if necessary, to extend the TEXT or INSERT data to meet the POS or PLENG specifications respectively.

REVERSE
Specifies that the data coded for the TEXT operand will be reversed.

SPACE iref refid=space02.
Specifies that the data coded for the TEXT operand will have a space of length PLENG between each word. If the PAD operand is coded, the PAD value is used instead of a space.

STRIP
Specifies that the data coded for the TEXT operand will have both leading and trailing blanks removed. If the PAD operand is coded, the PAD value will be used as the character to remove instead of blanks.

STRIPL
Specifies that the data coded for the TEXT operand will have leading blanks removed. If the PAD operand is coded, the PAD value will be used as the character to remove instead of blanks.

STRIPT
Specifies that the data coded for the TEXT operand will have trailing

blanks removed. If the PAD operand is coded, the PAD value will be used as the character to remove instead of blanks.

SUBWORD
Specifies that a substring of the data coded for the TEXT operand will start at the *n*th word coded by the *n* value of the POS operand and up to the PLENG value coded for the number of words. If the PLENG value is not coded, the default will be the number of remaining words in the data coded for the TEXT operand.

X2B Specifies that the data coded for the TEXT operand will be converted to binary. Each hexadecimal character is converted to a string of four binary digits. Blanks are ignored.

X2C Specifies that the data coded for the TEXT operand will be converted to character. A leading 0 will be added if necessary to make an even number of hexadecimal digits.

INSERT
Specifies that the data coded for the INSERT operand is to be inserted into the data specified by the TEXT operand. If the PLENG operand is also coded, the data specified by the INSERT operand is padded or truncated to that length before it is inserted. If the POS operand is also specified, the INSERT data is inserted after that position in the TEXT data. If the PAD operand is specified, the INSERT data is padded with the specified pad character.

DELETE
Specifies that a substring of the data coded for the TEXT operand is to be deleted, starting at the position specified by the POS operand. If the PLENG operand is also specified, its value is the number of characters deleted.

LEFT Specifies that the data coded for the TEXT operand is to be padded or truncated on the right to the length specified by the PLENG operand. If the PAD operand is specified, the data is padded with the specified pad character.

RIGHT
Specifies that the data coded for the TEXT operand is to be padded or truncated on the left to the length specified by the PLENG operand. If the PAD operand is specified, the data is padded with the specified pad character.

DBCSADD
Adds SO/SI characters to the data coded for the TEXT operand and saves the result in the user area or save area specified by the AREA operand.

DBCSADJ
Deletes SI/SO character pairs from the data coded for the TEXT operand and saves the result in the user area or save area specified by the AREA operand.

DBCSDEL
Deletes SO/SI characters from the data coded for the TEXT operand and saves the result in the user area or save area specified by the AREA operand.

DBCS2SB
Converts ward 42 (EBCDIC) DBCS data to SBCS data in the data coded for the TEXT operand and saves the result in the user area or save area specified by the AREA operand.

SB2DBCS
Converts SBCS data to ward 42 (EBCDIC) DBCS data in the data coded for the TEXT operand and saves the result in the user area or save area specified by the AREA operand.

SB2MDBCS
Converts SBCS data to ward 42 (EBCDIC) DBCS data with SO/SI characters wrapped around it in the data coded for the TEXT operand and saves the result in the user area or save area specified by the AREA operand.

TRANSLATE
Specifies that the data coded for the TEXT operand is to be translated using the TABLEI, TABLEO, and PAD operand values. The data specified by TABLEI is searched for each character of the TEXT string. If found, the corresponding character in the data specified by the TABLEO operand replaces the TEXT character in the result. If the TEXT character is not found in the TABLEI data, it is left unchanged in the result. If a character occurs more than once in the TABLEI data, the first occurrence is the one used. If the TABLEO data is shorter than the TABLEI data, the TABLEO data is padded with the character specified by the PAD operand.

If TABLEO, TABLEI, and PAD are all omitted, lower case characters in the TEXT data are translated to upper case and all other characters are unchanged.

For individual defaults for TABLEO, TABLEI, and PAD, see those operand descriptions.

Default: None. This operand is optional.

INSERT=(*data*)
Function: Specifies the string data to be inserted into the data coded in the TEXT operand.

Note: This operand is only valid when the TEXT operand is coded and when FUNCTION=INSERT or FUNCTION=OVERLAY is coded.

Format: You can code any amount of data for this operand. If the resulting string is longer than the space available in the save area, the data is truncated and a message is written to the log data set.

The data is enclosed by the text delimiting character specified on the MSGTXT statement. (The defaults are left and right parentheses.) You can also continue the data.

You can use the data field options (see Chapter 9, “Data field options,” on page 199). To enter hexadecimal data, enclose the digits within single quotes. To enter a single quote, a special control character (CONCHAR), or a text delimiter (TXTDLM) as data, enter two of the characters. If you enter two text delimiting characters, they must be on the same statement; you cannot continue the statement between the characters.

Default: None.

LENG={*value*|<u>100</u>}

Function: Specifies, in bytes, the amount of data to be saved for later recall. If the specified length is greater than the space available in the save area or user area, the available length is used, and an informational message is written to the log data set. If the LENG operand value is greater than the length of the data in the device buffer, only the available data is saved.

Note: This operand is not valid if the TEXT operand is coded.

Format: *value* can be an integer from 1 to 32767 or a counter specification whose value is within this range.

Default: 100

LOC={B±*value*|C±*value*|D+*value*|TH+*value*|RH+*value*|RU+*value*|(*row*,*col*)|*}

Function: Specifies the location of the data to be saved.

If the DATASAVE statement is encountered as a result of an IF statement execute function (that is, THEN=E*name-label*) with WHEN=IN coded, then data is saved from the device input buffer. Otherwise, data is saved from the device output buffer. For display devices, the output buffer is the screen image if B±, C±, or (*row,col*) is specified.

Note: This operand is not valid if the TEXT operand is coded.

Format: For the LOC operand, you can code one of the following options. *value* can be an integer from 0 to 32766 or a counter specification whose value is within this range. Zero is the offset to the first byte of the field for positive offsets (+*value*) and the offset to the last byte of the field for negative offsets (-*value*).

B±*value*
: +*value* specifies that the data to be saved is at an offset from the start of the device buffer, excluding any headers. For display devices, -*value* indicates an offset back from the end of the screen image buffer. For non-display devices, -*value* indicates an offset back from the end of the data in the device buffer.

C±*value*
: +*value* specifies an offset from the current cursor position. -*value* specifies an offset back from the cursor. You should use this value only with display devices.

D+*value*
: +*value* specifies an offset from the start of the data stream.

TH+*value*
: +*value* specifies an offset from the start of the transmission header.

RH+*value*
: +*value* specifies an offset from the start of the request header.

RU+*value*
: +*value* specifies an offset from the start of the request unit.

(*row***,***col***)**
: Indicates that the test is to be made at the specified row and column of the screen image of a display device, where *row* and *col* may be an integer from 1 to 255 or a counter specification whose value is within this range. You should use this value only with display devices.

*
: Specifies that data is to be saved from the input area location that was

coded on the last logic test that was executed and had its THEN action taken. This option is valid only when the DATASAVE statement is encountered as a result of a logic test execute function, (for example, THEN=E*name-label*). The data to be saved begins with the first character of the data string that satisfied the logic test. If the last logic test executed did not specify a device buffer location, or its THEN action was not an execute function, no data is saved and an informational message is written to the log data set.

Default: B+0

PAD={*char*|blank}

Function: Specifies the character to be used as padding when BITAND, BITOR, BITXOR, CENTER, OVERLAY, SPACE, STRIP, STRIPL, STRIPT, INSERT, LEFT, RIGHT, or TRANSLATE is specified for the FUNCTION operand.

Note: This operand is only valid when the TEXT operand is coded along with the specified functions.

Format: *char* is a 1-character string constant or a 2-character hexadecimal constant. When you use a special character (such as a quote or parenthesis) as padding, do not double it.

Default: When FUNCTION=BITAND is coded, the default is X'FF'. When BITOR or BITXOR is coded for the FUNCTION operand, the default is X'00'. Otherwise, the default is blank (X'40').

PLENG=*value*

Function: When INSERT or OVERLAY is coded for the FUNCTION operand, *value* specifies the number of characters to which the data specified by the INSERT operand is truncated or padded when data is inserted into, or overlaid on, the data specified by the TEXT operand. When FUNCTION=DELETE is coded, *value* specifies the number of characters to be deleted. When FUNCTION=SPACE is coded, *value* specifies the number of blanks or characters specified by the PAD operand, to place between the words. If *value* is 0, all blanks are removed. Leading and trailing blanks are always removed. When CENTER, LEFT or RIGHT is coded for the FUNCTION operand, *value* specifies the number of characters to which the data coded for the TEXT operand is truncated or padded. When DELWORD or SUBWORD is coded for the FUNCTION operand, *value* specifies the number of words to delete or return.

Note: This operand is only valid when the TEXT and FUNCTION operands are coded.

Format: An integer from 0 to 32767 or a counter specification whose value is within the range.

Default: When INSERT or OVERLAY is coded for the FUNCTION operand, the default is the length of the data specified by the INSERT operand. When FUNCTION=DELETE is coded, the default is the length of the TEXT data after the position specified by the POS operand. When FUNCTION=SPACE is coded, the default is 1. When CENTER, LEFT or RIGHT is coded for the FUNCTION operand, there is no default, and a value is required. When DELWORD or SUBWORD is coded for the FUNCTION operand, the default is the number of words left in the TEXT data after the word in the position specified by the POS operand.

POS=*value*

Function: When FUNCTION=INSERT is coded, *value* specifies the position in the TEXT data after which the data coded for the INSERT operand is inserted. When FUNCTION=OVERLAY is coded, *value* specifies the first position in the TEXT data to be overlaid with the data coded for the INSERT operand. When FUNCTION=DELETE is coded, *value* specifies the first position in the TEXT data to be deleted. When DELWORD or SUBWORD is coded for the FUNCTION operand, *value* specifies the word position in the TEXT data.

Note: This operand is only valid when the TEXT operand is coded and when DELETE, DELWORD, INSERT, OVERLAY or SUBWORD is coded on the FUNCTION operand.

Format: When FUNCTION=INSERT is coded, *value* can be an integer from 0 to 32766 or a counter specification whose value is within that range. If *value* is greater than the length of the data specified in the TEXT operand, pad characters are inserted after the TEXT data, with the INSERT data following. If *value* is zero, the data coded in the INSERT operand is inserted before the beginning of the data coded in the TEXT operand.

When FUNCTION=OVERLAY is coded, *value* can be an integer from 1 to 32767 or a counter specification whose value is within that range. If *value* is greater than the length of the data specified in the TEXT operand, pad characters are added to the TEXT data.

When DELETE, DELWORD or SUBWORD is coded on the FUNCTION operand, *value* can be an integer from 1 to 32767 or a counter specification whose value is within that range.

Default: When FUNCTION=INSERT is coded, the default is 0. When FUNCTION=OVERLAY is coded, the default is 1. When DELETE, DELWORD or SUBWORD is coded for the FUNCTION operand, there is no default and a value must be specified.

TABLEI=(data)

Function: Specifies the input table to be used with the translation of the data coded in the TEXT operand.

Note: This operand is only valid when the TEXT operand is coded and when FUNCTION=TRANSLATE is coded.

Format: You can code any amount of data for this operand, but you would not normally code more than 256 characters. See the description of the TRANSLATE function under the FUNCTION operand for details of how this data is used.

The data is enclosed by the text delimiting character specified on the MSGTXT statement. (The defaults are left and right parentheses.) You can also continue the data.

You can use the data field options (see Chapter 9, “Data field options,” on page 199). To enter hexadecimal data, enclose the digits within single quotes. To enter a single quote, a special control character (CONCHAR), or a text delimiter (TXTDLM) as data, enter two of the characters. If you enter two text delimiting characters, they must be on the same statement; you cannot continue the statement between the characters.

Default: If TABLEO or PAD is coded, the default for TABLEI is the 256 byte string of hexadecimal codes 00,01,02,..., FF. Otherwise, no default.

TABLEO=(data)

Function: Specifies the output table to be used with the translation of the data coded in the TEXT operand.

Note: This operand is only valid when the TEXT operand is coded and when FUNCTION=TRANSLATE is coded.

Format: You can code any amount of data for this operand, but you would not normally code more than 256 characters. See the description of the TRANSLATE function under the FUNCTION operand for details of how this data is used.

The data is enclosed by the text delimiting character specified on the MSGTXT statement. (The defaults are left and right parentheses.) You can also continue the data.

You can use the data field options (see Chapter 9, "Data field options," on page 199). To enter hexadecimal data, enclose the digits within single quotes. To enter a single quote, a special control character (CONCHAR), or a text delimiter (TXTDLM) as data, enter two of the characters. If you enter two text delimiting characters, they must be on the same statement; you cannot continue the statement between the characters.

Default: If TABLEI or PAD is coded, the default for TABLEO is a string of repeated PAD characters equal in length to the data specified or defaulted for TABLEI. Otherwise, no default.

TEXT=([*data*])

Function: Specifies the text data to be saved in the save area or user area based on the function specified for the FUNCTION operand.

Note: If this operand is coded, the LENG and LOC operands are not valid.

Format: You can code any amount of data for this operand. If the data is longer than the space available in the save area or user area, it is truncated, and an informational message is written to the log data set.

The data is enclosed by the text delimiting characters specified on the MSGTXT statement. (The default is left and right parentheses.) You can also continue the data.

You can use the data field options (see Chapter 9, "Data field options," on page 199). Enter hexadecimal data by enclosing the digits within single quotes. To enter a single quote, a special control character (CONCHAR), or a text delimiting character (TXTDLM) as data, enter two of the characters. If two text delimiting characters are entered, they must be on the same statement (no continuation between the characters).

If the TEXT operand specifies only two text delimiting characters with no intervening data (for example, TEXT=(),), a clear function is executed. Specifying TEXT=() also frees a dynamically allocated save area. If the AREA operand specifies a save area number, the length of data saved in that save area is set to zero. If the AREA operand specifies an offset into a user area, the user area is usually cleared to binary zeros from the specified offset to the end. This will not be the case if the FUNCTION operand or CONVERT=YES has also been specified. In those cases, the user area is unchanged.

Default: None. This operand is optional.

TEXT2=([*data*])

Function: Specifies the text data to be used when BITAND, BITOR or BITXOR is coded for the FUNCTION operand.

Note: If this operand is coded, the LENG and LOC operands are not valid.

Format: See the TEXT description above.

Default: None. This operand is optional.

DEACT - deactivate logic test and ON condition statement

```
[name] DEACT {IFS={(num,...)|ALL}}
             {ONEVENTS={(event,...)|ALL}}
```

Function

The DEACT statement globally or selectively deactivates message generation logic tests before the time that WSim normally performs logic test deactivation. This statement deactivates the indicated logic tests even if STATUS=HOLD was specified. It also globally or selectively deactivates ON conditions.

The DEACT statement does not affect network-level IF statements, and it does not affect message generation IF statements that specify WHEN=IMMED.

Note: Code either the IFS operand or the ONEVENTS operand. Both operands cannot appear on the same DEACT statement.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

IFS={(*num*,...)|ALL}

Function: Specifies the logic test numbers in the message generation deck that are to be deactivated.

Note: This operand cannot be continued. The number of *nums* allowed is limited by those that can be coded on a single statement.

Format: For the IFS operand, you can code one of the following values:

num,...	Specifies a list of unframed integers from 0 to 255, separated by commas and enclosed in parentheses. Each *num* corresponds to the name field of an IF statement.
ALL	Specifies all currently active message generation deck IF statements are to be deactivated.

Default: None. You must code either the IFS or ONEVENTS operand on a DEACT statement.

ONEVENTS={(*event*,...)|ALL}

Function: Specifies the events named by ON statements which have been activated for this terminal and which are to be deactivated.

Note: This operand cannot be continued. The number of *events* allowed is limited by those that can be coded on a single statement.

Format: For the ONEVENTS operand, you can code one of the following values:

event... Specifies a list of event names, separated by commas and enclosed in parentheses. Each *event* must be a 1- to 8-character alphanumeric name or a reference to a user area or save area.

ALL Specifies that all currently active ON conditions for this terminal are to be deactivated.

Default: None. Either the IFS or ONEVENTS operand must be coded on a DEACT statement.

DELAY - delay statement

```
[name] DELAY TIME={integer}
                 {A(integer)}
                 {F(integer)}
                 {R(value1[,value2])}
                 {T(integer)}
                 {cntr}
               [,UTI=uti]
```

Function

The DELAY statement specifies an inter-message delay that overrides the normal think time for the device, as specified by the DELAY operand on the network configuration statements. This statement affects only the delay after the message that is currently being generated.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

TIME={*integer*|A(*integer*)|F(*integer*)|R(*value1*[,*value2*])|T(*integer*)|*cntr*}

Function: Specifies a value that, when multiplied by the user time interval (UTI), provides a delay that overrides the DELAY operand on the NTWRK, DEV, TP, APPCLU, and LU statements.

Format: For the TIME operand, you can code one of the following values:

integer Specifies a fixed value, where *integer* is an integer ranging from 0 to 2147483647.

A(*integer*)
Specifies a delay to be chosen randomly from the range 0 to 2 times the integer, where *integer* is from 0 to 1073741823. The average delay will be *integer*.

F(*integer*)
Specifies a fixed value (the same as TIME=*integer*), where *integer* is from 0 to 2147483647.

R(*integer*)
Specifies a delay chosen randomly from the range on the RN statement referenced by the specified integer, where *integer* is from 0 to 225.

R(*value1*,*value2***)**
Specifies a delay chosen randomly in the range of low (*value1*) to high (*value2*), where *value1* is from 0 to 2147483646 and *value2* is from 1 to 2147483647 or counter specifications whose values are within these ranges. *value1* must be less than *value2*. If either value is a counter and *value1* is less than *value2*, the DELAY statement is processed as if DELAY TIME=F(0) was coded.

T(*integer***)**
Specifies a delay chosen randomly from the rate table on the RATE statement referenced by the specified integer, where *integer* is from 0 to 225.

cntr Specifies a delay chosen by the counter specification whose value is in the range from 0 to 2147483647.

Default: None. This operand is required.

UTI=*uti*
Function: Specifies a UTI which is to be used in calculating this delay. *uti* must reference a UTI statement defined within the network configuration statements.

Format: From one to eight alphanumeric characters.

Default: None. This operand is optional.

DELETE - delete key statement

```
[name] DELETE CHARS=value
```

Function

The DELETE statement simulates the action of the delete key on the 3270 display device.

Where

name
Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

CHARS=*value*
Function: Specifies the number of characters to be deleted from the simulated screen.

Format: For the CHARS operand, *value* can be one of the following:

integer Specifies a fixed value, where *integer* is an integer from 1 to 255.

cntr Specifies a counter specification whose value is within the range of 1 to 255.

Default: None. This operand is optional.

Note: Screen data will be shifted left on the simulated screen as data is deleted.

DUP - dup key statement

```
[name] DUP
```

Function

The DUP statement simulates the action of the duplicate key on a display device. This statement is valid for 3270 and 5250 simulation. This statement is a delimiter in some cases.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

ENDTXT - end message generation deck statement

```
[name] ENDTXT
```

Function

The ENDTXT statement indicates the end of a message generation deck. If no message generation deck calls are outstanding, the ENDTXT statement causes message generation processing to select a new path entry. If a message generation deck call is outstanding, the ENDTXT statement generates an automatic RETURN. This statement is required at the end of each message generation deck. This statement is a conditional delimiter.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

Note: If a message was built but not yet sent to the system under test and MORE=YES is not coded on the last TEXT statement of that message, then the ENDTXT statement will cause the message to be sent. If MORE=YES was coded on the TEXT statement, then the ENDTXT statement will *not* cause the message to be sent.

ENTER - enter key statement

```
[name] ENTER
```

Function

The ENTER statement simulates the action of the Enter key on a display device. This statement is valid for 3270 and 5250 simulation. This statement is a delimiter in some cases.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

EREOF - erase to end of field key statement

```
[name] EREOF
```

Function

The EREOF statement simulates the action of the Erase to the End of Field key on a display device. This statement is valid for 3270 simulation. This statement is a delimiter in some cases.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

ERIN - erase input key statement

```
[name] ERIN
```

Function

The ERIN statement simulates the action of the Erase Input key on a display device. This statement is valid for 3270 and 5250 simulation. This statement is a delimiter in some cases.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

ERROR - error simulation statement

```
[name] ERROR STATUS='xxxx'
```

Function

The ERROR statement performs the following functions:
- Provides logical error simulation for 3270 devices
- Generates the SNA sense bytes for an LU2 device.

This statement is valid for 3270 simulation.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

STATUS='*xxxx*'

Function: Specifies the sense and status information to be entered.

Format: Four hexadecimal digits enclosed in single quotes.

Default: None. This operand is required.

EVENT - event statement

```
[name] EVENT {POST=event}
             {QSIGNAL=event}
             {RESET=event}
             {SIGNAL=event}
             [,EVENTTAG=tag]
             [,TIME=ssssssss]
```

Function

The EVENT statement performs a post, reset, or signal action on the named event.

Note: You must code either the POST, QSIGNAL, RESET, or SIGNAL operand. However, only one of these operands can appear on the EVENT statement.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

POST=*event*

Function: Specifies the name of an event that is to be posted complete.

Format: For the POST operand, you can code one of the following options. *value* can be any integer from 0 to 32766 or a counter specification whose value is within this range. Zero is the offset to the first byte of the field for positive offsets (+*value*) and the offset to the last byte of the field for negative offsets (-*value*).

event Specifies the name of the event to be posted, where *event* is one to eight alphanumeric characters.

N±*value*
Specifies the event name to be referenced at an offset from the start (+*value*) or back from the end (-*value*) of the network user area.

U±*value*
Specifies the event name to be referenced at an offset from the start (+*value*) or back from the end (-*value*) of the device user area.

N*s***+***value*
Specifies an event name to be referenced at an offset from the start of the network save area where *s* is the network save area and can be any integer from 1 to 4095.

*s***+***value*
Specifies an event name to be referenced at an offset from the start of the device save area where *s* is the device save area and can be any integer from 1 to 4095.

Note: You can use N±*value*, U±*value*, N*s+value*, and *s+value* for variable event names. The first eight bytes of data beginning at the offset (*value*) comprise the name. For the network and device user area, code the name and then pad it with blanks if the length of the name is less than eight. If the area does not exist or no data is present, the name will consist of eight blanks. Because no validity checking is performed on the name, you can use a name that cannot be expressed as EBCDIC characters. You can put the name to be referenced into the save area or user area with a DATASAVE statement.

Default: None.

QSIGNAL=*event*

Function: Specifies the name of an event which is to be signaled. This operand will signal the event only for the device that issued the QSIGNAL.

Format: Refer to the POST operand format for more information.

Default: None.

RESET=*event*

Function: Specifies the name of an event that is to be marked not complete.

Format: Refer to the POST operand format for more information.

Default: None.

SIGNAL=*event*

Function: Specifies the name of an event which is to be signaled.

Format: Refer to the POST operand format for more information.

Default: None.

EVENTTAG=*tag*

Function: Specifies a tag to be assigned to an EVENT statement to be referenced by the CANCEL statement when canceling event actions associated with a timer expiration.

Format: Refer to the POST operand format for more information.

Default: Event name specification coded on the POST=, RESET=, QSIGNAL=, or SIGNAL= operand.

Note: The same tag may be assigned to one or more EVENT statements. If the TIME= operand is not coded, the EVENTTAG= operand is ignored. The tag value is resolved when the EVENT statement is executed. The event name is resolved when the timer expires.

TIME=*ssssssss*

Function: Specifies the number of seconds that the action specified by this EVENT statement is to be delayed.

Format: *value* can be an integer from 1 to 21474836 or a counter specification whose value is within this range.

Default: The action occurs immediately if you do not code this operand.

EXIT - user exit statement

```
[name] EXIT MODULE=name
          [,PARM=(data...)]
```

Function

The EXIT statement invokes a user exit routine during message generation processing. It allows the exit routine to control message generation by using return codes.

Note: Refer to , SC31-8950 for more information on user exit routines.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

MODULE=*name*

Function: Specifies the member (user exit load module) in the load library that was loaded during initialization and is to gain control when this statement is encountered during message generation.

Format: A 1- to 8-character name that conforms to standard JCL member naming conventions.

Default: None. This operand is required.

PARM=(*data*...)

Function: Specifies the user parameter to be passed to the user exit when it is called during message generation.

Format: From 1 to 100 characters enclosed in the text delimiting characters for this message generation deck. (The default is left and right parentheses). You can code hexadecimal data by enclosing the characters within single quotes. To enter the delimiting characters or a single quote as data, enter two of the characters. You can also continue the data. See "Continuing statements" on page 5 for information about continuing statements.

Default: None. This operand is optional.

The user exit routine must set one of the following return codes in register 15:

Code	Meaning
0	Continue in message generation as if the user exit had not been called.
4	Continue in message generation as though a delimiter had not been previously processed. For 3270 and 5250 simulation, a null attention indicator (AID) is set.
8	A message was generated by the user exit. Continue processing as if the message had been generated by a TEXT statement.
12	Set the wait indicator for the device and stop message generation processing. If a message was generated before calling the exit routine, transmit it now.
16	Stop message generation processing at this point but do not set the wait indicator for the device. If a message was generated before calling the exit routine, transmit it now.

FLDADV - field advance key statement

```
[name] FLDADV
```

Function

The FLDADV statement simulates the action of the Field Advance key on a 5250 display device. This statement is valid only for 5250 simulation. This statement is a delimiter in some cases.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

FLDBKSP - field backspace key statement

```
[name] FLDBKSP
```

Function

The FLDBKSP statement simulates the action of the Field Backspace key on a 5250 display device. This statement is valid only for 5250 simulation. This statement is a delimiter in some cases.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

FLDMINUS - field minus key (F-) statement

```
[name] FLDMINUS
```

Function

The FLDMINUS statement simulates the action of the Field Minus (F-) key on a 5250 display device. This statement is valid only for 5250 simulation. This statement is a delimiter in some cases.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

FLDPLUS - field exit or field plus key (F+) statement

```
[name] FLDPLUS
```

Function

The FLDPLUS statement simulates the action of the Field Exit or Field Plus (F+) key on a 5250 display device. This statement is valid only for 5250 simulation. This statement is a delimiter in some cases.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

FM - field mark key statement

```
[name] FM
```

Function

The FM statement simulates the action of the Field Mark key on a display device. This statement is valid for 3270 simulation. This statement is a delimiter in some cases.

Where

name

Function: Specifies a name to be used in branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

HELP - help key statement

```
[name] HELP [CODE={cccc|0000}]
```

Function

The HELP statement simulates the action of the Help key on a 5250 display device. This statement is valid only for 5250 simulation. This statement is a delimiter in some cases.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

CODE={*cccc*|0000}

Function: Specifies the error code to be entered on the error line.

Format: Any four EBCDIC characters.

Default: 0000

HIGHLITE - display highlight select statement

```
[name] HIGHLITE {BLINK|FIELD|REVERSE|UNDERLINE}
```

Function

The HIGHLITE statement simulates the action of a 3270 device key that selects the highlighting option for displaying subsequent data input. This statement is valid for 3270 simulation.

Note: If you do not code the HIGHLITE statement, the highlighting option will be selected by the extended field attribute byte value. This statement is a delimiter in some cases.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

{BLINK|FIELD|REVERSE|UNDERLINE}

Function: Specifies the highlighting option to be used for displaying subsequent data input from this device.

Format: You can code one of the following keywords for this statement:

BLINK
Specifies that the display of the input data will alternate between display and non-display modes.

FIELD Specifies that the highlighting option defined by the extended field attribute byte will be selected.

REVERSE
Specifies that the input data will be displayed as reversed image characters.

UNDERLINE
Specifies that the displayed input data will be underlined.

Default: FIELD

HOME - home key statement

```
[name] HOME
```

Function

The HOME statement simulates the action of the Home key on a display device. This statement is valid for 3270 and 5250 simulation. This statement is a delimiter in some cases.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

IF - message generation deck logic test statement

```
integer IF {CURSOR=(row,col)}
           {EVENT=event}
           {LOC=location}
           {LOCTEXT={cntr|(data)|integer}}
           [,AREA=area]
           [,COND={EQ|GE|GT|LE|LT|NE}]
           [,DATASAVE=(area,loc,leng)]
           [,DELAY=CANCEL]
           [,ELSE=action]
           [,LENG=value]
           [,LOCLENG=operand]
           [,LOG=(data)]
           [,RESP=NO]
           [,SCAN={YES|value}]
           [,SCANCNTR=cntr]
           [,SNASCOPE={ALL|LOC|REQ|RSP}]
           [,STATUS=HOLD]
           [,TEXT={RESP|cntr|(data)|'xx'|integer}]
           [,THEN=action]
           [,TYPE=type]
           [,UTBL=name]
           [,UTBLCNTR=cntr]
           [,WHEN={IMMED|IN|OUT}]
```

Function

The message generation deck IF statement performs the following functions:

- Specifies comparisons that are to be done on data sent or received by WSim or on terminal or device counter values, switch settings, or data areas
- Tests for event completions
- Sets switches, overrides normal SNA responses, cancels current delays, saves data, logs messages, or alters message generation paths based on the results of the comparisons.

The IF statement is activated and deactivated within the execution of a message generation deck and it is evaluated in the numerical order of the name fields.

Note: See Chapter 14, "Conditions logic test not evaluated," on page 217 for more information on conditions under which a logic test is not evaluated. Refer to , SC31-8945 for examples of logic tests.

Where

integer

Function: Specifies the number of the logic test. This field controls the order in which the message generation logic tests are evaluated for all IF statements that do not contain WHEN=IMMED. For example, if logic tests 3, 5, and 1 are active, the order of test evaluation is 1, 3, and 5.

Format: An integer from 0 to 4095.

Default: None. This field is required for all IF statements that do not contain WHEN=IMMED. The *integer* is not used for IF statements which specify WHEN=IMMED, or for internal deck branching.

Note: A message generation logic test is deactivated when the next TEXT statement is processed in the message generation deck, unless you code the STATUS=HOLD operand. If another IF statement is encountered with the same name field and WHEN=IMMED is not specified on the new IF statement, the first IF statement is deactivated, and the new IF statement is activated. You can also deactivate a message generation deck IF statement by specifying its name in a DEACT statement.

CURSOR=(*row*,*col*)

Function: Specifies the cursor position to be compared with the current cursor position.

Note: The logic test will not be evaluated if the device is not a display device.

Format: Two values, each of which can be either an integer between 1 and 255 or a counter specification whose value is within this range specifying the row and column positions, respectively.

Default: None. You must code either the CURSOR, EVENT, LOC, or LOCTEXT operand.

Note: If you code the CURSOR operand, do not code the AREA, COND, EVENT, LENG, LOC, LOCLENG, LOCTEXT, SCAN, SCANCNTR, TEXT, UTBL, and UTBLCNTR operands.

EVENT=*event*

Function: Specifies the name of a wait or post event that is to be tested.

Format: For the EVENT operand, you can code one of the following options. *value* can be an integer from 0 to 32766 or a counter specification whose value is within this range. Zero is the offset to the first byte of the field for positive offsets (+*value*) and the offset to the last byte of the field for negative offsets (-*value*). For save area references, *s* can be an integer from 1 to 4095.

name Specifies the name of the event to be tested for completion, where *name* is one to eight alphanumeric characters.

N±*value*
Specifies the event name to be referenced at an offset from the start (+*value*) or back from the end (-*value*) of the network user area.

U±*value*
Specifies the event name to be referenced at an offset from the start (+*value*) or back from the end (-*value*) of the device user area.

N*s***+***value*
Specifies an event name to be referenced at an offset from the start of the network save area where *s* is the network save area.

*s***+***value*
Specifies an event name to be referenced at an offset from the start of the device save area where *s* is the device save area.

Note: You can use N±*value*, U±*value*, N*s*+*value*, and *s*+*value* for variable event names. The first eight bytes of data beginning at the offset (*value*) comprise the name. For the network and device user area, code the name and then pad it with blanks if the length of the name is less than eight. If the area does not exist or no data is present, the name will consist of eight blanks. Because no validity checking is performed on the name, you can use a name that cannot be expressed as EBCDIC characters. You can put the name to be referenced into the save area or user area with a DATASAVE statement.

Default: None. You must code either the CURSOR, EVENT, LOC, or LOCTEXT operand.

Note: If you code the EVENT operand, do not code the AREA, COND, CURSOR, LENG, LOC, LOCLENG, LOCTEXT, SCAN, SCANCNTR, TEXT, UTBL, and UTBLCNTR operands.

LOC=*location*

Function: Specifies the starting location where the comparison is to take place. See Chapter 10, "Data locations," on page 209 for more device-specific information.

Note: If you code the LOC operand, do not code the CURSOR, EVENT, or LOCTEXT operands. If the LOC operand is coded and the location is not a switch or counter, the LOCLENG operand can also be coded.

Format: For the LOC operand, you can code one of the following options which represent offsets where *value* specifies the offset. *value* can be any integer from 0 to 32766 or a counter specification whose value is within this range. Zero is the offset to the first byte of the field for positive offsets (+*value*) and the offset to the last byte of the field for negative offsets (-*value*).

B±*value*	N*s*+*value*	TSW*n* | TSW*m* | ...	TSEQ
C±*value*	*s*+*value*	SW*n*	DSEQ
D+*value*	(*row*,*col*)	SW*n*&SW*m*&;..	NC*n*
TH+*value*	NSW*n*	SW*n* | SW*m* | ...	LC*n*
RH+*value*	NSW*n*&NSW*m*&;..	NSW*n*&TSW*m*&SW*n*&;..	TC*n*
RU+*value*	NSW*n* | NSW*m* | ...	NSW*n* | TSW*m* | SW*n* | ...	DC*n*
N±*value*	TSW*n*	NSEQ	
U±*value*	TSW*n*&TSW*m*&;..	LSEQ	

value can be an integer from 0 to 32766 or a counter specification whose value is within this range. Zero is the offset to the first byte of the field for positive offsets (+*value*) and the offset to the last byte of the field for negative offsets (-*value*).

Option Description

B±*value*
: For +*value* begin testing an offset from the start of data in the device buffer. (For display devices, the device buffer is the screen image buffer.) For non-display devices and -*value*, begin testing at an offset back from the end of the data in the device buffer. For display devices and -*value*, begin testing at an offset back from the end of the screen image buffer.

C±*value*
: Begin testing at an offset from the cursor for +*value* or at an offset back from the cursor for -*value*. Normally, use this location only with display devices.

D+*value*
: Begin testing at an offset from the start of the incoming or outgoing data stream. This includes the transmission header and the request header.

TH+*value*
: Begin testing at an offset from the start of the transmission header.

RH+*value*
: Begin testing at an offset from the start of the request header.

RU+*value*
: Begin testing at an offset from the start of the request unit.

N±*value*
: Begin testing at an offset from the start (+*value*) or back from the end (-*value*) of the network user area.

U±*value*
: Begin testing at an offset from the start (+*value*) or back from the end (-*value*) of the device user area.

N*s***+***value*
: Begin testing at an offset from the start of the network save area specified by *s*, where *s* is an integer from 1 to 4095.

*s***+***value*
: Begin testing at an offset from the start of the device save area specified by *s*, where *s* is an integer from 1 to 4095.

(*row***,***col***)**
: Indicates that the test is to be made at the specified row and column of the screen image of a display device, where *row* and *col* may each be an integer from 1 to 255 or a counter specification whose value is within this range. If specified for a non-display device type, the test will not be evaluated.

Note: You can combine network, terminal, and device level switches to be tested, according to the above rules (for example, TSW5|SW3|NSW7|NSW28). However, you cannot mix the & and | operators in the same LOC operand specification. Also, when one of the counter operands is coded, the corresponding value of the TEXT operand must be specified as numeric data or another counter.

Default: None. You must code either the CURSOR, EVENT, LOC, or LOCTEXT operand.

Notes:

- If, when a logic test is to be evaluated, the specified data location is not valid (the location is outside the buffer or user area or not within the data transferred), the logic test is not evaluated and no action is taken unless the LOCLENG operand is coded. For more information, see Chapter 14, "Conditions logic test not evaluated," on page 217.
- When multiple partitions are defined for a 3270 device, buffer or cursor offsets (B+, B-, C+, C-) will reference the data in the presentation space of the currently active partition. The combination (*row,col*) value will reference the display as you would see it, which could include data from more than one partition. The logic test will be performed against the presentation space data of the partition that owns the area of the display referenced by the (*row,col*) specification.

- For VTAMAPPL LUs, TSW, TSEQ, LSEQ, TC*n*, and LC*n* will reference a single set of switches and counters allocated to each VTAMAPPL.
- See Chapter 12, "Counters and switches," on page 213 for valid counter and switch specifications.

LOCTEXT={*cntr*|(*data*)|*integer*}

Function: Specifies the location value where the comparison is to take place. See Chapter 10, "Data locations," on page 209 for more device-specific information.

Note: If you code the LOCTEXT operand, do not code the AREA, CURSOR, EVENT, LENG, LOC, or LOCLENG operands.

Format: For the LOCTEXT operand, you can enter one of the following values:

cntr The counter to be used in the comparison. The valid values for *cntr* are NSEQ, LSEQ, TSEQ, DSEQ, NC*n*, LC*n*, TC*n*, and DC*n*, where *n* is an integer from 1 to 4095. These counters are explained under the LOC operand on this statement. The LOCTEXT operand can specify a counter value only if the TEXT operand specifies a counter or integer value.

(*data***)** The data coded within the text delimiter specified on the MSGTXT statement is to be used as the comparison data. The data field options can be used to specify the data (see Chapter 9, "Data field options," on page 199).

When comparing for specified data, enter hexadecimal data within the text delimiters by enclosing the digits within single quotes. Two digits compose one hexadecimal character. For example, LOCTEXT=(ABC) will generate a comparison for the three characters ABC. LOCTEXT=('AB'CD) is a comparison for three bytes including one hexadecimal character of AB and two EBCDIC characters of CD. A maximum of 32767 characters will be used for comparison.

To enter a single quote, text delimiter (TXTDLM), or data field option control character (CONCHAR) as data, enter two of the characters. You can also continue the data on the next line.

integer A 1- to 10-digit integer between 0 and 2147483647 is to be used for the comparison. This format is valid when the TEXT operand specifies a counter or integer value.

Default: None. You must code either the CURSOR, EVENT, LOC, or LOCTEXT operand.

Note: When the LOCTEXT operand is coded, the THEN or ELSE action on the IF statement is always executed as long as the IF statement meets the criteria set by the SNASCOPE, TYPE, and WHEN operands. The string data comparison allows for unequal or null strings, with the shorter string being padded with blanks, unless SCAN is also coded. In this case, the LOCTEXT data is scanned and substrings within it equal to the length of the TEXT data are compared to the TEXT data.

AREA=*area*

Function: Specifies a user area or save area location which contains the text value for which the test is to be made.

Format: For the AREA operand, you can code one of the following options. *value* can be any integer from 0 to 32766 or a counter specification whose value is within this range. Zero is the offset to the first byte of the field for positive offsets (+*value*) and the offset to the last byte of the field for negative offsets (-*value*). For save area references, *s* can be an integer from 1 to 4095.

N±*value*
: Specifies that the text is located at an offset from the start (+*value*) or back from the end (-*value*) of the network user area.

U±*value*
: Specifies that the text is located at an offset from the start (+*value*) or back from the end (-*value*) of the device user area.

N*s***+***value*
: Specifies that the text is located at an offset from the start of the network save area, where *s* is the network save area.

*s***+***value*
: Specifies that the text is located at an offset from the start of the device save area, where *s* is the device save area.

Default: None. The AREA, TEXT, or UTBL operand is required except when the LOC operand is specified for switch testing or the EVENT operand is specified. If you code this operand, you may also code the LENG operand. This operand may not be coded if the LOC operand specifies a sequence or index counter, or if the LOCTEXT or TEXT operand is coded.

COND={<u>EQ</u>|GE|GT|LE|LT|NE}

Function: Specifies the condition for which the comparison is to be made. The data field identified by the LOC or LOCTEXT operand is compared to the data specified in the TEXT or AREA operands, and the condition set. If the condition specified by the COND operand is met, the THEN action is taken. If the condition specified by the COND operand is not met, the ELSE action is taken.

Format: For the COND operand, you can code one of the following values:

EQ The two fields are equal.

GE The LOC data is greater than or equal to the TEXT or AREA data or the LOCTEXT data is greater than or equal to the TEXT data.

GT The LOC data is greater than the TEXT or AREA data or the LOCTEXT data is greater than the TEXT data.

LE The LOC data is less than or equal to the TEXT or AREA data or the LOCTEXT data is less than or equal to the TEXT data.

LT The LOC data is less than the TEXT or AREA data or the LOCTEXT data is less than the TEXT data.

NE The two fields are not equal.

Default: EQ

Note: This option is not valid for a logic test that tests switches, performs a test under mask, tests an event, or tests a cursor position.

DATASAVE=(*area***,***loc***,***leng***)**

Function: Specifies data to be saved in a save area when the logic test is made and the THEN action is taken. This operand is valid only if you also code a THEN operand on the same statement. If THEN=E(*name*), E(*name-label*), or E(*label*) is coded, the execute action is taken after the data save is performed.

Note: You cannot use the DATASAVE operand with IF statements that specify WHEN=IMMED.

Format: For the DATASAVE operand, you can code the following three values, enclosed in parentheses and separated by commas:

area	Specifies which of the save areas is to be used for data retention, where *area* is either a device save area *s*, or a network save area N*s*, where *s* is an integer from 1 to 4095.
loc	Specifies the location of the data to be saved. You can enter either B+*value*, C+*value*, D+*value*, TH+*value*, RH+*value*, or RU+*value* where *value* can be any integer from 0 to 32766 or a counter specification whose value is within this range. B (device buffer), C (cursor), D (data stream), TH (transmission header), RH (request header), and RU (request unit) are defined under the LOC operand.
leng	Specifies in bytes the amount of data to be saved, where *leng* is an integer from 1 to 32767 or a counter specification whose value is within this range.

Each time data is saved in a save area, the length of that data is also saved and is used when the data is recalled.

If the data to be saved is longer than the save area, the data will be truncated. If the data is shorter than what is specified by the *leng* parameter, only the available data will be saved.

Default: None. This operand is optional.

DELAY=CANCEL

Function: Specifies that the current active delay is to be canceled. The delay is canceled only when the THEN action specified on the IF statement is taken.

Format: CANCEL

Default: None. This operand is optional.

Note: This operand is valid only if a THEN operand is also specified on the same IF statement. This operand is invalid on IF statements that specify WHEN=IMMED.

ELSE=*action*

Function: Specifies the action to be taken if the condition was not met, the tested switches were off, or the tested event was not complete.

Note: If the ELSE operand is omitted and the condition is not met, no action is taken (all indicators and message generation paths are left as they were before the IF statement was encountered).

Format: For the ELSE operand, you can code one of the following options:

B*name-label*	CONT	NSW(ON)	SW(ON)
B*name*	WAIT	NSW(OFF)	SW(OFF)
B-*label*		NSW*n*(ON)	SW*n*(ON)
C*name-label*	QUIESCE	NSW*n*(OFF)	SW*n*(OFF)
C*name*	RELEASE		
C-*label*		TSW(ON)	WAIT(*event*)
E*name-label*	RETURN	TSW(OFF)	POST(*event*)
E*name*	IGNORE	TSW*n*(ON)	RESET(*event*)
E-*label*	ABORT	TSW*n*(OFF)	SIGNAL(*event*)

QSIGNAL(*event*)
VERIFY[-(*data*)]

Default: None. You must code either the THEN or ELSE operand.

For a description of these options, see the THEN operand.

When a BRANCH, CALL, RETURN, QUIESCE, RELEASE, or CONT action is taken, the WAIT condition is reset to OFF. However this does not reset the *event wait* condition.

LENG=*value*

Function: Specifies the length of the text in the user area or save area specified by the AREA operand.

Format: *value* can be an integer from 1 to 32767 or a counter specification whose value is within this range.

Default: None. The amount of data remaining in the area starting from the offset specified. This operand is not allowed if you code the LOCTEXT or TEXT operand.

LOCLENG=*operand*

Function: Specifies a length to be associated with the LOC operand data.

Note: When LOCLENG is coded, the THEN or ELSE action on an IF statement is always executed as long as the IF meets the criteria set by the SNASCOPE, TYPE, and WHEN operands. The string data comparison allows for unequal or null strings with the shorter string being padded with blanks. The LOCLENG operand cannot be coded with the CURSOR, EVENT, LOCTEXT, SCAN, and SCANCNTR operand or used with a test under mask condition.

Format: For the LOCLENG operand, you can code one of the following values:

*	Specifies that the length of the LOC operand data is all the data available in the specified area.
integer	Specifies that the length of the LOC operand data is the integer value (1-32767) specified.
cntr	Specifies that the length of the LOC data is the counter specification value (0-32767) specified.

Default: None.

LOG=(*data*)

Function: Specifies from 1 to 50 bytes of data to be written in a LOG record to the log data set when the test is made and the THEN action is taken for this IF statement.

Note: This operand is valid only if a THEN operand is also specified on the same IF statement.

Format: 1 to 50 bytes of EBCDIC data enclosed within the text delimiter specified on the MSGTXT statement. To enter a single quote or a text delimiter as data, enter two of the characters. You cannot continue this data to another statement. Also, you cannot code data field options for this operand.

Default: None.

RESP=NO

Function: Specifies that if the THEN action is taken for this IF statement, WSim will not generate an automatic SNA response for this message. Instead, WSim will set up the TH and RH for the normal response and go to message generation to get the response data from the message generation deck. The largest response that can be built is 256 bytes long. A response will always be sent after returning from message generation.

This operand is valid only if a THEN operand is also specified on the same IF statement. This operand is invalid on IF statements which specify WHEN=IMMED.

Note: This operand is ignored for non-SNA terminals. However, the THEN action will be performed for all terminal types. Therefore, code this operand only on logic tests evaluated for SNA terminals and devices.

Format: NO

Default: None. This operand is optional. If you do not code the RESP operand, WSim automatically builds the SNA response.

SCAN={YES|*value*}

Function: Specifies whether or not the data is to be scanned sequentially for the data specified in the TEXT, AREA, or UTBL operand. When scanning is specified, the data is searched starting at the location specified in the LOC operand or at the beginning of the text specified by the LOCTEXT operand.

Note: Using this operand can cause performance degradation.

Format: For the SCAN operand, you can code one of the following values:

YES Specifies that scanning continues until the condition is met or the end of the data is reached.

value Specifies that scanning continues until the condition is met, the number of positions specified by *value* has been scanned, or the end of data is reached. *value* can be any integer from 1 to 32767 or a counter specification whose value is within this range.

The data starting at the location as specified by the LOC or LOCTEXT operand is scanned and compared with the character string as specified by the TEXT or AREA operand for LOC or the TEXT operand for LOCTEXT. If data is found that meets the comparison condition before the specified number of positions have been scanned, the THEN action is taken. Otherwise, the ELSE action is taken. If LOCTEXT is coded, padding is not done for the compare.

Default: None. If this operand is omitted, no scanning is performed.

Note: The LOCLENG operand cannot be coded with the SCAN operand.

SCANCNTR=*cntr*

Function: Specifies a counter to be set to the offset of text data that caused the logic test condition to be met. If text is being compared and the IF condition is met, the specified counter is assigned the value of the offset into the save area, user area, buffer, or data stream which satisfies the condition if the LOC operand was coded or the value of the offset into the LOCTEXT data if the LOCTEXT operand was coded. If the IF condition was not met, the value of the counter is unchanged.

Note: If you code this operand, do not code the CURSOR, EVENT, or LOCLENG operands. Also, do not code the SCANCNTR operand if the LOC operand specifies a switch or counter to be tested or the LOCTEXT operand specifies an integer or counter to be tested. If SCANCNTR and UTBLCNTR are both coded and the same counter is specified on both operands, the SCANCNTR operand will take precedence if a match is found.

Format: The value coded for *cntr* can be any of the counter specifications as defined by the TEXT operand.

Default: None. This operand is optional.

SNASCOPE={ALL|LOC|REQ|RSP}

Function: Specifies which SNA flows to test for the data specified in the AREA, TEXT, or UTBL operand.

Format: For the SNASCOPE operand, you can enter one of the following values:

ALL	Specifies that logic testing is to be performed on both SNA request and response flows.
LOC	Specifies that logic testing is to be performed based on the LOC or LOCTEXT operand specification. See Chapter 10, "Data locations," on page 209 for more information.
REQ	Specifies that logic testing is to be performed on the SNA request flows.
RSP	Specifies that logic testing is to be performed on the SNA response flows.

Default: LOC

Note: This operand is ignored for non-SNA devices.

STATUS=HOLD

Function: Specifies that this IF statement is to be held active (the test is made on each eligible message), until a new path entry (from the PATH statement) is begun, until the test is overridden by another IF statement with the same name, or until the IF statement is explicitly deactivated by a DEACT statement.

Format: HOLD

Default: The test is made only until the next message is generated.

TEXT={RESP|*cntr*|(*data*)|'*xx*'|*integer*}

Function: Specifies the text value for which the test is to be made.

Format: For the TEXT operand, you can enter one of the following values:

RESP	The data to be used in the comparison was specified using the RESP operand on the previous TEXT statement. If no RESP was coded on the previous TEXT statement, a null value is used for the logic test.
cntr	The value of a counter is to be used in the comparison. The valid values for *cntr* are NSEQ, LSEQ, TSEQ, DSEQ, NC*n*, LC*n*, TC*n*, and DC*n*, where *n* is an integer from 1 to 4095. These values are explained under the LOC operand on this statement. The TEXT operand can specify a counter value only if the LOC operand specifies a counter value or the LOCTEXT operand specifies a counter or integer value.
(*data***)**	The data coded within the text delimiter specified on the MSGTXT statement is to be used as the comparison data. The data field options can be used to specify the data (see Chapter 9, "Data field options," on page 199). When comparing for specified data, enter hexadecimal data within the text delimiters by enclosing the digits within single quotes. Two digits

compose one hexadecimal character. For example, TEXT=(ABC) will generate a comparison for the three characters ABC. TEXT=('AB'CD) is a comparison for three bytes including one hexadecimal character of AB and two EBCDIC characters of CD. A maximum of 32767 characters will be used for comparison.

To enter a single quote, text delimiter (TXTDLM), or data field option control character (CONCHAR) as data, enter two of the characters. You can also continue the data on the next line.

'xx'	A test under mask on a byte of the data is executed using the mask specified by two hexadecimal digits within single quotes. The bits of the mask correspond one for one with the bits of the byte of data. A mask indicates that the corresponding bits in the byte of data are tested. If all bits tested are set to one, the THEN action is taken. Otherwise, the ELSE action is taken.
integer	A 1- to 10-digit integer ranging from 0 to 2147483647 is to be used for the comparison. This format is valid when the LOC operand specifies a counter value or the LOCTEXT operand specifies a counter or integer value. The value will be compared to this text numeric value.

Default: None. Either the TEXT, AREA, or UTBL operand is required for all tests except when the LOC operand is specified for switch testing or the EVENT operand is specified. You cannot code the TEXT operand when the LOC operand specifies switch testing or when the AREA or LENG operand is coded.

THEN=*action*

Function: Specifies the action to be taken if the specified condition was met, the switches tested were on, or the tested event was complete.

When you omit the THEN operand and the test condition is met, no action is taken (all indicators and message generation paths are left as they were before the IF statement was encountered). Also when you omit the THEN operand, you cannot code the DATASAVE, DELAY, LOG, and RESP operands for the IF statement.

Format: For the THEN operand, you can code one of the following options:

B*name-label*	CONT	NSW(ON)	SW(ON)
B*name*	WAIT	NSW(OFF)	SW(OFF)
B-*label*		NSW*n*(ON)	SW*n*(ON)
C*name-label*	QUIESCE	NSW*n*(OFF)	SW*n*(OFF)
C*name*	RELEASE		
C-*label*		TSW(ON)	WAIT(*event*)
E*name-label*	RETURN	TSW(OFF)	POST(*event*)
E*name*	IGNORE	TSW*n*(ON)	RESET(*event*)
E-*label*	ABORT	TSW*n*(OFF)	SIGNAL(*event*)
			QSIGNAL(*event*)
VERIFY[-(*data*)]			

The following list describes the options:

Option	**Description**
-*label*	Specifies a label of a statement within the message generation deck.
name	Specifies the name of the message generation deck that is the branch target.

name-label
Specifies a label in the named message generation deck.

ABORT
Causes the current message generation deck to be aborted, any active logic tests to be deactivated, and the next message generation deck to be selected according to normal PATH selection rules.

B
Specifies a branch to another location within the message generation deck. *-label* specifies the label of a statement within the current message generation deck. *name* specifies the name of the message generation deck that is the branch target. *name-label* specifies a label in the named message generation deck.

C
Specifies a call to another location within the message generation decks. *-label* specifies the label of a statement within the current message generation deck. *name* specifies the name of the message generation deck that is the call target. *name-label* specifies a label in the named message generation deck. Call differs from branch in that a return pointer is saved to allow message generation to return to the point of the call.

CONT
Specifies that message generation is to continue in the current message generation deck.

DLYCNCL
Cancels any active or pending intermessage delay.

E
Specifies an immediate execution of the statements beginning at the specified message generation deck location. *-label* specifies the label of a statement within the current message generation deck. *name* specifies the name of the message generation deck that is the execute target. *name-label* specifies a label in the named message generation deck. The statement types that can be executed with this action are BRANCH, CALC, CANCEL, DATASAVE, DEACT, EVENT, IF (other than WHEN=IMMED), LABEL, LOG, MONITOR, MSGTXT, ON, OPCMND, SET, SETSW, SETUTI, WTO, and WTOABRHD. Execution stops when any other statement type is encountered. This action is separate from, and does not affect, the normal message generation flow and does not reset the WAIT indicator. It takes place before the evaluation of any subsequent IF statements. It does not affect taking subsequent actions, and may itself be taken regardless of whether any other action has already been taken.

IGNORE
Specifies that no action is to take place. In addition, no other actions of the same class, such as WAIT, CONT, RETURN, will take place for the message being tested, even if a subsequent logic test condition is met.

NSW(ON)
Sets all network switches on, up to the maximum number referenced.

NSW(OFF)
Clears all network switches, up to the maximum number referenced.

NSW*n*(ON)
Sets the indicated network switch, where *n* is an integer from 1 to 4095.

NSW*n*(OFF)
Clears the indicated network switch, where *n* is an integer from 1 to 4095.

POST(*event*)
Specifies that the named *event* is to be posted.

QSIGNAL(*event*)
Specifies that the named *event* is to be signaled but only for the device which issued the QSIGNAL.

QUIESCE
Prohibits message generation until a release operation is performed. A quiesced device can receive messages and respond negatively to polls but cannot generate any data messages.

RELEASE
Specifies that a quiesced device is to proceed in message generation.

RESET(*event*)
Specifies that the named *event* is no longer to be considered posted.

RETURN
Specifies a return to message generation after the point of the last call. If no CALL statements have been issued, a message trace (MTRC) record is written to the log data set and the action is ignored.

SIGNAL(*event*)
Specifies that the named *event* is to be signaled.

SW(ON)
Sets all device switches on, up to the maximum referenced number.

SW(OFF)
Clears all device switches, up the the maximum referenced number.

SW*n*(ON)
Sets the indicated switch for the device, where *n* is an integer from 1 to 4095.

SW*n*(OFF)
Clears the indicated switch for the device, where *n* is an integer from 1 to 4095.

TSW(ON)
Sets all terminal switches on, up to the maximum referenced number.

TSW(OFF)
Clears all terminal switches, up to the maximum referenced number.

TSW*n*(ON)
Sets the indicated switch for the terminal, where *n* is an integer from 1 to 4095.

TSW*n*(OFF)
Clears the indicated switch for the terminal, where *n* is an integer from 1 to 4095.

VERIFY[-(*data*)]
Causes a VRFY log record to be logged on the log data set for this network. If *data* is coded, it will be included in the VRFY log record. The value coded for *data* can be one or more characters enclosed within the text delimiters specified on the MSGTXT statement. Although *data* may be longer than 50 characters, no more than 50

character will be included in the VRFY log record. To enter a single quote or text delimiter as data, enter two of the characters. You can continue the data and include data field options (see Chapter 9, "Data field options," on page 199).

The Loglist Utility (refer to , SC31-8947) can format these VRFY log records into Verification Reports.

WAIT Prohibits message generation.

WAIT(*event***)**
Specifies that the named *event* is to be posted before further messages can be generated.

When a BRANCH, CALL, RETURN, QUIESCE, RELEASE, or CONT action is taken, the WAIT condition is reset to OFF. However, this does not reset the *event wait* condition.

Refer to , SC31-8945 for more information on actions executed when multiple IF statements are coded.

The event name specified in WAIT(*event*), POST(*event*), SIGNAL(*event*), QSIGNAL(*event*), and RESET(*event*) can either be explicitly coded as a name (up to eight alphanumeric characters) or can reference a save area or user area. These are specified as:

- N±*value*
- N*s+value*
- U±*value*
- *s+value*

Reference the EVENT statement or EVENT operand for more information on these operand values.

Default: None. You must code either the THEN or ELSE operand.

TYPE=*type*
Function: Specifies the type of terminal for which this IF statement is to be evaluated.

Format: For the TYPE operand, you can code one of the following terminal and device types:

LU0	LU1	LU2	LU3	LU4
LU6	LU62	LU7	FTP	TN3270
TN3270E	TN3270P	TN5250	TNNVT	STCP
SUDP				

UTBL=*name*
Function: Specifies the number of the user table, as defined by a UTBL statement or the name coded on the MSGUTBL statement, containing the entries to be compared with the data defined by the LOC or LOCTEXT operand.

Note: If you code this operand, do not code with the AREA, EVENT, LENG, and TEXT operands. Also, do not code the UTBL operand if the LOC operand specifies a counter or switch to be tested or the LOCTEXT operand specifies a counter or integer value to be tested.

Format: An integer from 0 to 255 or from one to eight alphanumeric characters and the first character must be alphabetic.

Default: None. This operand is optional.

UTBLCNTR=*cntr*

Function: Specifies a counter to be set to the index of the user table entry that caused the logic test condition to be met. If the logic test condition was not met, the value of the counter is unchanged.

Note: If you code this operand, do not code the AREA, EVENT, LENG, and TEXT operands. Also, do not code the UTBLCNTR operand if the LOC operand specifies a counter or switch to be tested or the LOCTEXT operand specifies a counter or integer value to be tested. If SCANCNTR and UTBLCNTR are both coded and the same counter is specified on both operands, the SCANCNTR operand will take precedence if a match is found.

Format: The value coded for *cntr* can be any of the counter specifications as defined by the TEXT operand. The index of the first entry in a user table is zero.

Default: None. This operand is optional.

WHEN={IMMED|IN|OUT}

Function: Specifies when the logic test is to be evaluated.

Format: For the WHEN operand, you can code one of the following values:

IMMED
Specifies that the logic test will be evaluated immediately during message generation processing.

Note: If you specify WHEN=IMMED, do not code the DELAY, RESP, and DATASAVE operands.

IN Specifies that the logic test will be evaluated when data is received by WSim.

OUT Specifies that the logic test will be evaluated when data is transmitted by WSim.

IF statements that specify WHEN=IN or WHEN=OUT are ignored for CPI-C simulations.

Default: IN

INSERT - insert statement

```
[name] INSERT
```

Function

The INSERT statement simulates the 3270 INSERT key operation. Screen data will be shifted right on the simulated screen as the TEXT data is inserted. This statement is a delimiter in some cases.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

Note: Text data will be truncated and a message will be written to the log if there is not enough space in the data entry field of the screen to contain the inserted text.

JUMP - jump key statement

```
[name] JUMP [PID=nn]
```

Function

The JUMP statement simulates the Jump key on a 3270 display terminal. This statement is valid for 3270 simulation. This statement is a delimiter in some cases.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

PID=*nn*

Function: Specifies the partition identification number (PID) of the partition to be made active with the JUMP statement.

Format: For the PID operand, *nn* can be an integer from 0 to 15.

Default: None. This operand is optional.

Note: When you code the JUMP statement without the PID operand, the next sequential partition becomes active.

LABEL - label statement

```
name LABEL
```

Function

The LABEL statement establishes a label in a message generation deck to be used for a branch or call operation.

Where

name

Function: Specifies a name to be used when branching with logic tests or branching from the BRANCH and CALL statements.

Format: From one to eight alphanumeric characters.

Default: None. This field is required.

LCLEAR - local clear statement

```
[name] LCLEAR
```

Function

The LCLEAR statement simulates the Local Clear key on a display device. This statement is valid for 3270 simulation. This statement is a delimiter in some cases.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

LOG - write data to log statement

```
[name] LOG {(data)}
           {AREA=area[,LENG=value]}
           {BLOCK=name}
           {DISPLAY}
```

Function

The LOG statement writes the user-specified data, data areas or control blocks, or the 3270 or 5250 display to the log data set for formatting by the Loglist Utility.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

(*data***)**

Function: Defines the data to be written in a message to the log data set.

Note: If you code this operand, it must be the only operand in the LOG statement.

Format: You can code any amount of data for this operand, but a maximum of 32767 characters of user data will actually be written to the log data set. The data is enclosed by the text delimiting character specified on the MSGTXT statement (the default is left and right parentheses). You can also continue the data. However, if a single delimiting character is detected in column 71, it indicates the end of the operand and data past column 71 is ignored.

You can use the data field options. (See Chapter 9, "Data field options," on page 199.) Enter hexadecimal data by enclosing the digits within single quotes. To enter a single quote, the special control character (CONCHAR), or a text delimiting character (TXTDLM) as data, enter two of the characters. If you enter two text delimiting characters, they must be on the same statement (no continuation between the characters).

Default: None. If no data is entered, a null message will be logged.

AREA=*area*

Function: Specifies that a portion of a save area or user area is to be written to the log data set beginning with the specified offset.

Format: For the AREA operand, you can code one of the following options. *value* can be any integer from 0 to 32766 or a counter specification whose value is within this range. Zero is the offset to the first byte of the field for positive offsets (+*value*) and the offset to the last byte of the field for negative offsets (-*value*).

N±*value*
: Specifies the area to be written at an offset from the start (+*value*) or back from the end (-*value*) of the network user area.

U±*value*
: Specifies the area to be written at an offset from the start (+*value*) or back from the end (-*value*) of the device user area.

N*s***+***value*
: Specifies the area to be written at an offset from the start of the network save area, where *s* is the network save area and can be any integer from 1 to 4095.

*s***+***value*
: Specifies the area to be written at an offset from the start of the device save area, where *s* is the device save area and can be any integer from 1 to 4095.

Default: None. This operand is optional.

BLOCK=*name*

Function: Specifies the name of a control block or data area to be written to the log data set.

Note: If you code the BLOCK operand, it must be the only operand in the LOG statement.

Format: For the BLOCK operand, you can code one of the following keywords:

CNT	Sequence and index counters
DEV	Device or logical unit control block
NCB	Network control block
SCR	Screen image for displays
SWS	Switches

TRM Terminal control block

Default: None. This operand is optional.

DISPLAY

Function: Specifies that the display buffers are to be written to the log data set for formatting by the loglist program.

Note: If you code the DISPLAY operand, it must be the only operand on the LOG statement. The DISPLAY operand is valid only for 3270 and 5250 terminals.

Format: DISPLAY

Default: None. This operand is optional.

LENG=*value*

Function: Specifies the length of the data to be logged.

Note: The LENG operand is valid only if you also code the AREA operand.

Format: *value* can be an integer from 1 to 32767 or a counter specification whose value is within this range.

Default: The length from the offset specified by AREA to the end of the data in the save area or to the end of the user area.

MONITOR - monitor statement

```
[name] MONITOR
```

Function

The MONITOR statement causes the Display Monitor Facility to display the simulated 3270 display image, as it exists at this point in the message generation process, on the display monitoring this device, if any. The UPDATE=MONITOR option must be in effect when starting the monitor facility. Refer to , SC31-8948 for more information about display monitor facility. This statement is valid for 3270 simulation.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

MSGTXT - message generation deck begin statement

```
name MSGTXT [,CONCHAR={char|$}]
            [,COUNT={integer|1  }]
            [,PAD=digits]
            [,STLMEM=membername]
            [,TXTDLM=char]
```

Function

The MSGTXT statement performs the following functions:

- Defines the start of a message generation deck
- Specifies the control character used to delimit the data fields in this deck
- Specifies the control character used to delimit the data field options in this deck
- Specifies the number of times this deck will be executed before another deck is selected during message generation
- Specifies a character to be used in padding generated messages to a desired length

Where

name

Function: Specifies the name by which this message generation deck can be referenced. The name should be different from all other message generation decks.

Format: A 1- to 8-character name conforms to JCL member naming conventions.

Default: None. This field is required.

CONCHAR={*char*|<u>$</u>}

Function: Specifies the control character to be used as a delimiter for the data field options. See Chapter 9, “Data field options,” on page 199 for descriptions of the data field options and explanations of where they can be used.

Format: A single EBCDIC character. However, you cannot use a comma, parenthesis, quote, or blank for this character.

Default: $

Note: The values coded on TXTDLM and CONCHAR operands must be different characters.

COUNT={*integer*|<u>1</u>}

Function: Specifies the number of times this message generation deck will be processed successively during message generation.

Note: This operand is ignored for any message generation deck that is being executed because of a BRANCH or CALL command to the deck.

Format: An integer from 1 to 255.

Default: 1

PAD=*digits*

Function: Specifies the character to be used as a pad character for padding text messages to the desired length. When you code this operand, the normal alphabetic pad sequence is replaced with this character. For more information, see the LENG operand on the TEXT statement.

Format: Two hexadecimal digits enclosed in single quotes.

Default: The message will be padded with the alphabetic sequence (ABCDEF...).

STLMEM=*membername*

Function: Specifies the name of a special member of the MSGDD data set used by the STL trace facility.

Format: From one to eight alphanumeric characters.

Default: None.

Note: This operand is automatically generated by the STL translator and should not normally be coded.

TXTDLM=*char*

Function: Specifies the delimiting character to be used for text data on the TEXT, IF, CMND, WTO, LOG, DATASAVE, and OPCMND statements and PARM on the operand of the EXIT statement in the current message generation deck.

Format: A single EBCDIC character. However, you cannot code a comma, parenthesis, quote, or blank as the character. To use the delimiter as a data character, code two delimiters together.

Default: A left parenthesis, "(", to indicate the beginning of the text data and a right parenthesis, ")", to indicate the end of the text data.

Note: The values coded for TXTDLM and CONCHAR operands must be different characters.

MSGUTBL - user data table (Member) statement

```
name MSGUTBL (entry)[,...]
```

Function

The MSGUTBL statement defines a set of user data entries to be maintained as a member of a partitioned data set. This statement can be arranged in any order with the message generation decks, and is processed before the message generation decks by the WSim initiator.

The MSGUTBL statement is unique in that it is not a part of the network configuration definition or the message generation deck definition. It is syntactically equivalent to an entire message generation deck definition (MSGTXT ... ENDTXT).

All MSGUTBL statements must follow the network configuration statements or be previously stored in the partitioned data set named by the MSGDD DD statement. Whether a member is assumed to be a message generation deck or an MSGUTBL deck is determined by the first use of the member name in the network definition.

name

Function: Specifies the member name to be used to identify this table.

Note: MSGUTBL members and message generation decks reside in the same data set. Consequently, all MSGUTBL and MSGTXT names must be unique.

Format: From one to eight alphanumeric characters.

Default: None. This name is required.

(entry)

Function: Specifies the user data that can be referenced during message generation processing.

Format: Any amount of data enclosed in parentheses. However, it may be truncated during message generation. To enter hexadecimal data, enclose the digits between single quotes. To enter single quotes or parentheses in the data, enter two of the characters. You can continue the data by entering the data through column 71 and then starting in column 2 of the next statement, or by using the "+" continuation character.

Default: None. You must code at least one entry.

Note: The maximum number of entries is 2147483647.

NL - new line key statement

```
[name] NL
```

Function

The NL statement simulates the action of the New Line key on a display device. The NL statement is valid for 3270 and 5250 simulation. This statement is a delimiter in some cases.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

ON - on statement

```
[name] ON EVENT=event,THEN=action
```

Function

The ON statement sets up an action to be taken when a named event is signaled.

Note: If an ON statement is encountered which specifies the same *event* and *action* as a currently active ON statement for this terminal, the new statement is ignored.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

EVENT=*event*

Function: Specifies the name of the event that must be signaled to initiate the action specified by this ON statement.

Format: For the EVENT operand, you can code one of the following options. *value* can be any integer from 0 to 32766 or a counter specification whose value is within this range. Zero is the offset to the first byte of the field for positive offsets (+*value*) and the offset to the last byte of the field for negative offsets (-*value*).

name Specifies the name of the event, where *name* is one to eight alphanumeric characters.

N±*value*
Specifies the event name to be referenced at an offset from the start (+*value*) or back from the end (-*value*) of the network user area.

U±*value*
Specifies the event name to be referenced at an offset from the start (+*value*) or back from the end (-*value*) of the device user area.

N*s***+***value*
Specifies an event name to be referenced at an offset from the start of the network save area where *s* is the network save area and can be any integer from 1 to 4095.

*s***+***value*
Specifies an event name to be referenced at an offset from the start of the device save area where *s* is the device save area and can be any integer from 1 to 4095.

Note: You can use N±*value*, U±*value*, N*s+value*, and *s+value* for variable event names. The first eight bytes of data beginning at the offset (*value*) comprise the name. For the network and device user area, code the name and then pad with blanks if the length of the name is less than eight. If the area does not exist or no data is present, the name will consist of eight blanks. Because no validity checking is performed on the name, you can use a name that cannot be expressed as EBCDIC characters. You can put the name to be referenced into the save area or user area with a DATASAVE statement.

Default: None. This operand is required.

THEN=*action*

Function: Specifies an action to be taken when the event associated with this ON statement is signaled.

Format: Any of the actions that can be specified on the THEN and ELSE operands of the message generation deck IF statement can be specified here. See "IF - message generation deck logic test statement" on page 150.

Note: ON conditions, unlike message generation deck IF statements, are not deactivated when processing a new PATH entry begins. They remain active until explicitly deactivated or until deactivated as a result of the condition having been signaled.

Default: None. This operand is required.

OPCMND - operator command statement

```
[name] OPCMND (data...)
```

Function

The OPCMND statement specifies an operator control command to be executed by the simulated device. The operator control command is not executed until WSim leaves this pass through message generation.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

(*data***...)**

Function: Defines the operator command to be entered from the message generation deck. For example,

```
OPCMND (ZEND)
```

executes the ZEND operator command. Refer to , SC31-8948 for details of the various operator commands.

Note: You can execute all operator commands with the OPCMND statement, with the exception of console recovery subcommands, which must be entered by the system console operator.

You can enter any amount of data for this operand, but a maximum of 120 characters will actually be passed to the console routines.

Format: The data must be EBCDIC and must be enclosed by the text delimiting character specified on the MSGTXT statement, (the default is left and right parentheses).

You can use the data field options defined in Chapter 9, "Data field options," on page 199. To enter a single quote, the special control character (CONCHAR), or a text delimiting character (TXTDLM) as data, enter two of the characters. If you enter two text delimiting characters, they must be on the same statement (no continuation between the characters).

If necessary, you can continue the data on the next line using the standard continuation methods. However, if a single text delimiting character is detected in column 71, it indicates the end of the operand, and any data after column 71 is ignored.

Default: None. At least one data character is required.

PA - program access key statement

```
[name] PAn
```

Function

The PA statement simulates the action of one of the three Program Access keys on a display device. This statement is valid for 3270 simulation. This statement is a delimiter in some cases.

Where

name
: **Function:** Specifies a name to be used when branching during message generation.

  Format: From one to eight alphanumeric characters.

  Default: None. This field is optional.

PA*n*
: **Function:** Specifies which Program Access key is to be simulated, where *n* is the number of the key.

  Format: *n* is an integer between one and three.

  Default: None. You must specify a key to be simulated.

PF - program function key statement

```
[name] PFn
```

Function

The PF statement simulates the action of one of the 24 Program Function keys on a display device. This statement is valid for 3270 simulation. This statement is a delimiter in some cases.

Where

name
: **Function:** Specifies a name to be used when branching during message generation.

  Format: From one to eight alphanumeric characters.

  Default: None. This field is optional.

PF*n*
: **Function:** Specifies which of the 24 Program Function keys is to be simulated, where *n* is the number of the key.

  Format: *n* is a 1- to 2-digit number between 1 and 24.

  Default: None. You must specify a key to be simulated.

PRINT - print key statement

```
[name] PRINT
```

Function

The PRINT statement simulates the action of the Print key on a 5250 display device. This statement is valid only for 5250 simulation. This statement is a delimiter in some cases.

Where

name
: **Function:** Specifies a name to be used when branching during message generation.

  Format: From one to eight alphanumeric characters.

  Default: None. This field is optional.

PUSH - push statement

```
PUSH string [TO queue_name]
```

Function

The PUSH statement places *string* on *queue_name* on a last-in-first-out (LIFO) basis.

Where

string
: **Function:** Specifies a string to be placed on a queue.

  Format: From one to 32767 alphanumeric characters.

  Default: None. If no data is entered a null message is queued.

queue_name
: **Function:** Specifies a name of a queue.

  Format: From one to eight alphanumeric characters.

  Default: None. This field is optional.

QUEUE - queue statement

```
QUEUE string [TO queue_name]
```

Function

The QUEUE statement places *string* to *queue_name* on a first-in-first-out (FIFO) basis.

Where

string
: **Function:** Specifies a string to be added to a queue.

  Format: From one to 32767 alphanumeric characters.

  Default: None. If no data is entered a null message is queued.

queue_name

Function: Specifies a name of a queue.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

QUIESCE - quiesce statement

```
[name] QUIESCE
```

Function

The QUIESCE statement stops message generation and quiesces the device. This statement is an unconditional delimiter.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

RESET - reset key statement

```
[name] RESET
```

Function

The RESET statement simulates the action of the Reset key on a display device. This statement is valid for 3270 simulation.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

RESP - SNA response statement

```
[name] RESP SENSE='xxxxxxxx'
```

Function

The RESP statement provides for logical error simulation by SNA devices. It overrides the normal SNA response that WSim would transmit with an exception response containing user-specified sense data. This statement is valid only for SNA simulation, excluding CPI-C simulation.

Note: WSim will respond with an exception response to the next request received from the system under test.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

SENSE='*xxxxxxxx***'**

Function: Specifies the sense data to be included with the exception response being simulated.

Format: Eight hexadecimal digits enclosed in single quotes.

Default: None. This operand is required.

RETURN - return from subroutine statement

```
[name] RETURN
```

Function

The RETURN statement restarts message generation processing at the point where the last CALL statement or logic test call was issued. This statement is ignored if no call is outstanding.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

RH - request/response header statement

```
[name] RH [BB={ON|OFF}]
          [,CHAIN={FIRST|MIDDLE|LAST|ONLY}]
          [,CDI={ON|OFF}]
          [,CEB={ON|OFF}]
          [,DR1={ON|OFF}]
          [,DR2={ON|OFF}]
          [,EB={ON|OFF}]
          [,EDI={ON|OFF}]
          [,EXC={ON|OFF}]
          [,FMI={ON|OFF}]
          [,PACE={ON|OFF}]
          [,QRI={ON|OFF}]
          [,RESP={ON|OFF}]
          [,SNI={ON|OFF}]
          [,TYPE={DFC|FM|NC|SC}]
```

Function

The RH statement performs the following functions:

- Modifies the SNA request/response header built by WSim for a message generated with a TEXT statement
- Builds an RH for a user-specified command defined by a CMND statement
- Optionally specifies chaining of transmitted messages.

You must code the RH statement after the TEXT or CMND statement for which the RH is to be modified. This statement is valid only for non—CPI-C SNA simulation.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

CHAIN={FIRST|MIDDLE|LAST|ONLY}

Function: Specifies the setting of the chaining control flags in the RH.

Format: For the CHAIN operand, you can code one of the following keywords:

FIRST First RU of chain

MIDDLE
Middle RU of chain

LAST Last RU of chain

ONLY Only RU of chain.

Note: If CHAIN=FIRST is coded, CDI=OFF and EXC=ON are assumed unless otherwise specified.

Default: None. This operand is optional.

TYPE={DFC|FM|NC|SC}

Function: Specifies the type of request being built.

Format: For the TYPE operand, you can code one of the following keywords:

DFC	Data flow control
FM	FM data
NC	Network control
SC	Session control.

Default: None. This operand is optional.

The following operands are described as a group because they all have only ON and OFF as valid options. The definition of each keyword gives the meaning of the specified RH bit when the bit is set ON. ON indicates the specified bit value will be B'1' and OFF indicates the bit value will be B'0'.

BB	Set begin bracket.
CDI	Set change direction.
CEB	Set conditional end bracket.
DR1	Definite response 1 requested.
DR2	Definite response 2 requested.
EB	Set end bracket.
EXC	Exception response requested.
FMI	The RU is formatted.
QRI	Set the queued response indicator.
RESP	The data is a response.
SNI	The RU contains sense data.

This is how the RH statement maps to *SNA Formats* bits:

WSim	**SNA Terminology**
BB	BBI
CDI	CDI
CEB	CEBI
DR1	DR1I
DR2	DR2I
EB	EBI
EXC	ERI/RTI [6]
FMI	FI
QRI	QRI
RESP	RRI
SNI	SDI

6. The ERI and RTI bits are the same bit.

ROLLDOWN - rolldown key statement

```
[name] ROLLDOWN
```

Function

The ROLLDOWN statement simulates the action of the Roll Down key on a 5250 display device. This statement is valid only for 5250 simulation. This statement is a delimiter in some cases.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

ROLLUP - rollup key statement

```
[name] ROLLUP
```

Function

The ROLLUP statement simulates the action of the Roll Up key on a 5250 display device. This statement is valid only for 5250 simulation.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

SCROLL - scroll key statement

```
[name] SCROLL {DOWN|UP}
```

Function

The SCROLL statement simulates the Scroll keys on a 3270 display terminal. This statement is valid for 3270 simulation. This statement is a delimiter in some cases.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

{DOWN|UP}

Function: Specifies whether the displayed data is to be scrolled up or down in relation to the current viewport.

Format: For the SCROLL statement, you can code one of the following values:

DOWN
Specifies that the data is to be scrolled down.

UP Specifies that the data is to be scrolled up.

Default: None.

SELECT - selector pen detect statement

```
[name] SELECT [COLUMN={value|1}]
              [,OFFSET=value]
              [,ROW={value|1}]
```

Function

The SELECT statement simulates the action of the Selector Pen on a display device. This statement is valid for 3270 and 5250 simulation. This statement is a delimiter in some cases.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

COLUMN={*value*|1}

Function: Specifies the column of the character to be detected.

Format: *value* can be an integer from 1 to 255 or a counter specification whose value is within this range.

Default: 1

OFFSET=*value*

Function: Specifies the location of the character to be detected as an offset from the beginning of the display buffer, where *integer* is the offset. If multiple partitions are defined for a 3270 terminal, the character offset is from the beginning of the presentation space of the currently active partition.

Format: *value* can be an integer from 0 to 32766 or a counter specification whose value is within this range.

Default: None.

Note: This operand is invalid if COLUMN or ROW is coded.

ROW={*value*|1}
Function: Specifies the row of the character to be detected.

Format: *value* can be an integer from 1 to 255 or a counter specification whose value is within this range.

Default: 1

Note: When multiple partitions are defined for a 3270 terminal, the ROW and COLUMN specification will reference the display as you would see it, which could include data from more than one partition. The selection operation will be performed on the partition that owns the area of the display referenced by the ROW and COLUMN specification.

SET - set counters statement

```
[name] SET cntr=option[,...]
```

Function

The SET statement performs the following functions:
- Alters the values of the sequence and index counters for a simulated device
- Sets a counter to a specified integer, a random number, or the value of another counter
- Sets the counter to the sum of itself and another counter
- Sets the counter to the difference of itself and another counter
- Sets the counter to the product of itself and another counter
- Sets the counter to the quotient of itself and another counter
- Sets the counter to the remainder of itself and another counter
- Adds to or subtracts from the value of a counter or specified integer
- Multiplies to or divides from the value of a counter or specified integer
- Sets the counter to one or two bytes of hexadecimal data
- Sets the counter to the EBCDIC character representation of a number in data
- Sets the counter to the simulated cursor's current row, column, or offset
- Sets the counter to the index of the last UTBL item
- May alter multiple counters in a single statement
- Sets the counter to the position of the last occurrence of specified data in a save area
- Sets the counter to the position of the first occurrence of specified data in a save area
- Sets the counter to the position of the first character in the *n*th blank-delimited word in a save area
- Sets the counter to the word number of the first word of specified data found in a save area
- Sets the counter to the number of blank-delimited words in a save area or a user area

- Sets the counter to the number of text data items on the queue.

Note: To use the SET statement, you must code at least one operand.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

cntr

Function: Specifies which counter is to be set.

Note: You can set the same counter multiple times with the same SET statement. If you code multiple operands, the counters will be set in the order specified.

Format: The valid values for *cntr* are NSEQ, LSEQ, TSEQ, DSEQ, NC*n*, LC*n*, TC*n*, and DC*n*, where *n* is an integer from 1 to 4095. These values represent the sequence counters and the index counters for the network, line, terminal, and device levels.

Default: None. You must code a value for *cntr*.

Notes:
- For VTAMAPPL LU simulation, TSEQ, LSEQ, TC*n*, and LC*n* will reference a single set of counters allocated to each VTAM application (VTAMAPPL).
- For CPI-C transaction program simulation:
  - LSEQ and LCn will reference a single set of counters allocated to each APPC LU.
  - TSEQ and TCn will reference a single set of counters allocated to each transaction program.
  - DSEQ and DCn will reference a single set of counters allocated to each transaction program instance.
- See Chapter 12, “Counters and switches,” on page 213 for valid counter and switch specifications.
- WSim will allocate index counters (minimum 3) up to the maximum index counter referenced in a script associated with the network for each level of counters. For example, if DC4095 and TC77 are the maximum numbered index counters referenced in a script, all device counters from DSEQ to DC4095 will be allocated for each device level resource in the network and all terminal counters from TSEQ to TC77 will be allocated for each terminal level resource in the network. Only those counters allocated can be altered or queried.

option

Function: Specifies how the counter is to be set.

Format: For the *option*, you can code one of the following values:

integer Set the counter to the value specified by *integer* (0 to 2147483647).

+integer
Set the counter to the sum of its own value and the value specified by *integer* (0 to 2147483647).

-*integer*
: Set the counter to the difference of its own value and the value specified by *integer* (0 to 2147483647).

******integer*
: Set the counter to the product of its own value and the value specified by *integer* (0 to 2147483647).

/*integer*
: Divide the counter by the value specified by *integer* (0 to 2147483647) and set the counter to the integer quotient resulting from the division. Division by zero will cause the counter value to be unchanged. Message ITP468I will be logged indicating the reason for the SET failure.

//*integer*
: Divide the counter by the value specified by *integer* (0 to 2147483647) and set the counter to the remainder resulting from the division. This operator is also called the modulus operator. Division by zero will cause the counter value to be unchanged. Message ITP468I will be logged indicating the reason for the SET failure.

scntr
: Set the counter to the value of another counter. The values for *scntr* can be the same as the values for *cntr*.

+*scntr*
: Set the counter to the sum of *cntr* and *scntr*. The values for *scntr* can be the same as the values for *cntr*.

-*scntr*
: Set the counter to the difference of *cntr* and *scntr*. The values for *scntr* can be the same as the values for *cntr*.

******scntr*
: Set the counter to the product of *cntr* and *scntr*. The values for *scntr* can be the same as the values for *cntr*.

/*scntr*
: Divide the counter by the value of *scntr* and set the counter to the integer quotient resulting from the division. Division by zero will cause the counter value to be unchanged. Message ITP468I will be logged indicating the reason for the SET failure. The values for *scntr* can be the same as the values for *cntr*.

//*scntr*
: Divide the counter by the value of *scntr* and set the counter to the remainder resulting from the division. Division by zero will cause the counter value to be unchanged. Message ITP468I will be logged indicating the reason for the SET failure. The values for *scntr* can be the same as the values for *cntr*.

(*lo***,***hi***)**
: Set the counter to a random number in the range specified by the *lo* and *hi* values. *lo* is an integer from 0 to 2147483646 and *hi* is an integer from 1 to 2147483647 or counter specifications whose values are within these ranges. The value for *lo* must be less than the value for *hi*.

RN*n*
: Set the counter to a random number in the range specified by the RN statement with label *n* (0 to 255).

(X,*loc***,***leng***)**
: Set the counter to up to 31 bits of hexadecimal data. *loc* is the location of the data. If the specified location does not exist, the counter will not be changed, and message ITP468I will be logged. Any of the following locations may be specified:

- B±*value*
- C±*value*

- N±*value*
- N*s*+*value*
- U±*value*
- *s*+*value*
- (*row*,*col*)

These location notations are described on the IF statement LOC operand.

leng is the length of the data to copy in bytes and may be from 1 to 4. If you specify more than one byte of data and there is only one byte of data at the specified location, that single byte will be copied. The single byte is returned in the rightmost byte of the counter.

(E,*loc***,***leng***)**
Set the counter to a 1- to 10-digit EBCDIC number found in *loc*, the location of the data. If the specified location does not exist, the counter will not be changed, and message ITP468I will be logged. Any of the following locations may be specified:

- B±*value*
- C±*value*
- U±*value*
- N±*value*
- N*s*+*value*
- *s*+*value*
- (*row*,*col*)

These location notations are described on the IF statement LOC operand.

leng is the length of the numeric field to be translated. Leading, non-numeric characters (i.e., blanks, alphabetic characters, etc.) will be ignored. Trailing, non-numeric characters will be truncated. If no numeric character can be found within the specified text, or if the numeric field's value is greater than 2147483647, the counter value will not be changed. In such cases, message ITP468I will be logged. If the *leng* specified is longer than the data at the specified location, the available data will be used. If the *leng* specified is shorter than the data at the specified location, the rightmost *leng* digits are returned and the rest is truncated.

{CROW | CCOL | COFF}
Set the counter to the current position of the cursor's row, column, or offset. For 3270 partitioned devices with multiple partitions defined, the offset returned by the COFF option will represent the cursor offset from the beginning of the presentation space of the currently active partition. The values returned by the CROW and CCOL options will be the actual screen row and column numbers, without regard for partitions.

LASTPOS(*needle_area***,***haystack_area***[,***start***])**
Set the counter to the position of the last occurrence of the data specified in the *needle_area* save area within the data specified in the *haystack_area* save area. Returns 0 if data in *needle_area* is the null string or is not found. By default the search starts at the last character in *haystack_area* and scans backward. You can override the default by specifying *start*, the position to start the backward scan. *start* defaults

to the length of *haystack_area* if it is not specified or it is larger than the length of *haystack_area*. *start* can be an integer from 1 to 32767 or a counter specification whose value is within this range.

{NUMROWS | NUMCOLS}
Set the counter to the number of rows or columns on a display screen.

LENG(N | U | N*s* **|** *s***)**
Set the counter to the current length of data in a save area or user area. The value for *s* can be an integer from 1 to 4095.

POS(*needle_area***,***haystack_area***[,***start***])**
Set the counter to the position of the first occurrence of the data specified in the *needle_area* save area within the data specified in the *haystack_area* save area. Returns 0 if data in *needle_area* is the null string or is not found, or if *start* is greater than the length of *haystack_area*. By default the search starts at the first character in *haystack_area*. You can override the default by specifying *start*, the position to start the search. *start* can be an integer from 1 to 32767 or a counter specification whose value is within this range.

QUEUED([*queue_name***])**
Set the counter to the number of text data items on the queue, *queue_name*. *queue_name* can be specified as either a static one to eight character alphameric queue name or a save or user area + offset definition. The default for *queue_name* is a unique value for each simulated device.

WORDINDEX(*string_area***,***n***)**
Set the counter to the position of the first character in the *n*th blank-delimited word in the *string_area* save area. Returns 0 if fewer than *n* words are in *string_area*. *n* can be an integer from 1 to 32767 or a counter specification whose value is within this range. *n* is required.

WORDPOS(*phrase_area***,***string_area***[,***start***])**
Set the counter to the word number of the first word in the *phrase_area* save area, found in the *string_area* save area. Returns 0 if *phrase_area* contains no words or if the data in *phrase_area* is not found in *string_area*. Multiple blanks between words in either *phrase_area* or *string_area* are treated as a single blank for comparison, otherwise the words must match exactly. By default, the search starts at the first word in *string_area*. You can override the default by specifying *start*, the word at which to start the search. *start* can be an integer from 1 to 32767 or a counter specification whose value is within this range.

WORDS(*string_area***)**
Set the counter to the number of blank-delimited words in the *string_area* save area or user area.

UTBLMAX(*utblname***)**
Set the counter to the index of the last item in the UTBL. *utblname* specifies the name of an MSGUTBL or the label of a UTBL network definition statement.

Default: None. You must code a value for *option*.

Note: If the integer to be subtracted from a counter is greater than the value of the counter, the subtraction results in wrapping the counter value around the limit of 2147483647. This applies to other manipulation as well as subtraction.

SETSW - set switches statement

```
[name] SETSW {NSW[n]|SW[n]|TSW[n]}={ON|OFF}
```

Function

The SETSW statement sets any or all of the network, terminal, or device switches. It also clears any or all of the network, terminal, or device switches.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

{NSW[*n*]|SW[*n*]|TSW[*n*]}={ON|OFF}

Function: Specifies the switches that are to be set or cleared.

Format: For the SETSW operand, you can code one of the following values:

NSW*n*=ON
Specifies that one of the 4095 network switches is to be set on, where *n* is the switch number and can be any integer from 1 to 4095.

NSW*n*=OFF
Specifies that one of the 4095 network switches is to be cleared, where *n* is a switch number and can be any integer from 1 to 4095.

NSW=OFF|ON
Specifies that all 4095 of the network switches are to be cleared or set on.

SW*n*=ON
Specifies that one of the 4095 device switches is to be set on, where *n* is a switch number and can be any integer from 1 to 4095.

SW*n*=OFF
Specifies that one of the 4095 device switches is to be cleared, where *n* is a switch number and can be any integer from 1 to 4095.

SW=OFF|ON
Specifies that all 4095 of the device switches are to be cleared or set on.

TSW*n*=ON
Specifies that one of the 4095 terminal switches is to be set on, where *n* is a switch number and can be any integer from 1 to 4095.

TSW*n*=OFF
Specifies that one of the 4095 terminal switches is to be cleared, where *n* is a switch number and can be any integer from 1 to 4095.

TSW=OFF|ON
Specifies that all 4095 of the terminal switches are to be cleared or set on.

Default: None. This operand is required.

Notes:

- For VTAMAPPL LU simulation, TSW will reference a single set of switches allocated to each VTAM application (VTAMAPPL).
- For CPI-C transaction program simulation, TSW will reference a single set of switches allocated to each transaction program, and SW will reference a single set of switches allocated to each transaction program instance.
- WSim will allocate switches (minimum 32) up to the maximum switch referenced in a script associated with the network for each level of switches. For example, if SW4095 and TSW77 are the maximum switches referenced in a script, 4095 device switches will be allocated for each device level resource in the network and 77 terminal switches will be allocated for each terminal level resource in the network. Only those switches allocated can be Altered or Queried.

SETUTI - set UTI statement

```
[name] SETUTI UTI=uti
```

Function

The SETUTI statement alters the active UTI for a simulated device. This UTI is active until another SETUTI is encountered, or ENDTXT is processed.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

UTI=*uti*

Function: Specifies which UTI is to be used as the active UTI. *uti* must reference a UTI statement defined within the network configuration portion of the network or “NTWRKUTI”, which references the network-level UTI.

Format: From one to eight alphanumeric characters.

Default: None. This field is required.

STOP - stop statement

```
[name] STOP
```

Function

The STOP statement acts as an unconditional delimiter. It does not set the terminal wait indicator. The STOP statement unconditionally interrupts the message generation process for a particular terminal. It does not render the terminal inactive, nor does it stop WSim execution.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

STRIPE - magnetic stripe reader input statement

```
[name] STRIPE (data)
```

Function

The STRIPE statement defines message data to be transmitted to the system under test by a magnetic stripe reader. This statement is valid for 3270 simulation. This statement is a delimiter in some cases.

Note: The STRIPE statement will be ignored for non-display devices.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

(*data***)**

Function: Defines the secure data to be input to the display device. A field attribute is generated at the position of the cursor as long as it is an unprotected character location. The operator cannot alter the data. It will be sent to the host in a Read Modified operation when the next delimiter statement in the deck is encountered.

Format: You can code up to 125 characters for this statement. If you code more than 125 characters, the extra characters will be truncated. The data is enclosed by the text delimiting character specified on the MSGTXT statement (the default is left and right parentheses).

You can code the data field options (see Chapter 9, "Data field options," on page 199). Enter hexadecimal data by enclosing the digits within single quotes. To enter a single quote, the special control character (CONCHAR), or a text delimiting character (TXTDLM) as data, enter two of the characters. If two text delimiting characters are entered, they must be on the same statement (no continuation between the characters). You can continue the data on the next line.

Default: None.

SYSREQ - system request statement

```
[name] SYSREQ
```

Function

The SYSREQ statement simulates the action of the SYSREQ key on a Telnet device. This statement is valid for non—CPI-C SNA simulations. This statement is a delimiter in some cases.

Where

name
: **Function:** Specifies a name to be used when branching during message generation.

  Format: From one to eight alphanumeric characters.

  Default: None. This field is optional.

TAB - tab key statement

```
[name] TAB
```

Function

The TAB statement simulates the action of the Tab key on a display device. This statement is valid for 3270 simulation. This statement is a delimiter in some cases.

Where

name
: **Function:** Specifies a name to be used when branching during message generation.

  Format: From one to eight alphanumeric characters.

  Default: None. This field is optional.

TEXT - generate text statement

```
[name] TEXT [(data)]
            [,LENG={integer|(lo,hi)|RNn|cntr}]
            [,LOG={byte|Ns+value|s+value|N±value|U±value|00}]
            [,MORE={YES|NO}]
            [,RESP=(data)]
```

Function

The TEXT statement:

- Defines message data to be transmitted to the system under test by the simulated device
- Specifies a byte of data to be associated with the generated message when it is written to the log data set
- Specifies data to be used in a comparison with a response message received from the system under test.

Notes:
- This statement is a conditional delimiter.
- TEXT statements are ignored for CPI-C simulations.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

(*data***)**

Function: Defines the data to be entered into the terminal buffer. This data will be placed into the terminal buffer following any possible headers (such as TH, RH, and SDLC headers) required in the buffer.

Format: You can code any amount of data for this operand. The data is enclosed by the text delimiting character specified on the MSGTXT statement (the default is left and right parentheses).

You can code the data field options (see Chapter 9, "Data field options," on page 199). Enter hexadecimal data by enclosing the digits within single quotes. To enter a single quote, the special control character (CONCHAR), or a text delimiting character (TXTDLM) as data, enter two of the characters. If two text delimiting characters are entered, they must be on the same statement (no continuation between the characters).

You can continue the data on the next line. However, if a single text delimiting character is detected in column 71, it indicates the end of the operand, and any data after column 71 is ignored.

Default: If no data is entered and LENG is coded, the message generated consists of the alphabet repeated to make up the requested length.

LENG={*integer***|(***lo***,***hi***)|RN***n***|***cntr***}**

Function: Specifies the length of the message to be generated.

If the length is longer than the length of data entered, the message is padded with the operand corresponding to the PAD keyword on the MSGTXT statement or alphabetic characters, if the PAD operand is not coded (such as ABCDEF). If the data entered is longer than the LENG value, the message is truncated on the right to the value specified. If the data entered is longer than the terminal or device buffer size, the following takes place:

1. The data is truncated to the terminal buffer size specified by the BUFSIZE parameter on the NTWRK, TCPIP, VTAMAPPL, or APPCLU statement.
2. Informational message ITP404I is written to the log data set.

Format: For the LENG operand, you can code one of the following values:

integer An integer from 1 to 32767.

(*lo,hi***)** A random number in the range specified by the *lo* and *hi* values, where *lo* is an integer from 0 to 32766 and *hi* is an integer from 1 to 32767 or counter specifications whose values are within these ranges. The value coded for *lo* must be less than the value coded for *hi*.

RN*n* A random number in the range specified by the RN statement with label *n* (0 to 255).

cntr Valid values for *cntr* are NSEQ, LSEQ, TSEQ, DSEQ, NC*n*, LC*n*, TC*n*, and DC*n*, where *n* is an integer from 1 to 4095. These values represent the sequence counters and the index counters for the network, line, terminal, and device levels.

Note: For LENG=*cntr*, if the value of the counter is 0, LENG is not taken into account. If the counter value is greater than 32767, the value is used according to the BUFSIZE specified.

Default: If LENG is omitted, the message generated is equal to the amount of data entered.

LOG={*byte***|N***s+value***|***s+value***|N±***value***|U±***value***|00}**

Function: Specifies a single byte of data to be included in the message log header for this data transmission and for all records for this device until the next TEXT or CMND statement is generated.

Format: For the LOG operand, you can enter one of the following options. The value for *value* can be any integer from 0 to 32766 or a counter specification whose value is within this range. Zero is the offset to the first byte of the field.

byte One byte of data. If this byte is a single EBCDIC character, the EBCDIC character is logged. If this byte is two hexadecimal digits, the two digits are logged.

N*s+value*
Specifies an offset into a network save area, where *s* is the number of the network save area and is an integer from 1 to 4095.

s+value
Specifies an offset into a device save area, where *s* is the number of the device save area and is an integer from 1 to 4095.

N±*value*
Specifies an offset into a network user area, where *+value* is the offset from the start of the user area and *-value* is the offset back from the end of the user area.

U±*value*
Specifies an offset into a device user area, where *+value* is the offset from the start of the user area and *-value* is the offset back from the end of the user area.

Note: If an area has been specified and its length is longer than one byte, the first byte is used. If the area contains no data or *value* specifies an offset that is outside the area, X'00' is used for the log byte and an informational message is logged.

Default: A hexadecimal 00 will be logged.

Note: The loglist program (ITPLL) will cause both representations (EBCDIC and hexadecimal) of the log character to be printed on the formatted log report under the heading USER DATA. The log byte is reset by the detection of a TEXT or CMND statement during message generation.

MORE={YES|NO}
Function: Specifies that the next TEXT statement is not to be interpreted as a delimiter, but rather as a concatenation onto the current TEXT statement(s).

Format: YES or NO.

Default: NO

RESP=(*data*...)
Function: Specifies the text data to be used for comparison with a message when an IF statement is encountered with the TEXT=RESP operand, assuming that the IF statement is active for the message.

Format: You can code a maximum of 25 characters for the RESP operand, but you must enter at least one byte of data enclosed by the text delimiting character specified on the MSGTXT statement (the default is left and right parentheses).

Enter hexadecimal data within the text delimiting characters by enclosing the digits within single quotes. To enter a single quote or a text delimiting character (TXTDLM) as data, enter two of the characters.

Note: The data enclosed in text delimiting characters may not be continued. The data field options are not valid for this operand.

Default: None. This operand is optional.

TH - transmission header statement

```
[name] TH [,SNF=integer]
```

Function

The TH statement modifies the SNA transmission header built by WSim for a message generated with a TEXT statement. It also builds a TH for a user-specified command defined by a CMND statement. The TH statement must follow the TEXT or CMND statement for which the TH is to be modified. This statement is valid for SNA simulation, except for 3270 SNA, 5250 terminals, and CPI-C transaction programs.

Where

name
Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

SNF=*integer*
Function: Indicates the transmission header sequence number field is to be set.

Note: The sequence number field will be changed and can cause an error in the access method of the system under test.

Format: An integer from 0 to 65535.

Default: None. This operand is optional.

WAIT - wait for response statement

```
[name] WAIT [{EVENT=event}]
          [{TIME={ssssssss}
                 {cntr}
                 {A(integer)}
                 {F(integer)}
                 {R(value1[,value2])}
                 {T(integer)}}]
          [{,UTI=uti}]
```

Function

The WAIT statement simulates the action of a terminal operator waiting for a reply before entering the next message. This statement can specify an event that must be posted complete before further messages can be generated, or it can specify a delay to be observed until a reply is received or the delay expires. This statement is an unconditional delimiter.

Note: For CPI-C simulations, if a WAIT with no time is specified, an event will be needed to reset the wait.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

EVENT=*event*

Function: Specifies the name of an event that must be posted complete before further messages can be generated.

Format: For the EVENT operand, you can enter one of the following options. *value* can be any integer from 0 to 32766 or a counter specification whose value is within this range. Zero is the offset to the first byte of the field for positive offsets (+*value*) and the offset to the last byte of the field for negative offsets (-*value*).

name Specifies the name of the event to be posted, where *name* is one to eight alphanumeric characters.

N±*value*
Specifies the event name to be referenced at an offset from the start (+*value*) or back from the end (-*value*) of the network user area.

U±*value*
Specifies the event name to be referenced at an offset from the start (+*value*) or back from the end (-*value*) of the device user area.

N*s***+***value*

Specifies an event name to be referenced at an offset from the start of the network save area where *s* is the network save area and can be any integer from 1 to 4095.

*s***+***value*

Specifies an event name to be referenced at an offset from the start of the device save area where *s* is the device save area and can be any integer from 1 to 4095.

You can use N±*value*, U±*value*, N*s+value*, and *s+value* for variable event names. The first eight bytes of data beginning at the offset (*value*) comprise the name. For the network and device user area, code the name and then pad it with blanks if the length of the name is less than eight. If the area does not exist or no data is present, the name will consist of eight blanks. Because no validity checking is performed on the name, you can use a name that cannot be expressed as EBCDIC characters. You can put the name to be referenced into the save area or user area with a DATASAVE statement.

Default: None.

TIME={*ssssssss***|***cntr***|A(***integer***)|F(***integer***)|R(***value1***[,***value2***])|T(***integer***)}**

Function: Specifies the maximum amount of time to be delayed before continuing in message generation.

Except for CPI-C simulation, if a system reply arrives before the specified time delay has expired, message generation will continue immediately at the point after the WAIT, unless the wait indicator is set by an IF statement.

If the TIME operand is specified, the TIME value overrides the delay value for the message being generated. It is started immediately even if THKTIME=UNLOCK is specified for the terminal.

Format: For the TIME operand, you can code one of the following values:

ssssssss

Specifies a fixed number or seconds, where *ssssssss* is an integer from 0 to 21474836. If 0 is specified, the WAIT will be treated as if the TIME operand were not coded. If any other integer is specified, the maximum amount of time to wait will be that number of seconds. The delay will not be affected by the UTI value.

cntr A counter specification whose value is within the range of 0 and 2147483647. For any counter specification value, the maximum amount of time to wait will be the counter value multiplied by the UTI value.

A(*integer***)**

Specifies the maximum wait time will be chosen by randomly selecting a number in the range from 0 to two times the specified integer and then multiplying the number by the UTI value, where *integer* is an integer from 0 to 1073741823. The average delay will be *integer*.

F(*integer***)**

Specifies a fixed value, where *integer* is an integer from 0 to 2147483647. If F(0) is specified, the WAIT will be treated as if the TIME operand were not coded. If any other integer is specified, the maximum wait time will be the integer multiplied by the UTI value.

R(*integer***)**

Specifies that the integer will identify an RN statement, where *integer* is an integer from 0 to 255. The maximum wait time will be a randomly selected value from the range coded on the RN statement multiplied by the UTI value.

R(*value1***,***value2***)**
: Specifies that the maximum wait time will be a randomly selected value from the range of low (*value1*) to high (*value2*) multiplied by the UTI value, where *value1* is an integer from 0 to 2147483646 and *value2* is an integer from 1 to 2147483647 or counter specifications whose values are within these ranges. The value coded for *value1* must be less than the value coded for *value2*.

T(*integer***)**
: Specifies that integer will identify a RATE statement which defines a rate table, where *integer* is an integer from 0 to 255. The maximum wait time will be a randomly selected value from the rate table multiplied by the UTI value.

Default: None.

Note: You can code either the EVENT or the TIME operands, but not both. If you omit both TIME and EVENT operands, the device will not send another message until the wait indicator is reset. In this case, a normal delay will be used. Refer to , SC31-8945 for information on resetting the wait indicator.

UTI=*uti*

Function: Specifies a UTI which is to be used in calculating this delay. *uti* must reference a UTI statement defined within the network configuration statement.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

Note: The TIME operand must be coded on the WAIT statement for the UTI operand to be valid. If the TIME=*integer* with no prefix, and UTI=*uti* operands are coded on a WAIT statement, the statement will be flagged in error. You must code one of the prefix A, F, R, or T before *integer* when the UTI operand is specified. Furthermore, the UTI operand is not valid if the EVENT operand has been coded.

WTO - write data to console statement

```
[name] WTO (data...)
```

Function

The WTO statement writes user-specified data to the operator console.

Note: When using the WTO statement, make sure that the system console is not overloaded with these messages.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

(*data***...)**

Function: Defines the data to be written to the operator.

Format: You can enter any amount of data for this operand, but a maximum of 50 characters of user data will actually be written to the operator console. The data is enclosed by the text delimiting character specified on the MSGTXT statement, defaulting to left and right parentheses.

You can use the data field options (see Chapter 9, "Data field options," on page 199). Enter hexadecimal data by enclosing the digits within single quotes. To enter a single quote, the special control character (CONCHAR), or a text delimiting character (TXTDLM) as data, enter two of the characters. If two text delimiting characters are entered, they must be on the same statement (no continuation between the characters).

You can continue the data on the next line. However, if a single text delimiting character is detected in column 71, it indicates the end of the operand, and any data after column 71 is ignored.

Default: None. If no data is entered, no user data will be included in the console message.

WTOABRHD - write data with abbreviated header to console statement

```
[name] WTOABRHD (data...)
```

Function

The WTOABRHD statement writes user-specified data to the operator console using only an abbreviated header containing only a message number preceding the data.

Where

name

Function: Specifies a name to be used when branching during message generation.

Format: From one to eight alphanumeric characters.

Default: None. This field is optional.

(*data***...)**

Function: Defines the data to be written to the operator.

Format: You can enter any amount of data for this operand, but a maximum of 100 characters of user data will actually be written to the operator console. The data is enclosed by the text delimiting character specified on the MSGTXT statement, defaulting to left and right parentheses.

Enter hexadecimal data by enclosing the digits within single quotes. To enter a single quote, the special control character (CONCHAR), or a text delimiting character (TXTDLM) as data, enter two of the characters. If two text delimiting characters are entered, they must be on the same statement (no continuation between the characters).

You can continue the data on the next line. However, if a single text delimiting character is detected in column 71, it indicates the end of the operand, and any data after column 71 is ignored.

Default: None. If no data is entered, no user data will be included in the console message.

Chapter 9. Data field options

This chapter describes the data field options, which you can use to insert variable data into the message or data field being constructed during message generation processing.

Description of data field options

To use an option, enclose the option with the control character (CONCHAR) specified on the MSGTXT statement (the default is $). For truncations, integers are truncated on the left and strings are truncated on the right. The control character is not valid within the option itself.

The following options are valid only in the data field of a TEXT statement:
- FM
- NL
- TAB

You can use all other options in any of the following data fields:
- CMND statement DATA operand
- CMND statement URC operand
- DATASAVE statement TEXT operand
- IF statement LOCTEXT operand
- IF statement TEXT operand
- LOG statement data field
- OPCMND statement data field
- STRIPE statement data field
- TEXT statement data field
- WTO statement data field.
- WTOABRHD statement data field

The following list describes the data field options available with WSim:

$APPCLUID$
For CPI-C simulations, $APPCLUID$ returns the name of the APPC LU on which the transaction program is defined. The name of the APPC LU is the name field of the APPCLU statement with trailing blanks removed.

$ATTR,*loc*,*l*$
Specifies the character attribute values associated with a unique screen location that are to be indirectly tested for the 3270 message generation facility. The $ATTR,*loc*,*l*$ data option will generate up to 11 characters in length specified by *l*, at the specific screen location *loc*. The valid values for *loc* are B±*value*, C±*value*, and *row,col* where *value* is the offset and can be any integer from 0 to 32766 or a counter specification whose value is within this range. The values of *row* and *col* can each be an integer from 1 to 255 or a counter specification whose value is within this range. For example:

```
DATASAVE  AREA=U+0,TEXT=($ATTR,C+0,9$)
```

Refer to , SC31-8945 for more information.

$CMONTH$
The name of the current month in EBCDIC is inserted in the data. The name is in mixed case, for example January.

$CNTLRID$
The name field from the CNTLR statement is inserted into the data. The blanks used to pad the name to eight characters are deleted.

$CNTR,*cntr*[,*l*]$
Includes the value of any counter in text data without affecting the value of the counter, where *cntr* is a counter name, and *l* is an optional length value that specifies the number of counter digits to include in the text.

The valid values for *cntr* are SEQ, NSEQ, LSEQ, TSEQ, DSEQ, NC*n*, LC*n*, TC*n*, and DC*n*, where *n* is an integer from one to the value specified by the CNTRS operand on the NTWRK statement. These values represent the sequence counters and the index counters for the network, line, terminal, and device levels. If *l* is specified for the length value, its value must be from one to ten. If *l* is omitted, all significant digits of the counter value will be inserted in the text, but leading zeroes will be eliminated.

$CNTRX,*cntr*[,*l*]$
Includes the hexadecimal value of any counter in text data without affecting the value of the counter, where *cntr* is a counter name, and *l* is an optional length value that specifies the number of hexadecimal bytes of the counter to include in the text.

The valid values for *cntr* are SEQ, NSEQ, LSEQ, TSEQ, DSEQ, NC*n*, LC*n*, TC*n*, and DC*n*, where *n* is an integer from one to the value specified by the CNTRS operand on the NTWRK statement. These values represent the sequence counters and the index counters for the network, line, terminal, and device levels. If *l* is specified for the length value, its value must be one to four. If *l* is omitted, all significant bytes of the counter value will be inserted in the text, but leading zeros will be eliminated.

$DATE[±*offset*][,*format*]$
The current date (in the case of $DATE$ without the "+" or "-" or with a "+0" or a "-0") or the specified date (in any other case), in one of the formats specified below, is inserted into the text. *offset* is an integer from 0 to 65535 or a counter specification whose value is within this range and specifies the number of days to be added to or subtracted from the current date before formatting it. *format* is one of the following operands:

Operand	Description	Format
D	Number of days this year	ddd
E	Date in European format	dd/mm/yy
H	Packed Julian padded with X'F'	yydddF*
J	Date in Julian format	yyddd
M	Calendar month's name	January
N	Date in Normal format (dd mon yyyy)	05 Feb 2002
O	Date in ordered format	yy/mm/dd
P	Packed standard WSim date	mmddyy*
S	Date in sorted format	yyyymmdd
<u>**T**</u>	WSim standard date format	mmddyy
U	Date in USA format	mm/dd/yy

Operand	Description	Format
W	Day of the week	Monday

T is the default value for *format*.

Note: The options marked with an * return packed values.

DAY
The 2-character number for the day of the month is inserted in the data.

$DEVID$
The name field from the DEV statement is inserted in the data. The blanks used to pad the name to eight characters are deleted.

$DSEQ,*n*$
The device sequence counter is inserted in the data, where *n* is the length of the number inserted and can be any integer from one to ten. Each time the device sequence counter is referenced, the counter value is updated before it is inserted in the data. The $DSEQ,*n*$ option performs the same function as the $SEQ,*n*$ option. For terminals without devices, SEQ, DSEQ, and TSEQ are the same counter.

$DUP,*char*,*value*$
The single byte specified by *char* is duplicated in the data *value* times, where *value* can be an integer from 1 to 32767 or a counter specification whose value is within this range. To enter hexadecimal data, specify two hexadecimal digits for *char*. When duplicating a quote or a parenthesis, do not enter two quotes or two parentheses.

EL
Specifies the end of a 1-byte (L) or 2-byte (LL) length field special option. The EL option is valid only for non-display devices. See "Example of the length data field options" on page 207 for examples of the EL, L, and LL data field options.

FM
Simulates the 3270 Field Mark key. Each time this option is detected, a Field Mark character is entered into the buffer. The FM option is ignored for device types other than 3270. It is valid only in the TEXT statement data field.

$ID,*n*$
Specifies that all or part of the name field for the terminal is to be inserted in the data, where *n* is an integer from one to eight that specifies the amount of data to be inserted. If *n* is less than eight, the name is truncated on the right. If *n* is greater than the length of the name, blanks are added on the right.

L
Specifies the start of a 1-byte length field with a value of one plus the length of the text data following the L special option up to a EL special option or to the end of the data. The L option is valid only for non-display devices. See "Example of the length data field options" on page 207 for examples of the EL, L, and LL data field options.

$LASTVERB$
For CPI-C simulations, $LASTVERB$ returns the name of the last CPI-C statement (verb) that was issued by this message deck.

LL
The start of a 2-byte length field with a value of two plus the length of the text data following the LL special option up to a EL special option or to the end of the data. The LL option is valid only for non-display devices. See

"Example of the length data field options" on page 207 for examples of the use of the EL, L, and LL data field options.

$LSEQ,*n*$
The line sequence counter is inserted in the data, where *n* is an integer from one to ten that specifies the length of the number inserted. The number is padded with zeroes, if necessary.

Each time the line sequence counter is referenced, the counter value is updated before it is inserted in the data. This counter is a fullword and automatically wraps to 0 after 2147483647. However, zero will not appear as the sequence counter because the counter is updated to one when it is next referenced.

$LUID$
The name field from the LU statement is inserted in the data. The blanks used to pad the name to eight characters are deleted.

$MONTH$
The 2-character number for the current month is inserted in the data.

$MSGTXTID$
Allows you to include in the text data the name of the message generation deck in which the text resides. The message generation deck name is inserted with trailing blanks deleted.

$NETID$
The name field from the NTWRK statement is inserted in the data. The blanks used to pad the name to eight characters are deleted.

NL
Simulates the action of the New Line key on the 3270. The cursor is set to the first unprotected character location of the next line. If no unprotected fields exist, the cursor is set to character location zero. If the display contains no fields, the cursor is set to the first position of the next line. The NL option is ignored for device types other than 3270 and is valid only in the TEXT statement data field.

$NSEQ,*n*$
The network sequence counter is inserted in the data, where *n* is an integer from one to ten that specifies the length of the number inserted. The number is padded with zeros, if necessary.

Each time the network sequence counter is referenced, the counter value is updated before it is inserted in the data. This counter is a fullword and automatically wraps to 0 after 2147483647. However, zero will not appear as the sequence counter because the counter is updated to one when it is next referenced.

$PATHID$
The PATHID function returns the name field from the PATH statement currently being executed. The name field from the PATH statement is inserted in the data. Blanks used to pad the name to eight characters are deleted.

$PULL[,*queue_name*]$
The PULL function returns the next text/string item from the queue specified by *queue_name*. *queue_name* can be specified as either a static one-to-eight character alphanumeric queue name or an area+offset definition. The *queue_name* field conforms to the same rules as event names.

$RECALL,
{*s*}$
{N*s*}

```
{N±value[,leng]}
{Ns+value[,leng]}
{s+value[,leng]}
{U±value[,leng]}
{B±value[,leng]}
{C±value[,leng]}
{D+value[,leng]}
{TH+value[,leng]}
{RH+value[,leng]}
{RU+value[,leng]}
{(row,col)[,leng]}
```

Data from the indicated save area or user area is placed in the data. This option allows the retrieval of data previously saved with the DATASAVE statement, the DATASAVE operand on the IF statement, or a user exit. The following lists define the RECALL option parameters. The value for *value* can be any integer from 0 to 32766 or a counter specification whose value is within this range. Zero is the offset to the first byte of the field for the positive offsets (+*value*) and the offset to the last byte of the field for negative offsets (-*value*).

s
: Specifies an integer from 1 to 4095 that specifies the number of the device save area from which data is recalled. The length of data recalled is equal to the length of the data previously saved in the device save area.

N*s*
: Specifies an integer from 1 to 4095 that specifies the number of the network save area from which data is recalled. The length of data recalled is equal to the length of the data previously saved in the network save area.

N±*value*
: Specifies an offset into the network user area from the beginning or end of the area, respectively.

N*s***+***value*
: Specifies an offset into a network save area, where *s* is the number of the network save area and is an integer from 1 to 4095.

*s***+***value*
: Specifies an offset into a device save area, where *s* is the number of the device save area and is an integer from 1 to 4095.

U±*value*
: Specifies an offset into the device user area from the beginning or end of the area, respectively.

B±*value*
: Specifies an offset into the device buffer from the beginning or end of the buffer, respectively.

C±*value*
: Specifies an offset from the current cursor position for +*value* or an offset back from the current cursor position for -*value*.

D+*value*
: Specifies an offset from the beginning of the data stream.

TH+*value*
: Specifies an offset from the beginning of the transmission header.

RH+*value*
: Specifies an offset from the beginning of the request header.

RU+*value*
: Specifies an offset from the beginning of the request unit.

(*row,col***)**
: Specifies the row and column of the screen image of the display device.

leng
: Specifies an integer from 1 to 32767 or a counter specification whose value is within this range that is the length of data to be recalled from the save area or user area. If the length plus the offset is longer than the available data in the save area or is beyond the end of the user area, only the available data is recalled.

Note: If the save area or user area was not defined in the network definition, or no data has been previously saved in the save area, an informational error message is written to the log data set, and the option is ignored. Data is recalled from an existing user area even if no data has been previously saved in that area.

The following examples present the valid formats of the RECALL option:

```
$RECALL,10$
$RECALL,N6$
$RECALL,N1+4,20$
$RECALL,1+DC4,20$
$RECALL,N+0,DC5$
$RECALL,N+TC2,5000$
$RECALL,U-10,5$
$RECALL,U-NSEQ,5$
$RECALL,N10+DC2$
$RECALL,U+0$
$RECALL,B+10,3$
$RECALL,C-5$
$RECALL,D+6,45$
$RECALL,(15,22)$
$RECALL,RH+4,1$
$RECALL,RU+3,1000$
$RECALL,TH+1$
```

$RNUM,*lo***,***hi***,***n***$**

A random number between the specified low (*lo*) and high (*hi*) values and of length *n* is inserted in the data, where *n* is an integer from 1 to 10, *lo* is an integer from 0 to 2147483646 or a counter specification whose value is within this range, and *hi* is an integer from 1 to 2147483647 or a counter specification whose value is within this range. If the number generated is shorter than *n*, it is padded with leading zeros. *lo* must be less than *hi*.

$RNUM,*m***,***n***$**

A random number between the limits specified on the RN statement referenced by *m* and of length *n* is inserted in the data, where *m* is an integer from 0 to 255, and *n* is an integer from 1 to 10. If the number generated is shorter than *n*, it is padded with leading zeros.

$SEQ,***n***$

A terminal sequence counter is inserted in the data, where *n* is an integer from one to ten that specifies the length of the number inserted. The number is padded with leading zeros if necessary.

Each time the specified sequence counter is referenced, the counter value is updated before it is inserted in the data. This counter is a fullword and automatically wraps to zero after 2147483647. However, zero will never appear as the sequence counter in the data since the counter is updated to one the next time it is referenced.

$SESSNO$
The session number for the LU is inserted into the data. The number inserted is formatted as -*nnnnn* or null if no session number exists.

TAB
Simulates the 3270 Tab key. The cursor is moved to the first byte of the next unprotected field. The TAB option is ignored for device types other than 3270 and is valid only in the TEXT statement data field.

$TCPIPID$
The name field from the TCPIP statement is inserted in the data. The blanks used to pad the name to eight characters are deleted.

$TOD,*n*$
The time of day in EBCDIC is inserted in the data, where *n* specifies the number of bytes to be inserted and is an integer from one to eight. The time of day is formatted as HHMMSSTH (hours, minutes, seconds, tenths, and hundredths of seconds). The most significant digits of the time are inserted if *n* is less than 8.

$TPID$
For CPI-C simulations, $TPID$ returns the transaction program name for this message deck.

$TPINSTNO$
For CPI-C simulations, $TPINSTNO$ returns the transaction program instance number for this message deck. The number inserted is formatted as -nnnnn, or null if no instance number exists.

$TSEQ,*n*$
A terminal sequence counter is inserted in the data, where *n* is an integer from one to ten that specifies the length of the number inserted. The number is padded with leading zeros if necessary.

Each time the specified sequence counter is referenced, the counter value is updated before it is inserted in the data. This counter is a fullword and automatically wraps to zero after 2147483647. However, zero will never appear as the sequence counter in the data since the counter is updated to one the next time it is referenced.

$UTBL,*id*,*sel*$
User data specified on a UTBL statement is inserted in the data, where *id* is the number from the label field of the UTBL statement of the UTBL or the name from the label field of the MSGUTBL statement which begins with an alphabetic character from which the data is to be extracted, and *sel* is the data entry in the table.

Note: Table entries are indexed beginning with 0. If *n* is too large and the entry does not exist, no data is inserted.

You can choose one of the following options for *sel*:

S*n*, SD*n*
The data is chosen sequentially using one of the device index counters, where *n* is an integer from one to the value specified by the CNTRS operand on the NTWRK statement and indicates which counter to use. Each time the counter is referenced, the counter value is used to index into the specified table to select the data, and then the counter value is incremented. The counter value is reset to 0 when it references a value past the last element in a user table.

ST*n* The data is chosen sequentially using one of the terminal index counters. The counters are used and altered as in SD*n* above.

SL*n* The data is chosen sequentially using one of the line index counters. The counters are used and altered as in SD*n* above.

SN*n* The data is chosen sequentially using one of the network index counters. The counters are used and altered as in SD*n* above.

C*n*, CD*n*
The data is chosen from the user table using the device index counter specified by *n*, where *n* is an integer from one to the value specified by the CNTRS operand on the NTWRK statement. The index counter is not incremented when referenced. The counter value is reset to 0 when it references a value past the last element in a user table.

CT*n* The data is chosen from the user table using one of the terminal index counters. The counters are used and altered as in CD*n* above.

CL*n* The data is chosen from the user table using one of the line index counters. The counters are used and altered as in CD*n* above.

CN*n* The data is chosen from the user table using one of the network index counters. The counters are used and altered as in CD*n* above.

DC*n* The data is chosen from the user table using the device index counter specified by *n*, where *n* is an integer from one to the value specified by the CNTRS operand on the NTWRK statement. The index counter is not incremented when referenced. The counter value is not reset to 0 when it references a value past the last element in a user table. If *n* is too large and the entry does not exist, no data is inserted.

TC*n* The data is chosen from the user table using one of the terminal index counters. The counters are used and altered as in DC*n* above.

LC*n* The data is chosen from the user table using one of the line index counters. The counters are used and altered as in DC*n* above.

NC*n* The data is chosen from the user table using one of the network index counters. The counters are used and altered as in DC*n* above.

F*n* The entry is chosen by fixed selection (the same entry is to be chosen each time), where *n* is an integer specifying the entry in the UTBL. If *n* is too large and the entry does not exist, no data is inserted.

R The data is chosen from the table randomly. A random number is generated and the table entry referenced is placed in the generated message.

R*n* The data is chosen from the table randomly using the distribution defined by the UDIST statement referenced by *n*, where *n* is an integer from 0 to 255.

$VTAMAPPLID$
The name field from the VTAMAPPL statement is inserted in the data. The blanks used to pad the name to eight characters are deleted.

$YEAR$
The last two characters of the number of the current year are inserted in the data.

Example of the length data field options

The examples below show the use of the $EL$, $L$, and $LL$ data field options:

Text Statement	**Results**
TEXT ($L$ABC$EL$)	X'04C1C2C3'
TEXT ($LL$ABC$EL$)	X'0005C1C2C3'
TEXT ($LL$ABC)	X'0005C1C2C3'
TEXT ($L$ABC$L$DEF$EL$$EL$)	X'08C1C2C304C4C5C6'

Chapter 10. Data locations

Use the following tables when coding IF statements or recalling data from some location to identify the data location. For additional information, refer to , SC31-8945.

Table 10. Data locations for non-SNA terminals

Option	Data Location for Non-display Devices	Data Location for Display Devices
B + 0	First byte of incoming or outgoing data or message being built	First position of screen image (row 1, column 1)
B - 0	Last byte of incoming or outgoing data or message being built	Last position of screen image
D + 0	First byte of incoming or outgoing data or message being built	First byte of incoming or outgoing data or message being built including control information
N + 0	First byte of network user area	First byte of network user area
N - 0	Last byte of network user area	Last byte of network user area
U + 0	First byte of terminal user area	First byte of terminal user area
U - 0	Last byte of terminal user area	Last byte of terminal user area
C + 0	Unpredictable results	Current position of cursor
C - 0	Unpredictable results	Current position of cursor
s + 0	First byte of device save area specified by *s*	First byte of device save area specified by *s*
N*s* + 0	First byte of network save area specified by N*s*	First byte of network save area specified by N*s*
(*row*,*col*)	Not executed if coded on an IF when LOCTEXT or LOCLENG is not specified. If not coded on an IF then the result will be unpredictable.	Specified row and column of the screen image

Table 11. Data locations for SNA terminals

Option	Data Location for Non-display Devices	Data Location for Display Devices	Test SNA Responses
B + 0	First byte of data in the device buffer or message being built, excluding SNA headers	First position of screen image (row 1, column 1)	No
B - 0	Last byte of incoming or outgoing data or message being built	Last position of screen image	No
D + 0	First byte of incoming or outgoing data or message being built, including TH and RH	First byte of incoming or outgoing data or message being built, including TH and RH	Yes
N + 0	First byte of user area	First byte of user area	Yes
N - 0	Last byte of user area	Last byte of user area	Yes

Table 11. Data locations for SNA terminals (continued)

Option	Data Location for Non-display Devices	Data Location for Display Devices	Test SNA Responses
U + 0	First byte of user area	First byte of user area	Yes
U - 0	Last byte of user area	Last byte of user area	Yes
C + 0	Unpredictable results	Current position of cursor	No
C - 0	Unpredictable results	Current position of cursor	No
TH + 0	First byte of transmission header	First byte of transmission header	Yes
RH + 0	First byte of request header	First byte of request header	Yes
RU + 0	First byte of request unit	First byte of request unit	Yes
s + 0	First byte of device save area specified by *s*	First byte of device save area specified by *s*	Yes
N*s* + 0	First byte of network save area specified by N*s*	First byte of network save area specified by N*s*	Yes
(*row*,*col*)	Not executed if coded on an IF when LOCTEXT or LOCLENG is not specified. If not coded on an IF then the result will be unpredictable.	Specified row and column of the screen image	No

Note:

- Logic tests on switches, counters, or locations specified by the AREA operand will not be executed when SNA responses are sent or received when SNASCOPE=LOC is specified on the IF statement. SNASCOPE=LOC is the default. This can be changed by using the SNASCOPE operand such that any *loc* option specification can be tested on any SNA flow.
- When multiple partitions are defined for a 3270 (LU2) terminal, buffer or cursor offsets (B+, B-, C+, C-) will reference the data in the presentation space of the currently active partition. The combination (*row*,*col*) value will reference the display as you would see it, which could include data from more than one partition. The logic test will be performed against the presentation space data of the partition that owns the area of the display referenced by the (*row*,*col*) specification.
- Tests of data against the data stream, B±, D+, TH+, RH+, or RU+ are dependent upon where the test is performed in the deck. For example, if you display the RH while in message generation, you will see the default RH, which may or may not be the actual RH sent. Displaying these areas prior to the initial intermessage delay being set will yield unpredictable results.

Chapter 11. Terminal, device, and logical unit types

The following lists define the types of terminals simulated by WSim according to the values specified by the TYPE and LUTYPE operands on the network configuration statements.

The list below defines the valid terminal types simulated by the TYPE operand on the DEV statement.

Operand	**Terminal Simulated**
FTP	TCP/IP FTP Client Device
LU0	SNA logical unit type 0
LU1	SNA logical unit type 1
LU2	SNA logical unit type 2 (3270 Display)
LU3	SNA logical unit type 3 (3270 Printer)
LU4	SNA logical unit type 4
LU6	SNA logical unit type 6
LU7	SNA logical unit type 7 (5250 Display)
STCP	Simple TCP Client
SUDP	Simple UDP Client
TN3270	Telnet 3270 Client Display
TN3270E	Telnet 3270E Terminal
TN3270P	Telnet 3270E Printer
TN5250	Telnet 5250 Terminal
TNNVT	Telnet Line Mode Network Virtual Terminal Client

The list below defines the valid terminal types simulated by the LUTYPE operand on the LU statement.

Operand	**Terminal Simulated**
LU0	SNA logical unit type 0
LU1	SNA logical unit type 1
LU2	SNA logical unit type 2 (3270 Display)
LU3	SNA logical unit type 3 (3270 Printer)
LU4	SNA logical unit type 4
LU6	SNA logical unit type 6
LU62	SNA logical unit type 6.2
LU7	SNA logical unit type 7 (5250 Display)

Chapter 12. Counters and switches

A WSim network has a set of switches and counters associated with it. One set of network counters and one set of network switches is allocated for each network. Line, terminal, and device counters are allocated for the resources defined within your network.

The following table shows the sets of counters that are allocated to each simulated device.

Table 12. Line, terminal, and device counters

Simulation type	LSEQ LC1 - LC*n*	TSEQ TC1 - TC*n*	DSEQ DC1 - DC*n*
CPI- C TP			
APPCLU	X		
TP		X[Note 1]	X[Note 2]
VTAM Application			
VTAMAPPL	X	X	
LU			X
TCP/IP Connection			
TCPIP	X	X	
DEV			X

Notes:

1. This set of counters is available to all instances of a given CPI-C transaction program.
2. This set of counters is unique for each CPI-C transaction program instance.

The valid specifications for counters and switches are shown below.

NSW*n*
NSW*n*&NSW*m*&..
NSW*n*|*m*|...
: Specify the testing of one or a combination of the 4095 network level switches. You can test combinations of switches by using the & (and) or | (or) logic operators. The specified switches are always tested for being on.

TSW*n*
TSW*n*&TSW*m*&..
TSW*n*|TSW*m*|...
: Specify the testing of one or a combination of the 4095 switches available to you for each terminal. You can test combinations of switches by using the & (and) or | (or) logic operators. The specified switches are always tested for being on.

SW*n*
SW*n*&SW*m*&..
SW*n*|SW*m*|...
: Specify the testing of one or a combination of the 4095 switches available to you for each device. You can test combinations of switches by using the & (and) or | (or) logic operators. The specified switches are always tested for being on.

You can use the following to compare data against the sequence and terminal or device counters during message generation.

NSEQ	NTWRK sequence counter.
LSEQ	LINE sequence counter.
TSEQ	TERM sequence counter.
DSEQ	DEV sequence counter.
NC*n*	NTWRK index counter where *n* is an integer from 1 to 4095.
LC*n*	LINE index counter where *n* is an integer from 1 to 4095.
TC*n*	TERM index counter where *n* is an integer from 1 to 4095.
DC*n*	DEV index counter where *n* is an integer from 1 to 4095.

Chapter 13. Format control statements

You can use the following statements to format the network listings for better readability. You can place these statements anywhere within the network configuration definition statements or the message generation statements.

Statement	Action
%EJECT	Eject the listing to a new page.
%SPACE	Skip one blank line.
%SPACE 1	Skip one blank line.
%SPACE 2	Skip two blank lines.

Note: You must code the % control character in the first statement column. You can code the remainder of the command anywhere on the statement before column 72.

Chapter 14. Conditions logic test not evaluated

The following rules outline all of the conditions for which a logic test is not evaluated for a transmitted or received message:

- The IF statement specifies WHEN=IN for a transmitted message or WHEN=OUT for a received message.
- The terminal type specified by the IF statement TYPE operand does not match the type of terminal associated with the message.
- The SNA flow type specified by the IF statement SNASCOPE operand does not match the flow type.
- A user area or a save area was specified by the IF statement LOC or AREA operand, and the area does not exist for the terminal and the LOCLENG operand is not coded.
- The IF statement LOC operand specifies (*row,col*) for a non-display terminal and the LOCLENG operand is not coded.
- The beginning location for the test as specified by the LOC or AREA operand is not within the available data and the LOCLENG operand is not coded.
- The ending location for the test as specified by the LOC and TEXT operands or the AREA and LENG operands is not within the available data and the LOCLENG operand is not coded.
- The LOC operand is set to a null value (no data) and the LOCLENG operand is not coded.
- The TEXT operand is set to a null value (no data) and the LOCLENG or LOCTEXT operand is not coded.
- The TEXT=RESP operand is coded but a RESP was not coded on the previous TEXT statement and the LOCLENG or LOCTEXT operands are not coded.
- When the value of LOC is greater than the length of the save area and the LOCLENG operand is not coded.
- The IF statement CURSOR operand specifies (*row*,*col*) and the value is not a valid screen position for a display device or it is a non-display device.
- Network-level IF statements are not evaluated for CPI-C simulations. If they are specified, WSim ignores them when simulating CPI-C transaction programs. For combined networks, they are evaluated for all resources except CPI-C transaction programs.
- Message deck IF statements that specify WHEN=IN or WHEN=OUT are ignored for CPI-C simulations.

Part 2. Guide to using STL and the STL Translator

Chapter 15. Introducing the Structured Translator Language

This chapter introduces the Structured Translator Language (STL) and the STL Translator. It explains the role of message generation decks in your network simulation and describes how the STL Translator is used to create these decks as well as the network definitions.

What is STL?

STL is a high-level, structured programming language that enables you to create message generation decks and define terminals and devices you want to simulate with WSim. Like other structured languages, STL uses constants, variables, expressions, and control structures as program elements.

Your STL programs specify the data entered and activities performed by a terminal operator at a terminal or by a device such as a magnetic stripe reader. For example, you can set up a simulation to order inventory items using catalog numbers generated randomly, simulate the actions of operators logging on to a system, or design terminal interactions so that one terminal waits until another has completed certain actions. You can even simulate orderly shutdown in the event of a system failure message. The system under test responds to these messages just as it would to messages generated by actual network resources.

The STL Translator is a utility that translates STL programs into message generation decks. You can also include network definitions in the STL input data set. The STL Translator invokes the Preprocessor to validate these statements for you. Refer to , SC31-8947 for more information about the Preprocessor.

STL provides a convenient way to create message generation decks and include network definitions; however, you can still use existing message generation decks and network definitions.

What do STL programs contain?

STL programs can contain all the messages required for the simulation of an entire network, or they can contain only those messages needed to accomplish specific tasks. STL programs contain procedures, which are similar to modules and subroutines in other programming languages. Procedures can call other procedures, enabling you to code a procedure only once and use it as a subroutine.

STL programs are made up of statements, which are composed of keywords, functions, assignments, expressions, and declarations. The elements that make up STL programs are described in Chapter 17, “Understanding the elements of an STL program,” on page 233.

Like other structured programming languages, STL uses variables to represent data items. You can also use symbolic names to represent constants. STL allows three data types for variables: integer, string, and bit (binary). To control conditional and repetitive execution of statements, STL provides structured flow-of-control concepts such as IF/THEN/ELSE and DO WHILE statements. STL also enables you to send and receive messages asynchronously through a set of statements designed specifically for this purpose. See Chapter 18, “Controlling STL program flow,” on page 255 page 255 for more information about structured flow-of-control statements. See “Testing asynchronous conditions” on page 291 for more information about handling messages asynchronously.

How do STL programs relate to a script?

To simulate network resources with WSim, you must code a script, which consists of a network definition and message generation decks.

The network definition defines the terminals being simulated and the various options used for the different lines, terminals, and devices that compose the simulated system. Message generation decks define the messages sent to the system under test, decisions on messages sent or received, some aspects of the timing of the messages, and the interaction among terminals.

You can code an entire script in one STL input data set. This data set can contain both a network definition and STL programs.

The STL Translator translates your STL programs into message generation decks. Each STL procedure is equivalent to one message generation deck. Since an STL program may contain more than one procedure, an STL program may be translated into more than one message generation deck.

This book describes how to write and translate STL programs into message generation decks. However, you cannot create decks in isolation. You must consult the test plan for the test being conducted to find out exactly what your decks should accomplish. See , SC31-8948 for information about typical contents of a test plan. Refer to the network definition to find out what options are coded for various terminals and to determine how this affects your program planning.

Using STL

STL enables you to create messages using familiar programming structures and conventions Designed for ease of use, it is intended to simplify the message generation process. In addition, you should find it easier to understand and maintain programs written in STL. New and experienced users alike should find STL convenient and easy to learn.

This book enables you to learn to write STL programs and use the STL Translator to create message generation decks and network definitions. Because of the similarity of STL to other programming languages, you will probably find that it is easy to learn and enables you to create message generation decks quickly and accurately.

Using message generation statements with STL

Although STL is convenient for writing messages and data transactions for your simulated terminals, it does not replace message generation statements. In fact, the STL Translator translates your STL programs into message generation statements. You may need to be familiar with these statements to understand fully the behavior of the terminals you are simulating, especially when you are debugging a complicated problem.

Existing message generation statements can be included into STL programs, so that you do not have to re-create pieces of code you already have available. You can simply place the sections you want to keep in the STL programs, using appropriate interface coding. For information about how to include message generation statements in STL programs, see “Including message generation statements in STL programs” on page 251.

Using the STL Translator

The STL Translator translates your STL program into the message generation decks required to run a simulation and places the message generation decks into the data set you specify. In addition, the STL Translator will identify any syntax errors in your program, making it easier to debug.

The STL Translator also invokes the Preprocessor for you to verify any network definition statements you included. These are placed in the data set you specify to contain network definitions.

When using the STL Translator, you can place your translated message generation decks directly in the MSGDD, the data set for message generation decks that are ready to use. Alternatively, you can store them in a sequential data set and use the Preprocessor to place them in the proper data set. Your network definitions are placed directly in the INITDD data set, the data set for network definitions that are ready to use. Typically, you will debug your STL programs using the STL Translator and then use the translator to place the translated message generation decks and verified networks directly in the partitioned data set MSGDD and INITDD, respectively.

Figure 2 on page 224 shows the three ways that you can use the STL Translator.

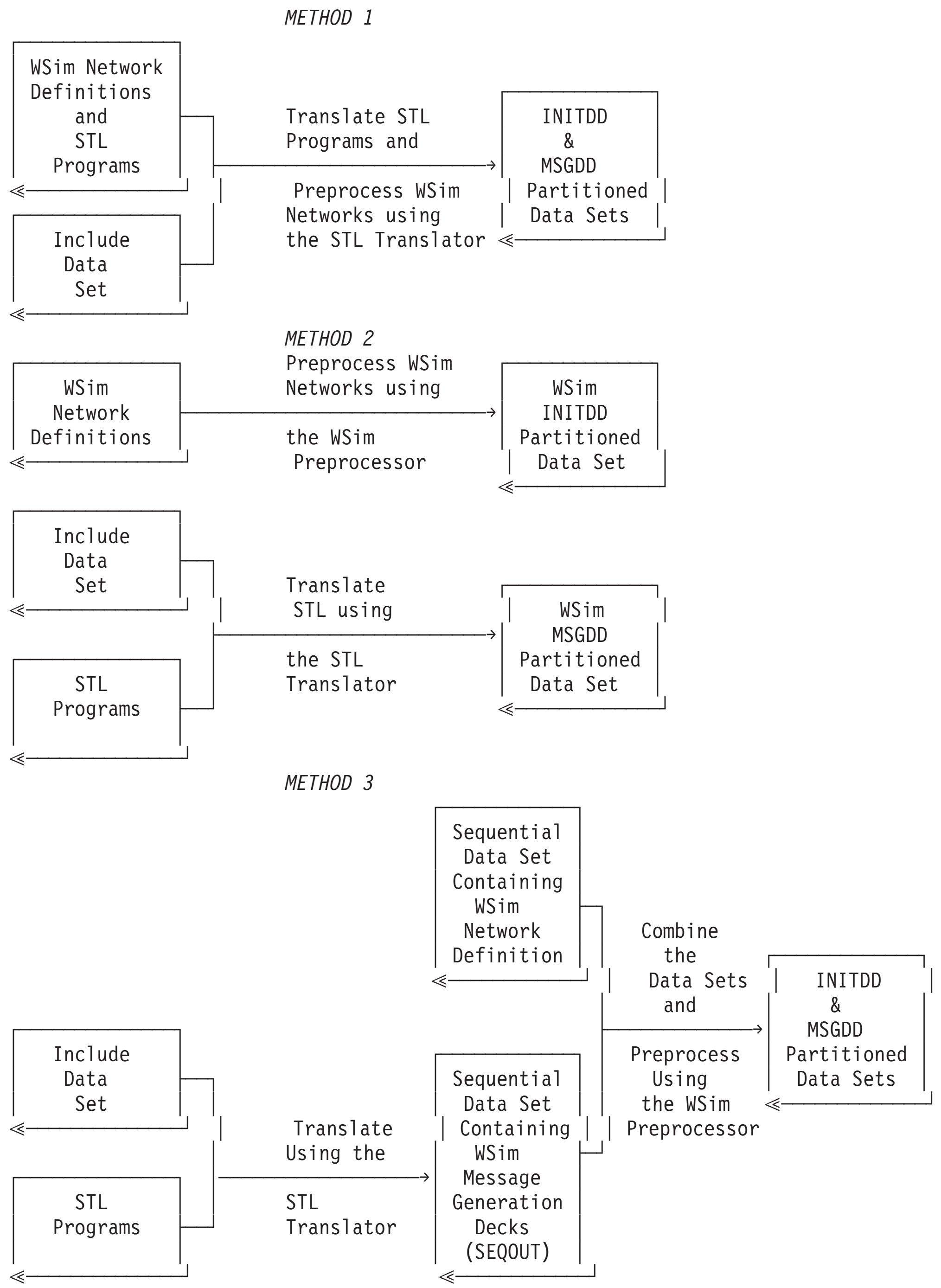

METHOD 1
WSim Network Definitions and STL Programs
Include Data Set
Translate STL Programs and
Preprocess WSim Networks using the STL Translator
INITDD & MSGDD Partitioned Data Sets
METHOD 2
WSim Network Definitions
Preprocess WSim Networks using the WSim Preprocessor
WSim INITDD Partitioned Data Set
Include Data Set
STL Programs
Translate STL using the STL Translator
WSim MSGDD Partitioned Data Set
METHOD 3
Sequential Data Set Containing WSim Network Definition
Include Data Set
STL Programs
Translate Using the STL Translator
Sequential Data Set Containing WSim Message Generation Decks (SEQOUT)
Combine the Data Sets and
Preprocess Using the WSim Preprocessor
INITDD & MSGDD Partitioned Data Sets

Figure 2. Three methods of using the STL Translator

What does an STL input data set look like?

The example below shows a sample STL input data set. It contains a network definition and two STL programs. When the STL programs are translated by the STL Translator, the network definition and associated message generation decks make up a script.

```
@network
testnet   ntwrk   uti=100,bufsize=2400,delay=f(5)
mypath    path    test1
appl1     vtamappl
mylu      lu      lutype=lu2,path=(mypath),
                  frsttxt=logon
@endnetwork
/* Program #1 */
@program
logon: msgtxt          /* Procedure to log on to the terminal.  */
initself('MYAPPL')
endtxt
@endprogram
/* Program #2 */
@program
test1: msgtxt          /* Procedure to test program execution.  */
do i = 1 to 5
   type "Hello"
   transmit using PF1,
      and wait until onin substr(screen,40,17) = 'Hello to you too.'
end
endtxt
@endprogram
```

The network definition shown above defines the network named TESTNET. It contains the following information about the network:

- The NTWRK statement provides the network name and defines various attributes of the network.
- The PATH statement indicates that the procedure named TEST1 in program number 2 is used for the path named MYPATH.
- The VTAMAPPL statement defines the VTAM application symbolic name.
- The LU statement defines one logical unit in the network.

  On the LU statement, the FRSTTXT operand indicates that the LOGON procedure is used once to start the terminal. The LUTYPE operand defines the type of logical unit being used, and the PATH operand indicates that this logical unit will use the procedure named MYPATH defined on the PATH statement, which is the main procedure for program number 2, after it finishes with the logon procedure, LOGON.

The network definition determines the main procedures representing each program which will be used by particular terminals in a network and establishes the order in which they are used. You can design a network definition and associated procedures to be always used together. You can also create more generic procedures that can be used with different network definitions.

See Chapter 23, “Combining STL programs and network definitions,” on page 325 for more information about combining network definitions and STL programs into scripts.

Each STL procedure begins with an MSGTXT statement and ends with an ENDTXT statement. The name of the procedure appears immediately in front of the MSGTXT statement followed by a colon. The first program in the example logs on to the VTAM application. The second sends and receives the given messages five times.

Note: The STL programs shown in the above example have not been translated into message generation statements. Before you can use an STL program, you must run the STL Translator to translate the program into message generation statements. For convenience, in this book the term “running a program” means running the message generation statements produced by the STL Translator.

Unlike a typical program, an STL program does not just execute once and stop when it reaches the end of the program. The program containing the procedure named TEST1 identified on the PATH statement in the network definition (shown above) repeats until the test operator stops the simulation run. The program does not run continuously, however. In WSim, a terminal enters program execution to send a message. After one message is sent, the terminal enters a delay state, called the intermessage delay. During one terminal's intermessage delay, WSim generates messages for other simulated terminals and handles the actual sending and receiving of data. When the delay has expired and the terminal is in a state that the operator can enter data, WSim returns to the first terminal and continues executing its procedures until the terminal sends another message or executes the SUSPEND statement, halting program execution again.

As discussed in , SC31-8945, creating network definitions and message traffic is an iterative process. To code network definitions, you must understand the requirements for messages for each terminal. To code the message traffic, you must understand the network definition and the options coded for the various terminals.

Chapter 16. Designing STL programs

WSim provides flexibility in designing messages. For example, you can set up data tables to simulate a data base. Then, you can use data from the table to generate messages or to compare values from the table against data received in messages from the system under test. Messages received asynchronously, that is, not on an anticipated time schedule, can be tested for certain conditions when they are received. Additionally, you can simulate keystrokes and cursor movements on display terminals.WSim maintains a screen image so that you can simulate display devices.

Designing STL programs is like designing a program in any other programming language. Before beginning to code, you should have a specific plan for what you expect your program to do, how you will divide it into procedures, how you will handle errors, how you will test received messages, and how you will manage any other requirements for the program.

When planning your STL programs, you must:
- Understand the test plan for the network and the messages to be simulated
- Identify any special requirements for your simulated network
- Plan your programs
- Determine documentation procedures
- Plan program testing strategies.

This chapter discusses each of these steps in detail.

Understanding the test plan for the simulated network

A careful definition of what is to be simulated is essential for your design process to be successful. The test plan for the simulation you are conducting provides information about the resources and applications to be simulated and the tasks to be performed. Consult your test planner to find out test objectives and how the message traffic to be simulated contributes to those objectives. If you do not have a test plan for the simulation you are conducting, see , SC31-8948 to determine the information you must have when developing your simulation.

Since the detail in a test plan varies from installation to installation, you may need to define some of the requirements yourself based on your knowledge of the network and the applications you are testing.

Once you have an overview of the requirements from the test plan, you can translate that knowledge into specifics. For example, you may have devices or terminals with particular restrictions. Your network may have specific timing or message-load requirements. You may want to use the same messages for each terminal, or you may need to have each terminal perform different tasks. You will need procedures to handle errors and to send messages to the test operator.

For example, a test plan might call for two terminals that send five different messages with at least a 10-second delay between messages. Each message is sent only after a designated message is received. The test plan indicates that both terminals should send the same messages. To design a program to meet these specifications, you need to answer questions such as the following:

- How are simulated messages created—are they typed in at a terminal, or perhaps obtained from a device such as a magnetic stripe reader?
- What type of terminal are you using?
- What type of network are you using and what are its requirements?
- What kind of data are you sending?
- What kind of data will you receive from the application program?
- What timing is anticipated for messages to be received? Will they arrive on a set schedule or at unanticipated times?
- What timing is anticipated for messages to be transmitted?

The answers to these and similar questions will determine what you need to include when designing and coding your STL programs.

Identifying special requirements

To plan your simulation, you have to know not only what you expect the terminals to accomplish but also what special requirements exist for the system. You may need to consider the following:
- Requirements for products, programs, and specific resources you are testing
- Requirements for running the test
- Use of existing message generation statements.

Using products, programs, and specific resources

When designing your STL programs, you must ensure that you provide the information needed to connect to other products and programs. For example, if you are connecting to an application program, you must identify any special message or logon requirements for the application.

STL enables you to simulate the functions of most IBM terminals. However, many functions that you can simulate using STL are specific to display terminals and cannot be used for other terminal types. As another example, if you are using an SNA network, you may need to use the specific commands available to begin and end SNA sessions.

Deciding how to run the test

Normally, an operator runs simulations. The operator can start and stop various terminals, alter operating parameters, and monitor network activity, among other tasks. For details about operator commands, see , SC31-8948.

You may also incorporate operator commands into your STL programs, so that the simulation can run independently of an operator. For example, you may want to incorporate operator commands when you have an established test that you want to use to collect statistics continuously, or you may have a long test that you want to be able to run after regular working hours.

WSim also enables you to code messages in your STL program that are sent to the operator indicating test progress. Because such messages can be an important test monitoring capability, be sure to plan for them when designing your STL programs.

Including message generation statements in STL programs

You may want to use existing message generation statements as part of your STL programs. For example, you may already have message generation decks for logon procedures. You will need to plan to use these statements and determine how they can be incorporated.

You may also find that you need to incorporate these statements when using specific types of devices. See "Including message generation statements in STL programs" on page 251 for more information about using these statements.

Planning your programs

When designing your STL programs, you must define the program structure and overall organization before you begin to develop the specifics. You should design your simulation beginning with the major tasks to accomplish and then break each task down into smaller and smaller subtasks. In this way, you can identify common subtasks to be structured as separate procedures with the major tasks as separate programs.

You can also determine if any existing message generation decks accomplish tasks that you want to incorporate in your programs.

Follow these steps to develop an STL program:

1. Decide what you want the program to accomplish.
2. Divide the task into its major component parts.
3. Fill in specifics for these parts using pseudocode or ordinary English statements.
4. Code specific parts, repeatedly checking that the parts you have defined are compatible with the complete program design.
5. Test individual program elements and then the program as a whole.

When you plan your programs, remember that STL uses structured programming concepts so that you can move through a program in a linear fashion. By using these concepts, you can avoid the possibility of skipping around and creating a confusing path through the programs. Each procedure in an STL program is treated similarly to subroutines in other programming languages. Like subroutines, procedures can be called. After the called procedure is completed, program execution resumes at the point where the calling procedure left off. See Chapter 17, "Understanding the elements of an STL program," on page 233 and Chapter 18, "Controlling STL program flow," on page 255 for details about the elements and structures of STL programs.

Structuring programs and procedures

The ways that you plan to use your programs determine how you structure them. The way you plan to integrate your network definition and procedures determines how you divide message traffic into different programs. For information about integrating network definitions and STL programs, see Chapter 23, "Combining STL programs and network definitions," on page 325. The uses for various procedures determine how you divide your programs into procedures.

As you create STL programs, remember that the network definition determines which terminals use which procedures. All terminals can use the same procedure or different terminals can use completely different procedures. Procedures can be used in random order or can be run in a predefined order with some procedures being used more frequently than others. The PATH statements and operands in the network definition define which procedures are used for each terminal and the order in which they are run. Information about using the PATH statement with STL programs appears in Chapter 23, “Combining STL programs and network definitions,” on page 325.

You may want to code some standard programs to perform frequently used functions. For example, you can develop programs to put a terminal into an indefinite wait state or to log terminals on to various applications.

Other functions require specific programs. For example, you may want to code one program that invokes a full-screen editor application and another that broadcasts messages to other users of the system under test. You can match programs to the different types of terminals you are simulating and the applications you are using.

Determining message content

What you simulate as message traffic is limited only by the capabilities of the system you are testing and the resources you are simulating. With WSim, you can generate message data, use keyboard keys, send messages at specified intervals, and take actions based on messages received.

The content of your messages is determined by your application. For example, an inventory program would probably require order numbers, part numbers, and prices. Bank programs would require check numbers, money amounts, account numbers, and so forth.

In addition to providing data to your application, you may want to simulate operator actions. Operators may make mistakes in entering information, respond improperly to application data requests, or forget to log off their terminals. You can code your messages to simulate these types of errors.

You can design your system to assign messages to terminals randomly, in a sequence you specify, or in a proportion you specify. The key is that the message content must be appropriate to your application.

To code message content and keyboard activity, you must be familiar with the application being tested and the terminals being simulated. For example, if you are simulating a terminal interacting with a panel-driven application, you need to know field locations for the application. If you are moving the cursor on the panel, you must know the cursor locations you want to use and the keys that are used to move the cursor.

Documenting your STL programs

As in all programming, the STL programmer's task includes more than coding. STL programs must be documented so that other people can maintain and use them.

Each program should include comments at the beginning of the program, preferably in a block surrounded by comment symbols. These comments should indicate:

- What type of terminal the program is used for
- What the program does
- Any special dependencies
- What test the program is part of

- What network definition is to be used with the program.

In addition to comments at the beginning of a program, you should comment the code for your programs extensively. It will help other programmers if you indicate what the program does and the logic that it follows. Using meaningful names for procedures, variables, and constants will also help to document the function of your programs.

In addition to documenting your programs, you or the test planner should keep documentation about the test itself. For example, you should keep copies of the test plans with notations of specifics on how the plan was carried out. These plans should indicate which STL programs were used for particular tests and how the programs were combined with network definitions. If you code the network definition in a separate data set from the STL programs that generate messages for it, you may want to keep copies of all the code together with your test plans, so that all documentation is in one place.

Complete, accurate, and up-to-date test documentation facilitates test repetition. Thorough documentation also makes it easier to build upon existing network definitions and STL programs when modifying simulations or designing new ones.

Testing your STL programs

To ensure that programs are performing as anticipated, test your programs as you develop them. Testing is a two-step process, involving syntax testing and logic testing.

The STL Translator checks program syntax when it translates your STL programs into message generation decks. See Chapter 22, “Using the STL Translator,” on page 311 for more information about how programs are translated and how syntax errors are treated. Correct syntax is only part of creating functioning programs. It does not necessarily mean that programs will act as expected. It only means that your programs will not stop executing because of syntax errors. You must also test the logic of the programs.

To test the logic of your programs, you must know what message traffic and terminal activity you are expecting. Your requirements for each terminal should be specified in the test plan. Before testing your programs, you must combine network definitions and STL programs if they are not included in the same data set. See Chapter 23, “Combining STL programs and network definitions,” on page 325 for information about how to combine them.

Then, request STL message traces by coding STLTRACE=YES in your network definition.WSim will log special STL trace messages during the simulation run. After you run your simulation, run the Loglist Utility requesting STL message traces. Information about using the Loglist Utility appears in , SC31-8947. Specific information about using the Loglist Utility with STL message traces appears in “Obtaining STL trace records” on page 341.

Facilitating STL program development

To facilitate program development, you should begin with a small network and STL program and ensure that they are running correctly before expanding your programs to produce more complex simulations. For example, you might begin by defining one terminal of the type you will simulate. Then, have it log on to the application and send and receive a message. After you accomplish that task, add messages so that the terminal can perform more tasks. After you have one terminal functioning successfully, add more terminals.

You should proceed in an iterative fashion, changing only one network definition or STL program each time. This process makes it easier to identify errors if they occur. Although you may be tempted to move immediately to producing the complete system simulation so that full-scale testing can begin, it is best to develop a comprehensive understanding of how network definitions and STL programs are created on a smaller scale. In the long run, this understanding simplifies development of complex simulations and enables you to correct errors and misunderstandings in a smaller, simpler environment before moving to a larger, more complex simulation.

Chapter 17. Understanding the elements of an STL program

This chapter describes the elements that make up an STL program and introduces the basic concepts that you must understand to create a program. It also discusses how you can incorporate existing message generation statements into an STL program.

What are the basic elements of an STL program?

Each program consists of a controlling procedure and other procedures "called" either by that procedure or by other procedures in the program. Include only one procedure in each program that is not called by another procedure. See Chapter 23, "Combining STL programs and network definitions," on page 325 for more information on how STL programs should be structured. See Chapter 22, "Using the STL Translator," on page 311 for more information about translating programs.

An STL program contains one or more procedures that share a common set of resources, such as variable names. Each procedure is like a subroutine in other languages. Procedures are made up of STL statements. They can be as long or as short as you want depending on the demands of your system.

The following example shows a simple procedure:

```
greeting: msgtxt
type "Hello"
transmit using enter
endtxt
```

Each procedure in a program begins with an MSGTXT statement and ends with an ENDTXT statement. The MSGTXT statement names the procedure. In the preceding example, the name for the procedure is "greeting". Each line is an STL statement. The statements between the beginning and ending statements simulate the action of typing "Hello" at a keyboard and pressing the Enter key to transmit the message to the system under test.

Procedures cannot be nested, that is, you cannot include an MSGTXT-ENDTXT pair of statements inside another MSGTXT-ENDTXT pair.

In addition to procedures, you can include declarative statements in your programs to define tables and symbolic names such as variables and constants. These statements are coded before the procedures that use the elements they define.

What does an STL statement include?

STL statements are made up of the following elements:

- **Variables and constants** represent data items and must be defined as specific data types. "Using variables and constants" on page 237 explains requirements for variables and constants.
- **Keywords** are words that have special meaning to the STL Translator. They are like commands or instructions in other programming languages. See "Using keyword statements" on page 247 for more information about using keywords.

- **Function names** call functions available in STL. A function performs the task or retrieves the data identified by the function name and returns a result. "Using functions" on page 249 describes how to use STL functions.
- **Operators** enable you to define relationships between various elements. Information about operators appears in "Using expressions" on page 248.
- **Punctuation** enables you to separate parts of your statements. Punctuation is described in "Using STL syntax."
- **Labels** enable you to identify STL statements. See "Using STL syntax" for information about labels.
- **MSGTXT** and **MSGUTBL names** enable you to refer to your procedures and tables at a later time. See "Using STL syntax" for information about these items.

STL uses three types of statements:

- **Declarative statements** enable you to declare variable types and classes; allocate counters, switches, and save areas; define constants; and define user tables. For more information about using declarative statements, see "Using declarative statements" on page 242.
- **Assignment statements** assign a value to a variable. STL variables represent storage locations in the computer's memory. The data in one of these locations can be modified by assigning it a new value. For more information about assignments, see "Using assignment statements" on page 246.
- **Keyword statements** control what the program does, similar to instructions or commands in other languages. These statements always begin with a keyword. They are discussed in "Using keyword statements" on page 247.

To obtain data values, you can use the following constructions:

- **Expressions** are a part of a statement that determines or calculates a value. The value can be numeric, character, or binary. An expression can include variables, constants, operators, and functions. With an expression, you can define relationships between constants and variables and calculate new values for variables. Expressions can be a part of any of the three types of statements. See "Using expressions" on page 248 for more information about expressions.
- **Functions** return a specific data value. STL does not allow user-written functions; all functions are built-in. A function performs the task or retrieves the data identified by its name and returns a result. Functions can be used as part of an expression. For more information about using functions, see "Using functions" on page 249.

Using STL syntax

You must follow the syntax rules described in this section when coding STL programs.

In STL programs, statements can begin in columns 1 through 254 and can extend through column 255. Thus, you can use standard programming style with indenting. A statement can appear alone or with several other statements on the same line. Statements can also be continued over more than one line. Each statement ends with a semicolon (;) or with the end of the line if it is not being continued. The following two statements are identical:

```
message_count = 0
message_count = 0;
```

More than one statement can be included on the same line as long as you separate the statements with a semicolon:

```
message_count = 0; error_count = 0
```

To write long statements, continue statements on more than one line by placing a comma (,) after each line to be continued, for example:

```
type "The quick brown fox jumps over the",
     "lazy dog."
```

Note: There will be a space between “the” and “lazy” because strings are concatenated with a space unless you use the concatenation operator. See “Constant types” on page 239 for more information about the concatenation operator.

Place spaces before and after variable names, operators, and keywords. Although spaces are not always required by the language syntax, this convention is recommended for readability. Parentheses must be used in pairs.

To continue functions or statements requiring commas to separate a list of items, you must code two commas on the end of the statement, as shown below.

```
/* In this case, the first comma separates string3 and string4, while */
/* the second comma indicates that the statement will be continued.   */
string string1,string2,string3,,
       string4,string5
```

STL uses a number of reserved words. These are words that have special meaning to the language and cannot be used in a different context. Reserved words include keywords, function names, special reserved variables, and bit values. All STL reserved words are listed alphabetically in Chapter 29, “STL reserved words,” on page 507.

Note: In the text of this book, all reserved words are displayed in uppercase for ease of reference. You do not have to use uppercase when you code them. See “Typographic conventions” on page xv for complete information about the typographic conventions used in this book.

You can give a label to most statements. The label enables you to refer to the statement from elsewhere in your program.

Labels must follow these rules:

- Labels can include these characters: uppercase and lowercase alphanumeric characters and the special characters $, @, and #.
- Labels cannot begin with a number.
- Labels must be from 1 to 8 characters long.
- Labels cannot be STL reserved words or variable names used in the program.
- Labels cannot begin with the character combinations $LA, $SET, or $INC. (These characters are used by the STL Translator.)

Use the following syntax for labels:

```
label: statement
```

You must provide a name on the MSGTXT statement to name each procedure and on MSGUTBL statements to name each table.

MSGTXT and MSGUTBL names must follow these rules:

- Names can include these characters: uppercase and lowercase alphanumeric characters and the special characters $, @, _(underscore), ?, and #.

- Names cannot begin with a number.
- Names must be from 1 to 8 characters long.
- Names cannot be STL reserved words or variable names used in the program.
- Names cannot begin with the character combinations $LA, $SET, or $INC. (These characters are used by the STL Translator.)

It does not make a difference whether you use uppercase or lowercase letters when you are programming in STL except when you use strings and in certain designated situations explained in later chapters. However, the STL Translator translates all variable names to uppercase. Thus, the STL Translator regards “Message” and “message” as the same variable.

You can place comments anywhere in an STL source data set. Comments begin with the characters /* and end with the characters */. Each comment must have these beginning and ending characters, but they do not have to be on the same line. Anything between the characters is regarded as a comment. The following examples show how comments can be coded.

```
/****************************************************/
/* This is a comment.  The statement below contains  */
/* a comment on the same line as the statement.      */
/****************************************************/
message_count = 0                  /* Initialize message count.       */
/****************************************************/
/* Comments may continue over several lines.  See   */
/* the next statement for an example.               */
/****************************************************/
message_count = message_count + 1   /* Increment the count of messages
                                       received.                     */
```

You can nest comments, that is, a comment can contain another comment. This capability is useful if you want to deactivate a section of an STL program temporarily, as shown in the following example.

Before Deactivation

```
message_count = 0           /* Initialize count of messages received. */
error_count = 0             /* Initialize count of errors detected.   */
```

After Deactivation

```
/*
message_count = 0           /* Initialize count of messages received. */
error_count = 0             /* Initialize count of errors detected.   */
*/
```

In the preceding example, enclosing the two statements and their comments with comment characters deactivates the statements. By removing the added comment characters, you can quickly reactivate the two statements. To ensure that your comments appear with the intended member associated with each procedure or user table of the MSGDD you should always code your comments following the MSGTXT or MSGUTBL statements. For example:

```
ATEST: msgtxt
/*************************************************************/
/* These are the block comments for this procedure.          */
/*************************************************************/
&#38;#8942;
endtxt
mtable: msgutbl
/*************************************************************/
```

```
/* This table is used for test purposes.                     */
/**************************************************************/
&#38;#8942;
endutbl
```

Notes:

- STL comments `/*` and `*/` cannot be coded within the network definition. See “Including network definition statements in STL” on page 325 for more information.
- Double-byte character set data must be enclosed in SO and SI characters and must be coded on the same input line. The SI/SO pairs of characters that result from DBCS data in a string constant ending on one input line and continuing again, with the `||` symbol, on another input line are removed from the resulting string.

  For example:

  ```
  A = '<.A.B.C>'||,
      '<.D.E.F>'
  ```

  results in:

  ```
  A = '<.A.B.C.D.E.F>'
  ```

  For more information about DBCS, see “Simulating DBCS terminals” on page 271.

Using variables and constants

In STL, you can use variables and constants as data items in your programs. A variable contains data that is used by a program in a certain way, but whose value can vary. In a program, each variable has a unique symbolic name. The value of a variable changes, but not its name.

Constants are values that do not change in the course of program execution. Constants can be used to initialize variables, to test the contents of variables, and to generate messages. They can also make up part of STL expressions. You can assign names to most constants. STL substitutes the actual constant value for the constant name.

Variables are classified as one of three data types: integer, string, or bit. Constants are classified as integer, string, hexadecimal string, or bit data types. Once a type is associated with a variable or named constant, you cannot change the type in the course of a program. Variables are also defined as being a “shared” or “unshared” class.

You can declare variable types and classes explicitly using the appropriate declarative statement. You can also declare constant names with declarative statements. You can also define variable types and classes implicitly by assignment. See “Using declarative statements” on page 242 for an explanation of how variable types and classes and constant names are declared explicitly. See “Using assignment statements” on page 246 for an explanation of how to use assignment statements.

The names you assign to variables and named constants follow rules similar to those for statement labels. Variable and constant names must follow these rules:

- Names can include these characters: uppercase and lowercase alphanumeric characters and the special characters $, @, _ (underscore), ?, and #.

- Names cannot begin with a number.
- Names must be from 1 to 32 characters long.
- Names cannot be STL reserved words.

Variable types

The type of a variable refers to the type of data represented by the variable. STL uses three variable types:

- **Integer variables** can take any positive integer value from 0 to 2147483647. When a network is first initialized, the value of an integer variable is 0. Integer variables translate into counters in the scripting language.

  Note: An integer variable's value will wrap if incremented beyond 2147483647 or decreased below 0. That is, if a variable has a value of 2147483647 and 1 is added to it, the result is 0. If a variable has a value of 2 and 3 is subtracted from it, the result is 2147483647.
- **String variables** can contain only characters. They must be from 0 to 32767 characters long. A character can be any 1-byte value. (A 1-byte value is X'00' to X'FF'.) Double-byte character set data must be enclosed in SO and SI characters and must be coded on the same record. For more information on DBCS, see "Simulating DBCS terminals" on page 271. When a network is first initialized, the value of a string variable is '' (the null string, which is represented by a pair of single or double quotation marks). String variables translate into save areas in the scripting language.
- **Bit variables** represent binary data items—that is, data items that can take one of two possible values. STL bit variables can have the value of ON or OFF. When a network is initialized, all bit variables have a value of OFF. Bit variables translate into switches in the scripting language.

You should initialize integer, string, and bit variables explicitly whenever possible since the save area, counter, or switch represented by the variable may have been altered before the execution of an STL program.

Variable classes

Variables are defined by class as well as type. STL defines two variable classes:

- **Shared variables** can be used by all terminals in a network. Shared variables translate into network counters, save areas, and switches in the scripting language.
- **Unshared variables** can be used only by individual terminals; each terminal has its own private copy of an unshared variable. Unshared variables translate into device counters, save areas, and switches in the scripting language.

Usually variables in STL programs are unshared. Sometimes you may want to keep statistics across an entire network or to communicate between terminals in a network. In these cases, you can use shared variables.

You must explicitly declare variables as shared. If variables are not explicitly declared as shared, they will be implicitly declared as unshared. See "Using declarative statements" on page 242 for more information about declaring variables explicitly.

You can declare up to 4095 shared integer variables, 4095 shared string variables, and 4094 shared bit variables. Also, in an STL program for a terminal, you can declare up to 4095 unshared integer variables, 4095 unshared string variables, and 4095 unshared bit variables.

The following example illustrates the difference between shared and unshared variables.

```
integer unshared terminal_error_count   /* This statement declares an */
                                        /* unshared integer variable. */
integer shared network_error_count      /* This statement declares a  */
                                        /* shared integer variable.   */
 .
 .
 .
terminal_error_count = terminal_error_count + 1
network_error_count = network_error_count + 1
```

Terminals A and B execute the statements in the preceding example; terminal A executes them first and the initial value of the variables is 0. When terminal A completes execution, its copy of “terminal_error_count” has a value of 1 and the shared variable “network_error_count” has a value of 1. When terminal B completes its execution, its copy of “terminal_error_count” has a value of 1 and the shared variable “network_error_count” has a value of 2, because it has been increased by both terminals.

Reserved variables

Reserved variables are variable names that have special meaning in STL. Like all other STL variables, these variables represent data items. Since their names are reserved, you cannot define your own variables using these names.

The reserved variable names are the following: BUFFER, DATA, RH, RU, SCREEN, and TH. The reserved variables BUFFER and SCREEN are treated identically by the STL Translator and can be used interchangeably. BUFFER is typically used for nondisplay terminals, and SCREEN is typically used for display terminals. These variables refer either to the terminal input or output buffer (excluding SNA headers) for nondisplay terminals or to the screen image buffer for display terminals. The other four reserved variable names refer to the RH, RU, TH, and DATA portion of the data stream. See “Testing asynchronous conditions” on page 291 for information about using reserved variables with asynchronous conditions.

The following list explains the type of data that exists in each of these reserved variables.

BUFFER or SCREEN
For nondisplay terminals, the device buffer. For display terminals, the screen image.

DATA or TH
Incoming or outgoing data, including SNA headers if present.

RH The SNA request/response header (RH) portion of incoming or outgoing data plus the SNA request/response unit (RU).

RU The SNA request/response unit (RU) portion of incoming or outgoing data.

Constant types

Constants are classified into four types: integer, string, hexadecimal string, and bit. You do not declare types for constants; the value of a constant determines its type. The types can take the following values:

- **Integer constants** are positive decimal integers from 0 to 2147483647.

```
message_count = 0       /* 0 is an integer constant assigned to the    */
                        /* integer variable "message_count."           */
if error_count = 5 then ...
                        /* Here the integer constant 5 is used to test */
                        /* the current value of the integer variable   */
                        /* "error_count."                              */
```

- **String constants** are any set of characters. A string constant must be enclosed in a pair of single or double quotation marks (the string delimiter character). If a string constant contains the string delimiter character (a single or double quotation mark), that character must be entered twice so that it will be recognized. You will probably want to use single quotes as your delimiter character for strings containing double quotes and double quotes as a delimiter for strings containing single quotes. You do not have to use the same delimiter character throughout your program; you can use whichever is most appropriate for each string you are enclosing.

  The following examples show how string constants and the delimiting characters are used.

```
message = 'Hello'     /* The string constant 'Hello' is assigned to   */
                      /* the string variable "message."               */
message = 'Isn''t this fun!!!'
                      /* Notice the doubling of the single quotation  */
                      /* mark in the contraction when it is used as   */
                      /* the string delimiter.                        */
message = "Isn't this fun!!!"
                      /* Notice the single quotation mark is not      */
                      /* doubled in the contraction when double       */
                      /* quotation marks are used as string           */
                      /* delimiters.                                  */
```

  Note: The string constants '' and "" have a length of zero and are called the null string.

  You can continue a string constant without a blank between the continued characters by ending a line with a quotation mark, entering the concatenation operator (||) and a comma, and continuing on a subsequent line with the rest of the string enclosed in quotation marks. If you do not use the concatenation operator, the continuation of the string follows an intervening blank (X'40').

  The following examples show how strings are concatenated.

  In the first example, the value of the variable "introduction" is "HelloGoodbye" without an intervening blank because the concatenation operator joins the two strings without a blank.

```
introduction = 'Hello'||'Goodbye'
                      /* Here a string constant is used as part of a */
                      /* string expression.                          */
```

  In the next example, there is one space between "men" and "to" because a space is included before the string delimiter character. The concatenation operator joins the strings without an intervening blank.

```
/*********************************************************************/
/* Below, a long, continued string constant is assigned to "message."    */
/*********************************************************************/
message = 'Now is the time for all good men '||,
          'to come to the aid of their country.'
/*********************************************************************/
/* The previous STL statement assigns the variable "message" the value:  */
/* Now is the time for all good men to come to the aid of their country. */
/*********************************************************************/
```

  Note that ending spaces are included as part of the string.

You could have included the blank and omitted the concatenation operator by coding the statement as follows:

```
message = 'Now is the time for all good men',
          'to come to the aid of their country.'
```

This statement would assign the same value to the variable "message".

There are no limits on the length of string constants, although string constants greater than 32767 characters may be truncated before use. If a string constant is too long, you will get a message indicating a possible truncation in your STL trace messages. See "Obtaining STL trace records" on page 341 for more information about STL trace messages.

Note: Double-byte character set data must be enclosed in SO and SI characters and must be coded on the same record.

- **Hexadecimal string constants** are specified by enclosing pairs of hexadecimal digits in string delimiters followed by the character x or X. Each pair of hexadecimal digits represents a single character in the string.

  The following examples show how hexadecimal strings can be used in STL statements. Both statements have the same effect. The first statement assigns a hexadecimal string constant to a string variable. The second statement assigns the same value using a normal character string constant.

  ```
  message = 'C8859393965A'x      /* This statement uses a hexadecimal string. */
  message = 'Hello!'             /* This statement uses a normal string.      */
  ```

  You must use hexadecimal string constants when the data cannot be expressed using printable characters, as shown in the following example.

  ```
  if substr(data,1,3) = '0030FF'x then ... /* Test for a special 3-byte code. */
  ```

- **Bit constants** can take the value ON or OFF. These constants can be assigned to bit variables or used to test the current setting of a bit variable. The following examples show how bit constants can be used.

  ```
  error_occurred = off             /* Bit variable "error_occurred"    */
                                   /* is initialized to OFF.           */
  if message_received = on then ... /* The bit constant ON is used to   */
                                   /* test the current setting of      */
                                   /* bit variable "message_received." */
  ```

Named constants

You can declare names for integer and string constants using declarative statements. See "Using declarative statements" on page 242 for information about using constant names.

Constant names follow the same rules as variable names. Once the name is declared, the STL Translator substitutes the actual constant for the name whenever the name is encountered during translation.

When possible, you should use named constants instead of string or integer variables because STL allocates resources—save areas, counters, or switches—for variables but not for named constants. Thus, named constants use fewer resources. See "Allocating WSim resources" on page 246 for information about allocating save areas, counters, and switches.

Named constants are useful when you use a constant repeatedly in an STL program, particularly if it is a long string constant. They also make it easier to change a constant globally throughout an entire program if necessary.

Using declarative statements

Declarative statements enable you to provide definitions for the STL Translator that are in effect for all the statements in a program following the declarations.

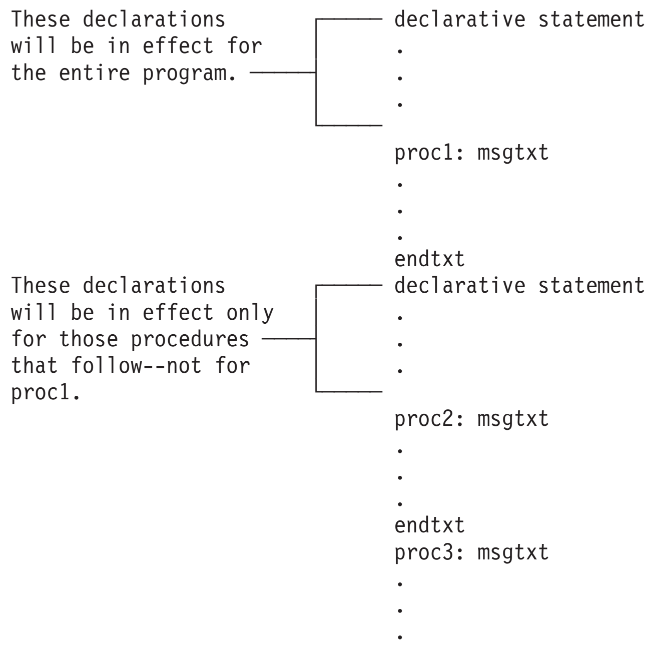
The four types of declarative statements enable you to define the following STL elements:

- Variable types and classes
- Constant names
- User tables
- Allocation of save areas, counters, and switches to variables.

You must code declarative statements outside procedures; they cannot be located between MSGTXT and ENDTXT statements. Declarative statements that you code at the beginning of a program are in effect for the entire program. Those coded between procedures in a program are only in effect for the procedures that follow.

It does not matter whether you code all declarative statements at the beginning of a program or intersperse them between the procedures that make up a program. However, you cannot use a variable or constant name defined on a declarative statement before its declaration in the source data set. For this reason, you may find it helpful to place all declarative statements preceding the first procedure in your program.

The example in Figure 3 shows how declarative statements and procedures are coded in an STL program.



```
These declarations          ┌────── declarative statement
will be in effect for       │       .
the entire program. ────────┤       .
                            │       .
                            └──────
                                    proc1: msgtxt
                                    .
                                    .
                                    .
                                    endtxt
These declarations          ┌────── declarative statement
will be in effect only      │       .
for those procedures ───────┤       .
that follow--not for        │       .
proc1.                      └──────
                                    proc2: msgtxt
                                    .
                                    .
                                    .
                                    endtxt
                                    proc3: msgtxt
                                    .
                                    .
                                    .
```

Figure 3. Placement of declarative statements in an STL program

Note: Variable declarations for one program do not apply to another program. For example, if you had four programs in your input data set, a variable defined in program 1 would not be usable by program 2. Each program has its own set of variables.

The following sections explain the four types of declarative statements.

Declaring variable types and classes

You can use declarative statements to explicitly declare the variables you will use in your program.

You can implicitly declare a variable's type simply by using the variable in an assignment statement. See Using Assignment Statements for more information about this method of declaring variables. Although it may be more convenient to declare variables implicitly, you must be careful to remember the variable type declared so that you do not accidentally mix types.

Explicit declarations enable you to assign the class SHARED to variables and override the default class, UNSHARED. If you want variables to be shared, you must declare them explicitly. If you do not use an explicit declaration, variables that are implicitly declared (declared on an assignment statement) have the class UNSHARED. Explicit declarations also provide a description of the variable type that you can refer to, which can be useful when debugging your programs.

You can explicitly declare one or more variables as integer, string, or bit data types by using the following declarative statements:

```
INTEGER ┌ {SHARED}    ┐ variable_list
        │ {UNSHARED}  │
        ≪             ┘

STRING ┌ {SHARED}    ┐ variable_list
       │ {UNSHARED}  │
       ≪             ┘

BIT ┌ {SHARED}    ┐ variable_list
    │ {UNSHARED}  │
    ≪             ┘
```

Note: In the preceding syntax examples, brackets, [ ], indicate that the items enclosed in brackets are optional. Braces, { }, mean that you should choose one of the items enclosed in braces. Underlined items are the default. Italics indicate items for which you must fill in information. See Typographic conventions for more information about typographic conventions used in this book.

The following examples show how these statements can be used.

```
string mydata, yourdata
integer shared total
bit unshared getout, stayout
```

In the preceding examples, “mydata” and “yourdata” are unshared string variables, “total” is a shared integer variable, and “getout” and “stayout” are unshared bit variables.

Declaring named constants

The CONSTANT statement enables you to give names to constants. This statement uses the following syntax:

CONSTANT *name constant_expression*

The *name* must follow the rules outlined for variable names in “Using variables and constants” on page 237. The *constant_expression* can be any constant expression. See “Using expressions” on page 248 for a definition of constant expressions.

For example, the following statement defines the name “Greetings” to represent a string constant:

```
constant Greetings 'Hello.  How are you?'
```

The examples that follow show how you can use CONSTANT expressions to declare named constants.

```
constant price 100
constant tax     5
constant total_cost price + tax
```

In the preceding example, the named constant “total_cost” has a value of 105. If in the future, “price” changes to 200, the value of “total_cost” will automatically change to 205 since its definition depends on the value of “price”.

Use named constants instead of variables whenever possible because named constants, unlike variables, are not allocated to save areas, counters, or switches. See “Allocating WSim resources” on page 246 for information about save areas, switches, and counters.

The following two examples have the same effect, but the second is less efficient because it requires more resources.

```
/* Example 1. */
constant greetings 'Hello.  How are you?'
⋮
message = greetings
/* Example 2. */
string greetings
⋮
greetings = 'Hello.  How are you?'
message = greetings
```

In the second case, the string “greetings” is explicitly declared as a string variable, rather than being declared as a named constant. In both cases, after execution the variable “message” contains the string 'Hello. How are you?'

Named constants can only be declared with the CONSTANT statement; they cannot be assigned a name implicitly with an assignment statement. An assignment of a string constant produces a string variable rather than a named constant.

All constants are assigned a class of UNSHARED.

Declaring user tables

A user table is a one-dimensional array of string constants. WSim enables you to select entries from this predefined array. You can select entries randomly, pick specific entries, or choose entries in a defined sequence. User tables can be especially useful when your simulated terminals must send messages containing information such as names, part numbers, account codes, or other items that can be specified logically in a list.

You can define a user table in one of three ways:

- Use a UTBL statement in your network definition.
- Use an MSGUTBL message generation statement.
- Use an MSGUTBL declarative statement group in your STL program.

Defining a table in your network definition enables you to use the table when simulating that network only. See *Creating WSim Scripts* for information about defining user tables for a network. However, you may want to use the same table with more than one network definition. To do this, use the MSGUTBL statement.

In your STL program, use the MSGUTBL statement group to define a user table. Place this statement group before the procedures that will use the table. As indicated in "Using declarative statements" on page 242, it is most convenient to place all declarative statements at the beginning of your program unless there is reason to do otherwise.

The syntax of the MSGUTBL statement group is the following:

```
msgutbl_name: MSGUTBL
   utbl_entry
 ┌            ┐
 │ utbl_entry │
 │ .          │
 │ .          │
 │ .          │
 └            ┘
  ENDUTBL
```

msgutbl_name is a 1-character to 8-character name for the table. The name must not have been previously used. The STL Translator remembers this name as an MSGUTBL name and will not permit it to be used as anything else.

utbl_entry is a string constant expression. It can be a simple string constant (enclosed in quotation marks), a named string constant, a hexadecimal string constant, or a concatenation of two or more string constants.

The following example shows a sample user table. Remember that the user table is coded as a part of your STL program but it cannot be a part of a procedure. Because the MSGUTBL statement group is a declarative statement group, you must code it outside pairs of MSGTXT and ENDTXT statements.

```
/* This user table contains customer names. */
customer: msgutbl
   'John Doe'
   'Mary Smith'
   'Sam Jones'
   'Susan Barnes'
   'Bob White'
endutbl
```

In this example, the table contains the following entries. Each entry is identified by a sequential number, called its index. Entry numbers begin with 0.

0.	John Doe
1.	Mary Smith
2.	Sam Jones
3.	Susan Barnes
4.	Bob White

You can have up to 2147483647 entries in each table.

This section explains how to define user tables. See Chapter 19, "Generating messages for an STL program," on page 267 for information about how to use information from user tables in STL programs.

Allocating WSim resources

When an STL program is processed by the STL Translator, WSim allocates save areas, counters, and switches to hold the values of the variables used by the STL program. See *Creating WSim Scripts* for general information about how WSim uses save areas, counters, and switches.

The ALLOCATE statement tells the STL Translator exactly which save area, counter, or switch resource should be allocated to an STL variable when your program is translated. Normally, STL does this allocation automatically. Use the ALLOCATE statement to combine your STL program with message generation decks that already exist or to define variables to be used at the line and terminal level. You may also need to use the ALLOCATE statement to combine several STL programs into one. See “Combining STL procedures from different STL programs” on page 333 for information about using more than one program.

The following examples show how to use the ALLOCATE statement.

```
allocate mydata '1'        /* Use device save area 1 to hold "mydata." */
allocate mycount 'DC1'     /* Use device counter 1 to hold "mycount."  */
allocate netflag 'NSW3'    /* Use network switch 3 to hold "netflag."  */
```

You can use the ALLOCATE statement to define variables as corresponding to network and device save areas; network, terminal, line, and device counters; and network, terminal, and device switches. See “ALLOCATE” on page 363 for more information about how to define specific resources.

The two variable classes used by the STL Translator, SHARED and UNSHARED, correspond to save areas, counters, and switches at the network and device levels, respectively. Consequently, the translator only knows about network and device level save areas, counters, and switches. That is, the STL Translator keeps track of resources—save areas, counters, and switches—at these two levels, rejecting attempts to define or allocate them more than once.

However, the STL Translator does not automatically keep track of line-level and terminal-level counters and terminal-level switches. You may need to use these resources to collect statistics for all the devices associated with a particular terminal or line. If so, you may allocate an STL variable to one of these resources.

For example, the following statement identifies the STL variable “group_count” with terminal counter 13.

```
ALLOCATE group_count 'TC13'
```

The STL Translator uses 'TC13' (terminal counter 13) whenever it encounters the variable “group_count” in a program. You must define this usage in your program because the translator does not track the use of terminal and line resources. In other words, you can declare multiple STL variables that use the same terminal or line resource.

Using assignment statements

STL variables represent storage locations in computer memory. The data in one of these storage locations can be modified by assigning it a new value. An assignment names a variable and gives it a value. In STL, assignments are specified by an equal sign (=). An assignment statement uses the following syntax:

```
variable_name = expression
```

For example, you might make the following assignment:

```
message = 'Take care!'
```

This means that the string 'Take care!' is to be put into the location called “message” in the computer's memory.

If the type of the variable has been declared before the assignment is processed, the type of the variable must be the same as the type of the expression. If the variable type has not been declared, it will be implicitly declared as having the same type as the expression and a class of UNSHARED.

For information about what can be contained in an expression, see “Using expressions” on page 248.

Using keyword statements

Keyword statements advance the progress of the program and complete actions. The following are examples of keyword statements:

```
type "Hello"
transmit using pf8
say 'Test complete.'
```

Each keyword statement begins with a keyword. In the preceding examples, TYPE, TRANSMIT, and SAY are keywords. Syntax and complete descriptions for keyword statements appear in Chapter 25, “Reference to STL statements,” on page 355.

You can code a simple STL program using four basic statements. These statements are the following:

- MSGTXT
- TYPE
- TRANSMIT
- ENDTXT.

The MSGTXT-ENDTXT pair signals the beginning and end of an STL procedure. The TYPE statement simulates an operator typing information at a keyboard, and the TRANSMIT statement simulates the operator sending information to the system under test.

Each keyword statement can have a label. Code the label before the keyword and follow the label with a colon. Use a label if you will want to refer to a statement from elsewhere in your program. The following keyword statement includes a label:

```
test1: type "Hello"
```

Structured flow-of-control statements are a special type of keyword statement that enable you to control the order in which statements in your program are executed. The keywords that begin structured flow-of-control statements are the following:

- CALL
- IF/THEN/ELSE
- DO
- SELECT.

Chapter 18, “Controlling STL program flow,” on page 255 discusses these statements and their use in detail.

Using expressions

You can obtain values to use in your program by using expressions. An expression can be a variable, a constant, or the result of an operation or STL function. STL uses three types of expressions: integer, string, and bit.

Integer expressions

An integer expression can be composed of a single integer variable or constant or these can be joined by one or more arithmetic operations.

The valid operations that you can use for integer expressions are addition (+), subtraction (-), multiplication (*), division (/), and remainder division (//). Expressions involving more than one operation are evaluated by the following rules:

1. High-precedence operations (*, /, //, and functions that return an integer) are evaluated first, left to right as they are found in the expression.
2. Low-precedence operations (+ and -) are evaluated after high-precedence operations, left to right as they appear in the expression.
3. You can use parentheses to alter this order of evaluation.

An integer constant expression contains only integer constants and integer operators. The STL Translator performs any operation involving only integer constants when the program is translated and flags any errors it finds. For instance, any all-constant operation that results in a value less than 0 or greater than 2147483647 is flagged as an error. Division by the constant 0 is also flagged as an error.

WSim performs all other operations (that is, those involving variables and functions) when the program executes. Any of these operations resulting in a value greater than 2147483647 will wrap to 0. Values less than 0 will wrap to 2147483647.

All arithmetic operations are performed using integer arithmetic, that is, fractional portions of a number, whether the result or an intermediate value, will be truncated.

The following are examples of assignments that show valid integer expressions:

```
count = 5                 /* Expression is a single integer constant. */
count = old_count         /* Expression is a single integer variable. */
count = old_count + 1     /* Expression is an addition operation.     */
count = old_count - 2 * 5   /* Expression includes multiple operations: */
                          /* first the product (2 * 5) is evaluated,  */
                          /* then that value (10) is subtracted from  */
                          /* the value of variable "old_count."       */
count = (old_count + 2) * 5  /* Expression includes multiple operations: */
                          /* first the sum of (old_count + 2) is      */
                          /* evaluated, then that sum is multiplied   */
                          /* by 5.  The order of evaluation is        */
                          /* different from the previous example      */
                          /* because parentheses are used.            */
count = 5/2               /* Expression is the result of 5 divided by */
                          /* 2.  Since arithmetic operations are      */
                          /* integer only, the result is 2.           */
```

String expressions

A string expression can be composed of a single string variable, string constant, hexadecimal string constant, or one or more string operations. You can concatenate, or join, strings with or without blanks between them. To concatenate without a blank, use the concatenation operator (||). To concatenate with a blank between strings, simply leave a blank (X'40') between the string expressions. (A single blank is inserted even if more than one blank appears between the two strings being concatenated.) You can use parentheses to group the operations in a string expression, but they do not affect the order of evaluation.

The following examples of assignments show valid string expressions.

```
msg = 'Hello!'              /* Expression is a single string constant. */
msg = greetings             /* Expression is a single string variable. */
msg = 'Hello!'||greetings   /* Expression is the concatenation of a    */
                            /* string constant and a string variable   */
                            /* without an intervening blank.           */
msg = 'Hello!' greetings    /* Expression is the concatenation of a    */
                            /* string constant and a string variable   */
                            /* with an intervening blank.              */
msg = greetings body closing  /* Expression is the concatenation (with   */
                            /* blanks) of three string variables.      */
```

Note: Double-byte character set data must be enclosed in SO and SI characters and must be coded on the same record. The SI/SO pairs of characters that result from DBCS data ending on one record and continuing again on another record, however, are removed from the resulting string. For more information on DBCS, see "Simulating DBCS terminals" on page 271.

Bit expressions

A bit expression can be one of the following: a bit variable, a bit constant, or a bit function. Bit expressions can be evaluated either to the value ON or OFF.

The following examples show bit expressions used in assignments:

```
got_it = found         /* A bit variable is assigned to a bit variable.  */
found = on             /* A bit constant is assigned to a bit variable.  */
too_late = posted('MYEVENT') /* The value of a bit function is assigned   */
                             /* to a bit variable.                        */
```

Using functions

STL provides a number of built-in functions that can be used to access and manipulate data. All functions return a value, which can be an integer, string, or bit value. You can use functions as expressions or parts of expressions wherever variables can be used. STL function names are reserved words. You cannot use them as names or labels.

A function consists of the name of an STL function, followed by a left parenthesis, any arguments for the function, and a right parenthesis. The function arguments consist of values you are supplying to the function. Even if the function does not use arguments, you must code the parentheses. If you specify more than one argument, you must separate them with commas. You can continue argument lists on another line by ending the current line at the end of an argument and continuing with the next argument on a new line. You indicate the continuation of that current line by ending it with two commas (one as an argument separator and one as the continuation indicator). You can continue string constant arguments using the continuation rules for string constants given in "Constant types" on page 239.

Function syntax and details on arguments and the values each function returns appear in Chapter 26, "Reference to STL functions," on page 449.

A number of functions perform translations between data types. A list of these functions and their actions follows:

Function	Action
B2X()	Converts a binary string to a hexadecimal string value.
C2D()	Converts a hexadecimal string to its decimal (integer) value.
C2X()	Converts an EBCDIC character string to its hexadecimal string value.
CHAR()	Converts an integer value to its EBCDIC character representation.
D2C()	Converts a decimal integer value to its hexadecimal string value.
E2D()	Converts an EBCDIC number to its decimal (integer) value.
HEX()	Converts a decimal value to its hexadecimal string value.
X2B()	Converts a hexadecimal string to a binary string.
X2C()	Converts a hexadecimal string to a character string.

A number of functions perform translations between single- and double-byte character set data. A list of these functions and their actions follows:

Function	Action
DBCSADD()	Adds SO/SI characters to string data.
DBCSADJ()	Deletes SI/SO pairs from string data.
DBCSDEL()	Deletes SO/SI characters from string data.
DBCS2SB()	Converts ward 42 (EBCDIC) double-byte character set data to single-byte character set data.
SB2DBCS()	Converts single-byte character set data to ward 42 (EBCDIC) double-byte character set data.
SB2MDBCS()	Converts single-byte character set data to ward 42 (EBCDIC) double-byte character set data and wraps SO/SI characters around the data.

For more information on DBCS, see “Simulating DBCS terminals” on page 271.

Several functions enable you to locate portions of strings. These functions are discussed in the following sections.

The SUBSTR function

You can use the SUBSTR function (substring function) to reference a portion (or a substring) of a string expression. The syntax for this expression is the following:

SUBSTR(*source*,*starting_position*[,*length*])

This function returns a string expression that you can use wherever string expressions can be used. The *source* is the string expression in which the substring is to be found. The *starting_position* describes where to look for the beginning of the string. Optionally, you can include the *length* of the substring you are looking for.

For example, you can use the SUBSTR function as follows:

```
mydata = 'Now is the time'
substring = substr(mydata,5,2)
```

In this example, the variable “substring” is assigned a value of 'is'.

You will often use the SUBSTR function to test asynchronous conditions. The function has a number of limitations when used with asynchronous conditions. See “SUBSTR” on page 488 for more information about these limitations.

The INDEX function

The INDEX function returns an integer value that gives the position of a target string in a source string. If the target string is not found in the source string, the function returns a value of 0.

The syntax of this function is the following:

```
INDEX(source,target)
```

You can use this function to determine whether specific information is included in a string you received, as shown in the following example.

```
source = "What a nice day."
location_found = index(source,'nice')
```

In this example, the variable “location_found” would be assigned the value “8” because “nice” begins at position 8 in the source string.

Including message generation statements in STL programs

You may want to include one or more message generation statements in an STL program. For example, you have existing message generation decks that you want to include into an STL program. Message generation statements to be included in an STL program must be preceded by an @GENERATE statement (which can also be coded @GEN) and followed by an @ENDGENERATE statement (which can also be coded @ENDGEN). The STL Translator uses the message generation statements between the @GENERATE and @ENDGENERATE statements as input lines and places them unchanged in the output. The statements included in the message generation deck produced by the STL Translator are the output lines.

When the STL Translator finds an @GENERATE statement, it enters “generate mode.” While in this mode, the translator processes subsequent lines in your STL program according to the following rules:

1. You can include only string constants, STL variable names, and STL comments in the message generation statements included in the STL program.
2. The STL Translator generates one output line in a message generation deck for each input line in the STL program.
3. When the STL program is translated, the STL Translator copies string constants directly into the output line. You can use named constants in your input statements. Their declared values are substituted for the constant name.
4. STL variable names are replaced by the save areas, counters, and switches used to represent them in message generation statements.
5. No syntax checking is done and offsets are not added for save areas when the save area number is included as a variable name.
6. The output line resulting from the translation must not contain more than 72 characters. Longer output lines result in an error message. Thus, when coding input lines, you must be sure that expansion of named constants and variable names will not result in an output line that is too long.
7. When the STL Translator finds an @ENDGENERATE statement, generate mode ends.

8. While in generate mode, the STL Translator assigns a new message generation statement number to each output line that does not begin with 15 blanks. Be sure to code 15 blanks at the beginning of each line that does not begin a new message generation statement. For example, if you want to continue a line, make sure the continued line begins with at least 15 blanks. Failure to do this can cause errors when tracing your STL program.
9. The LABEL message generation statement should not be coded. It causes errors when tracing your STL program.

Note: WSim interprets a non-blank character in column 1 of an output message generation statement as a label. Be careful to include leading blanks in your generate mode input where necessary.

The example below shows how to use the @GENERATE statement in an STL program. This example calls a user exit routine.

Input lines are coded as string constants. Input lines must be enclosed in STL string delimiter characters (either single or double quotation marks).

Rules for using string delimiter characters in input lines differ from those used for STL statements. See "@GENERATE" on page 355 for an explanation of these rules.

```
example: msgtxt
/*********************************************************************/
/*                      @GENERATE Example                            */
/* Notice that message generation statements are coded as string    */
/* constants.                                                        */
*********************************************************************/
 .
 .
 .
@generate
'         DATASAVE AREA='stringvar',TEXT=($RECALL,U+0,8$)'
@endgenerate
 .
 .
 .
endtxt
```

Note: If you code a network definition in your STL input, you can also include message generation decks between the @NETWORK and @ENDNETWORK statements. See "Including network definition statements in STL" on page 325 for more information.

Including data from other data sets

You can include data from other partitioned data sets in your STL input data set, such as common ALLOCATE statements or procedure headers, by coding the @INCLUDE statement. This statement retrieves the member specified and inserts the contents of that member where you coded the statement.

You can code the @INCLUDE statement anywhere within your STL input data set. When you code multiple statements on the same line with the @INCLUDE statement, you must code the @INCLUDE statement last. Comments can be coded at the end of the line.

You must code complete STL statements in an included member. You can code @INCLUDE statements within an included member; however, you cannot include members recursively (that is, you cannot code @INCLUDE statements in two included members that reference each other). You cannot continue statements across members.

For example, suppose that you wanted to include common user tables in your STL input data set. You can code the @INCLUDE statement as shown below.

```
@program = myprog
@include myutbls    /* Include my common set of MSGUTBLs */
mytest: msgtxt
⋮
endtxt
```

You can code your common user tables in the member named MYUTBLS in the include data set, as shown below.

```
myusers: msgutbl
'user 1'
'user 2'
endutbl
mypws: msgutbl
'password 1'
'password 2'
endutbl
```

When you translate your STL input data set, the STL Translator inserts the user tables, specified by the @INCLUDE statement, where you coded the statement.

For more information on the @INCLUDE statement, see "@INCLUDE" on page 358. For information on the SYSLIB DD data set used to define the include data set, see "Input to the STL Translator" on page 314.

Defining user exits

You can invoke a user exit routine during the execution of your program with the USEREXIT statement. It allows the exit routine to produce messages and control execution of the program by setting return codes.

When you code the USEREXIT statement, you must specify the member (user exit load module) in the load library that was loaded during initialization and that gains control when this statement is encountered during program execution.

You can also specify the parameters to be passed to the user exit when it is called. The following example shows how to code the USEREXIT statement. In this example, the exit "MYEXIT" is called with the parameter "NOWAIT".

```
userexit('MYEXIT','NOWAIT')
```

For more information on the USEREXIT statement, see "USEREXIT" on page 444. Concatenate the user exit data set to the STEPLIB DD JCL statement. For information about coding user exits, refer to , SC31-8950.

Using CPI-C verbs

You can emulate CPI-C verbs with STL statements; the STL statements for CPI-C begin with CMACCP on page "CMACCP — Accept_Conversation" on page 369 and end with CMTRTS on page "CMTRTS — Test_Request_To_Send_Received" on page 394.

You can specify input verb parameters in any of the following ways:

- STL variables (shared or unshared)
- String or integer named constants
- Literal values.

You must specify output verb parameters as unshared STL variables.

Note: The STL Translator requires a free device save area for each string literal or named constant referenced, and a free device counter for each integer literal or named constant referenced. If a free save area or counter is not available when required, the STL Translator issues error message ITP3027I or ITP3028I.

The syntax for the CPI-C STL statements is

```
verbname({string_variable|string_name_constant|'literal'},,           <character input>
        {integer_variable|integer_named_constant|numeric_literal},,   <numeric input>
        unshared_integer_variable,,                                   <character output>
                .
                .
        unshared_integer_variable)                                    <numeric output>
```

Note: CPI-C verb parameters are separated by a single comma. STL also uses a comma to indicate continuation. If a CPI-C verb is continued across more than one line, you must break the line at the end of a parameter, and you must specify two commas.

The following example shows how to code CPI-C verbs as STL statements:

```
Example:
/* Set the symbolic destination name to "SERVER". */
sym_dest_name="SERVER"
/* Initialize a conversation with "SERVER". */
CMINIT (conversation_id, sym_dest_name, return_code)
/* Allocate a conversation with "SERVER". */
CMALLC (conversation_id, return_code)
/* Setup the send buffer and length. */
send_buffer = 'Data to send'
send_length = length(send_buffer)
/* Send data to "SERVER". */
CMSEND (conversation_id,,
        send_buffer,,
        send_length,,
        request_to_send_received,,
        return_code)
/* Deallocate the conversation with "SERVER". */
CMDEAL (conversation_id, return_code)
```

For more information about defining CPI-C networks and scripts, refer to *Creating WSim Scripts*.

Chapter 18. Controlling STL program flow

STL enables you to control the flow of your program by using a subset of keyword statements called structured flow-of-control statements. These statements enable WSim to move through a program in a nonlinear fashion, making it possible for you to program more efficiently.

The following statements and structures are used to control program flow:

- Structured flow-of-control statements
- Conditions.

Structured flow-of-control statements give you the ability to have your procedure call another procedure, passing control to the second procedure. They also enable you to take a specific action depending upon conditions that occur. "Using structured flow-of-control statements" describes the flow-of-control statements available in STL.

Conditions are included in many flow-of-control statement groups. Depending on whether these conditions are evaluated as "true" (the condition exists) or "false" (the condition does not exist), control of program execution may pass to a different location. For information about setting up the conditions used in many of these statement groups, see "Using conditions and relational operators" on page 261.

Using structured flow-of-control statements

Structured flow-of-control keyword statements enable you to execute statements conditionally and repetitively. Unlike the message generation statements, STL does not include a branching or "go to" statement. (See , SC31-8945 for more information about message generation statements.) In STL, you must use the structured control statements to alter program flow.

STL uses four types of control statements:

- CALL
- IF/THEN/ELSE
- SELECT
- DO.

The CALL statement shifts control of program execution from one procedure to another procedure. IF/THEN/ELSE and SELECT statement groups execute statements selectively depending on conditions that exist. DO statement groups enable you to group statements logically and in some cases to execute statements repetitively.

The following sections discuss the use of these statements.

The CALL statement

The CALL statement enables your procedure to execute another procedure, and return to the original, or calling, procedure. Thus, a procedure can be used as a subroutine, called by many other procedures when appropriate. Programming and maintenance are more efficient, because the procedure used as a subroutine does not have to be copied into each procedure that uses it.

Unlike subroutines in other programming languages, in STL you cannot pass data as an argument to a called procedure. When you call a second procedure, you are simply passing control of execution to the called procedure. When the called procedure finishes its execution, control returns to the calling procedure at the statement following the CALL statement.

The syntax of the CALL statement is the following:

```
CALL procedure_name
```

The *procedure_name* is the name of the procedure being called. This name is coded on the MSGTXT statement that begins the called procedure.

Use a RETURN statement in the called procedure to return control to the calling procedure. Although an ENDTXT statement also acts like a RETURN statement, it is good practice to include the RETURN statement in a procedure that is called. Any RETURN statements encountered when a CALL has not been made are ignored.

The following example shows how the CALL and RETURN statements are used:

```
procA: msgtxt
 .
 .
 .
call procB
 .
 .
 .
endtxt
procB: msgtxt
 .
 .
 .
if done_yet? then
   return
 .
 .
 .
endtxt
```

An STL program includes one procedure that is entered on a PATH statement in a network definition, called the main procedure. All other procedures in that program are “called” by a procedure included in the program. See Chapter 23, “Combining STL programs and network definitions,” on page 325 for information about the relationship between STL programs and PATH statement entries.

The IF/THEN/ELSE statement group

The IF/THEN/ELSE statement group enables your program to execute a statement or group of statements conditionally, depending on whether the condition following the IF keyword is true or false.

The syntax of the IF statement is the following:

```
IF condition THEN[;] statement[;]
           [ELSE[;] statement]
```

The statement following the THEN is executed only if the result of the condition is true. See “Using conditions and relational operators” on page 261 for an explanation of what you can include in a condition.

You do not have to include an ELSE and its accompanying statement in your IF statement group. If an ELSE is included, the statement after the ELSE is executed only if the result of the condition following the IF is false.

You can code the statements following the THEN and ELSE keywords on the same line as the keyword or on the following line. The semicolon following the THEN keyword is optional if the ELSE is on the following line. However, if the ELSE is coded on the same line as the THEN, the semicolon is required.

If you want your program to do nothing when a given condition exists or not, code the NOP statement following the THEN or ELSE keyword. Since THEN and ELSE must be followed by some type of statement, NOP enables the program to move to the next statement without taking action.

The following example shows a simple IF statement group:

```
if count = 1 then
   say 'count has a value of 1'
else
   say 'count does not have a value of 1'
```

The SAY statement shown in the previous example displays the given string at the operator's terminal. If “count” equals 1, the message “count has a value of 1” is displayed at the terminal; otherwise, the message “count does not have a value of 1” is displayed.

The statement following a THEN or ELSE keyword can be another structured flow-of-control statement group or a single statement.

An ELSE is associated with the nearest preceding unmatched IF. When you use ELSE statements, be careful either to include an ELSE statement for each IF or to be aware of which IF the ELSE is associated with. The following example shows how you can use structured statement groups within other statement groups and illustrates the proper use of the ELSE keyword.

```
if count = 1 then
   if quiet = off then
      say 'count has a value of 1'
   else
      nop
else
   do
      say 'count does not have a value of 1'
      say 'count =' char(count)
   end
```

In the preceding example, the first ELSE is associated with the second IF, and the second ELSE is associated with the first IF.

The SELECT statement group

The SELECT statement group enables you to execute one of several alternative statements depending on the condition that exists.

The syntax of this group of statements is the following:

```
SELECT
    WHEN condition THEN[;] statement

  ┌                                  ┐
  │ WHEN condition THEN[;] statement │
  │  .                               │
  │  .                               │
  .
   OTHERWISE[;] statement
END
```

The condition following the first WHEN is evaluated as true or false. If the condition is true, the statement following the THEN is executed and execution continues at the statement following the END keyword. The statement following the THEN can be a statement or statement group such as IF/THEN/ELSE, DO, or SELECT. If the condition is false, the program moves to the next WHEN statement group. See "Using conditions and relational operators" on page 261 for an explanation of what you can include in a condition.

If none of the WHEN conditions are true, the program executes the statement or statement group following the OTHERWISE keyword. You must have at least one WHEN statement and an OTHERWISE statement in a SELECT statement group. A SELECT statement group must end with an END statement.

The following example displays the use of the SELECT statement group.

```
data_length = length(message)
select
   when data_length > 100 then
      say 'Data is longer than 100 characters'
   when data_length > 50 then
      say 'Data is longer than 50 characters and shorter than 101 characters'
   when data_length > 10 then
      say 'Data is longer than 10 characters and shorter than 51 characters'
   otherwise
      say 'Data is shorter than 11 characters'
end
```

The DO statement groups

STL provides four types of DO statements:

- Simple DO statement groups
- DO WHILE statement groups
- DO FOREVER statement groups
- Iterative DO statement groups.

Simple DO statement groups execute statements only once, allowing a group of statements to be executed as a single statement. The other types of DO statement groups control the repetitive execution of a group of statements. DO FOREVER statement groups loop forever. DO WHILE statement groups are conditional loops, which continue to execute as long as some condition is satisfied. The iterative DO statement group specifies the number of times the statements in a group are repeated.

Each type of DO statement group must end with an END statement.

Note: @GENERATE and @ENDGENERATE statements also act as an implied DO statement group.

Simple DO statement group

A simple DO statement group enables you to group the statements it includes together so that the statements can be thought of and processed as a single statement. The statements in a simple DO statement group are executed once, unlike the other DO statements discussed in this chapter.

The syntax of a simple DO statement group is the following:

```
DO
   statement
   .
   .
   .
END
```

Simple DO statement groups are often used after a THEN, ELSE, or WHEN statement to allow the conditional execution of multiple statements, as seen in the following example.

```
if count = 1 then   /* Both statements in this DO group        */
   do               /* will be executed if "count" has a       */
      b = 0         /* value of 1.                             */
      c = 0
   end
```

In the preceding example, both assignment statements (b = 0 and c = 0) are executed if “count” has a value of 1. In the next example, however, the assignment c = 0 is always executed.

```
if count = 1 then
   b = 0
c = 0
```

The DO WHILE statement group

The DO WHILE statement group enables you to execute statements repetitively while some condition is true. The DO WHILE statement group uses the following syntax:

```
DO WHILE condition
   statement
   .
   .
   .
END
```

When WSim executes the DO WHILE statement, the condition is evaluated. If the condition is true, WSim executes the statements between the condition and the END statement. Control returns to the DO WHILE statement and WSim evaluates the condition again. When the condition is false, the execution continues with the statement following the END statement. See “Using conditions and relational operators” on page 261 for an explanation of what you can include in a condition.

The following example shows a DO WHILE statement:

```
get_out = off
do while get_out = off
   .
   .
   .
    if a = 100 then
       get_out = on
   else
       nop
end
```

You can end execution of a DO WHILE statement group prematurely by using the LEAVE statement. You can end the current iteration through the statement group by using the ITERATE statement. See “LEAVE” on page 414 and “ITERATE” on page 413 for details and examples of the use of the LEAVE and ITERATE statements.

The DO FOREVER statement group

The DO FOREVER statement group enables you to repeat a statement group forever. WSim executes the statements until a LEAVE statement is encountered or the simulated network is canceled.

This statement group enables your program to perform actions continuously. This capability can be useful if you want a terminal to repeat an action for as long as your simulation is running.

The DO FOREVER statement group has the following syntax:

```
DO FOREVER
   statement
   .
   .
   .
END
```

In the following example, WSim executes the statements in the DO FOREVER statement group until the network is canceled. Thus, it repeatedly types the 10 numbers, transmits them, and waits until a message is received.

```
do forever
   type '1234567890'
   transmit and wait until onin   /* Wait until something is received. */
end
```

In the next example, WSim executes the statements until the message UNRECOVERABLE ERROR appears on the screen. When WSim receives this message, it executes the LEAVE statement, and program execution continues at the statement following the END statement.

```
do forever
   type '1234567890'
   transmit and wait until onin         /* Send message and wait until */
                                        /* something is received.      */
   if index(screen,'UNRECOVERABLE ERROR') > 0 then
      leave                             /* Get out of loop when error  */
                                        /* occurs.                     */
   else                                 /* Continue looping if error   */
      nop                               /* did not occur.              */
end
```

Iterative DO statement

By using the iterative DO statement group, you can repeat a group of statements a specified number of times. You determine the number of times the statements are repeated by defining a control variable and providing initial and exit values for this variable. The control variable is increased by a specified increment value each time the statement group is executed. You can optionally supply this increment value in your statement group. If you do not specify an increment value, the control variable is increased by a value of 1 each time the statements are executed.

When the control variable value exceeds the exit variable value, execution continues with the statement following the END statement.

The iterative DO statement group uses this syntax:

```
DO control_variable = initial_value TO exit_value [BY increment_value]
   statement
   .
   .
   .
END
```

The *control_variable* is a name for an integer variable. The *initial_value* and *exit_value* are integer expressions. The *increment_value* is an integer constant or an integer variable.

The following example shows a simple iterative DO statement group.

```
do i=1 to 5
   type "Hello"
   transmit using PF8
end
```

This example types “Hello” and transmits it using the PF8 key five times.

Controlling DO statement group execution

You can use the LEAVE and ITERATE statements to execute parts of DO loops selectively. These statements are not valid for simple DO statement groups. As indicated previously, the LEAVE statement causes you to leave a DO loop. The ITERATE statement enables you to skip to the next iteration of a DO loop without completing the current iteration of the entire statement group.

You can use the LEAVE and ITERATE statements only inside repetitive DO statements; they apply only to the innermost loop of which they are a part.

In the following example, the LEAVE statement enables the program to write the string “Hello” to the operator three times.

```
a = 1
do forever
   if a = 4 then
      leave
   else
      nop
   say 'Hello'
   a = a + 1
end
```

In the next example, the numbers 0-19 and 21-99 are transmitted; the number 20 is not.

```
do a = 0 to 99
   if a = 20 then
      iterate
   else
      nop
   type char(a)
   transmit
end
```

Using conditions and relational operators

Statements that control program flow use conditions to determine what the program does next. Conditions in statement groups are evaluated as being either true or false. A condition is true if the relationship specified in the condition exists (for example, a = b is true if the value of a is the same as the value of b). A condition is false if the specified relationship does not exist.

An STL condition consists of relational expressions. A relational expression contains two expressions with a relational operator between them. Relational operators establish relationships between expressions. You can evaluate whether expressions are equal, unequal, greater than or less than, greater than or equal to, or less than or equal to one another. The relational operators enable you to compare the expressions on either side of the operator.

A condition involving a single relational expression is called a simple condition. A condition involving two or more relational expressions joined by logical operators is called a complex condition.

Note: If you use DBCS data in your conditions, see "Logic testing DBCS data" on page 272.

Simple conditions

The syntax for a simple condition is the following:

```
expression relational_operator expression
```

Note: There are some restrictions on expressions used in relational expressions. These restrictions are discussed later in this section.

An expression can be a variable, constant, function, or any combination of these joined by arithmetic operators (for integers) or string operators (for strings). Note that you must have the same type of expression (bit, integer, or string) on both sides of the relational operator.

Table 13 lists the STL relational operators and the data types each can be used with.

Table 13. STL relational operators and associated data types

Operator	Meaning	Data Types
=	equality	integer, string, bit
¬=, ><, <>	inequality	integer, string, bit
<=, =<, ¬>	less than or equal	integer, string
>=, =>, ¬<	greater than or equal	integer, string
<	less than	integer, string
>	greater than	integer, string
&=	test under mask	string

Simple bit conditions

Bit conditions test only equality and inequality because there are only two possible values for a bit variable: ON or OFF. The syntax for a bit condition is the following:

```
bit_expression1 [relational_operator bit_expression2]
```

Note: In a bit expression, *bit_expression1* cannot be a bit constant.

The following examples show bit conditions used in IF statements. These conditions can also be used in other statements that control program flow such as DO statement groups and SELECT statement groups.

```
if something_received = on then         /* A bit variable is compared */
                                        /* for equality with the      */
                                        /* bit constant ON.           */

if something_received –= on then        /* A bit variable is compared */
                                        /* for inequality with the    */
                                        /* bit constant ON.           */
```

```
if current_state = desired_state then    /* A bit variable is compared */
                                         /* for equality with          */
                                         /* another bit variable.      */
```

To simplify comparisons, STL provides a shorthand method of comparing whether a bit variable is equal to the bit constant ON. Using this method, the condition only includes *bit_expression1*.

The two IF statements shown in the following example have the same effect. One uses the normal bit condition syntax and the other uses the shorthand.

```
if something_received  = on then       /* Normal bit condition syntax. */

if something_received then             /* Shorthand syntax.            */
```

Simple integer conditions

Simple conditions involving integer expressions can use all of the STL relational operators except the test under mask operator (&=). The syntax for integer expressions is the following:

```
integer_expression relational_operator integer_expression
```

The following examples illustrate simple integer conditions used with IF statements.

```
if count = 5 then             /* An integer variable is compared for   */
                              /* equality with the integer constant 5. */

if count >= max_count + 5 then /* An integer variable is compared for   */
                              /* being greater than or equal to the    */
                              /* value of an integer expression.       */

if count + 1 = max_count then  /* An integer expression is compared for */
                              /* equality with an integer variable.    */
```

Simple string conditions

You can use all of the STL relational operators in simple conditions involving string expressions. The syntax of a simple string condition is shown below:

```
string_expression relational_operator string_expression
```

Note: If the strings are of unequal length, the shorter string will be padded logically on the right with blanks in evaluating the condition.

The following examples show how to use string conditions with IF statements.

```
if message = 'Target not found.' then  /* String variable is compared   */
                                       /* for equality with a string    */
                                       /* constant.                     */

if message = test_message then         /* String variable is compared   */
                                       /* for equality with a           */
                                       /* string variable.              */

if message ¬= msgA||msgB then          /* String variable is compared   */
                                       /* for inequality with a string  */
                                       /* expression.                   */

if substr(message,4,1) = '*' then
                      /* The fourth character of a string variable is   */
                      /* compared for equality with the character '*'.  */
```

Test under mask operation

Use the test under mask operator to compare a string variable with a mask byte. A mask byte is a 2-digit hexadecimal string constant. You may want to make this type of comparison when you want to test only certain bits in a byte (character) of string data.

The test under mask operator uses the &= symbol. The syntax for the comparison is either of the following:

```
string_variable &= mask_byte

substring_function &= mask_byte
```

The test under mask operator compares the mask byte with the leftmost byte in the string on the left side of the comparison. You can use the substring function to select a particular byte in a string. See "The SUBSTR function" on page 250 for information about the substring function.

The condition is true if each bit specified as ON in the mask is set ON in the leftmost byte on the left side of the comparison. In other words, the system performs a bitwise logical AND operation on the left-side byte and the mask byte. If the result of the AND operation is the same as the mask byte, the condition is true, unless the mask byte is X'00'.

The following example illustrates how this comparison is used.

```
a = '0F'x
if a &= '0C'x then ...  /* In this example, the condition is true. */
```

In the preceding example, the string variable "a" has a value of X'0F'. The bit configuration of "a" is B'00001111'. The mask byte is X'0C' which has a bit configuration of B'00001100'. When WSim performs the bitwise logical AND operation, the result is the following:

```
00001111 (This is variable "a," which has the value '0F'x.)
00001100 (This is the mask byte, which has the value '0C'x.)
--------
00001100 (This is the result of the bitwise logical AND operation.)
```

Note that each bit that is set ON (has a value of 1) in the mask byte is also set ON in the left-side byte (the variable "a"). Thus, the comparison is true.

The following condition is false:

```
a = '0F'x
if a &= '1C'x then ...                 /* This comparison is false. */
```

The bitwise logical AND operation for this example gives the following result:

```
00001111 (This is the variable "a," which has the value '0F'x).
00011100 (This is the mask byte, which has the value '1C'x.)
--------
00001100 (This is the result of the logical AND operation.)
```

In this case, every bit that is set ON in the mask byte is not set ON in the leftmost byte of the comparison ("a"). The comparison is false.

The following examples show additional test under mask conditions, indicating which are true and which are false.

```
a = '0F'x

if a &= '01'x then ...            /* Condition is true.             */
if a &= '09'x then ...             /* Condition is true.            */

if a &= '11'x then ...             /* Condition is false.           */

b = 'Hello'                        /* In hex, Hello is 'C885939396'x. */

if b &= 'C0'x then ...             /* The leftmost byte is 'C8'x.    */
                                   /* The condition is true.         */

if substr(b,2) &= 'C0'x then ...   /* The SUBSTR function selects    */
                                   /* '85'x as the byte to be tested. */
                                   /* The condition is false.        */
```

Complex conditions

A complex condition involves two or more simple conditions joined with one of the logical operators: AND (&) or OR (|). If a complex condition involves two simple conditions joined with AND, the condition is true only if both simple conditions are true. If a complex condition consists of two simple conditions joined with OR, the condition is true if one or both of the simple conditions are true.

You can code multiple simple conditions in a complex condition. The logical operators do not have an order of precedence like the arithmetic operators do, so complex conditions are evaluated left to right unless you use parentheses to group simple conditions.

In the following example, the condition is true if both “a” and “b” are 1 or if “c” is 1.

```
if a = 1 & b = 1 | c = 1 then ...
```

The condition in the following example is true only if “a” is 1 and either “b” or “c” is 1.

```
if a = 1 & (b = 1 | c = 1) then ...
```

Chapter 19. Generating messages for an STL program

Defining messages to be transmitted is a major part of any STL program. For simulated display terminals, you can think of this task as mimicking the actions of the terminal operator. For nondisplay terminals, you will code the type of messages they normally send.

This chapter discusses how to perform the following tasks:
- Simulate keyboard actions
- Simulate DBCS data entry
- Simulate other types of text entry
- Obtain data from outside sources
- Simulate operator decisions
- Generate message text with user tables
- Identify cursor position and display characteristics
- Log on and off terminals.

Simulating keyboard actions

When simulating a terminal operator's activity, you can simulate the following actions:
- Entering text from a keyboard
- Using keyboard keys to perform functions
- Using the cursor movement keys to position the cursor.

Simulating keyboard text entry

You can simulate an operator entering text at the keyboard by using the TYPE statement. The syntax of the TYPE statement is the following:

```
TYPE string_expression
```

The TYPE statement puts the specified information into a buffer, but it does not transmit the message or stop program execution for the terminal. WSim does not transmit the message until it executes a TRANSMIT statement. (See "Using the TRANSMIT statement" on page 285 for a discussion of the TRANSMIT statement.) Thus, you can build long messages with TYPE statements and the various device key statements before transmitting them. See "Simulating device keys" on page 268 for information about the device key statements.

When using the TYPE statement, you can use any combination of string constants and variables to make up your string expression. You can also use multiple type statements. For example, to simulate an operator entering the phrase EDIT MYDATA at a keyboard, you can use any of the four following sections of code.

```
type "EDIT MYDATA"

 or

file = "MYDATA"
type "EDIT" file

 or
```

```
action = "EDIT"
file   = "MYDATA"
type action file

 or

type "EDIT "          /* Note: Messages may be "built"   */
file = "MYDATA"       /* using multiple TYPE statements. */
type file
```

When building messages using multiple type statements, string expressions on subsequent TYPE statements are concatenated onto the previous messages without an intervening blank. This is why the string expression on the first TYPE statement in the last example includes a terminating blank. However, string expressions on the same TYPE statement (as shown in the third example) include blanks as they are coded. This is because the string variables or string constants are separated by the blank operator X'40', which inserts a blank when the strings are concatenated. See Chapter 17, "Understanding the elements of an STL program," on page 233 for more information about string concatenation operators in string expressions.

Note: If you are simulating SNA devices, WSim generates chaining headers automatically for your TYPE statements. If you need more exact control of what goes in each chain, you will need to use the SETRH statement to define the chains.

If you want to repeat a particular character in your message, you can use the REPEAT function. Its syntax is the following:

`REPEAT(`*`character`*`,`*`count`*`)`

The *character* is the character to be repeated; *count* is the number of times to repeat the character. For example, you could repeat the character A five times in your message:

```
type repeat('A',5)
```

Simulating device keys

STL provides a number of statements that simulate the operator pressing various keys. These device key statements are used primarily for simulated terminals interacting with full-screen applications.

The device key statements are listed alphabetically with other keyword statements in Chapter 25, "Reference to STL statements," on page 355. Special purpose keys are available for the following display devices:

- IBM 3270 Information Display System
- IBM 5250 Display System

The device key statements and AID identifiers that can be used for each terminal are listed in Chapter 27, "Keys valid for particular devices," on page 501. AID identifiers are discussed in "Using the TRANSMIT statement" on page 285.

There are several types of device key statements:

- Keys to change screen attributes
- Keys to edit text
- Keys to move around the screen
- Keys that perform other miscellaneous functions.

Chapter 25, "Reference to STL statements," on page 355 provides details about syntax for these statements, how they are used, and which devices they can be used with. The statements are listed here by their functions so that you will know which keys can be simulated.

Keys to Change Screen Attributes

Device Key Statement	Function
CHARSET	Select a character set for data input
COLOR	Select a color for displaying data input
HIGHLITE	Select highlight options for displaying data input.

Keys to Edit Data

Device Key Statement	Function
DELETE	Delete beginning with the character at the cursor
DUP	Duplicate
EREOF	Erase to end of field
ERIN	Erase input
FM	Field mark
INSERT	Insert
LCLEAR	Local clear

Keys to Move the Cursor Around on the Screen

Device Key Statement	Function
BTAB	Back up one input field
CTAB	Move to next input field if not at the beginning of an input field (conditional tab)
FLDADV	Move to next input field on 5250 terminal
FLDBKSP	Move to previous input field on 5250 terminal
FLDMINUS	Field minus (F-) on 5250 terminal
FLDPLUS	Field exit or Field plus (F+) on 5250 terminal
HOME	Move cursor to beginning of first input field on screen
JUMP	Move cursor to specified partition and make it active
NL	Put cursor on a new line
SCROLL	Scroll up or down in displayed data
TAB	Move to the next input field on the display screen.

Keys for Other Purposes

Device Key Statement	Function
CURSRSEL	Select the field the cursor is in
RESET	Simulate action of Reset key

SYSREQ	Simulate action of SNA SYSREQ key.

Note: CURSRSEL stops program execution for some data fields. See “CURSRSEL” on page 398 for information about limitations on using the CURSRSEL statement.

The LIGHTPEN statement performs much the same action as the CURSRSEL device key statement. It simulates selection of the cursor location field using a Selector Light Pen on a display terminal. This statement can be used for simulating 3270 and 5250 terminals. Like CURSRSEL, the LIGHTPEN statement stops program execution for some data fields. See “LIGHTPEN” on page 415 for details about syntax and usage.

Like other device key statements, the keywords listed in this section are the first element in a statement. For example, to simulate an operator pressing the HOME and EREOF keys, you could code the following statements:

```
home
ereof
```

The FM, NL, and TAB statements can also be coded as functions within a TYPE statement. Note that these functions can only be used in a TYPE statement. These functions enable you to have the operator press a key in the middle of a TYPE statement, as shown in the following example.

```
type 'This is field 1'tab()'and this is field 2.'
```

You can use the field mark (FM) and newline (NL) functions in the same way. These three functions return strings equivalent to the effect of executing the TAB, FM and NL statements.

The following statements produce the same results as the single TYPE statement shown in the previous example:

```
type 'This is field 1'
tab
type 'and this is field 2.'
```

You can use whichever method is most convenient for you.

You can simulate device keys that transmit messages to the system under test by including them on a TRANSMIT USING statement. See “Using the TRANSMIT statement” on page 285 for more information about this statement and the keys you can simulate with it.

Simulating cursor movement

Simulating movement on a panel is especially useful when your simulated terminal is interacting with a full-screen application such as Interactive System Productivity Facility (ISPF). These applications usually define certain areas of the screen as fields that will accept operator input. The operator must move the cursor to these fields using the cursor movement keys, tab keys or some other key that alters the cursor location, such as NL for a new line.

You can also position the cursor with the CURSOR statement. You can use this statement to simulate the arrow keys or to place the cursor where you want it without simulating keys.

You can use the CURSOR statement either to position the cursor at an absolute screen location or to move the cursor up, down, left, or right relative to its current location. The syntax for this statement follows:

```
CURSOR({integer_expression}[,integer_expression])
       {direction}
```

If you use two integer expressions, the first is the row and the second is the column. If you use one of the directional words ("UP", "DOWN", "LEFT", or "RIGHT") with an integer, the cursor moves the given number of characters in the specified direction. If you use an integer alone, the cursor is positioned at that offset into the screen.

The following examples position the cursor at the locations given.

```
cursor(1,1)                /* Positions cursor at row 1, column 1.    */
cursor(1000)               /* Positions cursor at screen offset 1000. */
row_number = 12
column_number = 40
cursor(row_number,column_number + 5)    /* Positions cursor at row 12, */
                                        /* column 45.                  */
```

The following examples move the cursor relative to its current screen location.

```
cursor("up")                        /* Moves cursor up one row.      */
cursor("down",5)                    /* Moves cursor down 5 rows.     */
count = 20
cursor("right",count*3)             /* Moves cursor right 60 columns. */
```

The following statements show the different ways that device key statements can be coded to move the cursor. For example, to place the cursor at the beginning of the third input field on the current screen, you could code:

```
home
tab
tab
```

Simulating the SNA attention key

You can simulate the SNA attention key by using the following statement:

```
snacmnd(signal,'00010000'x)
```

This statement causes a transmit interrupt. See “Interrupting program execution” on page 286 for information about how and when terminals transmit messages.

Simulating DBCS terminals

WSim supports simulation of 3270 DBCS terminals that send and receive messages with DBCS data. STL supports entry of DBCS data by either entering the DBCS data directly from a 3270 DBCS display into the STL program or using the built-in DBCS functions such as SB2MDBCS(*string*) to create DBCS data at STL program execution time.

When you enter DBCS data into the program from a 3270 DBCS display, the DBCS data is identified by wrapping Shift-Out (SO) (X'0E') and Shift-In (SI) (X'0F') characters around the DBCS data. DBCS data entered in this manner is called ***literal text DBCS data***. You can enter literal text DBCS data as string data and within comments. DBCS data identified using SO and SI characters is also referred to as a “DBCS subfield” or DBCS data in a mixed string.

In the following examples and discussion, the SO character is represented using a "<", the SI character is represented using a ">", and the first byte of each DBCS character, which is referred to as the ward byte, is represented using a "." character.

The following examples enter DBCS data into the simulated screen. The SB2MDBCS(*string*) function creates a mixed string containing a DBCS subfield with ward 42 (EBCDIC) DBCS data from the input string.

```
type '<.A.B.C>'       /* DBCS data entered using 3270 DBCS display */
type sb2mdbcs('ABC')  /* Ward 42 (EBCDIC) DBCS data via function   */
```

You can code literal text DBCS data and single-byte character set (SBCS) data within the same mixed string, as shown here.

```
type '<.D.B.C.S> and SBCS'           /* Mixed string from 3270      */
type sb2mdbcs('DBCS') || ' and SBCS' /* Mixed string using function */
```

WSim uses the SO and SI characters to identify DBCS data. The only exception to this rule is for a 3270 DBCS field when any string data is accepted and assumed to be DBCS data.

Note: Functions that are not specifically for DBCS data perform at the byte level and not at the character level.

Logic testing DBCS data

DBCS data is identified on a simulated 3270 screen by being in a DBCS field, having SO and SI characters wrapped around the DBCS data, or having the character attributes indicate DBCS data. The DBCS data in a message sent or received may or may not have SO and SI characters included.

Because of the above situation, you must carefully think out the logic testing of DBCS data. For example, if the DBCS data ".A.B.C" is located in a DBCS field or identified by character attributes as DBCS data, you need to code the logic test as follows to obtain a true result.

```
if index(screen,dbcsdel('<.A.B.C>')) > 0 then say 'True'
```

Note: The DBCSDEL(*string*) STL function deletes the SO and SI characters from the literal text DBCS data.

If the DBCS data on the screen contains SO and SI characters such as "<.A.B.C>", the logic test does not require the use of the DBCSDEL(*string*) function to delete the SO and SI characters. The following logic test obtains a true result in this case.

```
if index(screen,'<.A.B.C>') > 0 then say 'True'
```

Note: SI/SO pairs of characters that result from DBCS data ending on one record and continuing again on another record, however, are removed from the resulting string.

Simulating other types of text entry

You can use the STRIPE statement to generate text to simulate message data to be transmitted to the system under test by a magnetic stripe reader. This statement can be used only when you are simulating 3270. You must provide the data to be transmitted. The syntax of this statement is the following:

```
STRIPE stripe_data
```

The *stripe_data* must be a string expression. The following example shows how this statement is used.

```
stripe '123456789'                          /* Send code to the host. */
```

Unlike the TYPE statement, which only puts text in a buffer, the STRIPE statement puts the text in a buffer and transmits it. The simulated terminal stops moving through the STL program until the next time it is ready to generate a message. See "Interrupting program execution" on page 286 for information about how and when terminals transmit messages.

Obtaining data

You can use string functions to generate data to include in your messages or to evaluate incoming messages. The data provided can be used to simulate data from a display terminal, to compare against incoming data, or to provide data to the operator or the log data set.

Using random numbers

You may want to use random numbers as part of your message text. This can be useful when generating identification numbers, order numbers, or other information entered by the operator. STL provides two functions that generate a random number:

- RANDOM()
- RNUM().

The RANDOM function returns an integer random number. The syntax for this function is the following:

```
RANDOM({'RN',rn_number}|{low,high})
```

Inputs to this function can be either a range specified by the RN statement on the network definition or a *low* and *high* value specifying the range in which the random number must lie. The *rn_number* is the label for the RN statement defining the range. See Part 1, "WSim language statements," on page 1 for more information about the RN statement. If you specify a low and high value, the low value can be an integer expression from 0 to 2147483646 and the high value from 1 to 2147483647. The low value must be less than the high value.

The RNUM function returns an EBCDIC string representing a random number. The syntax for this function is the following:

```
RNUM({'RN',rn_number[,length]}|{low,high[,length]})
```

The inputs to this function are the same as for the RANDOM function except that you can specify a *length* for the returned string. The length can be from 1 to 10. If you do not specify a value, 10 is used. If the random number is shorter than the length, it is padded with leading zeros.

You could assign a random number from 0 to 300 to a string variable like this:

```
random_number = rnum(0,300)
```

Obtaining the date and time

Frequently, terminal input includes the current date and time, whether the operator types it or whether the system provides it. Additionally, you may want to use the date and time as elements of conditions in your programs. STL provides functions that supply date and time information. Only the DATE and TOD functions use arguments. The following list displays the date and time functions and the data they return.

Function	Returns
CMONTH()	Name of the current month in mixed case (January, February, ... , December)
DATE()	Current date. **Note:** There are numerous formats for displaying the date. See "DATE" on page 463 for more information.
DAY()	Number of the current day of month as a 2-character string (01-31)
MONTH()	Number of the current month as a 2-character string (01-12, where 01=January and 12=December)
TOD()	Current time in the form HHMMSSTH (hours, minutes, seconds, tenths, hundredths)
YEAR()	Last two numbers of the current year as a 2-character string.

See Chapter 25, "Reference to STL statements," on page 355 for more information about these functions.

Using device IDs

STL supplies a number of functions that return IDs for the network, terminals, lines, devices, physical units, and logical units that you are simulating. These functions are listed in "Identifying network resources" on page 309. They are discussed in detail in Chapter 26, "Reference to STL functions," on page 449.

Use these functions whenever you want to use a terminal or device name in a message you send to the terminal operator, the log data set, or as a message to your system under test. You can also use these functions when setting up conditions to test messages sent and received.

For example, you might want to use the device name when logging on to your system. You could do this with the following statement:

```
type 'Logon' id()
```

The ID function returns the name of the device that is executing the STL program.

Simulating operator decisions

The data entered by a terminal operator is often influenced by the contents of the current panel or by other factors such as the name of the current application or the last message received. A real operator makes decisions and enters data accordingly. As an STL programmer, you can simulate operator decisions by controlling the logical flow of your program. See Chapter 18, "Controlling STL program flow," on page 255 for information about controlling program flow in STL.

For example, if you want your STL program to generate and transmit the message "Hello there" if the value of the integer variable "count" is 1 and to generate the message "Goodbye there" if the value is anything else, you can use the following code:

```
if count = 1 then
  type 'Hello'
else
  type 'Goodbye'
type ' there'
transmit
```

Using user tables

WSim enables you to set up user tables so that you can use predefined data either in a random or a defined sequence as a part of your messages. A user table is a list of string constant entries in a table. The table is a one-dimensional array.

You can use entries in user tables as data for your messages or as data against which incoming information can be compared. For example, you could include a list of order quantities in one table and a list of colors in another. You could combine information from these tables to define various orders in your messages. Or, you might want use a table to provide logons and passwords for your simulated terminals and its operators.

WSim provides statements enabling you to define user tables. In addition, STL functions provide the ability to select table entries and to compare entries with a given source string. The following sections discuss these capabilities.

Defining user tables

In STL, you define user tables with the MSGUTBL statement group. See "Using declarative statements" on page 242 for information about declaring user tables in your STL program.

Selecting entries in a user table

After you use the MSGUTBL statement group in your STL program to define a table, you can select entries from the table in several ways.

Each entry in a user table has an index, or number, that WSim uses to identify the entry. The first entry in the table has an index of 0. The UTBL function enables you to select table entries in the following ways:

- By index number
- Randomly
- Randomly using a defined distribution.

The syntax of this function is the following:

```
UTBL(utbl_name,{index_number})
               {'R'}
               {'Rn'}
```

The *utbl_name* is either the name of an MSGUTBL declarative statement or the number of a UTBL defined in the network definition. WSim will look in this table for the entries. The second argument describes how the entry is to be selected. The *index_number* option enables you to enter the number of the entry to be used. The 'R' option means that the entry should be selected randomly. The 'R*n*' option means that you enter a number *n* that refers to the label of a UDIST statement on your network definition. The UDIST statement defines a probability distribution to use in choosing entries from a user table. See , SC31-8945 for information about this statement.

The following example shows how the various selection methods are used. Remember that the first entry in the table has the index number 0.

```
mynames:  msgutbl
   'Mary'
   'Joe'
   'John'
   'Sue'
```

```
endutbl
   .
   .
   .
a = 1
b = 2
name = utbl(mynames,0)         /* Mary is returned.                   */
name = utbl(mynames,a+b)       /* Sue is returned.                    */
name = utbl(mynames,'R')       /* A random entry is returned.         */
name = utbl(mynames,'R3')      /* A random entry with the distribution */
                               /* defined by UDIST number 3 in the     */
                               /* network definition is returned.      */
   .
   .
   .
```

The UTBLMAX function enables you to loop through an entire user table sequentially. It returns the index (the number) of the last entry in your user table. Its syntax is the following:

```
UTBLMAX(user_table)
```

For *user_table*, enter the name of the table in which WSim will look for entries. The following example shows how you can use this function:

```
do i = 0 to utblmax(mynames)    /* Loop through all entries of         */
                                /* the user table named "mynames".     */
   type utbl(mynames,i)         /* During each loop, simulate the      */
                                /* operator typing the next            */
                                /* entry in the table named "mynames". */
end
```

Comparing entries

Use the UTBLSCAN function to compare each entry of a user table with a source string. This function enables you to determine if a message you receive matches a table entry.

The UTBLSCAN function returns a bit value of ON if an entry matches the source string and OFF if one does not. Optionally, it sets an integer variable to the index number of the user table entry that matches the target string.

The syntax of the UTBLSCAN function is the following:

```
UTBLSCAN(source,utbl_name[,integer_variable])
```

In this function, *source* is the string expression for which the specified UTBL or MSGUTBL is to be searched. The *utbl_name* is either the number of a UTBL defined in the network definition or the name of an MSGUTBL declarative statement. The *integer_variable* is an optional argument indicating the name of an integer variable to which the matching UTBL entry's index number is assigned if a table entry match is found. If no match is found, the value of *integer_variable* remains unchanged. When using this number, remember that user tables begin with entry number 0.

This function is usually used in conditions. The following example uses the table defined previously.

```
if utblscan('Mary',mynames) = on then                /* Condition met. */
```

Identifying cursor position and display characteristics

When you are generating data to send, you may need to know the current cursor position. You may also want to know the size of your display screen. Additionally, since an operator or an application can change screen attributes on a 3270 display terminal, you may want to be able to check the attributes of a particular location. For example, you could use the cursor position or screen attribute as a point for an operator decision in your program. If the cursor is not at the top of the screen, you could have the operator press the Home key. If a particular field is green, you could have the operator change it to red.

Three functions enable you to identify the current cursor position: CCOL, CROW, and COFF. These functions return the current cursor column, row, and offset, respectively, for display devices. Results of these functions for nondisplay devices are unpredictable. These functions do not use arguments.

The CCOL and CROW functions return the row or column relative to the entire screen size for the simulated device beginning with row 1 and column 1. The COFF function returns the cursor offset. The cursor offset is the cursor location relative to the beginning of the screen or the currently active partition for devices that support partitioning. The first position of a screen or partition is offset 0.

The CPOS function enables you to determine if the current cursor row and column matches those specified as function arguments. It returns a bit value: ON if the column and row specified match the current cursor position, OFF if they do not. Its syntax is the following:

```
CPOS(row_number,column_number)
```

In this function, *row_number* is the number of the row to be matched and *column_number* is the number of the column. Both must be integer expressions. If an integer constant is used, it must be a number from 1 to 255.

You may want to use the number of columns or the number of rows on your display terminal in your program. The NUMCOLS function returns the number of columns available on the display terminal you are simulating. The NUMROWS function returns the number of rows available on the display terminal you are simulating. These functions return unpredictable results for nondisplay terminals.

The ROWCOL function enables you to determine the screen offset of the specified row and column. Its syntax is shown below:

```
ROWCOL(row,column)
```

In this function, *row* and *column* reference the position on the screen for which you want to determine the offset. This function can only be used when coded as the offset for the SUBSTR or ATTR3270 functions.

Your application or your terminal operator may be able to change screen attributes. To determine screen attributes for a particular screen location on a 3270 display terminal, you can use the ATTR3270 function. This function provides information about the standard field, extended field, and character attribute values associated with a screen location. These attributes provide information about screen format and field and character highlighting and color, among others.

The syntax for this function is the following:

```
ATTR3270(screen_location[,length])
```

The *screen_location* identifies the point on the screen for which the attributes are to be returned. The first screen position is offset 0. Optionally, you can specify a *length* for the attribute string to be returned. This length determines how many characters and thus how much information is returned by the function. The length can be an integer from 1 to 11. If you do not include a length, nine characters are returned.

See "ATTR3270" on page 449 for more information about the values returned by this function.

Logging on and off an application

Logging on and off your system is a unique task. If you are using an SNA network, special STL commands exist that enable you to perform these tasks efficiently. These commands are discussed in the following sections. Also provided are general logon instructions for other types of terminals. See , SC31-8945 for specific information about how to log on and off other terminal types, sample scripts, coding requirements for specific terminal types, and information about network definition requirements.

As with real terminals, a terminal that WSim simulates must first log on to the system (or application) under test before it can generate meaningful messages. Your application and the type of network you are using determine the logon code that you must use.

Typically, your logon code is placed in a procedure by itself. This procedure can be identified in the FRSTTXT operand on various device statements in your network definition. See Chapter 23, "Combining STL programs and network definitions," on page 325 for more information on integrating network definitions and message generation decks.

If you are simulating LUs in an SNA network, you may need to use the special SNA logon and logoff statements provided by STL. If you are simulating other types of terminals, you will have to provide the necessary logon and logoff commands.

Logging on SNA LUs

Logging on an LU on an SNA network may require a special logon sequence, depending on the values that have been established for the LU.

In SNA, an LU can initiate a session with another LU. Normally, each LU initiates a session by sending an INITIATE SELF request unit (RU) to the host. An LU that sends the INITIATE SELF RU is called an initiating LU (ILU). An ILU can be a primary or a secondary LU.

Use the following operands in your network definition to define your LU sessions. These operands should be coded on the device statement in your network definition.

Operand	**Function**
MAXSESS	Specifies the number of sessions and whether a simulated LU is to be secondary or primary.
INIT	Specifies whether the primary or secondary LU is to initiate sessions.
RESOURCE	Defines the name of the simulated LU's partner.

For a further explanation of these operands, see , SC31-8945.

If the ILU is a primary LU, it eventually receives information from its host that it converts into a BIND RU and sends to its secondary LU partner. If the ILU is a secondary LU, it must wait after sending the INITIATE SELF RU until it receives a BIND from its primary LU partner.

If your simulated LU is the initiating LU and the RESOURCE operand was coded on the DEV or LU statement, you do not need to include session establishment code in your STL program; your STL program will begin executing **after** the BIND RU has been successfully sent (and responded to), whether your simulated LU is primary or secondary. If, however, your simulated LU is to be the initiating LU and the RESOURCE operand was **not** coded on the DEV or LU statement, you **must** include session establishment code at or near the beginning of your STL program.

When the RESOURCE operand is coded, execution of the STL program may begin before the LU receives the BIND. To handle this situation, the program should wait until the simulated resource has completed the logon. You can make the program wait by creating a loop in which you wait until the BIND is received. See "The sample STL program and network definition" on page 346 for an example of this.

If the RESOURCE operand is not coded, use the INITSELF, SNACMND, or TYPE statement to establish an LU session for an ILU. The INITSELF statement sends an INITIATE SELF format 0 RU to the simulated LU's host. When using the INITSELF statement, you must specify the name of the desired partner LU and can optionally specify the name of a MODE table entry and user data, for example, a password. The syntax of the INITSELF statement is the following:

```
INITSELF(resource[,[mode][,[user_data][,log_byte]]])
```

resource is the name of the LU with which you want to establish a session. If the resource is a string constant expression, it must be from 1 to 8 characters long. The *mode* is an entry from a logon mode table. Your simulation determines whether you need to include this argument. You may also need to include *user_data* such as a user password. The *log_byte* specifies a byte of user data to be associated with all data transmitted and received. The *log_byte* remains active until data is "typed" on a TYPE statement and transmitted on a TRANSMIT statement or until an INITSELF, TERMSELF, or SNACMND statement is issued. This byte gives users of the Response Time Utility a way to identify transactions when gathering statistics by the various "log_byte" categories.

The SNACMND statement sends an SNA command to the simulated LU's host. The function of the INITSELF statement is a subset of this statement's function. See "Using the SNACMND statement" on page 284 for more information about the SNACMND statement.

For more information about the TYPE statement, refer to "Simulating keyboard actions" on page 267. The TYPE statement can only be used to establish a session if you are not running a VTAMAPPL configuration and you are running unformatted system services.

Suppose, for example, you want your simulated terminal to log on to the application MYAPPL. You could code the following STL statement to accomplish this:

```
initself('MYAPPL')
```

In some cases, you may need to include the name of a logon mode table entry and to set a log byte for the Response Time Utility. The following example will log your simulated terminal on to MYAPPL using the logon mode table entry MYMODE and set the log byte to X'AB'.

```
initself('MYAPPL','MYMODE',,'AB'x)
```

In still other cases, you may need user data (such as a password) with the INITIATE SELF RU. The following examples show how you can specify user data on the INITSELF statement.

```
initself('MYAPPL','MYMODE','MYPASSWD') /* Includes mode entry and user */
                                       /* data.                        */

initself('MYAPPL',,'MYPASSWD')         /* Includes user data only.     */
```

The INITSELF statement interrupts the execution of your STL program. The terminal resumes program execution after the session has been established with the partner LU.

The following example shows how the INITSELF statement can be coded for a secondary initiating LU. Note that when execution resumes after the INITSELF statement, the STL program must check to see if an initial message (a WELCOME message in the example) has been received. If not, the LU should wait until the initial message is received, as shown in the following example:

```
bit logged_on
secinit:  msgtxt
/* Logon procedure for secondary LU, INIT=SEC, RESOURCE not coded.     */

check: onin index(screen,'WELCOME') > 0 then logged_on = on

logged_on = off
initself('MYAPPL','MYMODE','MYPASSWD')

/* Now wait for the WELCOME message, if it has not already             */
/* been received.                                                      */

do while logged_on = off             /* Wait until the WELCOME        */
  wait until onin                    /* message is received.          */
  end
deact check
```

The following example shows how the INITSELF statement might be used within an STL program for a primary initiating LU.

```
priinit:  msgtxt
/* Logon procedure for primary LU, INIT=PRI, RESOURCE not coded.       */

initself('MYAPPL')

/* The following code will be executed after the LU-LU session is      */
/* established.                                                        */

type 'WELCOME.  Please enter your name.'
transmit and wait until ...
```

Table 14 on page 281 summarizes when to use the INITSELF and SNACMND statements and describes how SNA session establishment affects STL program execution.

Table 14. Establishing sessions—SNA LUs

LU Session Type	**INIT= Operand Value**	**RESOURCE= Operand Coded?**	**STL Statement**
Primary MAXSESS=(x,y) where $x > 0$	PRI	YES	None. (Logon is handled by WSim. STL execution begins after session is established.)
		NO	INITSELF or SNACMND (Interrupts STL execution. STL execution resumes after session is established.)
	SEC	YES or NO	None. (LU does not initiate session. STL execution begins after session is established.)
Secondary MAXSESS=(x,y) where $y > 0$	PRI	YES or NO	None. (LU does not initiate session. STL execution begins after session is established.)
	SEC	YES	None. (Logon is handled by WSim. STL execution begins after session is established.)
		NO	INITSELF or SNACMND (Interrupts STL execution. STL execution resumes after session is established.)

Logging off SNA LUs

STL also provides a special statement, TERMSELF, to terminate a session for an SNA terminal or an LU. TERMSELF ends a session in an orderly fashion and simulates the normal course of events for an SNA network or LU. The syntax of this statement is the following:

```
TERMSELF([resource][,log_byte])
```

The *resource* specifies the name of the LU's session partner. If it is not included, WSim uses the name of the LU's current partner. The *log_byte* specifies a byte of user data to be associated with all data transmitted and received. The *log_byte* remains active until data is “typed” on a TYPE statement and transmitted on a TRANSMIT statement or until an INITSELF, TERMSELF, or SNACMND statement is issued. This byte gives users of the Response Time Utility a way to identify transactions when gathering statistics by the various “log_byte” categories.

The TERMSELF statement interrupts execution of the STL program and stops program execution temporarily for the terminal currently executing the program while WSim transmits the TERMINATE SELF format 0 RU. Execution of your STL program resumes with the next STL statement. If you want your terminal to participate in another SNA session at that point, you can include session establishment code. Otherwise, you can put your terminal in a quiesce state (a state where it can receive but not respond to messages) by using the QUIESCE statement.

The SNACMND statement can also be used to terminate a session for an SNA terminal or an LU. See “Using the SNACMND statement” on page 284 for more information about the SNACMND statement.

Logging on a TCP/IP Telnet 3270 terminal

The following example shows how to log on to a simulated Telnet 3270 terminal. Notice that the INITSELF statement is not used, since INITIATE SELF is used by SNA terminals only. This example assumes the application being logged on to will send an "ENTER LOGON" message when it is ready for the terminal to log on.

```
wait until onin index(screen,'ENTER LOGON') > 0 /* Wait for logon message. */

/* The above wait condition ends when the logon message has been received. */

ereof                                       /* Clear input field.      */
type 'LOGON 'id()                           /* Type the logon command. */
transmit using enter,                       /* Transmit the logon      */
  and wait until onin index(screen,'READY') > 0 /* and wait for TSO.       */
```

The TRANSMIT keyword is described in "Using the TRANSMIT statement" on page 285 and the ONIN keyword is described in "Using the ONIN and ONOUT statements" on page 292.

Note: The text of logon messages will vary from installation to installation. When following the above example, be sure the strings specified on the INDEX functions are correct for your installation.

For information about how to log on other terminal types and examples, see , SC31-8945.

Logging off a TCP/IP Telnet 3270 terminal

You can log off your simulated Telnet 3270 terminal by simulating the activity of a real terminal operator terminating communication with an application. Since each application may differ in the way this is done, your STL code for logging off will vary from application to application.

The following examples show two possible logoff sequences. The first shows the STL code for logging off an application that requires the terminal operator to type "LOGOFF". The second shows the code for an application that requires the operator to press the PF3 key to log off.

```
/* Example 1 */

/***********************************/
/* Log off by sending the message */
/* LOGOFF to the application.     */
/***********************************/
type 'LOGOFF'
transmit and wait until onin index(data,'TERMINATED') > 0

/***********************************/
/* Put the terminal to sleep.     */
/***********************************/
quiesce

/* Example 2 */

/**********************************/
/* Log off by pressing PF3.      */
/**********************************/
transmit using PF3 and wait until onin index(data,'TERMINATED') > 0

/********************************/
/* Put the terminal to sleep.   */
/********************************/
quiesce
```

Notes:

- The procedure for logging off and the text of logoff messages varies from application to application. When following the preceding examples, be sure that the strings specified on the TYPE statement and the INDEX function and that the AID key specified on the TRANSMIT USING statement are correct for your application.
- Telnet 3270 and 3270E devices automatically reconnect 30 seconds after logoff, unless quiesced. If it reconnects, the Telnet device comes up as if it were just starting. Refer to , SC31-8945 for more detail.

Generating SNA terminal messages

To modify SNA messages transmitted by your simulated devices, you can use the SETTH statement to alter the SNA transmission header and the SETRH statement to control the chaining of messages and to alter the SNA request/response header. To send a specific SNA message to the system under test, use the SNACMND statement.

Using the SETTH statement

Use the SETTH statement to modify the SNA transmission header for a message built by a TYPE statement. Refer to “SETTH” on page 432 for more information.

Using the SETRH statement

Use the SETRH statement to modify the SNA request or response header for a message built by a TYPE statement. For example, you can use the SETRH statement to control the chaining of messages, as is shown in the following example.

```
chains: msgtxt
type 'first data'
setrh chain 'first' on(exc)
transmit
type 'more data'
transmit
type 'even more data'
transmit
type 'last data'
setrh chain 'last' off(exc)
transmit
type 'only data'
transmit
endtxt
```

When the STL program executes, the first message sent is a first-in-chain RU. The next two RUs transmitted for the logical unit are marked as middle-in-chain. If you omit the SETRH statement and a first-in-chain RU has been sent for a logical unit, WSim automatically sends subsequent RUs as middle-in-chain. The next message is transmitted as a last-in-chain RU that requests a definite response from the test system. WSim does not generate another message for the logical unit until the response is received. Setting EXC on indicates that this is an exception response. The last message generated from this procedure is sent as an only-in-chain RU.

When CHAINING=AUTO is specified in the LU definition for a non-display SNA device, WSim automatically performs the chaining operation on the message text data using the maximum RU size value from the BIND. Refer to Part 1, “WSim language statements," on page 1 for more information on these network definition statements. WSim can use BUFSIZE operand values up to 32767 when building messages.

For more information about this command, see "SETRH" on page 430.

Using the SNACMND statement

Use the SNACMND statement to specify an SNA command to be sent by a logical unit to the system under test. WSim builds default SNA headers for the command, but you can modify the headers with the SETTH and SETRH statements. (Note that the SETRH statement only works with the LUSTAT SNACMND statement.) The following example demonstrates how to use the SNACMND statement to build a SIGNAL RU.

```
snacmnd(signal,'00010000'x)   /* SNA attention key (SIGNAL). */
```

The SNACMND statement can be used to generate asynchronous SNA data flows. When the SNACMND statement is the next statement in an STL program, the SNACMND statement is executed, even if the normal requirements for executing the program have not been met.

See "SNACMND" on page 434 for more detailed information on the SNACMND statement.

Chapter 20. Transmitting and receiving messages from an STL program

WSim enables you to transmit messages from simulated terminals, receive messages, and take actions based upon messages sent or received.

This chapter discusses the following topics:
- Transmitting messages
- Controlling intermessage delays
- Receiving messages
- Testing asynchronous conditions
- Posting and signaling events.

Transmitting messages

Once a message has been generated, WSim must transmit the message to the system under test. When WSim transmits a message, program execution for a particular terminal is interrupted to simulate the time required before an operator takes another action.

The TRANSMIT statement is the simplest method of sending messages. However, you will need to use other statements to send messages and interrupt program execution in specific circumstances. The following sections discuss the use of the TRANSMIT statement and the other statements you can use to interrupt program execution.

Note: Do not use the TRANSMIT statement in CPI-C transaction program simulations. This statement causes program execution to be interrupted for the CPI-C transaction program. CPI-C transaction programs send messages by using the CMSEND statement.

Using the TRANSMIT statement

Normally, a message is transmitted from a real 3270 terminal to a host system whenever the terminal operator presses an attention identifier (AID) key. The TRANSMIT statement simulates the operator pressing an AID key and instructs WSim to transmit all previously entered message data, that is, data entered on a TYPE statement.

The syntax of the TRANSMIT statement is the following:

```
TRANSMIT [USING aid_key] [LOGGING log_byte]
[AND wait_statement]
```

The optional parts of the statement are called clauses. You can use these clauses to specify how the message will be transmitted and whether the terminal should wait after sending a message.

The valid AID keys appear in the following list.

CLEAR	HELP	ROLLUP
CLEARPTN	PA1-PA3	SEND
CMD1-CMD24	PF1-PF24	SENDLINE
CURSRSEL	PRINT	SENDMSG
ENTER	ROLLDOWN	SYSREQ

If you do not enter the clause that begins with USING to define the AID key, STL assumes the default key ENTER. The AID keys that can be used for specific IBM display devices are listed in Chapter 27, “Keys valid for particular devices,” on page 501. The LOGGING *log_byte* clause specifies a byte of user data to be associated with all data transmitted and received. The *log_byte* remains active until data is “typed” on a TYPE statement and transmitted on a TRANSMIT statement or until an INITSELF, TERMSELF, or SNACMND statement is issued. This byte gives users of the Response Time Utility a way to identify transactions when gathering statistics by the various “log_byte” categories.

The *wait_statement* clause is described in “Using the WAIT UNTIL and QUIESCE UNTIL statements” on page 298. As an example of how the TRANSMIT statement is used, consider the following STL code.

```
type 'hello'
tab
type "My name is" myname
transmit using enter
```

This set of statements enters data into the current screen field, tabs to the next screen field, enters data into that field, and instructs WSim to transmit the generated data to the host application using the ENTER AID.

Note: ENTER is the default AID for the TRANSMIT statement. Thus, you do not have to include it. The last line of the above program segment could have been simply coded as TRANSMIT.

You do not have to generate message data using the TYPE statement before a TRANSMIT statement. Often a real terminal operator presses an AID key without entering any text, for example, when scrolling through a file. You could include these statements in your code for a terminal that uses PF8 as its scroll key:

```
transmit using PF8
transmit using PF8
transmit using PF8
```

However, unlike the device key statements, the AID keys can only be coded on a TRANSMIT statement. For example, you cannot use the following code for AID keys:

```
PF8   /* This usage is not allowed. */
```

Interrupting program execution

In STL, each terminal in a simulated network continues through the STL programs specified for it in the network definition until it encounters a statement that “stops” program execution. Generally these statements are statements that transmit a message to the host system. When the terminal “stops” program execution, WSim interrupts execution of the STL program and sends accumulated message data. This interruption is called a Transmit Interrupt.

The length of the Transmit Interrupt is discussed in “Controlling intermessage delays” on page 288. The terminal starts program execution again after the interrupt expires. During the first terminal's Transmit Interrupt, another terminal may begin program execution and execute statements in the STL programs designated on the PATH statement for that terminal in the network definition.

The following statements cause a Transmit Interrupt.
- INITSELF
- SNACMND
- STRIPE
- TERMSELF
- TRANSMIT.

The following statements cause a Transmit Interrupt and stop program execution for a terminal until the condition associated with the statement is satisfied or the terminal is reset.
- QUIESCE
- SUSPEND
- WAIT.

The following statements cause a Transmit Interrupt when they are used in certain fields.
- CURSRSEL
- LIGHTPEN.

The fields that cause these statements to transmit information are determined by the application's trigger field. When using these fields, data is sent if the cursor is moved out of the trigger field after the data is entered. See “CURSRSEL” on page 398 for more information about the CURSRSEL statement and “LIGHTPEN” on page 415 for more information about the LIGHTPEN statement. Since these statements do not always cause a Transmit Interrupt, use them carefully while sending data.

The following CPI-C statements may cause a Transmit Interrupt:
- CMALLC
- CMCFM
- CMCFMD
- CMDEAL
- CMFLUS
- CMPTR
- CMRCV

  Note: This verb will cause an interrupt only if the receive type is “receive-and-wait” and no receive data is currently queued. When this verb causes an interrupt, program execution will stop until either data or status is received from the partner transaction program.
- CMRTS
- CMSEND
- CMSERR.

A transmit interrupt will only occur if the CPI-C verb results in a request being issued to VTAM. If the verb fails as a result of a local error that is detected prior to issuing a request to VTAM, a transmit interrupt will not occur. The types of errors that are typically detected as local errors are parameter checks and state checks.

Controlling intermessage delays

The TRANSMIT statement interrupts the execution of an STL program and stops program execution for the terminal. The amount of time that each terminal must wait following a Transmit Interrupt before it begins program execution again is called the intermessage delay. When conditions are right for the terminal to send another message and after the intermessage delay, your STL program continues executing at the statement following the TRANSMIT statement.

Note: Do not use the TRANSMIT statement in CPI-C transaction program simulations. This statement causes program execution to be interrupted for the CPI-C transaction program. Intermessage delays apply to the CPI-C verbs that cause transmit interrupts (as listed above).

The intermessage delay simulates the activity of a real operator, who would normally require time to view the screen, think about the information, and enter more data. The Transmit Interrupt and the intermessage delay give other simulated terminals the opportunity to execute their STL programs and provides time for WSim to transmit and receive messages, test asynchronous conditions, and execute asynchronous subset statements. The length of the delay is controlled by a number of factors. These factors are discussed in later sections.

The following conditions must be met before a terminal can continue executing an STL program after a Transmit Interrupt:

- Any outstanding wait condition must be satisfied.
- Any outstanding quiesce condition must be satisfied.
- The intermessage delay for the terminal must have expired.
- The INPUT INHIBITED indicator for the terminal must not be set.
- The terminal must not be in the console recovery state, or if it is in the console recovery state, the terminal operator must enter a message for the terminal from the console.
- An SNA logical unit must be in the correct SNA state for sending data (for example, the Change Direction indicator must be in its favor).

The length of the intermessage delay depends on these factors:

- Values for network definition operands:
  - DELAY
  - UTI or IUTI
  - THKTIME
  - EMTRATE.
- STL statements:
  - DELAY
  - WAIT UNTIL
  - QUIESCE UNTIL
  - SUSPEND
  - UTI.
- Load on the WSim system.

You can control some of these factors in your STL program. Others are controlled in your network definition or in the system.

Using the network definition to control delays

The following operands in your network definition control the length of the intermessage delay:

- DELAY
- UTI or IUTI
- THKTIME
- EMTRATE.

The DELAY, UTI, IUTI, and THKTIME operands can be coded on the DEV and LU statements in your network definition. They can also be coded on higher-level statements and will take effect for all terminals controlled by these definitions.

Note: Since the IUTI operand enables you to set a different UTI value for different terminals, you will not use it on an NTWRK statement.

The EMTRATE operand is coded only on the NTWRK statement in your network definition.

The following list describes the function of these operands.

DELAY
This operand, together with the user time interval (UTI or IUTI operand), specifies the length of the intermessage delay. This value is an integer. It can be a fixed value, a random value, or a value from a table of delays. It is multiplied by the UTI to provide the number of seconds for the delay.

UTI or IUTI
These operands are user-specified values that specify the number of hundredths of seconds by which the DELAY value is multiplied. A different UTI can be specified for each terminal using the IUTI operand. As an example, if the UTI for a terminal is 100 and a DELAY value of 5 is specified for the terminal, the intermessage delay for the device is 5 seconds (500 hundredths of a second).

THKTIME
This operand defines when the intermessage delay is started for a terminal. If you code THKTIME=IMMED on your network definition, the intermessage delay starts immediately when the current message is transmitted. If you code THKTIME=UNLOCK, the delay starts when the keyboard is unlocked by the host system.

Notes:

- If you are simulating primary logical units, it is recommended that you code THKTIME=IMMED.
- For CPI-C transaction program simulations, the THKTIME operand is not applicable. The intermessage delay starts immediately when the CPI-C request is transmitted to VTAM.

EMTRATE
This operand enables you to specify a message transfer rate for the network. WSim automatically adjusts the UTI for the network to maintain the desired message transfer rate.

Note: You should not use multiple UTIs if you are using the EMTRATE operand to control message transfer rates for your network. If you do use multiple UTIs with the EMTRATE operand, WSim automatically adjusts all of the UTI values. This adjustment may produce unexpected results if the operator changes the IUTI for a terminal.

See , SC31-8945 for more complete descriptions of these operands and for further information about their effect on message transfer rates.

Using STL statements to control delays

Intermessage delays are affected by the following STL statements:

- DELAY
- WAIT UNTIL
- QUIESCE UNTIL
- SUSPEND
- UTI.

These statements perform the following functions:

DELAY
This statement specifies the intermessage delay for the next message the simulated terminal transmits. The value specified overrides the default delay only for the next message transmitted by the terminal. WSim multiplies the value specified on this statement by the UTI value for the terminal to obtain the duration of the delay in hundredths of seconds. You may optionally specify a UTI value on this statement. The value for the DELAY statement can be a fixed value, a random value, or a value chosen from a rate table.

WAIT UNTIL
This statement can be used by itself or as a clause on the TRANSMIT statement. It allows you to specify a condition following the UNTIL that must be met before the terminal can begin program execution again. The intermessage delay for a terminal begins when the WAIT UNTIL condition is satisfied. Use of this statement is described in "Using the WAIT UNTIL and QUIESCE UNTIL statements" on page 298.

QUIESCE UNTIL
This statement enables you to quiesce a terminal, that is, put it to sleep. The terminal is still logged on and can still receive messages, but it cannot take any action until either the condition specified following the UNTIL is met or the operator issues a command to release the terminal.

SUSPEND
This statement enables you to suspend STL execution for a designated time interval for the terminal.

UTI This statement enables you to change the IUTI value for your terminal. See "UTI" on page 445 for more information.

If you use a WAIT or QUIESCE statement without specifying a condition, you may place the terminal in a wait or quiesce state with no obvious means of getting out. an operator can end a wait state using the F (Console Recovery) command. The operator can end a quiesce state using the A (Alter) command with the RELEASE operand.

Evaluating the load on the WSim system

In addition to the network operands and STL statements, the load on the WSim system also affects the message transfer rate. The number of terminals WSim is simulating, the complexity and efficiency of the programs the simulated terminals are executing, and the overall load on the WSim host system can all affect the length of the intermessage delay for a given simulated terminal. If the load on the WSim system is too great, a simulated terminal may have to wait for other simulated terminals to reach an interrupt point in their STL programs before beginning program execution again, even though its own intermessage delay has expired. This can be minimized by giving WSim the resources it needs to sustain the desired rate, such as host processor priority, authorization, and paging.

Consider the potential effects of the system load when designing your simulation. If longer than expected delays are a possibility, you may want to set up your system to compensate for them.

Receiving messages

WSim can receive messages for a simulated terminal between messages transmitted. For all terminal types, WSim will test any outstanding asynchronous conditions for received messages and then free the received message buffer. For display terminals, the screen image buffers are updated. Asynchronous conditions are described in “Setting up asynchronous conditions” on page 299.

Your STL program cannot access messages received during an intermessage delay when it resumes execution. However, these messages (or portions of them) can be assigned to STL string variables asynchronously and referenced when the STL program resumes. This can be especially useful for nondisplay terminals since these terminals do not have screen buffers that are updated. For more information about asynchronous conditions and statements, see “Testing asynchronous conditions.”

For a display terminal, WSim updates the terminal's display buffer when a message is received. When the STL program resumes after the intermessage delay expires, the STL reserved variable SCREEN (or BUFFER) reflects the newly received information.

Note: For nondisplay devices, when the STL program resumes, these variables do not necessarily contain the information received during the intermessage delay.

Testing asynchronous conditions

WSim uses the time between messages to test whether specified conditions have been met while the terminal was in the delay state and to take actions based upon those conditions. The actions also take place while the terminal is in the delay state. The actions that take place are coded as a special set of statements called asynchronous subset statements because they can only be coded on specified asynchronous statements. The conditions and statements are called asynchronous because they occur outside the flow of normal STL program execution.

Asynchronous statements enable you to specify conditions to be tested when messages are sent or received and to take action based on the content of the messages. The messages may be those sent to and received from the host application, or they may be messages from other terminals indicating that specified events have been completed. WSim evaluates the message and compares its content to your specifications.

The following asynchronous STL statements enable you to test messages received and transmitted.

ONIN Defines an asynchronous condition that is to be tested when data is received by a simulated terminal and the actions that are to be taken if the condition is true.

ONOUT
Defines an asynchronous condition that is to be tested when data is transmitted by a simulated terminal and the actions that are to be taken if the condition is true.

ON SIGNALED
Defines an asynchronous action that is to be taken when the specified event is signaled.

WAIT UNTIL
Interrupts STL execution and defines an asynchronous condition that, when met, allows STL program execution to resume. This statement can also appear as the WAIT UNTIL clause of the TRANSMIT statement.

QUIESCE UNTIL
Interrupts STL execution and defines an asynchronous condition that, when met, releases the terminal and allows STL program execution to resume.

Note: The ONIN and ONOUT statements do not apply to CPI-C transaction program simulations, and will be ignored if specified.

Using the ONIN and ONOUT statements

The ONIN and ONOUT statements enable you to test messages received (ONIN) or sent (ONOUT) for the existence of specified conditions. The syntax of these statements is the following:

```
[label:] {ONIN} [asynchronous_condition] THEN[;] asynchronous_subset_statement
         {ONOUT}
```

If you like, you can include a *label* to identify each statement. For information about what can be included in an asynchronous condition, see “Setting up asynchronous conditions” on page 299. The statement following the THEN is an asynchronous subset statement that will be executed during the intermessage delay when the condition is met. These statements are described in “Using asynchronous subset statements” on page 294.

The following example shows how to use an ONIN statement:

```
onin then data_received = buffer
```

When a message is received, the data in the reserved string variable BUFFER (which contains the device display buffer) is saved in the string variable “data_received”.

Testing ONIN and ONOUT conditions

Conditions on ONIN and ONOUT statements are available for testing when the first statement following the ONIN or ONOUT statement is executed. ONIN conditions are tested for every incoming message destined for the simulated terminal and ONOUT conditions are tested for every message transmitted by the simulated terminal.

All ONIN and ONOUT statements in your program are tested each time a message is received or transmitted. If you do not want an ONIN or ONOUT statement tested, you must explicitly deactivate the ONIN or ONOUT statement with a DEACT statement. See “Deactivating ONIN and ONOUT conditions” for more information about this statement. ONIN and ONOUT statements are automatically deactivated by WSim at the end of your program (that is, when an ENDTXT statement is reached and all CALL statements in the program have been completed).

If more than one ONIN or ONOUT statement is currently active and thus available for testing, they will be tested in the order of their appearance in your STL program. The description of these statements in “ONIN and ONOUT” on page 420 provides an example of the testing order.

Deactivating ONIN and ONOUT conditions

ONIN and ONOUT conditions remain active throughout the execution of an STL program. Thus, when a terminal begins program execution, all ONIN and ONOUT conditions prior to the point of execution are active and are tested when a message is received or sent. To prevent testing of conditions, you must explicitly deactivate them. STL enables you to deactivate specific statements or all ONIN and ONOUT conditions in a program.

To deactivate a particular statement or group of statements, use the DEACT keyword followed by the label of the statement or statements, as shown in the following examples:

```
/* Example 1 */

test1: onin substr(data,1,4) = 'XXXX' then gotdata = on
 .
 .
 .
deact test1

/* Example 2 */
acheck: onin index(data,'A') > 0 then afound = on
bcheck: onin index(data,'B') > 0 then bfound = on
ccheck: onin index(data,'C') > 0 then cfound = on
 .
 .
 .
deact acheck, bcheck, ccheck
```

The statement is used in the same way for ONOUT conditions:

```
outb: onout substr(data,1,4) = 'xxxx' then outdata = on
 .
 .
 .
deact outb
```

If you want to deactivate all ONIN and ONOUT conditions in an STL program, use the following statement:

```
DEACT ALL IO ONS
```

Using asynchronous subset statements

Asynchronous subset statements are STL statements that can be executed when an asynchronous condition is satisfied on an ONIN, ONOUT, or ON SIGNALED statement. Asynchronous subset statements are executed during the transmit interrupt.

The asynchronous subset statements are the following:

- For ONIN, ONOUT, and ON SIGNALED:
  - ABORT
  - CALL
  - CANCEL {DELAY|SUSPEND}
  - EXECUTE
  - Any statement allowed within an EXECUTE procedure except RETURN.
- For ONIN or ONOUT only:
  - NORESP.

***ABORT Statement*:** The ABORT statement forces the current STL program to terminate immediately. All outstanding asynchronous conditions for the program are deactivated and the next program specified for this terminal on the PATH network definition statement will begin executing when the current intermessage delay expires.

For example, you could use the ABORT statement as follows:

```
onin index(ru,'UNRECOVERABLE ERROR') > 0 then abort
```

If the SNA request/response unit contains the specified message, the current STL program is terminated immediately.

***CALL Statement*:** The CALL statement calls the specified procedure when the terminal resumes program execution. Use this statement sparingly since it abnormally alters the flow of control of your STL program. It also resets the wait condition of your simulated terminal, causing any outstanding wait conditions to be prematurely satisfied.

You could use the CALL statement as an asynchronous subset statement like this:

```
onin index(ru,'UNRECOVERABLE ERROR') > 0 then call errproc
```

If the SNA request/response unit contains the specified message, the current STL program calls the procedure “errproc” to terminate the program. See “Reserved variables” on page 239 for information about using STL reserved variables.

***EXECUTE Statement*:** The EXECUTE statement specifies the name of an STL execute procedure that is to be executed immediately, during the transmit interrupt. Execute procedures are a special type of STL procedure that can contain only a limited set of STL statements. You may use an EXECUTE procedure if you want to be able to execute the same block of code in separate places in your program or among several STL programs.

An EXECUTE procedure can perform bit, integer, and string assignments and can post, signal, and reset events. Statements and functions used with events are discussed in “Posting and signaling events” on page 300.

The EXECUTE asynchronous subset statement has the following syntax:

```
EXECUTE execute_procedure_name
```

The *execute_procedure_name* is the name for the execute procedure. An execute procedure is coded like an STL procedure, beginning with an MSGTXT statement and ending with an ENDTXT statement.

The following statements are not allowed in an execute procedure:

- Device key statements (None of the statements that simulate device keys pressed by the operator can be used.)
- Flow-of-control statements
  - DO FOREVER, DO WHILE, iterative DO, SELECT, and IF
  - CALL statement except when coded as an asynchronous subset statement immediately following the THEN keyword on an ONIN, ONOUT, or ON SIGNALED statement.
- Transmit Interrupt statements:
  - CURSRSEL
  - INITSELF
  - LIGHTPEN
  - QUIESCE
  - SNACMND
  - STRIPE
  - SUSPEND
  - TERMSELF
  - TRANSMIT
  - WAIT.

The following statements also are not allowed:

- CURSOR
- DELAY
- SETRH
- SETTH
- TYPE
- VERIFY.

The following functions are not allowed except as part of an ONIN or ONOUT statement.

- CPOS
- INDEX
- POSTED
- UTBLSCAN.

The following example shows how the EXECUTE statement can be used to identify a procedure to be executed asynchronously.

```
myproc: msgtxt
 .
 .
 .
onin index(screen,'XXX') > 0 then execute execproc
onin index(screen,'YYY') > 0 then execute execproc
 .
 .
 .
endtxt
```

```
execproc: msgtxt                /* This is an execute procedure. */
number_received = number_received + 1
post 'RCVEVENT'
endtxt
```

Any Statement Allowed within an EXECUTE Procedure*:** This allows you to enter any statement that you can code within an EXECUTE procedure except RETURN. See "EXECUTE Statement***" on page 294 for a list of statements not allowed within an execute procedure. For example, you may want to post an event and write out a message to the operator whenever a message is received. This is shown in the following example.

```
onin then do                       /* If a message is received     */
   post 'MYEVENT'                  /* Post the event 'MYEVENT'     */
   say 'MYEVENT posted'            /* Notify the operator          */
   end
```

Likewise, you may want to save a data stream asynchronously and use this data when program execution resumes. The example below shows you how to do this.

```
onin then gotdata = on             /* Get something.               */
onin rh &= '80'x then gotdata = off   /* Do not save SNA responses.   */
onin gotdata then thisdata = buffer   /* Save the buffer when a       */
                                   /* non-SNA response is          */
                                   /* received.                    */
type 'Hello'                       /* Type text to be transmitted. */
gotdata = off                      /* Prepare to wait for data.    */
transmit and wait until onin gotdata   /* Transmit text and wait for a */
                                   /* non-SNA response.            */
gotdata = off                      /* Reset for next wait.         */
if thisdata = 'Goodbye' then       /* Partner sent Goodbye.        */
   do
   .
   .
   .
   end
else
   if thisdata = 'So long' then    /* Partner sent So long.        */
   .
   .
   .
```

***CANCEL Statement*:** The CANCEL statement enables you to cancel the delay currently in effect. This delay may have been set by a DELAY or SUSPEND statement or it may be the standard intermessage delay for the simulated terminal that is coded in the network definition.

The syntax for this statement is the following:

```
CANCEL {DELAY}
       {SUSPEND}
```

Use the SUSPEND option when you are canceling a SUSPEND statement; otherwise, use the DELAY option.

When the delay or suspend is canceled, the terminal should resume execution of the STL program shortly unless the terminal is waiting for a condition to be met or the terminal is not in a state in which it can send messages.

***NORESP Statement*:** The NORESP statement tells WSim *not* to send an SNA response automatically for the current message. Instead, WSim sets the transmission header (TH) and the response/request header (RH) for the normal response and then resumes STL execution, enabling your program to construct its own SNA response immediately.

Notes:

- WSim automatically generates and transmits SNA responses unless you include the NORESP statement as your asynchronous subset statement on an ONIN or ONOUT statement. Use this statement only when you want to send an abnormal response. Coding this statement ensures that a response is sent.
- Do not send your response by using the TRANSMIT statement. The TRANSMIT statement sets the AID byte. Instead, use the SUSPEND statement. This causes the data to be sent without setting the AID byte.

The following example shows how the NORESP statement is used.

```
norespx: msgtxt

/*****************************************************/
/* Activate ONIN conditions.                         */
/*****************************************************/

onin then something_received = on
onin substr(data,1,6) = 'xxxxxx' then generate_response = on
onin substr(data,1,6) = 'xxxxxx' then noresp

something_received = off          /* Prepare to receive something.    */
generate_response = off           /* Prepare to receive something.    */
wait until onin something_received /* Wait till something is received. */
/*********************************************************************/
/* Something has been received.  See if response should be generated.  */
/*********************************************************************/

if generate_response then         /* Program needs to generate       */
                                  /* response.                       */
 do
  type '082E0000'x                /* This is an SNA-generated        */
                                  /* response to send the intervention*/
                                  /* required sense data.            */

  setrh on(exc,sni)               /* Indicate this is an exception   */
                                  /* response.                       */

  suspend()                       /* Required to send the response   */
                                  /* without setting an AID.         */
 end
else                              /* Program does not need to        */
                                  /* generate response.              */
 .
 .
 .
endtxt
```

When coding ONIN or ONOUT statements, it is good programming practice to follow these steps:

1. Set up the conditions.
2. Set the values to OFF.
3. Continue with the synchronous execution of the program.

Using network-level IF statements

You can also test messages transmitted and received with network IF statements in your network definition. These statements enable you to test message conditions for all terminals in a network rather than just the terminal currently executing an STL program. See , SC31-8945 for information about how to code these statements and how these statements interact with terminal IF statements.

Using the WAIT UNTIL and QUIESCE UNTIL statements

The WAIT UNTIL statement and the QUIESCE UNTIL statement (or these clauses on the TRANSMIT statement) stop execution of a terminal's STL program. The terminal then waits or quiesces until the specified conditions are met. When the condition is met, the STL program continues synchronous execution after the intermessage delay has expired.

The WAIT UNTIL and QUIESCE UNTIL statements do not use asynchronous subset statements.

The WAIT UNTIL statement uses the following syntax:

```
WAIT ┌                                      ┐
     │ UNTIL {ONIN [asynchronous_condition]} │
     │       {ONOUT [asynchronous_condition]}│
     │       {POSTED(event_name)}            │
     │       {SIGNALED(event_name)}          │
     ≪                                       ┘
```

The *asynchronous_condition* can be any condition that meets the requirements described in “Setting up asynchronous conditions” on page 299. The POSTED and SIGNALED functions enable you to test whether an event to be posted or signaled has occurred. These functions are discussed in “Posting and signaling events” on page 300. When the condition is satisfied or the event specified by *event_name* has occurred, the terminal is ready to begin program execution when its intermessage delay has expired.

The WAIT UNTIL statement may imply a TRANSMIT USING ENTER. If TYPE statements have been processed since the last Transmit Interrupt, the WAIT UNTIL statement sends the messages.

The following two sections of code do exactly the same thing:

```
transmit using clear
type 'Hello'
wait until onin substr(data,1,1) = 'X'

or

transmit using clear
type 'Hello'
transmit using enter and wait until onin substr(data,1,1) = 'X'
```

In both cases, the message is transmitted before the terminal begins waiting.

The QUIESCE UNTIL statement uses this syntax:

```
QUIESCE ┌                                      ┐
        │ UNTIL {ONIN [asynchronous_condition]} │
        │       {ONOUT [asynchronous_condition]}│
        │       {SIGNALED(event_name)}          │
        ≪                                       ┘
```

The QUIESCE statement enables you to specify optionally the conditions that release a terminal from a quiesced state, as shown in the following example.

```
/* Quiesce until the terminal receives the string 'WAKE UP'. */

quiesce until onin index(screen,'WAKE UP') > 0
```

Like the WAIT UNTIL statement, the QUIESCE UNTIL statement may imply a TRANSMIT.

Setting up asynchronous conditions

Asynchronous conditions, like regular conditions, can be simple (a single relational expression consisting of an expression or two expressions joined by an arithmetic or string operator) or complex (two or more relational expressions joined by logical operators). See "Using conditions and relational operators" on page 261 for information about regular conditions.

The following examples show how you can code asynchronous conditions.

```
onin substr(data,1,3) = '111111'x then ...
                                         /* Compares the first        */
                                         /* 3 bytes of the incoming   */
                                         /* data with the hexadecimal  */
                                         /* string '111111'x           */

wait until onin substr(ru,100,5) = 'Hello'
                                         /* Interrupts program         */
                                         /* execution until an SNA RU  */
                                         /* is received with the       */
                                         /* string 'Hello' starting at */
                                         /* position 100.              */
```

The following sections discuss the restrictions on expressions and the SUBSTR function when used in asynchronous conditions.

Restrictions on expressions in asynchronous conditions

Unlike regular conditions, expressions used in asynchronous conditions are limited to those that do not require computation of intermediate results. The following expressions require computation of intermediate results: integer arithmetic expressions involving at least one variable and SUBSTR functions that specify a variable starting position value. The expressions that cannot be used are listed in Chapter 28, "Expressions not allowed in asynchronous conditions," on page 503.

The following examples show coding that **cannot** be used because it requires computation of an intermediate result or uses a variable starting position on a SUBSTR function.

```
onin a = b * c then . . .    /* Cannot be used because it requires      */
                             /* computation of an intermediate result,  */
                             /* the product of two integer variables.   */

wait until onin substr(data,a,5) = 'hello' /* Cannot be used because a   */
                                           /* variable starting position */
                                           /* on a SUBSTR function       */
                                           /* requires a temporary       */
                                           /* counter.                   */
```

Restrictions on the SUBSTR function in asynchronous conditions

There are a number of restrictions on the SUBSTR function when it is used as a part of an asynchronous condition. The use of the SUBSTR function is described in "The SUBSTR function" on page 250.

The SUBSTR function enables you to use a part of a source string expression. The syntax of the SUBSTR function is the following:

```
SUBSTR(source,starting_position[,length])
```

When you are using this function in an asynchronous condition, the following restrictions apply:

1. The *source* expression must be a string variable or STL reserved variable; it cannot be any other type of string expression such as a string constant, string concatenation, or function that returns a string.
2. Only integer constants and integer constant expressions can be used as *starting_position* arguments.
3. Only integer constants, integer constant expressions, or single integer variables can be used as *length* arguments. You cannot use expressions that include variables.

Examples of these restrictions appear in “SUBSTR” on page 488.

Posting and signaling events

WSim uses events to coordinate the activities of simulated terminals. Events enable you to:

- Control the activity of a single terminal
- Synchronize the activities of two or more terminals
- Allow the operator to coordinate terminal activity.

When you coordinate terminal activity, you can simulate a terminal operator who must wait for a particular event or an action by another terminal operator before continuing.

You define an event by assigning it a name and informing the terminals when the event is posted or signaled. This is done as follows:

- **Posting** an event means that WSim is posting a notice that the event has occurred, like posting a notice on a bulletin board. The posting is available to any terminal that inquires about the event and is in effect until it is explicitly reset, much like a notice, which can be consulted until it is removed.
- **Signaling** an event means that WSim is signaling that an event has happened. Unlike a post, a signal cannot be consulted later to determine if an event had occurred. The event must be signaled again. You can also qsignal an event. A qsignal is a qualified signal that affects only the terminal that issued the qsignal.

The following sections describe these two methods of managing events. Note that the two methods are completely independent of each other. Posting an event has no effect on terminals waiting for a signal, and a signal has no effect on terminals waiting until an event is posted, even if the same event name is used for both events.

Posting events

You post events when you want simulated terminals to be able to check the status of the event at different times. When the event has occurred, terminals will use that information as they continue executing their STL program.

To use this method of managing events, you must take these steps:

- Post the event.
- Request information about the status of the event.

Posting the event

Use the POST statement to post an event. This statement uses the following syntax:

```
POST event_name [AFTER time [TAG event_name]]
```

event_name is a string expression that specifies the name of an event. If it is a string constant expression, *event_name* must be 1 to 8 alphanumeric characters and enclosed in single or double quotation marks. (Strings containing hexadecimal constants are exempt from this restriction.) If you specify a nonconstant string expression, the first 8 characters of the string are used. If the string is shorter than 8 characters, the available characters are used. To satisfy conditions using event names, the first 8 characters must match exactly. If you specify a string variable, it cannot be the name of one of the reserved variables (for example, BUFFER).

Note: See "Specifying variable event names with a time delay" on page 306 for information about specifying variable event names with a time delay.

The AFTER clause enables you to post an event after a specified period of time has elapsed. The value for *time* is an integer value in seconds.

The TAG clause enables you to tag events that you may want to cancel before the specified time elapses. You can use the TAG clause to assign the same *event_tag* to multiple events. You can assign the same tag to events that are posted, qsignaled, reset, or signaled, enabling you to cancel multiple events at one time. If an event tag is not specified, a default tag is assigned that has a value that is the same as the event name. See "Canceling events" on page 305 for information about canceling events. You must code an AFTER clause if you want to code a TAG clause on the POST statement. If you specify a string variable, it cannot be the name of one of the reserved variables (for example, BUFFER).

If an invalid string expression is used for the event name or tag, a value of eight blanks (X'4040404040404040') is used. This value will be assigned if the event name or tag is the null string or a expression containing a substring function with an invalid *starting_position*.

If you wanted to post an event, you could use this code:

```
if a = 1 then
  post 'MYEVENT'
else
   nop
```

In a more complex example, you could post the same event after 5 seconds and give it a tag using the code in the following example.

```
if a = 1 then
  post 'MYEVENT' after 5 tag 'EVENT1'
else
  nop
```

Note: You cannot code an AFTER value of 0.

When you post an event, the post remains in effect until explicitly reset. To reset a post, use the RESET statement. The RESET statement uses the following syntax:

```
RESET event_name [AFTER time [TAG event_name]]
```

The AFTER and TAG clauses are used in the same way as for the POST statement. If you tag an event, you can cancel the activity for the event (in this case, resetting the event) before the specified time has elapsed. In the following example, the event 'MYEVENT' is reset.

```
reset 'MYEVENT'
```

The operator can also post events using the A (Alter) command and reset them using the R (Reset) command. See , SC31-8948 for information about operator commands.

Determining if an event is posted

To direct your simulated terminal to take action when an event is posted, use the POSTED function. You can use this function on a WAIT statement or as a condition in structured flow-of-control statements.

For example, the following statement forces your terminal to wait until the named event takes place.

```
wait until posted('RIGHT')       /* The terminal continues execution   */
                                 /* when event RIGHT occurs.           */
```

When the event is posted, the condition is satisfied and the terminal continues executing the STL program after the intermessage delay has expired.

You can also use the POSTED function in structured flow-of-control statements. For example, the statements following the THEN in the following example will be executed if the event 'HELLO' has been posted before the program execution reaches this point.

```
if posted('HELLO') then
   say 'HELLO has been posted'
else
   say 'HELLO has not been posted'
```

You can also wait to take action until a combination of events occurs: an event is posted and a message is received or sent. Use the following syntax:

```
WAIT UNTIL {ONIN} POSTED(event_name)
           {ONOUT}
```

If you are waiting for a message to be received (ONIN), the terminal continues synchronous execution when a message is received after the named event has been posted. If you are waiting for a message to be sent (ONOUT), the terminal continues synchronous execution after you send a message and the event is posted. Obviously it is not typically possible to send a message when the terminal is waiting. This capability is primarily useful when waiting to send messages that your terminal sends automatically, such as SNA responses.

You can also use this construction on the QUIESCE statement. For example:

```
quiesce until onin posted('MYEVENT')  /* Quiesce this terminal  */
                                      /* until a message is     */
                                      /* received while MYEVENT */
                                      /* is posted.             */
```

The following example shows how you could coordinate the activities of two terminals by using a post.

```
string shared message        /* "message" is shared by all terminals. */

master: msgtxt
/************************************************************/
```

```
/*                        MASTER                            */
/*                                                          */
/* This procedure is for the "master" terminal.  It is      */
/* responsible for building a message to be transmitted,    */
/* posting event READY once the message has been built, and */
/* transmitting the message.                                */
/**********************************************************/
message_number = message_number + 1        /* Building a new message. */
message = 'This is message number' char(message_number)
post "READY"
type message
transmit                                   /* Send "message."         */
endtxt
```

```
mimic:  msgtxt
/**********************************************************/
/*                          MIMIC                         */
/*                                                        */
/* This procedure is for the terminal which mimics the    */
/* "master" terminal.  It must wait until event READY is  */
/* posted, then send the message previously built         */
/* and saved in the shared variable "message".            */
/**********************************************************/
wait until posted("READY")       /* Wait for event READY to be posted. */
reset "READY"                    /* Reset event READY for next use.    */
type message                     /* "message" is a shared variable.    */
transmit                         /* Send "message".                    */
endtxt
```

Using signals

Once an event is signaled, any actions the terminal takes as a result of the signal must be taken immediately. You can signal an event using the SIGNAL statement or the QSIGNAL statement in your STL program. The SIGNAL statement affects all terminals in your network. The QSIGNAL statement affects only the terminal that issues the QSIGNAL. The operator can also signal an event using the A (Alter) operator command. See , SC31-8948 for information about operator commands.

Signaling an event

To signal an event, use the SIGNAL or QSIGNAL statement. The syntax for these statements is the following:

```
SIGNAL event_name [AFTER time [TAG event_name]]
```

```
QSIGNAL event_name [AFTER time [TAG event_name]]
```

event_name is a string expression that specifies the name of an event. If it is a string constant expression, *event_name* must be 1 to 8 alphanumeric characters and enclosed in single or double quotation marks. (Strings containing hexadecimal constants are exempt from this restriction.) If you specify a nonconstant string expression, the first 8 characters of the string are used. If the string is shorter than 8 characters, the available characters are used. To satisfy conditions using event names, the first 8 characters must match exactly. If you specify a string variable, it cannot be the name of one of the reserved variables (for example, BUFFER).

Note: See "Specifying variable event names with a time delay" on page 306 for information about specifying variable event names with a time delay.

The AFTER clause enables you to specify a time interval that must elapse before the event is signaled. The TAG clause identifies a name for the event that can be used to cancel the event if the time interval has not expired. If an event tag is not specified, a default tag is assigned that has a value that is the same as the event name.

Once an event is signaled, your terminals have two ways to act on the signal:

- Wait or quiesce until an event is signaled and then continue synchronous program execution
- Take action on the signal asynchronously, that is, whenever the event is signaled, regardless of whether the terminal is currently executing the program.

These methods are described in the following sections.

Waiting or quiescing until an event is signaled

Just as with a post, you can wait for a signal and continue synchronous program execution when the event is signaled. Use the following statement to direct the terminal to wait until an event is signaled:

```
WAIT UNTIL SIGNALED(event_name)
```

Synchronous program execution continues when the condition is satisfied. You can use the QUIESCE UNTIL statement in the same way. The condition for the UNTIL SIGNALED statement is met with either a SIGNAL or a QSIGNAL. The following example shows how to use this statement:

```
wait until signaled('MYEVENT')       /* The terminal will wait until   */
                                     /* the event MYEVENT is signaled. */
```

Taking action on a signal asynchronously

The ON SIGNALED statement activates an asynchronous condition. The condition is satisfied when the event is signaled. When the event is signaled and the condition satisfied, the asynchronous subset statement associated with the statement is executed immediately.

ON SIGNALED is an asynchronous statement, like ONIN and ONOUT, that is active while a program is executing. Thus, an action can be taken during the terminal's intermessage delay.

The syntax of the ON SIGNALED statement is the following:

```
ON SIGNALED(event_name) THEN[;] asynchronous_subset_statement
```

The *event_name* is an event name defined by a QSIGNAL or SIGNAL statement. The *asynchronous_subset_statement* can be one of the following statements: EXECUTE, CALL, ABORT, or any statement allowed within an EXECUTE procedure, except RETURN. See "Using asynchronous subset statements" on page 294 for information about these asynchronous subset statements.

Once an ON SIGNALED condition has been satisfied, it is no longer active. If you want to reactivate the condition, you must include another ON SIGNALED statement in your program.

Only those ON SIGNALED conditions that are active at the time an event is signaled are affected by that signal. Earlier signals of the same event have no effect on an ON SIGNALED condition that is subsequently activated specifying that event. Statements are activated as program execution reaches them.

You can deactivate any number of ON SIGNALED statements in your program. To deactivate all ON SIGNALED conditions referring to particular events, enter:

```
DEACT event_names
```

where *event_names* is a list of event names, separated by commas. If you want to deactivate all ON SIGNALED conditions in a program, enter this statement:

```
DEACT ALL EVENT ONS
```

The following example shows how the ON SIGNALED statement is used.

```
runit: msgtxt
myevent = "HELLO"
on signaled(myevent) then say 'Event "HELLO" has been signaled.'
 .
 .
 .
endtxt
```

The condition is tested immediately when the event is signaled.

The following example illustrates a more complex use of this condition. In this example, suppose that you want a simulated terminal to write a report to the operator each time an event is signaled. The operator signals the event periodically using the following command:

```
a network,SIGNAL=MYEVENT
```

The following portions of an STL program will report the information to the operator.

```
proc1: msgtxt
/**********************************************************/
/* This procedure keeps a count of the number of times     */
/* various events occur.                                   */
/**********************************************************/

on signaled('MYEVENT') then execute report
 .
 .
 .
endtxt
report: msgtxt
/************************************************************/
/* This procedure will be executed when MYEVENT is signaled. */
/* It reestablishes the ON SIGNALED condition and           */
/* then reports various types of information                */
/* to the operator.                                         */
/************************************************************/

on signaled('MYEVENT') then execute report

number_of_signals = number_of_signals + 1

say 'Total number of signals =' char(number_of_signals)
say 'Total messages received =' char(messages_received)
say 'Total messages sent     =' char(messages_sent)
 .
 .
 .
endtxt
```

Canceling events

You can cancel events by using the CANCEL statement and the tag assigned to the events to be canceled. The syntax of this statement is the following:

```
CANCEL event_tag
```

The *event_tag* is named following the same rules for event names discussed in "Posting the event" on page 301. When WSim encounters a CANCEL statement, it cancels all events with the specified tag named on POST, QSIGNAL, RESET, and

SIGNAL statements for which the specified time has not elapsed. Thus, if the named events have not yet been posted, qsignaled, reset, or signaled, the action is canceled.

If the event was posted, qsignaled, reset, or signaled without an event tag, the event can still be canceled if the time specified on the AFTER clause has not expired. If you did not specify an event tag, a default event tag is assigned. The default event tag is the same as the event name assigned to the event.

When used to cancel events, the CANCEL statement is not an asynchronous subset statement.

For examples of how to use the CANCEL statement to cancel events, see "CANCEL" on page 366.

Specifying variable event names with a time delay

If *time* is specified, the *event_name* will not be assigned if it is a variable until the time expires. For example, assume the following code:

```
my_event = 'KEY1'
post my_event after 10
my_event = 'KEY2'
post my_event after 10
```

In this case the event KEY2 will be posted twice because the name of the event is not resolved until after the time expires.

Chapter 21. Monitoring and automating your test

STL uses the WSim facilities to enable you to perform the following tasks:

- Monitor the test by:
  - Writing messages to the operator
  - Displaying messages at a display monitor using the Display Monitor Facility
  - Using the printed listing to trace program execution and check the values of variables.
- Log test data to the log data set
- Write verify records to the log data set to produce Verification Reports
- Automate the test so that it can run without an operator
- Identify network resources being used.

These capabilities enable you to monitor the test as it is running, to obtain data to review performance after the test is complete, and to automate the test.

Monitoring the test

The simplest way to monitor your test is to write statements to the operator's terminal at certain points in your program. In STL, you can write these messages using the SAY statement. For example, the following statement sends the message "The test is running" to the operator's terminal.

```
SAY "The test is running"
```

The message can be a string variable or constant expression. You must enclose a string constant in a pair of single or double quotation marks.

If you are using the Display Monitor Facility, you can send the simulated 3270 display image as it exists at a particular point in your program to the display monitor. You can send these images by including a MONITOR statement in your STL program at the appropriate point. For example, you could display a screen you received using these commands:

```
transmit and wait until onin     /* Send data and wait for new screen. */
monitor                          /* Display the screen you received at */
                                 /* the Display Monitor Facility.      */
```

You can use the Display Monitor Facility in other ways as well. For more information about the Display Monitor Facility, see , SC31-8948.

You can also use operator commands to monitor your simulation by tracing the execution of your program. The operator commands enable you to display the statement number currently being executed.

You may also wish to check or modify the value of one of your variables using operator commands while the test is running. For example, you may want to find out how many times a loop has been executed. You can use the printed listing to determine how STL variables map to save areas, counters, and switches by consulting the variable dictionary at the end of the listing. See "Printed listing" on page 314 for more information about the printed listing produced by the STL Translator.

Logging test data

WSim automatically logs all message traffic to and from your simulated terminals unless you code MLOG=NO for your network. You can also direct WSim to log STL trace information when a simulation is running by using operator commands. See , SC31-8948 for more information on these commands. After the information is written to the log data set, you must format it using the Loglist Utility. See , SC31-8947 and "Obtaining STL trace records" on page 341 for information about using this utility.

In your STL program, you can define other messages to be written to the log data set. Typically these are informational in nature and report on the progress of the STL program. You can also write the display image for the following terminals to the log data set: 3270 or 5250. You can log information by using the LOG statement. You can log a display image to the log data set as shown in the following example.

```
log display                                /* Logs the current screen. */
log 'I have just logged the display image for 'devid()
```

WSim logs the display and then the message identifying the device. The function DEVID provides the name of the simulated device currently executing the STL program.

Writing verify records

In your STL program, you can also write Verify records to the log data set (used by the Loglist Utility to produce Verification Reports) with the VERIFY statement. Verification Reports provide, at a glance, information about a simple condition. You can also use them to quickly determine if an error occurred and, if so, how many times.

Note: Verification Reports reference resources: counters, save areas, and switches. Therefore, you should refer to the variable dictionary to determine how STL variables map to the resources.

The format of the VERIFY statement is as follows:

```
VERIFY simple_condition [FOR description]
```

The following examples show you how to use the VERIFY statement.

```
/* Serial numbers, which are found at offset 30 on the screen        */
/* cannot contain a "9" as the first digit.  If this occurs           */
/* then log a verify record.  The description, "Serial number        */
/* in error" can be used when looking at the Verification Summary     */
/* Report to determine how many errors of this nature occurred.      */

verify substr(screen,30,1) = '9' for 'Serial number in error'

verify amount = 0 /* Log Verify record if "amount" equals zero.      */
```

Automating your test

On occasion, you may want to run a test without having an operator present. To create a self-controlled test, you can include operator commands in your STL programs. With these commands, you can control the test without intervention from the operator; changes can be made just as if the operator were there. By coding operator commands in your STL program, a simulated terminal can start and stop other network resources, query other terminals, activate traces, alter message rates and delays, post events, and even stop a network or cancel WSim.

You can also use operator commands to automate regression testing with an existing set of networks. You could write a master network to initialize, start, and stop the networks used in the regression test.

To incorporate operator commands, use the OPCMND statement followed by the appropriate operator command. See , SC31-8948 for details of operator command usage and syntax.

Note: To incorporate operator commands you **must** code the OPTIONS=(MONCMND) network definition operand.

For example, the following statement automatically alters the user time interval (UTI) to 100 when the STL program is running. It does not require intervention from the operator.

```
OPCMND 'A 'NETID()',U=100'                /* Alter UTI to 100. */
```

Notice that NETID() is an STL function that inserts the network name in the operator command.

Although the command can be any length, a maximum of 120 characters is actually used.

You can use all operator commands except console recovery commands with the OPCMND statement. Console recovery commands must be entered by an operator.

Identifying network resources

In your messages to the operator and the log data set, you may want to identify the terminals currently running the STL program or the network being used. STL provides a set of functions that return this type of information. The following list indicates the functions available and the information each returns.

Function	Returns
APPCLUID()	Name of the APPC LU associated with a simulated transaction program
DEVID()	Name of the simulated device executing the STL procedure
ID()	All or part of the name of a terminal
LASTVERB()	Name of the last CPI-C verb issued by this message deck in a CPI-C simulation
LUID()	Name of the simulated LU executing the STL procedure
MSGTXTID()	Name of the STL procedure currently being executed
NETID()	Name of the simulated terminal's network
SESSNO()	Session number of the simulated LU executing the STL procedure
TCPIPID()	Name of the TCP/IP connection associated with a simulated device
TPID()	Transaction program name for this message deck
TPINSTNO()	Transaction program instance number for this message deck
VTAMAPID()	Name of the VTAMAPPL associated with a simulated LU.

These functions return the names identified on the network definition for these terminals and devices. Each function returns the name of the terminal or device currently executing the STL program.

These functions are described in detail in Chapter 26, “Reference to STL functions,” on page 449.

Chapter 22. Using the STL Translator

The STL Translator translates your STL programs into the message generation statements required to run a simulation. If your programs do not have any syntax errors, it also places the translated message generation decks into the data set you specify.

If you include a network definition in your STL input, the STL Translator invokes the Preprocessor to verify the network definition statements. If there are no errors, the Preprocessor stores the network definition in the data set you specify.

To run a simulation, you must store the network definition and message generation decks that make up the script for the network in the appropriate data sets. This chapter discusses storage options, explains how to run the STL Translator, and provides example output. Chapter 23, “Combining STL programs and network definitions,” on page 325 discusses factors to consider when combining network definitions and STL programs to create scripts.

This chapter discusses the following topics:
- Methods for storing scripts
- Input to the STL Translator
- Output created by the STL Translator
- How to run the STL Translator
  - Execution parameters
  - Sample JCL and TSO CLISTs for running the translator
  - Data set requirements
  - Return codes for the translator.

Methods for storing scripts

During a simulation, the network definition and message generation decks must be in specific data sets.

Figure 4 on page 312 shows the three methods of storing scripts.

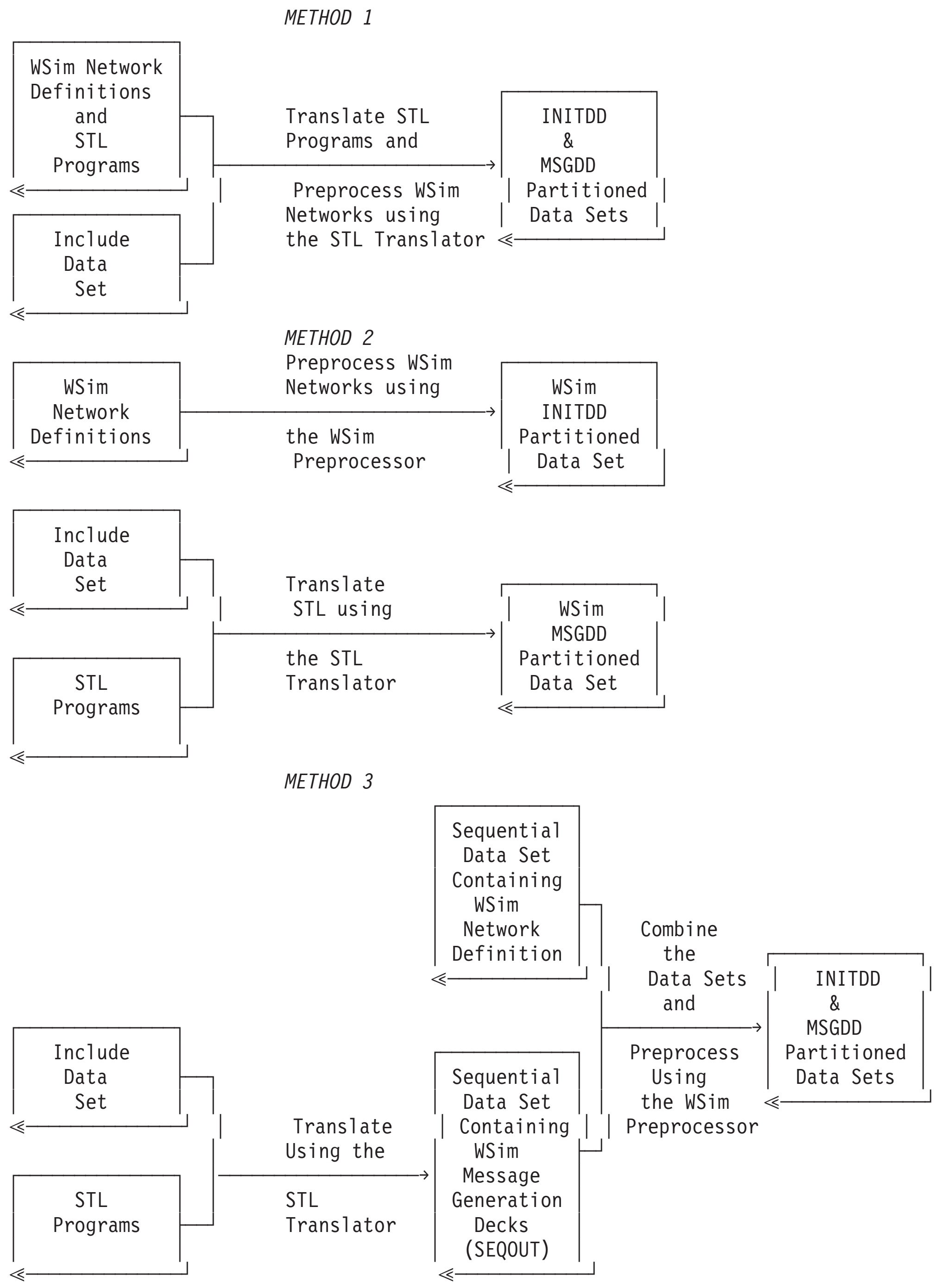

METHOD 1
WSim Network Definitions and STL Programs
Include Data Set
Translate STL Programs and
Preprocess WSim Networks using the STL Translator
INITDD & MSGDD Partitioned Data Sets
METHOD 2
WSim Network Definitions
Preprocess WSim Networks using
the WSim Preprocessor
WSim INITDD Partitioned Data Set
Include Data Set
STL Programs
Translate STL using
the STL Translator
WSim MSGDD Partitioned Data Set
METHOD 3
Sequential Data Set Containing WSim Network Definition
Combine the Data Sets and
Preprocess Using the WSim Preprocessor
INITDD & MSGDD Partitioned Data Sets
Include Data Set
STL Programs
Translate Using the
STL Translator
Sequential Data Set Containing WSim Message Generation Decks (SEQOUT)

Figure 4. Using the STL Translator and the Preprocessor to store scripts

Method 1 lets you create message generation decks from your STL programs and store them in the MSGDD data set. It also lets you include your network definition in the same data set as your STL programs. This way, when you run the STL Translator, it invokes the Preprocessor to store the network definition in the INITDD data set. This is the recommended method of storing scripts.

Method 2 lets you store the network definition in a separate data set and run the Preprocessor on it as a separate step from translating the STL programs. The Preprocessor stores your network definition in the INITDD data set.

Method 3 lets you store the message generation decks output by the STL Translator in a sequential data set. Only the message generation decks are written to this data set. These decks can then be combined with the network definition and run through the Preprocessor as a separate step. The Preprocessor integrates the network definition and message generation deck portions of the script and stores the output in the MSGDD and INITDD data sets. If you include the network definition as part of your STL Translator input, you may want to specify the NOPREP execution parameter when using this method.

You must specify the NOSEQOUT execution parameter when running the STL Translator if you do not want the sequential output data set. The STL Translator creates this data set by default.

Specify the NOPDSOUT execution parameter if you do not want the STL procedures and user tables to be stored in the MSGDD data set. Note that the trace correlation data set, which is required by the STL trace facility and the Q (Query) operator command to display STL statement numbers, will not be available. If you code a network definition in your STL input and specify the NOPDSOUT execution parameter, members could be stored in the MSGDD data set during Preprocessor execution.

You may want to run the Translator for preliminary debugging of your STL programs without requesting either a sequential data set or a MSGDD partitioned data set. You can then examine the STL Translator printed listing to debug syntax errors.

If you choose the recommended method for storing your script (Method 1), syntax errors for the network definition appear in the Preprocessor output, which is appended to the STL Translator listings. If the Preprocessor runs with no errors, it stores your network definition in the INITDD data set. If you do not change the network definition after it is preprocessed, you can then run the STL Translator with the NOPREP execution parameter. This eliminates unnecessary invocations of the Preprocessor.

After you have completed preliminary debugging, you will usually send translated STL procedures directly to an existing MSGDD. The STL Translator will not add STL program output to the MSGDD data set until the program translates without errors. Each procedure in the program is translated as a separate partitioned data set member. See “Structuring STL programs” on page 326 for a definition of an STL program and procedure.

Although storing a complete program at one time is simpler, it is not a requirement. If you do choose to write a program in several parts and translate them into your MSGDD data set separately, you must follow the procedures outlined in “Combining STL procedures from different STL programs” on page 333.

You can define an INITDD data set for network definitions and a separate MSGDD data set for message generation decks. If you like, you can define INITDD and MSGDD as the same data set and store your entire script together. See , SC31-8948 for more information about how to run WSim.

Input to the STL Translator

The source data set containing your network definition and STL programs is the primary input to the STL Translator. This source data set must be a sequential data set or a member of a partitioned data set of fixed, fixed-block, variable, or variable-blocked record format with a maximum logical record length of 255 bytes. This data set is defined to the operating system on the SYSIN DD statement when running the translator.

Sequence numbers placed in the data set by some editors will cause translation errors if they are not removed.

Also, a data set containing members to be included by the STL Translator may be appended. This data set must be a partitioned data set. You define this data set to the operating system with the SYSLIB DD statement when you run the Translator.

Output created by the STL Translator

The STL Translator produces the following types of output:

- Printed listing
- MSGDD partitioned data set members containing translated STL procedures, user tables, message generation decks, and the STL trace correlation records (STL to message generation deck statement number correlations required to use the STL trace facility and the Q (Query) operator command to display STL statement numbers - optional)
- INITDD partitioned data set members containing network definitions
- Sequential output data set (must be defined unless you specify the NOSEQOUT execution parameter)
- Temporary work data sets.

You will obtain the MSGDD partitioned data set members and the sequential output data set unless you suppress them with your execution parameters. See "Using STL Translator execution parameters" on page 319 for information about execution parameters. You must specifically request the MSGDD partitioned data set member containing the variable dictionary and statement correlations when you run the STL Translator. You request the variable dictionary and statement correlations by coding @PROGRAM statements in your STL input or by running the translator with the PROGRAM execution parameter specified.

The STL Translator outputs are described in the following sections.

Printed listing

The printed listing created by the STL Translator contains the message generation statements the translator produces and the STL input source lines. The input source lines appear in the listing as message generation statement comments (an asterisk is added to the beginning of each line). If you do not want message generation statements in the listing, you can suppress them by specifying the NOWSIM execution parameter when you run the translator.

The listing also contains Preprocessor output if you included a network definition in your STL input and did not specify the NOPREP execution parameter. If you specify NOLIST, only Preprocessor errors appear. If you specify SUMMARY, an extra report summarizing the network definition options and defaults appears. You also get a cross-reference report unless you specify NOXREF or NOLIST.

Below is a summary of the listing formats:

1. STL Translator Listings (one for each program):
   - Error messages
   - Source statements, unless you specify NOSOURCE
   - Scripting language statements, unless you specify NOWSIM
   - Variable and Event Dictionary
   - Statistics Report.
2. Preprocessor listings (one for each network):
   - Error messages
   - Preprocessor listing, unless you specify NOLIST
   - Cross-reference report, unless you specify NOXREF or NOLIST
   - Network Definition Summary Report, if you specify SUMMARY.

The data set to contain the printed listing is defined to the operating system on the SYSPRINT DD statement when you run the STL Translator.

Figure 5 on page 316 shows an example listing resulting from the translation of an STL input data set that includes a network definition. Both the STL listing and the Preprocessor output are shown as they appear in the SYSPRINT data set.

In the STL listing, statement numbers are included at the left side of the listing. The STL statement numbers and the message generation statement numbers are printed in separate columns. The STL statement numbers are referenced on the STL trace records (STRC) produced by the Loglist Utility since STL trace records are requested in the network definition (by coding STLTRACE=YES). The message generation statement numbers are referenced on the message trace records (MTRC) produced by the Loglist Utility since message trace records are requested in the network definition (by coding MSGTRACE=YES).

At the end of the STL listing, the STL Translator appends a dictionary of the variables used in the source program, detailing their use, class, and associated resource name. A dictionary of the events used, if any, is also appended.

Note: The variable dictionary, event dictionary, and statement numbers are for your use when reading the STL listing and debugging your programs. For more information, see "Reading the variable dictionary" on page 338 and "Reading the event dictionary" on page 340.

The following list shows the contents of the STL listing by column:

Columns	**Contents**
1	Carriage control character for printing
2-6	Message generation statement number (message generation statements only)
8-12	STL statement number (STL statements only)
14-133	Statement text

The network definition statements included in the STL Translator input do not appear in the STL listing because the STL Translator does not process these statements. Instead, they appear in the Preprocessor output because this is where the Preprocessor flags errors.

A cross-reference report also appears in the Preprocessor output, indicating that NOXREF was not specified when the STL Translator was run. A report summarizing the network definition options and defaults also appears, indicating that SUMMARY was specified.

```
STL LISTING                                                     TIME 12.48.45,      MAY 13, 2002   PAGE     1

WSim
# STL#   STATEMENTS
----- ----- ------------------------------------------------------------------------------------------------------------------

      00001 * @network
      00002 * @endnetwork
00000       *** MESSAGE ***
            ITP3172I NETWORK DEFINITION IS IN PREPROCESSOR OUTPUT BELOW

      00003 * @program=myprog
      00004 * mytest: msgtxt
            MYTEST   MSGTXT   STLMEM=MYPROG
      00005 * do i = 1 to 10
00001                SET      DC1=1
            $LA1     LABEL
00002                IF       WHEN=IMMED,
                              LOC=DC1,TEXT=10,COND=LE,
                              THEN=B-$LA2
00003                BRANCH   LABEL=$LA3
            $LA2     LABEL
      00006 *    transmit using pf8
00004                PF8
00005                STOP
      00007 *    if i = 5 then
00006                IF       WHEN=IMMED,
                              LOC=DC1,TEXT=5,COND=EQ,
                              THEN=B-$LA5
00007                BRANCH   LABEL=$LA6
            $LA5     LABEL
      00008 *       signal 'MYEVENT'
00008                EVENT    SIGNAL=MYEVENT
      00009 * end
            $LA6     LABEL
00009       $LA4     SET      DC2=DC1
00010                SET      DC1=+1
00011                IF       WHEN=IMMED,
                              LOC=DC1,TEXT=DC2,COND=LT,THEN=B-$LA3
00012                BRANCH   LABEL=$LA1
            $LA3     LABEL
      00010 * endtxt
00013                ENDTXT
STL LISTING        MYPROG                                       TIME 12.48.45,      MAY 13, 2002   PAGE     2
VARIABLE DICTIONARY
NAME                               USE      CLASS    MAPPING            DEFINED    REFERENCED ON STATEMENT NUMBER
--------------------------------   -------- -------- ------------------ -------    ----------------------------------------------

I                                  INTEGER  UNSHARED DC1                           5, 7
MYPROG                             PROGRAM           MYPROG                   3

MYTEST                             PROC              MYTEST                   4

(WORK VARIABLE)                                      DC2

LARGEST DEVICE COUNTER USED  = 2
LARGEST NTWRK COUNTER USED   = 0
LARGEST DEVICE SAVEAREA USED = 0
LARGEST NTWRK SAVEAREA USED  = 0
LARGEST DEVICE SWITCH USED   = 0
LARGEST NTWRK SWITCH USED    = 0

LARGEST IF NUMBER USED       = 0
STL LISTING        MYPROG                                       TIME 12.48.45,      MAY 13, 2002   PAGE     3
EVENT DICTIONARY
NAME                               USE     REFERENCED ON STATEMENT NUMBER
--------------------------------   ------  -------------------------------------------------------------------------------------

'MYEVENT'                          SIGNAL  8
```

Figure 5. Sample printed listing from STL translator, part 1 of 2

```
WSim PREPROCESSOR OUTPUT                                                TIME 12.48.45,      MAY 13, 2002   PAGE     4

   LINE    STMT    ---------1---------2---------3---------4---------5---------6---------7-

      1            mynet    ntwrk uti=100,bufsize=10000,delay=f(1),logdsply=both,
      2                           conrate=yes,mlog=yes,msgtrace=yes,
      3                           options=(debug,moncmnd),stltrace=yes
      4            slu      path  mytest
      5            myappl   vtamappl
      6            appl01   lu    lutype=lu0,path=(slu),init=sec,
      7                           dlogmod=s3270,thktime=immed
      8            * 00001 * @network
      9            * 00002 * @endnetwork
     10            * 00003 * @program=myprog
     11            * 00004 * mytest: msgtxt
     12            MYTEST   MSGTXT   STLMEM=MYPROG
     13            * 00005 * do i = 1 to 10
     14       1             SET      DC1=1
     15            $LA1     LABEL
     16       2             IF       WHEN=IMMED,
     17                              LOC=DC1,TEXT=10,COND=LE,
     18                              THEN=B-$LA2
     19       3             BRANCH   LABEL=$LA3
     20            $LA2     LABEL
     21            * 00006 *    transmit using pf8
     22       4             PF8
     23       5             STOP
     24            * 00007 *    if i = 5 then
     25       6             IF       WHEN=IMMED,
     26                              LOC=DC1,TEXT=5,COND=EQ,
     27                              THEN=B-$LA5
     28       7             BRANCH   LABEL=$LA6
     29            $LA5     LABEL
     30            * 00008 *       signal 'MYEVENT'
     31       8             EVENT    SIGNAL=MYEVENT
     32            * 00009 * end
     33            $LA6     LABEL
     34       9    $LA4     SET      DC2=DC1
     35      10             SET      DC1=+1
     36      11             IF       WHEN=IMMED,
     37                              LOC=DC1,TEXT=DC2,COND=LT,THEN=B-$LA3
     38      12             BRANCH   LABEL=$LA1
     39            $LA3     LABEL
     40            * 00010 * endtxt
     41      13             ENDTXT
CROSS-REFERENCE REPORT                                                      TIME 12.48.45,      MAY 13, 2002   PAGE     5

NAME     (DECK)      TYPE         DEFINED    REFERENCED ON LINE NUMBER
------------------  -----------  -------    ---------------------------------------------------------------------------------------

$LA1     (MYTEST)    LABEL            15    38

$LA2     (MYTEST)    LABEL            20    18

$LA3     (MYTEST)    LABEL            39    19, 37

$LA4     (MYTEST)    LABEL            34

$LA5     (MYTEST)    LABEL            29    27

$LA6     (MYTEST)    LABEL            33    28

DC1                  COUNTER                14, 17, 26, 34, 35, 37

DC2                  COUNTER                34, 37

MYEVENT              ON EVENT               31

MYTEST               DECK NAME        12    4

NTWRKUTI             UTI NAME          1

ITP3181I          22,704 BYTES ARE REQUIRED FOR THIS NETWORK

ITP3182I            1 BLOCKS OF TEXT DATA REQUIRED FOR THIS NETWORK

ITP3179I MEMBER ADDED TO DATA SET
ITP3179I NAME = INITDD(MYNET   )
```

Figure 6. Sample printed listing from STL Translator, part 2 of 2

Notice that the translator uses labels beginning with $LA. Thus, you cannot use $LA at the beginning of labels or names in an STL program. Refer to , SC31-8947 for more detail on reading the Preprocessor output part of the printed listing.

MSGDD partitioned data set members

The MSGDD partitioned data set contains message generation decks and user tables that are executed by WSim during a simulation. This data set is defined to the operating system on the MSGDD DD statement when you run the translator and WSim.

When you run the STL Translator, it automatically adds your translated STL procedures and user tables to the MSGDD data set unless you code the NOPDSOUT execution parameter or errors are found during the translation.

Each STL procedure processed by the STL Translator is added to the MSGDD data set as a separate member. Members are also added for each MSGUTBL group included in the STL source data set. Member names are the same as the names coded on the MSGTXT or MSGUTBL statements and must be unique.

Like the printed listing, the MSGDD members created include STL source lines as comments and the message generation statements generated by the STL Translator. You can eliminate the STL source lines by using the NOSOURCE execution parameter.

Note: MSGDD members created by the STL Translator are executable message generation decks. You do not need to preprocess these decks before using them in a simulation.

If you specify the PROGRAM execution parameter when running the translator or code @PROGRAM statements in your STL input, the STL Translator creates MSGDD data set members containing the records that correlate the STL statements in your programs with the message generation statements they generate. (The PROGRAM execution parameter specifies that trace information is to be stored for the first program defined in your input data. Code the @PROGRAM statement for any additional programs for which you want trace information.) These correlation records must be available if you want to use the STL trace facility or if you want to see the STL statement numbers with the Q (Query) operator command. Also, you must code STLTRACE=YES in your network definition. See “Obtaining STL trace records” on page 341 for more information about the STL trace facility.

These MSGDD data set members enable the Loglist Utility to produce STL trace records that refer to STL variable names and statement numbers. This information is essential when debugging STL programs.

INITDD partitioned data set members

The INITDD partitioned data set contains network definitions used by WSim during a simulation. You define this data set to the operating system on the INITDD DD statement when you run the translator and WSim.

When you include a network definition in your input data to the STL Translator, it invokes the Preprocessor, which verifies and stores each network in the INITDD data set. If you specify the NOPREP execution parameter, the STL Translator does not invoke the Preprocessor. Thus, no members are stored in the INITDD data set.

The Preprocessor only stores members in the INITDD data set when it successfully preprocesses the entire network definition.

Sequential output data set

The STL Translator creates a sequential output data set that you define to the operating system on the SEQOUT DD statement. It contains only the message generation decks translated from your STL programs. You could merge this data set with a network and then use the Preprocessor to place the members in the MSGDD and INITDD. This may be useful if you want to transfer the translated STL programs to another system.

A SEQOUT DD for this data set is required unless you code the NOSEQOUT execution parameter when invoking the translator. Like the printed listing and MSGDD, this data set contains the STL source lines with statement numbers as comments and the message generation statements generated by the STL Translator. You can eliminate the STL source lines by using the NOSOURCE execution parameter.

Once you debug your STL programs, the message generation statements included in this data set are ready to be combined with a network definition and processed by the Preprocessor. This data set **should not** be changed because the STL trace facility depends upon the order of the statements.

Note: Message generation decks in the sequential (SEQOUT) data set cannot be executed by WSim without first being preprocessed.

Temporary work data sets

The STL Translator uses three temporary partitioned data sets that you define to the operating system on the SYSUT1, SYSUT2, and SYSUT3 DD statements.

The translator uses SYSUT1 set as a workspace for storing STL procedures and user tables. Make sure that the data set you define is large enough to contain all procedures and user tables translated.

SYSUT2 and SYSUT3 are only needed if you include network definition statements in your STL input data set. The Preprocessor uses these data sets as workspace for storing network definition statements and message generation statements. You need SYSUT2 only if you code NTWRK statements in your network definition. SYSUT2 must be large enough to contain all the networks you define. You need SYSUT3 only if you code MSGTXT or MSGUTBL statements in your network definition. SYSUT3 must be large enough to contain all the message generation decks and user tables you define in your network definition.

Running the STL Translator

You can run the STL Translator using JCL on MVS or a CLIST or EXEC under TSO or the WSim/ISPF Interface. The name of the STL Translator program is ITPSTL.

Using STL Translator execution parameters

You can code the following execution parameters to run the STL Translator:

Note: When execution parameters are included in the PARMDD data set, they are overridden by those passed when ITPSTL is called.

NOSOURCE
When you specify NOSOURCE, STL source statements are not included in your MSGDD or SEQOUT data sets. If this parameter is not specified, STL source statements are printed and written to the output data sets as comments in the message generation statements.

NOWSIM
When you specify NOWSIM, the translated message generation statements are not included in the printed listing. Only the STL source statements are included in your printed listing. This can be useful when you are debugging the syntax of your STL programs.

NOSEQOUT

When you specify NOSEQOUT, the STL Translator does not create the sequential data set. Otherwise, the STL Translator creates a sequential output data set to contain all message generation decks and user tables translated from your STL programs. This does not include any message generation statements coded in the network definition.

NOPDSOUT

When you specify NOPDSOUT, the STL Translator does not place any message generation decks or user tables translated from your STL programs into the MSGDD partitioned data set.

Notes:

- If you specify NOPDSOUT, variable dictionaries, which are required by the STL trace facility and the Q (Query) command to display STL statement numbers, are not available. To create variable dictionaries, you must either specify the PROGRAM= parameter when you run the translator or code the @PROGRAM statement in your STL input data set, and *not* specify NOPDSOUT.
- If you do not specify NOPREP and you code a network definition, the Preprocessor may store members in the MSGDD partitioned data set while running.
- If you do not specify NOPREP and you code a network definition, the Preprocessor will fail if it expects a translated message deck or user table to be in the MSGDD partitioned data set that does not exist.

NOIMPLICIT

When you specify NOIMPLICIT as an execution parameter for STL, implicit definitions of BIT, INTEGER, or STRING variables are considered errors and are flagged with a new error message (ITP3206I). Refer to , SC31-8951 for a detailed description of this error message. Implicit definitions are most commonly found in an assignment statement where a previously unused and undefined variable is assigned a value of a particular type. They can also be encountered when previously unused variables are used as output variables on CPI-C STL statements or on the UTBLSCAN function.

If NOIMPLICIT is not specified, implicitly defined variables are assigned a type based on the context of the first usage of the variable.

Variables are explicitly defined by using ALLOCATE statements or BIT, INTEGER, or STRING statements.

The NOIMPLICIT option is particularly useful in avoiding inadvertent resource conflicts when STL programs are developed separately using a common set of variables distributed in a common member which is included by all such programs.

PROGRAM=name

Specifying the PROGRAM=*name* execution parameter associates the given program name with the translated STL source statements for the first program defined in the input data set. This program name appears in the title of the printed listing created by the STL Translator. In addition, this program name is assigned to the member of the MSGDD partitioned data set that contains the variable dictionary and statement correlation records. The name is not assigned if you have not requested MSGDD partitioned data set members (that is, you specified the NOPDSOUT execution parameter).

You can also specify program names using the STL @PROGRAM statement. If you specify a name on an STL @PROGRAM statement for the first program defined in the input data set and code a different name on the PROGRAM execution parameter, the PROGRAM execution parameter overrides the @PROGRAM statement.

Note that the name you use must be different from all procedure and MSGUTBL names contained in the MSGDD data set.

PRTLNCNT=nnn
Use the PRTLNCNT parameter to specify the maximum number of lines to be printed on an output page before ejecting to a new page. The value for *nnn* is an integer from 35 to 255. The default value is 60.

NOLIST
You can use the NOLIST execution parameter when you include a network definition in your STL input. When you specify NOLIST, no Preprocessor output is listed in the SYSPRINT data set. However, if errors are detected while preprocessing, these errors are still shown.

SUMMARY
You can use the SUMMARY execution parameter when you include a network definition in your STL input. When you specify SUMMARY, a report summarizing the network definition options and defaults is listed in the SYSPRINT data set.

XREF You can use the XREF execution parameter when you include a network definition in your STL input. When you specify XREF, a cross-reference report is listed in the SYSPRINT data set for each network you define, unless you also specified the NOLIST execution parameter. XREF is the default.

NOXREF
You can use the NOXREF execution parameter when you include a network definition in your STL input. When you specify NOXREF, a cross-reference report is not listed in the SYSPRINT data set for any network you define.

NOPREP
You can use the NOPREP execution parameter when you include a network definition in your STL input. When you specify NOPREP, your network definition statements are not validated or added to the MSGDD and INITDD data sets.

NOTRAN
You can use the NOTRAN execution parameter when you include a network definition in your STL input data set. When you specify NOTRAN, the STL Translator only verifies the @NETWORK and @ENDNETWORK statements and ignores the rest of the STL input data set. The STL Translator invokes the Preprocessor to verify the network definition and add members to the MSGDD and INITDD data sets.

Using JCL to run the STL Translator

The example below shows JCL that you can use to run the STL Translator (member STLJOB in the SAMPLE data set shipped with WSim).

```
//STLJOB   JOB
//*********************************************************************
//* Workload Simulator (WSim)   5655-I39                              *
//*********************************************************************
//*                     STLJOB JCL                                    *
//* Sample JCL to execute the WSim STL Translator (ITPSTL).           *
//*********************************************************************
//STL      EXEC PGM=ITPSTL,REGION=4096K
//STEPLIB  DD  DSN=WSIM.SITPLOAD,DISP=SHR
//PARMDD   DD  DSN=WSIM.PARMDD,DISP=SHR
//RATEDD   DD  DSN=WSIM.SITPRTBL,DISP=SHR
//INITDD   DD  DSN=WSIM.TESTFILE,DISP=SHR
//SYSPRINT DD  SYSOUT=A
//MSGDD    DD  DSN=WSIM.MSGFILE,DISP=SHR
//SEQOUT   DD  DSN=WSIM.STL.SEQOUT,DISP=SHR
//SYSLIB   DD  DSN=WSIM.STLIN,DISP=SHR
//SYSUT1   DD  UNIT=SYSDA,SPACE=(TRK,(10,10,3))
//SYSUT2   DD  UNIT=SYSDA,SPACE=(TRK,(10,10,3))
//SYSUT3   DD  UNIT=SYSDA,SPACE=(TRK,(10,10,3))
//* member_name = name of member with STL statements to translate
//SYSIN    DD  DSN=WSIM.STLIN(member_name),DISP=SHR
```

Using a TSO CLIST to run the STL Translator

The following example displays a TSO CLIST that you can use to run the STL Translator (member STL in the CLIST data set shipped with WSim).

```
/*******************************************************************/
/* Workload Simulator (WSim)   5655-I39                            */
/*******************************************************************/
/*                        STL CLIST                                */
/* Sample CLIST to execute the WSim STL Translator (ITPSTL).       */
/*                                                                 */
/* Format:  STL member_name                                        */
/*******************************************************************/
PROC 1 MEMBER_NAME
CONTROL NOMSG
FREE  DDNAME(SYSPRINT SYSIN MSGDD INITDD RATEDD PARMDD -
             SYSLIB SYSUT1 SYSUT2 SYSUT3)
CONTROL MSG
ALLOC DDNAME(SYSPRINT) DATASET('WSIM.STLPRINT') SHR
ALLOC DDNAME(SYSIN)    DATASET('WSIM.STLIN(&MEMBER_NAME)') SHR
ALLOC DDNAME(MSGDD)    DATASET('WSIM.MSGFILE')  SHR
ALLOC DDNAME(INITDD)   DATASET('WSIM.TESTFILE')  SHR
ALLOC DDNAME(RATEDD)   DATASET('WSIM.SITPRTBL')  SHR
ALLOC DDNAME(PARMDD)   DATASET('WSIM.PARMDD')  SHR
ALLOC DDNAME(SYSLIB)   DATASET('WSIM.STLIN') SHR
ALLOC DDNAME(SYSUT1)   NEW SPACE(10,10) TRACKS DELETE -
                       DIR(3) UNIT(SYSDA)
ALLOC DDNAME(SYSUT2)   NEW SPACE(10,10) TRACKS DELETE -
                       DIR(3) UNIT(SYSDA)
ALLOC DDNAME(SYSUT3)   NEW SPACE(10,10) TRACKS DELETE -
                       DIR(3) UNIT(SYSDA)
CALL  'WSIM.SITPLOAD(ITPSTL)' 'NOSEQOUT,NOWSIM'
WRITE STL RETURN CODE IS &LASTCC
FREE  DDNAME(SYSPRINT SYSIN MSGDD INITDD RATEDD PARMDD -
             SYSLIB SYSUT1 SYSUT2 SYSUT3)
END
```

Using the WSim/ISPF Interface

You can also run the STL Translator by way of the WSim/ISPF Interface. To do this, follow these steps:

1. Invoke the WSim/ISPF Interface main panel from ISPF. The method you use to do this depends on how the WSim/ISPF Interface application is installed at your site. If you are not sure how to do this, see your system programmer for assistance.
2. Select option 1 from the WSim/ISPF Interface main panel and press **Enter**. The Process Networks and STL Programs panel is displayed.

Note: You can also type “STL” on the WSim/ISPF Interface main panel command line and press **Enter** to display this panel.

3. Fill in the appropriate fields on this panel and press **Enter**.

For more information about the WSim/ISPF Interface, refer to , SC31-8947.

Data set requirements

The following data sets are used to run the STL Translator:

SYSIN	Defines the data set containing your STL programs. You may also include a network definition in this data set.
SYSPRINT	Defines the output printer.
SEQOUT	Defines the sequential data set to contain the message generation decks translated from your STL programs. This data set must be defined unless the NOSEQOUT or NOTRAN option is coded.
MSGDD	Defines the partitioned data set to contain the translated message generation decks, user tables, and STL statement correlation records. If you include a network definition in your STL input and you code message generation decks within it, they also appear in the MSGDD data set. This data set must be defined unless the NOPDSOUT option (and not NOTRAN) is coded and you did not include message generation decks or user tables in your network while omitting the NOPREP option. This data set can be the same as the INITDD data set.
SYSUT1	Defines a temporary partitioned data set used as a work space for storing the translated STL programs. It must be defined if you are using JCL or running under TSO and you do not specify NOTRAN.
SYSUT2	Defines a temporary partitioned data set used as a work space for storing network definition statements. This data set is required only if you include a network definition in your STL input containing at least one NTWRK statement and you are not running the STL Translator with the NOPREP execution parameter. It must be defined if you are using JCL or running under TSO.
SYSUT3	Defines a temporary partitioned data set used as a work space for storing message generation decks and user tables. This data set is required only if you include a network definition in your STL input containing at least one MSGTXT or MSGUTBL statement and you are not running the STL Translator with the NOPREP execution parameter. It must be defined if you are using JCL or running under TSO.
STEPLIB	Defines a data set containing the WSim host processor load modules and, optionally, any user exit data sets.
RATEDD	Defines an optional rate tables data set. This is required only if your network contains a RATE statement and the network definition is included in the STL input data set.
INITDD	Defines a partitioned data set containing the preprocessed network definition. This data set is required only if you include a network definition in your STL input and you are not running the STL Translator with the NOPREP execution parameter. This data set can be the same as the MSGDD data set.

PARMDD	Defines an optional sequential data set containing the execution parameters. The following syntax rules apply to the records in this data set: • An asterisk (*) in column 1 denotes a comment record. • One or more parameters may be coded on each record, delimited by commas. • Any data following a trailing blank is considered a comment. • Leading blanks are allowed. • A trailing comma is not required to indicate continuation of parameters on the next record. The BLKSIZE for this data set must be a multiple of 80. This data set is optional.
SYSLIB	Defines an optional partitioned data set whose members contain data to be included in the STL input data set. This data set is required if you code the @INCLUDE statement in your STL input data set.

STL Translator return codes

At the end of execution, the STL Translator sets a return code to indicate the status of the execution. Execution ends prematurely for all return codes except 0. The STL Translator can return the following codes:

Code	Meaning
0	Execution completed successfully.
4	An invalid execution parameter was specified.
8	The SYSPRINT data set failed to open.
12	The SYSIN data set failed to open.
16	The MSGDD, INITDD, SEQOUT, or a work data set failed to open.
20	An error occurred while writing to an output data set.
24	An error occurred while translating the STL program or preprocessing the network definition.
28	The trace correlation member of the MSGDD data set failed to open.
32	Not enough storage was available to run.
36	An error occurred while storing a partitioned data set member.

Chapter 23. Combining STL programs and network definitions

This chapter explains how to:
- Include network definitions in your STL programs
- Structure STL programs
- Reference STL programs in your network definition
- Combine STL procedures from different STL programs.

Including network definition statements in STL

As you create your STL programs, you can facilitate development of your entire simulation by including your network definition in the same data set. In this way, you can easily refer to your network definition as you are creating the messages that will run through it. You will also find it easier to debug and modify your simulation using this method.

To define your networks in STL, you must code network definition statements exactly as you would if they were input to the Preprocessor. This means that you **may not** code STL comments within the network definition. You can only code scripting language comments. Refer to , SC31-8945 for more detailed information on coding the network definition.

You must code the STL @NETWORK and @ENDNETWORK statements to include the network definition in your STL input data set. You must place them around the statements. Also, you cannot code any STL statements after the @NETWORK statement on the same line. However, you may code STL comments there. You must code the @ENDNETWORK statement in column one so that the STL Translator can detect it. Code only one block of network definition statements in your input data set and place it at the top of the data set following any STL comments. You may define multiple networks within a single network definition block.

An example of an STL input data set including network definition statements is shown below.

```
/* My STL Input Data Set */
@network
netx     ntwrk     uti=100,delay=f2,stltrace=yes,mlog=yes
apath    path      amsg
wsim1    vtamappl  bufsize=3000
* LU Definition
wsim1lu  lu        lutype=lu2,display=(24,80),init=sec,path=(apath)
@endnetwork
@program=amsgx
amsg:    msgtxt
         say 'Hi'
         opcmnd 'zend'
         quiesce
         endtxt
```

Note: You can abbreviate the @NETWORK and @ENDNETWORK statements as @NET and @ENDNET.

The STL Translator also invokes the Preprocessor to validate your network definition. If you have not made any errors, the STL Translator stores your networks into the INITDD output data set and any other message generation decks you may have coded along with them into the MSGDD data set. It replaces any members that already exist in these data sets by the same name.

If you made errors when creating your STL programs, the STL Translator does not add or replace those members in the MSGDD data set. However, it will try to preprocess the network definition unless you specify the NOPREP execution parameter. If you made any errors in the network definition, the STL Translator does not add or replace any members in the output data sets.

Structuring STL programs

The following topics give you guidelines on how to structure your STL programs.

Organizing your STL programs

In your simulation, you will probably want to create many STL programs that each perform a unique function. The simplest STL program you can create consists of a single STL procedure beginning with a MSGTXT statement and ending with an ENDTXT statement. However, you will usually code STL programs that include multiple procedures.

The best way to structure a program is to code one main procedure that calls the other procedures defined in the program. Here, you can have multiple interrelated procedures whose activity can be coordinated by one main procedure.

Coding multiple programs in one STL input data set

You should code all the procedures for one program together in one STL input data set. You can even code multiple programs in one STL input data set. If you also want to include the network definition with your STL programs, you can write your entire simulation in one data set.

To place multiple STL programs in the same data set, you must code the @PROGRAM and @ENDPROGRAM statements to separate the programs. The STL Translator does not accept statements between programs, although you may code comments.

For each program, code the @PROGRAM statement before all the variable declarations and procedures that make up the program. The STL Translator considers the @PROGRAM statement to be the first statement in a new program. Code the @ENDPROGRAM statement as the last statement in the program.

You can also use the @PROGRAM statement to specify a name to be used to store trace information for a program. See “Obtaining STL trace records” on page 341 for more information about the STL trace facility.

If you have only one program in your STL input data set, you do not need to code the @PROGRAM and @ENDPROGRAM statements. You also do not need to code the @ENDPROGRAM statement for the last program in the data set.

To help you write your simulation in STL, first think of what you want to simulate in terms of separate tasks. Each task can be thought of as a separate STL program. This also lets you maintain a library of STL programs that you can use for other simulations.

Another reason to organize your simulation into multiple programs is that it lets you reuse variable, constant, and label names. This does not include procedure, user table, or program names.

When coding multiple programs in your simulation, you must be careful to initialize the variables before you use them since the resources are shared among programs.
For example, the first program in Figure 7 declares an integer variable “i” that is

```
         STL                             WSim Message Decks

 @program=first
 integer i
 abc:  msgtxt                   ABC      MSGTXT   STLMEM=FIRST
       i=5                               SET      DC1=5
       endtxt                            ENDTXT
 @endprogram
                      -------->
 @program=second
 integer j
 string  s
 def:  msgtxt                   DEF      MSGTXT   STLMEM=SECOND
       s = char(j)                       DATASAVE AREA=1,
                                                  TEXT=($CNTR,DC1$)
       endtxt                            ENDTXT
 @endprogram
```

Figure 7. STL input data set translated to message generation statements

assigned to device counter 1 (or DC1) by the STL Translator. It assigns a value of 5 to “i” that, at runtime, places a value of 5 into DC1. The second program also declares an integer variable but calls it “j”. This program uses the value of “j” before initializing it in the statement “s = char(j)”. As you can see in the corresponding DATASAVE statement, “j” is also referred to as DC1.

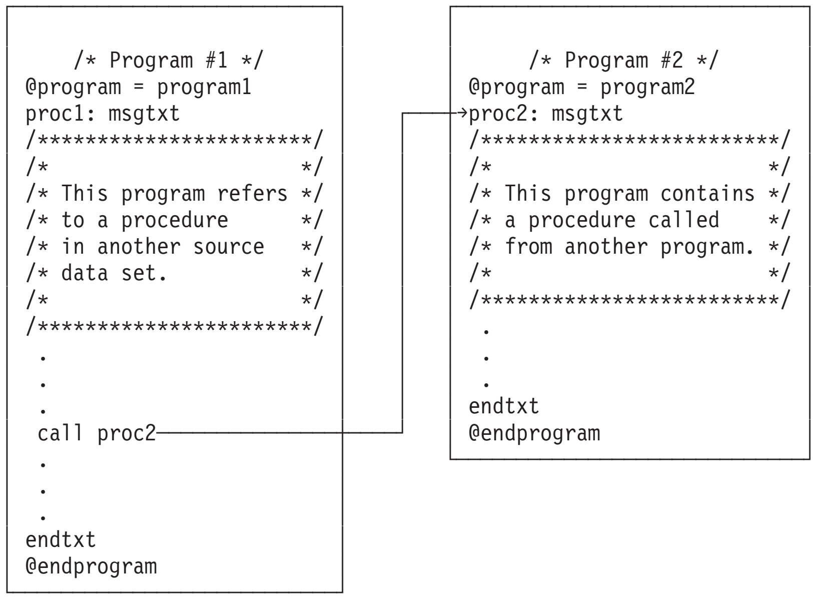
If a device in the simulation executes procedure DEF immediately after ABC, you may expect that “s” would be set to character “0” just by looking at the STL statements. However, since DC1 is used for both variables “i” and “j”, it would still be set to 5 upon entering procedure DEF. So “s” is assigned character “5”.

Therefore, **always initialize variables if you have multiple programs in your simulation**.

Avoiding misuse of procedure calls

As a general rule, an STL procedure in one program should not call a procedure in another program. To manage asynchronous conditions, the STL Translator assigns IF numbers to message generation statements. These numbers are not unique from program to program. Calling a procedure located in a different source data set can produce an unwanted and unexpected conflict of these identifiers. Figure 8 on page 328 shows a situation that should be avoided. See “Combining STL procedures from different STL programs” on page 333 for information on how to combine programs that must be translated separately.

```
*******************
*                 *
* DO NOT DO THIS! *
*                 *
*******************
```



```
    /* Program #1 */
@program = program1
proc1: msgtxt
/***********************/
/*                     */
/* This program refers */
/* to a procedure      */
/* in another source   */
/* data set.           */
/*                     */
/***********************/
 .
 .
 .
 call proc2
 .
 .
 .
endtxt
@endprogram
```

```
     /* Program #2 */
@program = program2
proc2: msgtxt
/*************************/
/*                       */
/* This program contains */
/* a procedure called    */
/* from another program. */
/*                       */
/*************************/
 .
 .
 .
endtxt
@endprogram
```

Figure 8. Example of misuse of calls in separate STL programs

Naming programs, procedures, and user tables

WSim accesses the MSGDD partitioned data set when it needs message generation decks, user tables, and trace correlation records. You can generate message generation decks and user tables by running the STL Translator on your STL procedures and user tables. You can also code program names to let the translator produce trace correlation records. The names of the members placed in the MSGDD partitioned data set are the same as those that you code for STL procedures, user tables, and program names. Therefore, you must code unique names to produce unique members in the data set.

When you code procedures, user tables, and programs in your STL input with the same names as members in the MSGDD data set, the STL Translator overwrites those members as it translates your input. If you code your simulation in several different STL input data sets, you must run the STL Translator several times. Be sure that all procedure, user table, and program names are unique.

Finally, you can repeat label names used on STL statements other than MSGTXT and MSGUTBL in multiple programs without affecting the MSGDD data set or program execution.

Referencing STL programs in your network definition

A network definition defines resources and operation options for your simulated network. Your network definition statements tell WSim which STL programs generate messages for each simulated resource. As explained in , SC31-8945, you can specify a set of message generation decks or STL procedures for an entire network, or you can specify a different sequence for each line, terminal, and device.

If you create your STL programs as recommended previously, they each perform independent tasks that are part of the whole simulation. Also, they each have one main procedure call the other procedures defined in the program.

To refer to a program in a network definition statement, just code the main procedure name.

The following statements and operands let you specify where STL programs are to be used and other requirements for your network:

- **PATH statements** in your network definition list the main procedures representing STL programs to be used by each resource in your simulated network. For each simulated resource, you can code a PATH statement to specify the main procedure names representing the STL programs to be executed by the resource and the order of execution. The programs are executed repeatedly until the simulation is ended. See , SC31-8945 for more information about how the procedures (message generation decks) can be listed.
- The **PATH operand** specifies which PATH statement defines the main procedures representing the STL programs to be used for a particular resource.
- **INCLUDE statements** in your network definition let you include any procedures used by the network during the simulation run. You need to code an INCLUDE statement for any procedure used in your simulation that is not either included in a PATH statement or explicitly called or executed within your program. For example, this can be procedures called on a new path the operator specifies by issuing the A (Alter) operator command.
- The **FRSTTXT operand** defines the main procedure representing an STL program to be used as the first program in a simulation. It is used only once and not repeated. Typically this procedure contains your logon logic.
- The **ATRDECK operand** specifies the name of the main procedure representing an STL program to be invoked automatically to provide automatic terminal recovery.

Note: Be sure that the CNTRS operand is not coded in your network definition. This operand can cause problems when using STL programs.

The PATH and INCLUDE statements are coded in your network definition. The PATH and FRSTTXT operands are coded on the DEV, TP, or LU statement defining the resource for which they are used or on higher-level statements such as NTWRK, VTAMAPPL, TCP/IP, and APPCLU statements. The ATRDECK operand is coded on the DEV or LU statement.

See , SC31-8945 for a more complete explanation of these statements and operands. You should be familiar with these books so that you are aware of the range of possibilities available to you when designing the messages to be simulated with your STL programs.

The example below shows a portion of a network definition.

```
testnet  ntwrk   uti=100,bufsize=6000,delay=f(5),
                 stltrace=yes,mlog=yes,logdsply=both
mypath   path    test1,coffee
*                ***** ******
appl1    vtamappl
mylu     lu      lutype=lu2,path=(mypath),init=sec,maxsess=(0,1),
                 display=(24,80),frsttxt=logon
```

In the network definition shown above, the LU first executes the main procedure named LOGON referenced on the FRSTTXT operand. Then it executes the programs represented by the main procedures named on the PATH statement MYPATH: TEST1 and COFFEE.

Each main procedure, representing a program, named on a path statement is called a path entry. When a represented program is named on a path entry, WSim executes that main procedure representing the program and any procedures called by that program in the course of its execution.

Suppose that you have written two STL programs, one designed to start a full-screen editor and the other designed to start a compiler. Each program consists of a single STL procedure. Figure 9 on page 331 shows what the source data sets might look like.

```
     /* Program #1 */
@program = program1
edit: msgtxt

/*************************/
/*                       */
/* This data set contains */
/* an STL program that    */
/* edits a data set.      */
/*                       */
/*************************/
 .
 .
 .
endtxt
@endprogram


     /* Program #2 */
@program = program2
compile: msgtxt

/*************************/
/*                       */
/* This data set contains */
/* an STL program that    */
/* invokes a compiler.    */
/*                       */
/*************************/
 .
 .
 .
endtxt
@endprogram
```

Figure 9. Two STL programs

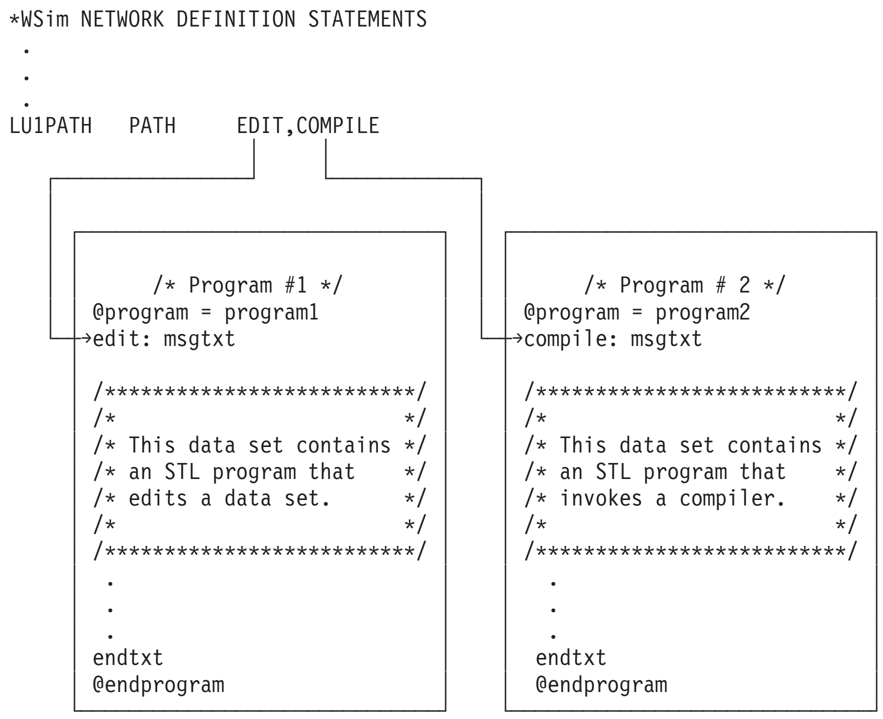
The main procedure in each program should be named on a PATH statement in your network definition. For example, if you have defined an LU named LU1 in your network definition, you can have it execute EDIT first and COMPILE second as shown in the network definition that appears in Figure 10.

The path entries specified on the PATH statement correspond to the procedure

```
*NETWORK DEFINITION STATEMENTS
*
     (This is
      the PATH --> LU1PATH  PATH     EDIT,COMPILE
      definition)  .
                   .
     (This is      .
      the LU   --> LU1      LU       PATH=LU1PATH,...
      definition)  .
                   .
                   .
```

Figure 10. Network definition to use two STL programs

names coded in the two example programs. Figure 11 on page 332 illustrates this relationship.



```
*WSim NETWORK DEFINITION STATEMENTS
 .
 .
 .
LU1PATH   PATH     EDIT,COMPILE
```

```
     /* Program #1 */
@program = program1
edit: msgtxt

/*************************/
/*                       */
/* This data set contains */
/* an STL program that    */
/* edits a data set.      */
/*                       */
/*************************/
 .
 .
 .
endtxt
@endprogram
```

```
     /* Program # 2 */
@program = program2
compile: msgtxt

/*************************/
/*                       */
/* This data set contains */
/* an STL program that    */
/* invokes a compiler.    */
/*                       */
/*************************/
  .
  .
  .
 endtxt
 @endprogram
```

Figure 11. Relationship of PATH statement and STL programs

As another example, you might design an editor program that contains three procedures: “edit”, the main procedure; “locate”, which performs an editor locate instruction; and “change”, which performs an editor change instruction. The program appears in Figure 12 on page 333.

```
       /* STL Program */
edit: msgtxt
/**********************************/
/*                                */
/* This data set contains an      */
/* STL program that edits a data  */
/* set.                           */
/*                                */
/**********************************/
 .
 .
 .
call locate
 .
 .
 .
call change
 .
 .
 .
endtxt
locate: msgtxt
 .
 .
 .
endtxt
change: msgtxt
 .
 .
 .
endtxt
```

Figure 12. STL program containing multiple STL procedures

The network definition for this program would look like the one shown below.

```
* NETWORK DEFINITION STATEMENTS
⋮
LUPATH  PATH  EDIT
```

Note that only the EDIT procedure is referenced on the PATH statement.

Combining STL procedures from different STL programs

Note: Combining procedures from multiple STL programs can be complex and the results can be difficult to maintain. For this reason, this approach is not recommended unless it is absolutely necessary (for example, your program is too large to edit as one data set). Another way to do this is to use the @INCLUDE statement. See “Including data from other data sets” on page 252 for more information.

You may need to place procedures that would normally be considered as a part of one program in more than one STL program. It may be more convenient to use separate source data sets or you may have written a large STL program that you now want to break into smaller components. In this case, you can put the STL statements for various procedures in separate data sets and use the STL Translator to translate the data sets separately.

To combine multiple programs into one program, declare and initialize all the variables used in each program in the first program that you translate.

When you translate the first program, the STL Translator includes a variable dictionary in the printed listing supplied as part of your STL Translator output. The variable dictionary indicates which save area, counter, or switch has been allocated for each variable used in that program and which are used as internal work variables (that is, there is no associated STL variable). Place ALLOCATE statements at the beginning of all other programs defining the save area, counter, or switch mapping specified by the variable dictionary. See “ALLOCATE” on page 363 for specific information about defining save areas, counters, and switches using the ALLOCATE statement.

Note: There are limits to the number of save areas, counters, and switches that can be mapped to variables for each STL program. Thus, if you exceed any of these limits, you may have to translate a large program as two or more separate programs and combine them. See “ALLOCATE” on page 363 for information about the numbers available.

See , SC31-8945 for information about how save areas, counters, and switches are used in message generation statements.

You must also ensure that IF numbers do not overlap among the various programs being combined and used as one program. Note the largest IF number used in each program. This number is found at the bottom of the variable dictionary in the printed listing. Use the @IFNUM statement at the beginning of each following program to ensure that unique IF numbers are assigned to each program.

You can code @INCLUDE statements in your STL input data set to include your common ALLOCATE statements and @IFNUM statements. See “@INCLUDE” on page 358 for more information.

The example below shows how you could code the first program to be translated.

```
/* First Program */

/*********************************************************/
/* All variables are explicitly declared below.          */
/*********************************************************/

string data_received
string message_list
integer error_count

proc1: msgtxt

/*********************************************************/
/* This procedure recognizes received error messages and   */
/* appends each one to a list.                            */
/*********************************************************/

error_count = 0
message_list = ''

onin then data_received = buffer  /* Save received data.              */
wait until onin                   /* Wait until something is received. */

if index(data_received,'ERROR') > 0 then    /* Error message received. */
   do
      error_count = error_count + 1
      message_list = message_list data_received    /* Append this      */
                                                   /* message to list. */
   end
else                                    /* Error message not received. */
   nop
endtxt
```

The following example shows a portion of the variable dictionary included in the printed listing for the first program:

The next program must include ALLOCATE statements based on the information

```
VARIABLE DICTIONARY
NAME                               USE      CLASS    MAPPING            DEFINED    REFERENCED ON STATEMENT NUMBER
--------------------------------   -------- -------- ------------------ -------    ----------------------------------------------

DATA_RECEIVED                      STRING   UNSHARED 1                        1    8, 10, 13

ERROR_COUNT                        INTEGER  UNSHARED DC1                      3    5, 12

MESSAGE_LIST                       STRING   UNSHARED 2                        2    6, 13

(WORK VARIABLE)                                      DC2


LARGEST IF NUMBER USED       = 2
```

included in the variable dictionary for the preceding program. These statements ensure that the same save areas, counters, and switches are used identically for each program.

The next program must also include an @IFNUM control statement. The example below shows how you could code the next program to be used in your combined program.

```
/* Next Program */

allocate data_received '1'
allocate error_count   'DC1'
allocate message_list  '2'
allocate workvar_dc2   'DC2'
@ifnum = 3     /* Begin numbering IF statements with 3. */
proc2: msgtxt
/************************************************************/
/* This procedure tells the operator how many error messages */
/* were received and logs the error message list.           */
/************************************************************/
say 'Total number of error messages received is' char(error_count)
log message_list
endtxt
```

Note: Remember that combining procedures from multiple STL programs into one program causes maintenance problems. For instance, if you change “proc1” in the previous example, you must ensure that any resulting changes in the STL variable dictionary or IF numbers are also reflected in “proc2”.

Chapter 24. Debugging your STL programs

When you run the STL Translator against your STL input data set, it checks for errors that prevent execution, such as incorrect syntax. You can find the errors you need to correct in the printed listing.

Once you correct these errors and rerun the translator, WSim can use the networks and the resulting message generation decks during a simulation. You may find that your simulation does not perform as you expect. This can be because of incomplete or faulty logic.

To identify these types of errors, you can use the Loglist Utility to examine the results of your simulation run. This procedure is described in , SC31-8947.

You can use the Loglist Utility to obtain information specific to your STL programs. "Finding and correcting STL Translator syntax errors" explains how to detect syntax errors in your STL programs.

The Loglist Utility can display various types of records, depending upon what you request when you run it. The records that trace activity caused by STL statements are called STL trace records (labeled STRC as the record type in the output). They are similar to the message trace records (MTRC) provided by the Loglist Utility for message generation statements.

This chapter explains how to find and correct errors identified during translation and how to trace program logic and correct logic errors in your programs.

Finding and correcting STL Translator syntax errors

If you code a network definition in your STL input data set and you run the STL Translator without the NOPREP execution parameter, the STL Translator invokes the Preprocessor to verify your network definition. You can determine if you made errors when coding your STL programs and network definition by examining the return code from the STL Translator. A return code of 24 indicates that you made a syntax error in one of your programs or in your network definition. Any other return code besides 0 indicates that some other error occurred, for example, your data set is full.

Reading error messages

To identify any errors you made when coding your STL programs or network definition, examine either the printed listing or the sequential output data set (if you requested one) for error messages. Errors in your STL programs are prefixed by ITP3*xxx*I, where 3*xxx* is the number of the error. Some ITP3*xxx*I messages are not errors, but are informational messages that may be output during the translation of your programs and after the preprocessing of your network definition.

Errors in your network definition appear in the Preprocessor Output section of your printed listing. The format of this section is the same as if you ran the Preprocessor yourself with the network definition as input. These errors are prefixed by ITP12*xx*I or ITP13*xx*I.

Explanations of the error messages can be found in , SC31-8951. Refer to , SC31-8947 for more details on reading the Preprocessor Output listing.

Errors detected by the translator indicate the element of the statement that is in error, as shown in the following example:

```
 WSim# STL#   STATEMENTS
 ----- ----- --------------------------------------------------------------
             .
             .
             .
       00006 * COUNT = COUNT + 1
             *** ERROR ***
             ITP3044I A VARIABLE OF UNKNOWN TYPE IS USED IN AN EXPRESSION
             ITP3044I NAME = COUNT
```

In this example, the program uses the variable “count” in an expression before the translator knows its type. The error message gives a brief explanation of the problem and identifies the variable that caused the error. Error messages are described in , SC31-8951 along with suggestions for correction.

Some error messages do not specify the statement element that caused the error. In the following example, the translator cannot identify a single element that is in error.

```
 WSim# STL#   STATEMENTS
 ----- ----- --------------------------------------------------------------
             .
             .
             .
       00005 * COUNT = 88
       00006 * MYDATA = 'MARK'
       00007 * COUNT = MYDATA
             *** ERROR ***
             ITP3047I TYPE MISMATCH
```

In this example, the variable “count” has previously been declared as an integer variable and the variable “mydata” has been declared as a string variable. STL does not permit assignment of a string value to an integer variable. The translator detects a type mismatch error.

Using the variable dictionary to find errors

You can use the variable dictionary, which is included at the end of each STL program listing produced by the STL Translator, to diagnose some errors. It may not be apparent why an error such as type mismatch occurs. An examination of the variable dictionary reveals the following information:
The variable dictionary shows that you used “count” as an integer variable, but

```
VARIABLE DICTIONARY
NAME                                USE      CLASS    MAPPING            DEFINED    REFERENCED ON STATEMENT NUMBER
--------------------------------    -------- -------- ------------------ -------    -----------------------------------------------
COUNT                               INTEGER  UNSHARED DC2                           5, 7
MYDATA                              STRING   UNSHARED 2                             6, 7
⋮
```

you used “mydata” as a string variable. The type mismatch error is explained.

Reading the variable dictionary

The variable dictionary contains six columns of information:

- NAME
- USE
- CLASS

- MAPPING
- DEFINED
- REFERENCED ON STATEMENT NUMBER.

The NAME column lists the names of all variables used in the STL program.

Note: The name “(WORK VARIABLE)” does not represent an STL variable. It lists internal resources (save areas, counters, and switches) used during the translation. You can use this when you combine STL procedures from different STL programs. For these names, only the MAPPING column contains a value.

The USE column lists how each variable is used by the STL program. Valid values for the USE column are:

BIT	Bit variable.
EXECPROC	Execute procedure.
INLABEL	Onin label.
INT CON	Integer constant.
INTEGER	Integer variable.
IO LABEL	DEACT ALL IO ONS statement.
LABEL	Any label that is not a procedure, user table, onin, or onout label.
MSGUTBL	User table.
OUTLABEL	Onout label.
PROC	Procedure.
PROGRAM	Trace information and querying of STL statement numbers.
STR CON	String constant.
STRING	String variable.
?	STL cannot determine the type for this variable. See the error messages in the printed listing to correct this.

The CLASS column lists the variable classes associated with each variable name. Valid values for the CLASS column are:

(blank)	This variable has no class associated with it.
SHARED	This is a shared variable.
UNSHARED	This is an unshared variable
?	The class of this variable cannot be determined by the STL Translator. See the error messages in the printed listing to correct this.

The MAPPING column lists the name of the associated resource. This can be used to determine the value contained in an STL variable when querying resources during a simulation run. The values in the MAPPING column can be any of the following:

- Any valid resource or label name. See Part 1, “WSim language statements,” on page 1 for information about resource and label names.
- (NULL STRING), which indicates the null string ('').
- The first 19 characters of string constants.
- The integer value for integer constants.

Note: If a PROC, MSGUTBL, or EXECPROC type of variable is referenced, UNRESOLVED may follow the mapping. If this occurs, it means that a procedure was called or executed in the STL program which was not found in the program. This is informational only. It may indicate that you misspelled a procedure name or it can serve as a reminder that you need to translate another program to place that procedure into the MSGDD data set.

The DEFINED column lists the statement number where the variable is defined. BIT, STRING, INTEGER, PROC, EXECPROC, and MSGUTBL statement numbers appear in this column only if they are explicitly defined in your program. IO LABEL statement numbers never appear in this column. PROGRAM statement numbers appear as *PARM* if they are passed as an execution parameter.

The REFERENCED ON STATEMENT NUMBER column lists the statement numbers where the particular variable is referenced.

Using the event dictionary to find errors

There are two different types of events: ON/SIGNAL events and WAIT/POST events. The event dictionary groups all uses of certain STL program resources together for easy reference. So, anywhere an ON/SIGNAL type of event is used (SIGNAL, ON SIGNAL, QSIGNAL, and WAIT UNTIL SIGNALED STL statements) it appears in the event dictionary as a SIGNAL type. Anywhere a WAIT/POST type of event is used (POST, RESET, POSTED, and WAIT UNTIL POSTED STL statements) it appears in the event dictionary as a POST type.

You can use the event dictionary, located after the variable dictionary, to diagnose event errors, such as using the same event name for two different types of events. For example, suppose you coded the following statements:

```
WSim# STL#   STATEMENTS
----- ----- -------------------------------------------------------------
⋮
      00009 * POST 'EVENT1'
⋮
      00030 * RESET 'EVENT2'
⋮
      00052 * SIGNAL 'EVENT2'
```

An examination of the event dictionary reveals the following information:

```
EVENT DICTIONARY
NAME                               USE     REFERENCED ON STATEMENT NUMBER
--------------------------------   ------  ------------------------------
'EVENT1'                           POST    9
'EVENT2'                           POST    30
'EVENT2'                           SIGNAL  52
⋮
```

The event dictionary shows that you used a posted event, “EVENT1”, in statement 9 and that you used “EVENT2” as a posted and signaled event in statements 30 and 52. “EVENT2” is a signaled event. Statement 30 is an error since you tried to reset a signaled event.

Reading the event dictionary

The event dictionary contains three columns of information:

- NAME
- USE

- REFERENCED ON STATEMENT NUMBER.

The NAME column lists the event names used in the STL program. String constant event names appear enclosed in single quotes (for example, "EVENT1"). Named constant and string variable event names appear without quotes (for example, MYEVENT). String expression event names appear as *STRING EXPRESSION* (for example, POST MYEVENT||"2" would appear this way). The statement DEACT ALL EVENTS ONS displays an event name of *ALL*.

The USE column lists how each event is used in the STL program. Valid values for the USE column are:

POST	POST type event names.
SIGNAL	SIGNAL type event names.
TAG	Event tags.

The REFERENCED ON STATEMENT NUMBER column lists the statement numbers where the particular event is referenced.

Correcting errors

When you find the reason for an error, correct the error in your STL input data set and retranslate the program. Repeat this process until the translator detects no errors. When no errors are found, the translator issues a return code of 0.

Correct and maintain STL programs in your STL input data sets. Do not attempt to correct errors by modifying the statements in the sequential or MSGDD output data sets created by the translator.

The reasons for this restriction are the following:
- By making all changes to the STL source code, you can centralize maintenance of your program. Otherwise, you must modify the output data sets each time you translate the STL input data set.
- The STL trace facility uses the statement numbers assigned by the STL Translator. If you modify the output data sets and execute the modified statements, the STL trace facility can produce confusing or misleading results. Likewise, statement numbers displayed by the Q (Query) operator command may be incorrect.

Obtaining STL trace records

To obtain information about your STL program when using the Loglist Utility, you must take the following steps:
- Create statement correlation records
- Log STL trace records
- Print STL trace records.

Creating statement correlation records

Since the STL Translator translates STL statements into message generation statements, WSim must be able to correlate these two sets of statements to obtain trace information related to a particular STL statement. To use STL traces, you must request statement correlation records (which correlate the two types of statements) when you run the STL Translator.

See the information about the PROGRAM execution parameter in “Using STL Translator execution parameters” on page 319 for information about obtaining these records. The methods of obtaining these records are summarized here for your convenience.

You can instruct the STL Translator to provide statement correlation records in two ways:

- Include @PROGRAM statements in your STL input data set. These statements associate the names you supply with STL programs and request statement correlation records as part of your translator output. If you include these statements, you will get correlation records each time your programs translate without errors.
- Specify the execution parameter PROGRAM=*name* when you run the STL Translator. This parameter also associates the name you supply with the first program in your STL input data set and requests statement correlation records as part of your translator output.

  The PROGRAM=*name* execution parameter overrides the @PROGRAM statement for the first program in your input data set.

  Note: The program name that you assign must not be the same as names assigned to STL procedures, MSGUTBL statements, or other program names already translated and stored in the MSGDD data set.

When you request statement correlation records, the STL Translator performs these actions:

- Includes the specified program name in the title line of each page of the STL printed listing
- Creates an extra member in the MSGDD data set containing statement correlation records. The program name you supply is the name of this data set member.

Logging STL trace records

The message logging facility, when active, writes messages to the log data set containing all data that simulated resources transmit or receive in a specified network. WSim automatically logs all message traffic to and from your simulated terminals unless you have coded MLOG=NO for your network.

WSim does not log STL trace records automatically. You can request these records in your network definition or with an operator command. To request these records in your network definition, you must code the STLTRACE=YES operand on DEV, TP, or LU statements for the terminals for which you want the trace recorded. You can also code it on higher-level statements (NTWRK, VTAMAPPL, APPCLU, or TCPIP) and it will default down to the lower-level terminals. The following example shows how STL trace records could be requested for an LU:

```
LU0001     LU    STLTRACE=YES,...
```

An operator can also request STL trace records using the A (Alter) command. See , SC31-8948 for information about this command.

If the STL trace is active for a simulated terminal as WSim executes message generation statements, it correlates them with the corresponding STL statements. To make this correlation, WSim uses the statement correlation records created by the STL Translator.

Note: Coding STLTRACE=YES uses more computing power and more log space. You will probably want to use it while writing and debugging your STL programs and remove it (or code STLTRACE=NO) when running your simulation after your programs are debugged.

Printing STL trace records

When you run the Loglist Utility, you use control commands to specify the output format you want. These commands can be part of your job input stream or you can enter them from the console. See , SC31-8947 for detailed information about these commands. If STRC records (STL trace records) are present in your log data set, the Loglist Utility will format and print them unless you include the NOSTRC control command or some other control command that suppresses printing of STL trace records when you run the Loglist Utility.

When you run the Loglist Utility, you can instruct it to print only STL trace records by using the STRC control command. Or, you can combine the STRC control command with other control commands for the Loglist Utility to select various combinations of record types for printing.

For example, if you use the following control commands (either by including them in your SYSIN data set or by entering them at the console) when running the Loglist Utility, it produces only data records and STL trace records for the terminal MYTERM.

```
DATA
STRC
T MYTERM
RUN
END
```

Reading STL trace output

In your Loglist Utility output, STL trace records have a record type of STRC, enabling you to identify them quickly. These records have the following format:

```
ITP35WWI PROGRAM= XXXXXXXX  STMT# = YYYYY  PROCEDURE= ZZZZZZZZ: trace data
```

The following information is contained in these records:

WW	A unique 2-character identifier for each message.
XXXXXXXX	The program name as specified on the PROGRAM execution parameter or an STL @PROGRAM statement
YYYYY	The number of the STL statement being traced by this STL trace record
ZZZZZZZZ	The name of the STL procedure currently being executed
trace data	A message that describes the activity for the statement being traced. See , SC31-8951 for information about the messages that can appear as trace data in these records.

Figure 13 on page 344 displays sample Loglist Utility output containing STL trace records along with other types of records. The following sections explain how to read this output and use it in tracing program activity.

```
NETWORK   APPCLU/TCPIP/VTAMAPPL   DEV/LU/TP        START    STOP     READY  RECORD   HEADER      DATA  TERM  MESSAGE   USER SEQUENCE
 NAME            NAME              NAME            TIME     TIME     TIME    TYPE    FLAGS       LENG  TYPE    DECK    DATA  NUMBER
                                                 14563706 0002026  11000000  CNSL  0800 000000     74
  Workload Simulator (WSim) Version 1, Release 1.0
-------------------------------------------------------------------------------------------------------------------------------------
                                                 14564335 0002026  11000000  CNSL  0800 000000     13
  I SLUECHO,S,L
-------------------------------------------------------------------------------------------------------------------------------------
                                                 14564379 0002026  11000000  CNSL  0800 000000     51
  ITP029I INITIALIZATION COMPLETE FOR NETWORK SLUECHO
-------------------------------------------------------------------------------------------------------------------------------------
                                                 14564379 0002026  11000000  CNSL  0800 000000     31
  ITP006I NETWORK SLUECHO STARTED
-------------------------------------------------------------------------------------------------------------------------------------
SLUECHO         APPL1            SLU-1         14564379 0002026  11000000  STRC  0004 081000     22   E2   LOGON     00        0
  ITP3515I PROGRAM=LOGECHO  STMT#=00006 PROCEDURE=LOGON   : EXECUTION RESUMES
-------------------------------------------------------------------------------------------------------------------------------------
                                                 14564379 0002026  11000000  CNSL  0800 000000     60
  ITP137I SLUECHO  SLU     -00001 - Logon processing begins...
-------------------------------------------------------------------------------------------------------------------------------------
SLUECHO         APPL1            SLU-1         14564379 0002026  11000000  STRC  0004 081000     22   E2   LOGON     00        1
  ITP3518I PROGRAM=LOGECHO  STMT#=00008 PROCEDURE=LOGON   : EXECUTION INTERRUPTED
  ITP3515I PROGRAM=LOGECHO  STMT#=00010 PROCEDURE=LOGON   : EXECUTION RESUMES
  ITP3518I PROGRAM=LOGECHO  STMT#=00010 PROCEDURE=LOGON   : EXECUTION INTERRUPTED
-------------------------------------------------------------------------------------------------------------------------------------
SLUECHO         APPL1            SLU-1         14564430 14564430 14564430 +XMIT  8000 880020     34   E2   LOGON     00        4
   XMIT INITIATE SELF REQUEST
       TH   2C0000010001         FID=2   WHOLE SEGMENT  NORMAL    FLOW  DAF=00   OAF=01   ODAI=0   SEQUENCE=1
       RH   0B8000   REQUEST    FM DATA-FM HEADER     ONLY IN CHAIN   RESPONSE TYPE=DEF1
       RU   01068101 C4F4C1F3 F2F7F8F2 F308C9E3    D7C5C3C8 D6400000 00                *..a.D4A327823.ITPECHO ...       *
-------------------------------------------------------------------------------------------------------------------------------------
SLUECHO         APPL1            SLU-1         14564430 14564430 14564430 +RECV  8000 080020     12   E2   LOGON     00        5
   RECV INITIATE SELF RESPONSE
       TH   2C0001000001         FID=2   WHOLE SEGMENT  NORMAL    FLOW  DAF=01   OAF=00   ODAI=0   SEQUENCE=1
       RH   8B8000   RESPONSE   FM DATA-FM HEADER     ONLY IN CHAIN   RESPONSE TYPE=DEF1
       RU   010681                                                                 *..a                             *
-------------------------------------------------------------------------------------------------------------------------------------
SLUECHO         APPL1            SLU-1         14564430 0002026  11000000  STRC  0004 081000     22   E2   LOGON     00        6
  ITP3514I PROGRAM=LOGECHO  STMT#=00008 PROCEDURE=LOGON   : ONIN  CONDITION NOT MET
-------------------------------------------------------------------------------------------------------------------------------------
SLUECHO         APPL1            SLU-1         14564430 14564430 14564430  RECV  8000 080000     48   E2   LOGON     00        7
   RECV BIND SESSION REQUEST
       TH   2D0001010001         FID=2   WHOLE SEGMENT  EXPEDITED FLOW  DAF=01   OAF=01   ODAI=0   SEQUENCE=1
       RH   6B8000   REQUEST     SESSION CONTROL      ONLY IN CHAIN   RESPONSE TYPE=DEF1
       RU   31010303 B1903080 000087C7 00000200    00000000 18500000 7E000008 C9E3D7C5   *..........gG.........&..=...ITPE*
 00000020   C3C8D640 000000                                                            *CHO ...                         *
        BIND SESSION   FORMAT 0   TYPE=NON-NEGOTIABLE   FM PROFILE 3     TS PROFILE 3
           PRIMARY PROTOCOLS:   RU CHAINING=MULTIPLE   REQUEST MODE=IMMEDIATE   RESPONSE REQUESTED=DEF OR EXC   END BRACKET SENT
           SECONDARY PROTOCOLS: RU CHAINING=MULTIPLE   REQUEST MODE=IMMEDIATE   RESPONSE REQUESTED=EXCEPTION    END BRACKET NOT SENT
           COMMON PROTOCOLS:    SEGMENTS SUPPORTED     FM HEADERS NOT ALLOWED   BRACKETS RESET BETB      BRACKET TERMINATION RULE 1
                                ALTERNATE CODE SET NOT USED   HALF-DUPLEX FLIP-FLOP    RECOVERY RESPONSIBILITY=PRIMARY
                                CONTENTION WINNER=SECONDARY
           SECONDARY SEND PACING COUNT=NONE         SECONDARY RECEIVE PACING COUNT=NONE   ADAPTIVE SESSION PACING NOT SUPPORTED
           SECONDARY MAXIMUM RU SEND SIZE=1024      PRIMARY MAXIMUM RU SEND SIZE=1536
           PRIMARY SEND PACING COUNT=NONE           PRIMARY RECEIVE PACING COUNT=NONE
           LU TYPE 2   DEFAULT SCREEN SIZE=024,080  ALTERNATE SCREEN SIZE=NONE
           PRIMARY LU NAME=ITPECHO                  CRYPTOGRAPHIC FIELD=NONE
```

Figure 13. Sample Loglist Utility Output with STL Trace Records, part 1 of 2

```
-----------------------------------------------------------------------------------------------------------------------------------
SLUECHO         APPL1            SLU-1           14564430 0002026  11000000  STRC  0004 081000     22   E2   LOGON     00        8
  ITP3514I PROGRAM=LOGECHO  STMT#=00008 PROCEDURE=LOGON   : ONIN  CONDITION NOT MET
-----------------------------------------------------------------------------------------------------------------------------------
SLUECHO         APPL1            SLU-1           14564430 14564430 14564430  XMIT  8000 880000     10   E2   LOGON     00        9
   XMIT BIND SESSION RESPONSE
       TH   2D0001010001          FID=2   WHOLE SEGMENT  EXPEDITED FLOW  DAF=01   OAF=01   ODAI=0   SEQUENCE=1
       RH   EB8000   RESPONSE    SESSION CONTROL      ONLY IN CHAIN   RESPONSE TYPE=DEF1
       RU   31                                                                   *.                               *
-----------------------------------------------------------------------------------------------------------------------------------
SLUECHO         APPL1            SLU-1           14564430 14564430 14564430  RECV  8000 080000     10   E2   LOGON     00       10
   RECV START DATA TRAFFIC REQUEST
       TH   2D0001010001          FID=2   WHOLE SEGMENT  EXPEDITED FLOW  DAF=01   OAF=01   ODAI=0   SEQUENCE=1
       RH   6B8000   REQUEST     SESSION CONTROL      ONLY IN CHAIN   RESPONSE TYPE=DEF1
       RU   A0                                                                   *.                               *
-----------------------------------------------------------------------------------------------------------------------------------
SLUECHO         APPL1            SLU-1           14564430 0002026  11000000  STRC  0004 081000     22   E2   LOGON     00       11
  ITP3514I PROGRAM=LOGECHO  STMT#=00008 PROCEDURE=LOGON   : ONIN  CONDITION NOT MET
  ITP3515I PROGRAM=LOGECHO  STMT#=00011 PROCEDURE=LOGON   : EXECUTION RESUMES
  ITP3512I PROGRAM=LOGECHO  STMT#=00011 PROCEDURE=LOGON   : DO WHILE CONDITION MET
  ITP3516I PROGRAM=LOGECHO  STMT#=00012 PROCEDURE=LOGON   : WAIT OR TRANSMIT INTERRUPT BEGINS
-----------------------------------------------------------------------------------------------------------------------------------
SLUECHO         APPL1            SLU-1           14564430 14564432 14564430  XMIT  8000 880000     10   E2   LOGON     00       12
   XMIT START DATA TRAFFIC RESPONSE
       TH   2D0001010001          FID=2   WHOLE SEGMENT  EXPEDITED FLOW  DAF=01   OAF=01   ODAI=0   SEQUENCE=1
       RH   EB8000   RESPONSE    SESSION CONTROL      ONLY IN CHAIN   RESPONSE TYPE=DEF1
       RU   A0                                                                   *.                               *
-----------------------------------------------------------------------------------------------------------------------------------
SLUECHO         APPL1            SLU-1           14564482 14564482 14564482  RECV  8000 080000    130   E2   LOGON     00       16
   RECV (DATA) REQUEST
       TH   2C0001010001          FID=2   WHOLE SEGMENT  NORMAL    FLOW  DAF=01   OAF=01   ODAI=0   SEQUENCE=1
       RH   0380A0   REQUEST         FM DATA          ONLY IN CHAIN   RESPONSE TYPE=DEF1   BEGIN BRACKET
                    INDICATORS=        CHANGE DIRECTION
       RU   F5C7114E 7F1DF8E6 C5D3C3D6 D4C540E3    D640C9E3 D7C5C3C8 D64B1D60 40C5D5E3   *5G.+".8WELCOME TO ITPECHO..- ENT*
 00000020   C5D97EC5 C3C8D640 4040C3D3 C5C1D97E    D9C5E2E3 D6D9C540 4040F57E E2E3D9C9   *ER=ECHO   CLEAR=RESTORE   5=STRI*
 00000040   D5C740D9 C5D7E340 4040F97E D9C5D7C5    C1E31150 50C5D5E3 C5D940C4 C1E3C140   *NG REPT   9=REPEAT.&&ENTER DATA *
 00000060   E3D640C5 C3C8D640 C2C5D3D6 E67A11D1    5F1D4013 115D7F1D F0                  *TO ECHO BELOW:.J–. ..)".0       *
-----------------------------------------------------------------------------------------------------------------------------------
SLUECHO         APPL1            SLU-1           14564482 0002026  11000000  STRC  0004 081000     55   E2   LOGON     00       17
  ITP3514I PROGRAM=LOGECHO  STMT#=00008 PROCEDURE=LOGON   : ONIN  CONDITION MET; LOGGED_ON = ON
  ITP3514I PROGRAM=LOGECHO  STMT#=00012 PROCEDURE=LOGON   : ONIN  CONDITION MET; WAIT CONDITION SATISFIED
-----------------------------------------------------------------------------------------------------------------------------------
SLUECHO         APPL1            SLU-1           14564483 14564483 14564480  XMIT  8000 880000      9   E2   LOGON     00       19
   XMIT RESPONSE
       TH   2C0001010001          FID=2   WHOLE SEGMENT  NORMAL    FLOW  DAF=01   OAF=01   ODAI=0   SEQUENCE=1
       RH   838000   RESPONSE        FM DATA          ONLY IN CHAIN   RESPONSE TYPE=DEF1
-----------------------------------------------------------------------------------------------------------------------------------
SLUECHO         APPL1            SLU-1           14564530 0002026  11000000  STRC  0004 081000     22   E2   LOGON     00       20
  ITP3515I PROGRAM=LOGECHO  STMT#=00013 PROCEDURE=LOGON   : EXECUTION RESUMES
  ITP3512I PROGRAM=LOGECHO  STMT#=00011 PROCEDURE=LOGON   : DO WHILE CONDITION NOT MET
-----------------------------------------------------------------------------------------------------------------------------------
                                                 14564530 0002026  11000000  CNSL  0800 000000     60
  ITP137I SLUECHO  SLU     -00001 - Logon processing complete.
-----------------------------------------------------------------------------------------------------------------------------------
SLUECHO         APPL1            SLU-1           14564530 0002026  11000000  STRC  0004 081000     22   E2   ECHO      00       22
  ITP3519I PROGRAM=SLUECHO  STMT#=00003 PROCEDURE=ECHO    : EXECUTION CONTINUES
  ITP3516I PROGRAM=SLUECHO  STMT#=00004 PROCEDURE=ECHO    : WAIT OR TRANSMIT INTERRUPT BEGINS
-----------------------------------------------------------------------------------------------------------------------------------
SLUECHO         APPL1            SLU-1           14564531 14564531 14564530  XMIT  8000 880000     20   E2   ECHO      00       24
   XMIT (DATA) REQUEST
       TH   2C0001010001          FID=2   WHOLE SEGMENT  NORMAL    FLOW  DAF=01   OAF=01   ODAI=0   SEQUENCE=1
       RH   039020   REQUEST         FM DATA          ONLY IN CHAIN   RESPONSE TYPE=EXCP
                    INDICATORS=        CHANGE DIRECTION
       RU   7DD1E511 D160C885 939396                                                 *'JV.J-Hello                     *
-----------------------------------------------------------------------------------------------------------------------------------
```

Figure 14. Sample Loglist Utility output with STL trace records, part 2 of 2

Tracing a sample STL program

To show how the STL trace records can be used to trace an STL program, the following example presents an STL program with a network definition included, and the resulting STL printed listing. To explain how the STL program relates to the Loglist Utility output, excerpts from the output presented in Figure 13 on page 344 and explanations of the output's meaning follow the sample listings.

The sample STL program and network definition

The example below shows the STL input data set containing the network definition and program that were used to create the sample STL trace records. This network consists of a single VTAMAPPL named APPL1 with a single secondary LU half-session named SLU. Notice that STLTRACE=YES has been coded on the NTWRK statement, instructing WSim to write STL trace records to the log data set. The operand MLOG=YES has been coded on the NTWRK statement to instruct WSim to log all messages sent and received in the network.

```
@network
sluecho  ntwrk bufsize=6000,display=(24,80),
               delay=f(1),uti=50,conrate=yes,
               stltrace=yes,mlog=yes,msgtrace=no
slupath  path  echo
appl1    vtamappl
slu      lu    lutype=lu2,init=sec,
               path=(slupath),frsttxt=logon
@endnetwork
@program = logecho
bit logged_on
logon: msgtxt
/* Logon to ITPECHO. */
say 'Logon processing begins...'
logged_on = off
check: onin index(screen,'WELCOME') > 0 then logged_on = on
initself('ITPECHO','D4A32782')
do while logged_on = off
   wait until onin
   end
deact check
say 'Logon processing complete.'
endtxt
```

This example traces the LOGON program specified on the FRSTTXT operand of the LU statement. The FRSTTXT operand specifies the name of the program to be executed when the device is first started. It is typically used to log on to an application. After the LOGON program finishes, execution continues with the programs listed on the PATH statement, in this case, the program ECHO. (See Chapter 23, “Combining STL programs and network definitions,” on page 325 for more information about path entries.)

Since the simulated LU named SLU initiates the session, INIT=SEC is coded on the LU statement. Notice, however, that the RESOURCE operand has not been coded. Thus, the LOGON program must use the INITSELF statement to specify an LU partner and initiate the session. See “Logging on and off an application” on page 278 for more information about initiating SNA sessions.

The other operands in the network definition define various options for the network and for the simulated LU.

The sample logon program consists of a single STL procedure named LOGON. Note that this is the name coded as the FRSTTXT operand on the LU statement in the network definition.

The first statement in LOGON is an @PROGRAM statement, which assigns the name LOGECHO to the program. This name is important because it will be referred to later in the STL trace records. Notice that the name of the program used for STL trace records is different from the name of the procedure, which is coded on the MSGTXT statement in the STL program.

This program logs on to the ITPECHO VTAM application that is provided with WSim. If you are not familiar with ITPECHO, see , SC31-8947 for a description.

The logic of LOGECHO is simple: Initiate a session with ITPECHO and wait until ITPECHO sends a screen containing the word “WELCOME”.

The sample printed listing

Figure 15 shows a portion of the STL printed listing generated for the sample program. This printed listing was generated with the NOWSIM execution parameter, so message generation statements are not included. The STL statement numbers are referenced in the STRC records included in the Loglist Utility output. In the example shown in Figure 15, notice the assignment of statement numbers.

```
WSim# STL#   STATEMENTS
----- -----  ------------------------------------------------------------------------------------------------------------------
      00001 * @network
      00002 * @endnetwork
            *** MESSAGE ***
            ITP3172I NETWORK DEFINITION IS IN PREPROCESSOR OUTPUT BELOW

      00003 * @program = logecho
      00004 * bit logged_on
      00005 * logon: msgtxt
      00006 * /* Logon to ITPECHO. */
            * say 'Logon processing begins...'
      00007 * logged_on = off
      00008 * check: onin index(screen,'WELCOME') > 0 then logged_on = on
      00010 * initself('ITPECHO','D4A32782')
      00011 * do while logged_on = off
      00012 *    wait until onin
      00013 *    end
      00014 * deact check
      00015 * say 'Logon processing complete.'
      00016 * endtxt
```

Figure 15. STL printed listing for STL trace output example

The ONIN statement is statement number 8 and the INITSELF statement immediately following is number 10. The skip in numbers occurs because the STL Translator assigns a statement number to the THEN clause of the ONIN statement, in this case, number 9.

If you include message generation statements in your printed listing, STL statement number 9 will appear beside the message generation statement equivalent to statement 9. When using the Loglist Utility output, remember that there may be times when an STL statement number refers only to a message generation statement. In these cases, refer back to the preceding STL statement.

The Loglist Utility output

You can use the output from the Loglist Utility to trace the logic used in your program. To help you understand this process, this section provides excerpts from the output shown in Figure 13 on page 344, which was generated from the log data set created when the sample program was executed.

In the sample output in Figure 13 on page 344, records for the LU named SLU are labeled SLU-1 because the records are for the first (and only) half session defined for the LU. The following discussion breaks the trace example into several steps and explains the actions that created the records for each step. Each record is followed by an explanation of the system activity. The Loglist Utility records have been shortened for presentation here.

Step 1:

```
------------------------------------------------------------------------------------------------------------------------------
SLUECHO         APPL1           SLU-1         14564379 0002026  11000000  STRC  0004 081000     22   E2   LOGON     00        0
  ITP3515I PROGRAM=LOGECHO  STMT#=00006 PROCEDURE=LOGON   : EXECUTION RESUMES
------------------------------------------------------------------------------------------------------------------------------
                                              14564379 0002026  11000000  CNSL  0800 000000     60
  ITP137I SLUECHO  SLU     -00001 - Logon processing begins...
------------------------------------------------------------------------------------------------------------------------------
SLUECHO         APPL1           SLU-1         14564379 0002026  11000000  STRC  0004 081000     22   E2   LOGON     00        1
  ITP3518I PROGRAM=LOGECHO  STMT#=00008 PROCEDURE=LOGON   : EXECUTION INTERRUPTED
  ITP3515I PROGRAM=LOGECHO  STMT#=00010 PROCEDURE=LOGON   : EXECUTION RESUMES
  ITP3518I PROGRAM=LOGECHO  STMT#=00010 PROCEDURE=LOGON   : EXECUTION INTERRUPTED
------------------------------------------------------------------------------------------------------------------------------
```

The first record shows that execution of the procedure LOGON has resumed (actually, in this case, execution has just started) with statement number 6, the first statement after the MSGTXT statement. This statement is the SAY statement. The second record is a log of the message written to the operator's console.

The remaining records show that execution is interrupted on statement number 8, resumed on number 10, and then interrupted again. This may seem strange at first, but the explanation illustrates some important points about how WSim processes statements.

The first point to remember is that the statement number included with the EXECUTION INTERRUPTED message is the number of the last statement executed. This means that the ONIN statement (number 8) was executed (that is, the ONIN condition was activated and ready to test incoming messages) and WSim stopped before executing the INITSELF. This brings up the second point. On the first execution cycle for a device, WSim implements an initial intermessage delay before executing the first statement that would cause a message to be sent (in this case, the INITSELF). Since any statements prior to this point have been executed, the initial intermessage delay could be changed by coding a DELAY statement at the beginning of the procedure.

The initial execution proceeded up to the point at which the INITSELF (statement number 10) would be executed and then stopped for the first intermessage delay. After the delay expired, execution resumed with the INITSELF, which caused a message to be sent. Execution was once again interrupted. An INITSELF statement always interrupts execution. Execution of the procedure will not resume until the session has been fully established.

Step 2:

```
------------------------------------------------------------------------------------------------------------------------------
SLUECHO         APPL1           SLU-1         14564430 14564430 14564430 +XMIT  8000 880020     34   E2   LOGON     00        4
   XMIT INITIATE SELF REQUEST
       TH   2C0000010001         FID=2   WHOLE SEGMENT  NORMAL    FLOW  DAF=00   OAF=01   ODAI=0   SEQUENCE=1
       RH   0B8000   REQUEST    FM DATA-FM HEADER     ONLY IN CHAIN   RESPONSE TYPE=DEF1
       RU   01068101 C4F4C1F3 F2F7F8F2 F308C9E3    D7C5C3C8 D6400000 00                 *..a.D4A327823.ITPECHO ...       *
------------------------------------------------------------------------------------------------------------------------------
SLUECHO         APPL1           SLU-1         14564430 14564430 14564430 +RECV  8000 080020     12   E2   LOGON     00        5
   RECV INITIATE SELF RESPONSE
       TH   2C0001000001         FID=2   WHOLE SEGMENT  NORMAL    FLOW  DAF=01   OAF=00   ODAI=0   SEQUENCE=1
       RH   8B8000   RESPONSE   FM DATA-FM HEADER     ONLY IN CHAIN   RESPONSE TYPE=DEF1
       RU   010681                                                                      *..a                             *
------------------------------------------------------------------------------------------------------------------------------
SLUECHO         APPL1           SLU-1         14564430 0002026  11000000  STRC  0004 081000     22   E2   LOGON     00        6
  ITP3514I PROGRAM=LOGECHO  STMT#=00008 PROCEDURE=LOGON   : ONIN  CONDITION NOT MET
------------------------------------------------------------------------------------------------------------------------------
```

In the next step, SLU sends the INITIATE SELF request to the host as a result of the INITSELF statement. The request contains the name of the requested session partner, ITPECHO.

SLU receives a positive INITIATE SELF response. An STL trace record is logged showing that the incoming response has been tested by the active ONIN condition on statement number 8. This condition tests the screen image buffer for the string "WELCOME" sent by ITPECHO. The condition is not met because SLU has not yet received this data. Since ONIN conditions remain active until a DEACT or ENDTXT statement is executed, this condition will be evaluated for all messages that SLU receives during execution of the LOGON procedure.

Step 3:

```
-----------------------------------------------------------------------------------------------------------------------------------
SLUECHO         APPL1             SLU-1          14564430 14564430 14564430  RECV  8000 080000     48   E2   LOGON     00        7
   RECV BIND SESSION REQUEST
       TH   2D0001010001         FID=2   WHOLE SEGMENT  EXPEDITED FLOW  DAF=01   OAF=01   ODAI=0   SEQUENCE=1
       RH   6B8000   REQUEST     SESSION CONTROL      ONLY IN CHAIN   RESPONSE TYPE=DEF1
       RU   31010303 B1903080 000087C7 00000200    00000000 18500000 7E000008 C9E3D7C5   *..........gG.........&.=...ITPE*
 00000020   C3C8D640 000000                                                             *CHO ...                        *
        BIND SESSION   FORMAT 0   TYPE=NON-NEGOTIABLE   FM PROFILE 3     TS PROFILE 3
           PRIMARY PROTOCOLS:   RU CHAINING=MULTIPLE   REQUEST MODE=IMMEDIATE   RESPONSE REQUESTED=DEF OR EXC   END BRACKET SENT
           SECONDARY PROTOCOLS: RU CHAINING=MULTIPLE   REQUEST MODE=IMMEDIATE   RESPONSE REQUESTED=EXCEPTION    END BRACKET NOT SENT
           COMMON PROTOCOLS:    SEGMENTS SUPPORTED     FM HEADERS NOT ALLOWED   BRACKETS RESET BETB      BRACKET TERMINATION RULE 1
                                ALTERNATE CODE SET NOT USED   HALF-DUPLEX FLIP-FLOP    RECOVERY RESPONSIBILITY=PRIMARY
                                CONTENTION WINNER=SECONDARY
           SECONDARY SEND PACING COUNT=NONE         SECONDARY RECEIVE PACING COUNT=NONE   ADAPTIVE SESSION PACING NOT SUPPORTED
           SECONDARY MAXIMUM RU SEND SIZE=1024      PRIMARY MAXIMUM RU SEND SIZE=1536
           PRIMARY SEND PACING COUNT=NONE           PRIMARY RECEIVE PACING COUNT=NONE
           LU TYPE 2   DEFAULT SCREEN SIZE=024,080  ALTERNATE SCREEN SIZE=NONE
           PRIMARY LU NAME=ITPECHO                  CRYPTOGRAPHIC FIELD=NONE
-----------------------------------------------------------------------------------------------------------------------------------
SLUECHO         APPL1             SLU-1          14564430 0002026  11000000  STRC  0004 081000     22   E2   LOGON     00        8
  ITP3514I PROGRAM=LOGECHO  STMT#=00008 PROCEDURE=LOGON   : ONIN  CONDITION NOT MET
-----------------------------------------------------------------------------------------------------------------------------------
SLUECHO         APPL1             SLU-1          14564430 14564430 14564430  XMIT  8000 880000     10   E2   LOGON     00        9
   XMIT BIND SESSION RESPONSE
       TH   2D0001010001         FID=2   WHOLE SEGMENT  EXPEDITED FLOW  DAF=01   OAF=01   ODAI=0   SEQUENCE=1
       RH   EB8000   RESPONSE    SESSION CONTROL      ONLY IN CHAIN   RESPONSE TYPE=DEF1
       RU   31                                                                          *.                              *
-----------------------------------------------------------------------------------------------------------------------------------
```

In this step, SLU receives a BIND SESSION request from ITPECHO. Again, the ONIN condition is evaluated, and the condition is not met. Next, SLU transmits a positive BIND SESSION response. WSim generates this response for SLU automatically. STL program execution has not yet resumed because session establishment is not complete.

Step 4:

```
-----------------------------------------------------------------------------------------------------------------------------------
SLUECHO         APPL1             SLU-1          14564430 14564430 14564430  RECV  8000 080000     10   E2   LOGON     00       10
   RECV START DATA TRAFFIC REQUEST
       TH   2D0001010001         FID=2   WHOLE SEGMENT  EXPEDITED FLOW  DAF=01   OAF=01   ODAI=0   SEQUENCE=1
       RH   6B8000   REQUEST     SESSION CONTROL      ONLY IN CHAIN   RESPONSE TYPE=DEF1
       RU   A0                                                                          *.                              *
-----------------------------------------------------------------------------------------------------------------------------------
SLUECHO         APPL1             SLU-1          14564430 0002026  11000000  STRC  0004 081000     22   E2   LOGON     00       11
  ITP3514I PROGRAM=LOGECHO  STMT#=00008 PROCEDURE=LOGON   : ONIN  CONDITION NOT MET
```

Here SLU receives the START DATA TRAFFIC request from ITPECHO, indicating that session establishment is complete. Once again, the ONIN condition is evaluated and not met.

Step 5:

```
  ITP3515I PROGRAM=LOGECHO  STMT#=00011 PROCEDURE=LOGON   : EXECUTION RESUMES
  ITP3512I PROGRAM=LOGECHO  STMT#=00011 PROCEDURE=LOGON   : DO WHILE CONDITION MET
  ITP3516I PROGRAM=LOGECHO  STMT#=00012 PROCEDURE=LOGON   : WAIT OR TRANSMIT INTERRUPT BEGINS
-----------------------------------------------------------------------------------------------------------------------------------
SLUECHO         APPL1             SLU-1          14564430 14564432 14564430  XMIT  8000 880000     10   E2   LOGON     00        12
   XMIT START DATA TRAFFIC RESPONSE
       TH   2D0001010001          FID=2   WHOLE SEGMENT  EXPEDITED FLOW  DAF=01   OAF=01   ODAI=0   SEQUENCE=1
       RH   EB8000   RESPONSE    SESSION CONTROL      ONLY IN CHAIN   RESPONSE TYPE=DEF1
       RU   A0                                                                     *.                               *
-----------------------------------------------------------------------------------------------------------------------------------
```

With session establishment complete, execution resumes with the next statement following the INITSELF. In statement number 11, the DO WHILE condition is met since the bit variable "logged_on" has not been turned ON by the ONIN condition. That is, SLU still has not received the "WELCOME" screen from ITPECHO. Since the condition "logged_on = off" is true, the statements in the DO WHILE group are executed. Execution of statement 12 (WAIT UNTIL ONIN) causes SLU to interrupt execution until it receives the next message from ITPECHO.

As for the BIND response in Step 3, WSim automatically generates a response to the START DATA TRAFFIC received in the previous step.

Step 6:

```
-----------------------------------------------------------------------------------------------------------------------------------
SLUECHO         APPL1             SLU-1          14564482 14564482 14564482  RECV  8000 080000    130   E2   LOGON     00        16
   RECV (DATA) REQUEST
       TH   2C0001010001          FID=2   WHOLE SEGMENT  NORMAL    FLOW  DAF=01   OAF=01   ODAI=0   SEQUENCE=1
       RH   0380A0   REQUEST         FM DATA          ONLY IN CHAIN   RESPONSE TYPE=DEF1   BEGIN BRACKET
                     INDICATORS=        CHANGE DIRECTION
       RU   F5C7114E 7F1DF8E6 C5D3C3D6 D4C540E3    D640C9E3 D7C5C3C8 D64B1D60 40C5D5E3   *5G.+".8WELCOME TO ITPECHO..- ENT*
 00000020   C5D97EC5 C3C8D640 4040C3D3 C5C1D97E    D9C5E2E3 D6D9C540 4040F57E E2E3D9C9   *ER=ECHO   CLEAR=RESTORE   5=STRI*
 00000040   D5C740D9 C5D7E340 4040F97E D9C5D7C5    C1E31150 50C5D5E3 C5D940C4 C1E3C140   *NG REPT   9=REPEAT.&&ENTER DATA *
 00000060   E3D640C5 C3C8D640 C2C5D3D6 E67A11D1    5F1D4013 115D7F1D F0                  *TO ECHO BELOW:.J—. ..)".0       *
-----------------------------------------------------------------------------------------------------------------------------------
SLUECHO         APPL1             SLU-1          14564482 0002026  11000000  STRC  0004 081000     55   E2   LOGON     00        17
  ITP3514I PROGRAM=LOGECHO  STMT#=00008 PROCEDURE=LOGON   : ONIN  CONDITION MET; LOGGED_ON = ON
  ITP3514I PROGRAM=LOGECHO  STMT#=00012 PROCEDURE=LOGON   : ONIN  CONDITION MET; WAIT CONDITION SATISFIED
-----------------------------------------------------------------------------------------------------------------------------------
SLUECHO         APPL1             SLU-1          14564483 14564483 14564480  XMIT  8000 880000      9   E2   LOGON     00        19
   XMIT RESPONSE
       TH   2C0001010001          FID=2   WHOLE SEGMENT  NORMAL    FLOW  DAF=01   OAF=01   ODAI=0   SEQUENCE=1
       RH   838000   RESPONSE        FM DATA          ONLY IN CHAIN   RESPONSE TYPE=DEF1
-----------------------------------------------------------------------------------------------------------------------------------
SLUECHO         APPL1             SLU-1          14564530 0002026  11000000  STRC  0004 081000     22   E2   LOGON     00        20
  ITP3515I PROGRAM=LOGECHO  STMT#=00013 PROCEDURE=LOGON   : EXECUTION RESUMES
  ITP3512I PROGRAM=LOGECHO  STMT#=00011 PROCEDURE=LOGON   : DO WHILE CONDITION NOT MET
-----------------------------------------------------------------------------------------------------------------------------------
```

Shortly, SLU receives a formatted 3270 data stream from ITPECHO. This contains the "WELCOME" screen SLU has been waiting for. The ONIN condition from statement 8 is finally met and "logged_on" is set to ON. The WAIT UNTIL ONIN condition from statement 12 is met (and would be met by any incoming message regardless of its content) and the WAIT condition is reset.

Execution resumes with the statement immediately following the WAIT, in this case, the END statement (13) and execution loops back to the DO WHILE statement (11). This time the DO WHILE condition "logged_on = off" is false and execution skips to the next statement after the DO WHILE group.

Step 7:

```
------------------------------------------------------------------------------------------------------------------------------------
                                                 14564530 0002026  11000000  CNSL  0800 000000     60
  ITP137I SLUECHO  SLU     -00001 - Logon processing complete.
------------------------------------------------------------------------------------------------------------------------------------
SLUECHO         APPL1             SLU-1          14564530 0002026  11000000  STRC  0004 081000     22   E2   ECHO      00        22
  ITP3519I PROGRAM=SLUECHO  STMT#=00003 PROCEDURE=ECHO    : EXECUTION CONTINUES
  ITP3516I PROGRAM=SLUECHO  STMT#=00004 PROCEDURE=ECHO    : WAIT OR TRANSMIT INTERRUPT BEGINS
------------------------------------------------------------------------------------------------------------------------------------
SLUECHO         APPL1             SLU-1          14564531 14564531 14564530  XMIT  8000 880000     20   E2   ECHO      00        24
   XMIT (DATA) REQUEST
       TH   2C0001010001          FID=2   WHOLE SEGMENT  NORMAL    FLOW  DAF=01   OAF=01   ODAI=0   SEQUENCE=1
       RH   039020   REQUEST         FM DATA          ONLY IN CHAIN   RESPONSE TYPE=EXCP
                     INDICATORS=        CHANGE DIRECTION
       RU   7DD1E511 D160C885 939396                                                    *'JV.J-Hello                     *
------------------------------------------------------------------------------------------------------------------------------------
```

In the last step, the execution of the SAY statement (statement 15) is logged. The next record shows that "EXECUTION CONTINUES" with statement 3 of the procedure ECHO. This means that WSim executed the ENDTXT statement (statement 16) in LOGON, and execution continued with the first program listed on the PATH statement in the network definition. Execution of the ENDTXT statement also deactivated the ONIN condition from statement 8. (WAIT UNTIL ONIN conditions like statement 12 are automatically deactivated when the WAIT is reset.)

Control has now passed to the ECHO procedure and SLU has sent data to ITPECHO, causing a Transmit Interrupt.

Part 3. Reference to STL statements and functions

Chapter 25. Reference to STL statements

This chapter contains detailed descriptions of all the STL statements that begin with an STL keyword. Statement syntax and usage considerations are included in the description of each statement.

You can code labels or names at the beginning of each of these statements. However, labels or names are not included in the syntax definitions, except in cases in which they have particular importance (MSGTXT, ONIN, ONOUT, and MSGUTBL).

Assignment statements are not included in this chapter because these statements do not begin with an STL keyword. For a description of assignment statements, see the information about assignment statements in “Using assignment statements” on page 246.

Note: This chapter lists only keywords that **begin** a statement. Keywords used within a statement are described with the keyword they follow.

@EJECT

```
@EJECT
```

Function

The @EJECT control statement forces a page eject for the STL printed listing.

Examples

```
proc1: msgtxt     /* I want this procedure to be on one page ...   */
&#38;#8942;
endtxt
@eject
proc2: msgtxt     /* ... and this procedure to be on another page. */
&#38;#8942;
endtxt
```

@GENERATE

```
@GENERATE
WSim_statements
@ENDGENERATE
```

Where

WSim_statements are message generation statements enclosed in single or double quotation marks. You may use STL variables and named string constants to form the statements (string operators and functions are not permitted).

Function

The @GENERATE control statement enables you to include message generation statements directly. When the STL Translator encounters an @GENERATE statement, it enters "generate mode," and will remain in this mode until it processes the @ENDGENERATE statement. While in generate mode, the translator processes only string constants, variables, and comments. Resources—counters, save areas, and switches—will be substituted for the variable names, and the resulting string constant expression will be output.

Examples

The following code includes message generation statements in an STL program.

```
@generate
'         WTO      (HELLO)'
'         SET      'a'=12345'   /* "a" is an integer variable. */
@endgenerate
```

The preceding example generates the following message generation statements:

```
         WTO      (HELLO)
         SET      DC1=12345
```

Notes

- The @GENERATE statement is valid only inside an STL procedure.
- The @GENERATE and @ENDGENERATE statements can function as a DO-END statement group when coded as part of an IF, ONIN, ONOUT, or ON SIGNALED statement.
- You can abbreviate @GENERATE as @GEN and @ENDGENERATE as @ENDGEN.
- For more information about using the @GENERATE control statement, see "Including message generation statements in STL programs" on page 251.
- While in generate mode, be careful when coding string constants containing single quotation marks (') especially when the single quotation mark is used as the string constant delimiter. WSim interprets pairs of quotation marks as one quotation mark. Therefore, to code a quotation mark in a string, you must double each instance of a quotation mark, as shown in the following example.

  Note that this rule only applies in generate mode. For STL coding, you would simply double a quotation mark that you did not intend to use as a string delimiter character.

  ```
  @gen
  '   TEXT  (It is 2 o'clock)'       /* WRONG! STL error.            */
  '   TEXT  (It is 2 o''clock)'      /* WRONG! OK for STL, but       */
                                     /* flagged by preprocessor.     */
  '   TEXT  (It is 2 o''''clock)'    /* RIGHT!                       */
  "   TEXT  (It is 2 o''clock)"      /* RIGHT!                       */
  "   TEXT  (It is 2 o'clock)"       /* WRONG!  OK for STL, but      */
                                     /* flagged by preprocessor.     */
  @endgen
  ```

- The statements must conform to the rules for coding WSim statements. See Part 1, "WSim language statements," on page 1 for more information about requirements for message generation statements.
- If you include a network definition in your STL input data set, you can also include entire message generation decks within it. See "@NETWORK" on page 360 for more information.

- No syntax checking is done and offsets are not added for save areas when the save area number is included as a variable name.
- In order to preserve the integrity of the STL trace correlation file, the statement must be started before column 16. Statements started after column 15 are assumed to be a continuation of the prior statement. The label statement should not be coded. The statement numbers in the STL trace output may not match your STL program if the rules for the `@GENERATE` statement are not followed.

  For example, the following `@GENERATE` statements would maintain the integrity of the trace correlation file and the statement sequence numbers.

  ```
  @generate
  '             WTO (A text message)      ' -> Starts in col 15
  '                                       ' -> Blank line
  '             TEXT (A message that ),   ' -> Starts in col 15
  '              (wraps)                  ' -> Continued in 16
  '* A comment line                       ' -> Comment line
  @endgenerate
  ```

  The following `@GENERATE` statements would cause confusion and possibly incorrect trace output messages during execution of an STL program containing the statements.

  ```
  @generate
  '              WTO  (A text message)    ' -> Starts in col 16
  '             TEXT (A message that ),   ' -> Valid start but
  '         (wraps)                       ' -> continued before
                                               col 16
  'MYLABEL LABEL                          ' -> Should not code
  @endgenerate
  ```

@IFNUM

```
@IFNUM = starting_number
```

Where

starting_number is an integer constant with a value from 1 to 4095.

Function

The @IFNUM control statement specifies the number of the first input (WHEN=IN) or output (WHEN=OUT) IF statement generated by the STL Translator. When this statement is coded in an STL program, generated IF statements will be assigned numbers beginning with *starting_number* up to a maximum of 4095.

This statement is valid only when coded outside of an STL procedure.

Examples

```
@ifnum = 30    /* Begin IF numbering with 30. */
```

Note

You should code the @IFNUM control statement only under these conditions:

1. You are trying to coordinate procedures in multiple STL programs.

   ```
                /* Program #1 */
   @program = prog1
   proc1: msgtxt
   ```

```
/*********************************************************/
/* WSim code generated for this program will begin with IF */
/* number 1.  This is the default.  PROC2 is called in     */
/* this program but is not included in this program.       */
/* You must be careful to coordinate IF numbering in       */
/* such cases.                                             */
/*********************************************************/
onin substr(ru,1,5) = 'Hello' then ready = on
call proc2
endtxt
@endprogram
```

```
                /* Program #2 */
@program = prog2
@ifnum = 20
proc2: msgtxt
/*********************************************************/
/* WSim code generated for this program will begin with IF */
/* number 20.  This is caused by the @IFNUM control state- */
/* ment.  This statement is needed because PROC2 is        */
/* called by a procedure that is not included in this      */
/* program.                                                */
/*********************************************************/
onin substr(ru,1,7) = 'Goodbye' then ready=off
 .
 .
 .
endtxt
@endprogram
```

2. You want to generate IF statements using the @GENERATE STL statement and need to reserve one or more IF numbers for your own use.

```
@ifnum = 11

proc1: msgtxt
/**********************************************/
/* Set aside IF numbers 0-10 for use in       */
/* code created by using @GENERATE            */
/* statements.                                */
/**********************************************/
/* Generate special IFs. */
@generate
"0       IF      WHEN=IN,LOC=RH+0,TEXT='80',THEN=B-RESP,STATUS=HOLD"
 .
 .
 .
"10      IF      WHEN=IN,LOC=B+0,TEXT=(REPLY 1),THEN=CONT"
@endgenerate
 .
 .
 .
endtxt
```

3. You exceeded the number of IFs and can reuse earlier IF numbers. You must be very careful when doing this to ensure that you do not override something you still need.

@INCLUDE

```
@INCLUDE member_name
```

Where

member_name is the name of a member of the partitioned data set specified by the SYSLIB DD statement. It is subject to the following limitations:

- Can include these characters: uppercase and lowercase alphanumeric characters and the special characters $, @, _ (underscore), ?, and #
- Cannot begin with a number
- Must be from 1 to 8 characters long.

Function

The @INCLUDE statement retrieves the *member_name* from the data set specified by the SYSLIB DD statement and processes the contents of the member as if it were actually coded in the STL input data set at the point where the @INCLUDE statement is coded. The lines in the include member are processed as if they were actually "included" in the STL input data set.

An included member can itself contain @INCLUDE statements referencing other members; however, members may not be included recursively.

The statements contained in an include member must be complete STL statements. If all records of an included member have been read and continuation is expected, an error will be flagged. Other structures (such as networks, do-end groups, and procedures) can cross boundaries of included members as desired.

This statement may be coded anywhere in the STL input data set. However, when coded, it must be the last statement on the line and be followed only by STL comments.

If this statement is coded between @NETWORK and @ENDNETWORK statements, it must begin in the first position of the line, must otherwise be coded with the syntax of a normal STL statement, and must be complete on the line.

Examples

```
---> STL input data set

/*******************************************************/
/* PROGRAM:   MYTEST                                   */
/* DATE:      06/20/2002                               */
@include header     /* Include my unique comment block   */
/*******************************************************/
mytest: msgtxt
&#38;#8942;
endtxt

---> HEADER member from include data set.  This member would
---> allow the user to include a common block of comments
---> in all of their input data sets.

/* USER:      JOHN DOE                                         */
/* TEST:      REGRESSION 45                                    */
/* .                                                           */
/* .                                                           */
/* .                                                           */
```

@NETWORK

```
@NETWORK
WSim_statements
@ENDNETWORK
```

Where

WSim_statements are all statements acceptable by the Preprocessor. These include network definition and message generation statements.

Function

The @NETWORK and @ENDNETWORK control statements let you include network definition statements in the same data set with your STL programs. When the STL Translator encounters an @NETWORK statement, it assumes that the following statements are in the scripting language, until the STL Translator encounters the @ENDNETWORK statement. After the translator completes processing all the STL statements, it calls the Preprocessor to verify all the statements found between the @NETWORK and @ENDNETWORK statements. You can specify the NOPREP execution parameter to suppress preprocessing of the network, or NOTRAN to suppress translation of the STL statements.

See "Including network definition statements in STL" on page 325 for more information.

Examples

```
/* My STL Data Set */

/* Network Definition */
@network
netx     Ntwrk     uti=100,delay=f2,stltrace=yes,mlog=yes
apath    Path      amsgWSim
1    Vtamappl  bufsize=3000
* LU DefinitionWSim
1lu  Lu        lutype=lu2,display=(24,80),init=sec,path=(apath)
@endnetwork

/* STL Program */
@program
integer i
string  c
amsg:    msgtxt
         do i=1 to 5
          c = char(i)
          say 'Message Number' c
         end
         opcmnd 'zend'
         quiesce
         endtxt
@endprogram
```

Notes

- The network definition must be placed at the top of the STL input data set. Only comments can come before it.
- The @ENDNETWORK statement must begin in column one.

- The statements must be coded **exactly** as you run the Preprocessor against the data set directly. For example, labels must begin in column 1.
- You must code at least one NTWRK or PREP statement in your network definition. Multiple networks may also be defined.
- STL comments may not be coded within the network definition. Only WSim comments are acceptable.
- The @NETWORK and @ENDNETWORK statements can be abbreviated @NET and @ENDNET.
- Only comments may be coded on the same line after the @NETWORK statement.

@PROGRAM

```
@PROGRAM[=program_name]
STL_statements
@ENDPROGRAM
```

Where

program_name is a 1- to 8-character alphanumeric name that conforms to STL variable naming conventions.

Function

The @PROGRAM control statement specifies the start of an STL program. If you code "=*program_name*", the STL trace facility stores trace information for the program under this name. See "Obtaining STL trace records" on page 341 for more information about the STL trace facility.

To code multiple programs in one STL input data set, you need the @ENDPROGRAM control statement. You must start each program in the input data set with an @PROGRAM statement and end each program with an @ENDPROGRAM statement. You can only code comments between programs. Defining multiple STL programs lets you reuse variable, constant and label names. (This does not include procedure, user table, or program names.)

For more information on using the @PROGRAM and @ENDPROGRAM statements to code multiple STL programs, see "Coding multiple programs in one STL input data set" on page 326.

Examples

```
/*******************************************/
/*                                         */
/* This program will be known as AMSGPROG. */
/*                                         */
/*******************************************/
@program = amsgprog
integer i                    /* The variable i is an integer
                                in this program.             */
amsg:    msgtxt
         do i = 1 to 5
           c = char(i)
           type 'Message Number' c
           transmit
         end
         endtxt
@endprogram                   /* Specifies the end of AMSGPROG. */
/******************************************/
/*                                        */
/* This new program is unnamed.           */
/* Therefore, STL trace information is not */
/* available for this program.  Also, STL  */
/* statement numbers are not displayed     */
/* when using the Q (Query) operator       */
/* command.                                */
/*                                        */
/******************************************/
@program
string  i                     /* The variable i is a string in
                                 this program.                 */
bmsg:    msgtxt
         i = 'some text'
         type i
         transmit
         endtxt
@endprogram                   /* Specifies the end of the
                                 previous program.             */
/******************************************/
/*                                        */
/* This program will be known as CMSGPROG. */
/*                                        */
/******************************************/
@program = cmsgprog
cmsg:    msgtxt
         opcmnd 'zend'
         quiesce
         endtxt
                              /* The @endprogram statement is
                                 optional for the last program. */
```

Notes

- The @PROGRAM statement, if coded, must be the first statement in a program. It must be placed before all the declarations and procedures that make up the program. The @ENDPROGRAM statement, if coded, must be the last statement in a program. No STL statements may follow it unless another @PROGRAM statement is coded. Neither the @PROGRAM statement nor the @ENDPROGRAM statement is valid within a procedure or user table.
- The @ENDPROGRAM statement is not required if the STL input data set contains only one program. If multiple programs are coded, it is also optional for the last program.
- You can specify a program name using the PROGRAM execution parameter. This overrides any name that was coded in an @PROGRAM statement for the *first* program defined in your STL input data set.
- Program names cannot be the same as names declared for other programs, procedures, or user tables.
- You cannot code STL statements after the @ENDPROGRAM statement on the same line. However, comments are valid.

ABORT

```
ABORT
```

Function

The ABORT statement is an asynchronous subset statement. You can use the ABORT statement to terminate the current STL program immediately. All outstanding asynchronous conditions will be deactivated and the next program specified for this terminal (on the PATH network definition statement) will begin executing when the current intermessage delay expires.

Examples

```
onin index(ru,'UNRECOVERABLE ERROR') > 0 then abort
                                             /* Abort this program  */
                                             /* if an unrecoverable */
                                             /* error ever occurs.  */
```

Notes

- This statement **must** be coded directly following the THEN keyword on the ONIN, ONOUT, or ON SIGNALED statement.
- You could use ABORT to handle the unexpected loss of a terminal session.

ALLOCATE

```
ALLOCATE variable_name {counter}
                       {save area}
                       {switch}
```

Where

variable_name is any valid STL variable name not previously declared.

counter is a counter name, for example, DC1 (for device counter 1) or NC25 (for network counter 25). You can code values in the ranges DC1-DC4095 and NC1-NC4095. (You can also code line-level, terminal-level, and sequence counters. See Note 4 on page 364.)

save area is a save area name, for example, 1 (for device save area 1) or N140 (for network save area 140). You can code values in the ranges 1-4095 and N1-N4095.

switch is a switch name, for example, SW1 (for device switch 1) or NSW8 (for network switch 8). You can code values in the ranges SW1-SW4095, NSW1-NSW31, and NSW33-NSW4095. STL reserves NSW32 for internal use. (You can also specify terminal-level switches. See Note 4 on page 364.)

All counter, save area, or switch names must be enclosed in single or double quotation marks.

Function

The ALLOCATE statement allows the STL programmer to specify which resource is to be used to represent a variable. If the ALLOCATE statement is not used, the STL Translator performs this variable-to-resource mapping automatically when a variable is declared.

Examples

```
allocate mydata '1'      /* Use device save area 1 to hold "mydata". */
allocate mycount 'DC1'   /* Use device counter 1 to hold "mycount".  */
allocate netflag 'NSW1'  /* Use network switch 1 to hold "netflag".  */
```

Notes

1. The ALLOCATE statement is a declarative statement. You can code it only **outside** an STL procedure.
2. If you use ALLOCATE statements, code them before all other declarative statements and before the first procedure in an STL program. This coding order is necessary to ensure that the resources are allocated properly.
3. The ALLOCATE statement should be necessary under only two conditions:
   - When you must coordinate separately translated STL programs. You can obtain the variable-to-resource mappings for a translated STL program from the variable dictionary included in the printed listing from the STL Translator.
   - When counters and switches unknown to the STL Translator are required. These include line-level and terminal-level counters, sequence counters, and terminal switches. For example:

     ```
     allocate line_count 'LC2'       /* Use line counter 2 to hold      */
                                     /* "line_count".                   */

     allocate group_flag 'TSW18'     /* Use terminal switch 18 to       */
                                     /* hold "group_flag".              */

     allocate net_sequence 'NSEQ'    /* Use the network sequence counter */
                                     /* (NSEQ) to hold the variable     */
                                     /* "net_sequence".                 */
     ```
4. STL does not automatically keep track of line-level and terminal-level counters, terminal-level switches, and sequence counters. Therefore, you can declare multiple STL variables that are mapped to the same resources.
5. Be sure that the CNTRS operand is not coded in your network definition. This operand can cause problems when using STL programs.

BIT

```
BIT [{SHARED|UNSHARED}] variable_list
```

Where

variable_list is any number of valid STL variable names not previously declared, separated by commas.

Function

The BIT statement explicitly declares one or more BIT variables, which can be specified as SHARED or UNSHARED. The default class is UNSHARED.

Examples

```
                  /* Declares the unshared bit variables,            */
                  /* "found_it" and "data_received".                 */
bit unshared found_it, data_received
```

```
bit myflag          /* Declares the unshared bit variable, "myflag".     */
bit shared netflag  /* Declares the shared bit variable, "netflag".      */
```

Note

The BIT statement is a declarative statement. You can code it only **outside** an STL procedure.

BTAB

```
BTAB
```

Function

The BTAB statement simulates the Back Tab key on a display terminal. The Back Tab key moves the cursor to the beginning of the preceding input field on the simulated panel if the cursor is currently positioned at the first character position in a field. If the cursor is currently positioned in an input field but not at the first character, the Back Tab key moves the cursor to the beginning of the current input field. This statement is valid for simulation of 3270 terminals only.

Examples

```
cursor(index(screen,'MYFILE'))  /* Position cursor on MYFILE. */
btab                            /* Back up to the input field */
                                /* before MYFILE.             */
```

CALL

```
CALL procedure_name
```

Where

procedure_name is the name of an STL procedure.

Function

The CALL statement passes control to the named STL procedure. All currently active ONIN, ONOUT, and ON SIGNALED conditions in the calling procedure remain active. When control returns from the called procedure (as a result of a RETURN or ENDTXT statement), execution of the calling procedure continues with the statement following the CALL.

Note: If this is coded as an asynchronous subset statement, the CALL will reset the WAIT condition of your simulated terminal, causing any outstanding WAIT conditions to be satisfied.

Examples

```
call editproc  /* Call the procedure that edits a file. */
```

Notes

- All STL variables are global in scope; therefore, their values are available to all procedures in a program. Thus, a variable's value may not be the same after the return from a called procedure because the called procedure may have modified the variable. For instance, in the example below, the SAY statement in “proc1” writes a “2” to the operator since “proc2” has modified the value of “a”.

```
proc1: msgtxt
a = 1
call proc2
say char(a)
endtxt

proc2: msgtxt
a = 2
endtxt
```

- You can make recursive calls (calls to the currently active STL procedure).
- WSim limits the number of outstanding CALLs permitted for a terminal. The MAXCALL operand on the DEV, TP, or LU statements specifies this limit. If you do not code MAXCALL, WSim uses the default limit of five. See Part 1, “WSim language statements,” on page 1 for more information about the MAXCALL operand.
- When coded as an asynchronous subset statement, this statement **must** be coded directly following the THEN keyword on the ONIN, ONOUT, or ON SIGNALED statement.

CANCEL

```
CANCEL  {event_tag}
        {DELAY}
        {SUSPEND}
```

Where

event_tag is a string expression that specifies the “tag” value assigned to a POST, QSIGNAL, RESET, or SIGNAL statement. If the *event_tag* is a string constant expression, it must be 1 to 8 alphanumeric characters and enclosed in single or double quotes. (Strings containing hexadecimal constants are exempt from this restriction.) If you specify a nonconstant string expression, the first 8 characters of the string are used. If the string is shorter than 8 characters, the available characters are used. To satisfy conditions using event names, the first 8 characters must match exactly. If you specify a string variable, it cannot be the name of one of the reserved variables (for example, BUFFER).

Function

When you use the DELAY or SUSPEND arguments, the CANCEL statement is an asynchronous subset statement. When encountered on an ONIN, ONOUT, or ON SIGNALED statement, CANCEL causes the current delay between messages or the current suspension interval to be canceled.

The STL Translator handles CANCEL DELAY and CANCEL SUSPEND identically. Therefore, either asynchronous subset statement will cancel the current delay interval, if one is in effect, or will cancel the current suspension interval, if one is in effect, regardless of which argument you specify.

This statement also enables you to cancel an action specified by a POST, QSIGNAL, RESET, or SIGNAL statement if a time interval was specified and the interval has **not** expired. The event can only be canceled by the same resource that established the event. For example, if terminal A signals an event, terminal B cannot cancel the signal. Only terminal A can cancel it.

Examples

The following example shows how to cancel a DELAY statement.

```
mainproc: msgtxt
onin index(ru,'More...') > 0 then cancel delay
                                     /* Don't delay if screen needs */
                                     /* to be cleared.              */
type 'LISTFILE * * *'                /* List all files.             */
delay(40)                            /* Delay 40 seconds after      */
                                     /* next message is sent.       */
transmit
do while index(screen,'More...') > 0       /* Keep clearing screen  */
                                           /* as needed.            */
delay(40)                                  /* Reestablish 40-second */
                                           /* delay.                */
transmit using pa2                         /* Clear screen.         */
end
&#38;#8942;
endtxt                                     /* End of Mainproc.      */
```

The following example shows how to cancel a SUSPEND statement.

```
sleep: msgtxt                  /* Go to sleep for one hour or until */
                               /* 'Hello' is received.              */
onin index(ru,'Hello') > 0 then cancel suspend
                               /* Wake up when 'Hello' is received. */
suspend(3600)                  /* Sleep for 1 hour (3600 seconds).  */
endtxt                         /* End of sleep.                     */
```

The following examples show how to cancel events.

```
signal 'SNOWFALL' after 5         /* Signal the event snowfall      */
                                  /* after 5 seconds have elapsed.  */
                                  /* The event tag defaults to the  */
                                  /* event name, SNOWFALL, since    */
                                  /* no event tag was specified.    */
&#38;#8942;
cancel 'SNOWFALL'                 /* Cancels the event snowfall     */
                                  /* if 5 seconds have not elapsed. */

weather = 'ALL'                     /* Assign ALL to this variable. */
post 'RAIN' after 10 tag weather    /* Set up each of these events. */
reset 'SNOW' after 20               /* For the POST and SIGNAL      */
signal 'SLEET' after 30 tag weather /* events, assign an event tag  */
                                    /* equal to the value of the    */
                                    /* variable "weather".          */
&#38;#8942;
cancel weather                    /* Cancels all events with the    */
                                  /* tag ALL provided the specified */
                                  /* time has not expired; the      */
                                  /* RESET event is not canceled    */
                                  /* because ALL does not match the */
                                  /* defaulted event tag SNOW for   */
                                  /* this event.                    */
```

Note

When coded as an asynchronous subset statement (CANCEL DELAY or CANCEL SUSPEND), this statement **must** be coded directly following the THEN keyword on the ONIN, ONOUT, or ON SIGNALED statement.

CHARSET

```
CHARSET character_set
```

Where

character_set is a string constant from the following list: APL, FIELD, PSA, PSB, PSC, PSD, PSE, PSF, or '*yy*'x (where *yy* is 2 hexadecimal digits with a value of "40"x through "EF"x). You can use uppercase or lowercase letters. You must enclose the string constant in single or double quotation marks.

Function

The CHARSET statement simulates the action of the 3270 terminal key that selects the character set for subsequent data input. The statement is valid for simulation of these terminals only.

If you do not use the CHARSET statement, the character set will be determined by the extended field attribute byte value.

You can choose one of the following character sets:

APL	Selects the special APL/Text character set for APL characters, which must be sent using the 2-character graphic escape sequence.
FIELD	Specifies that the character set is determined by the extended field attribute byte. Use the FIELD character set identifier to return to the standard EBCDIC character set after an APL selection.
PSA	Selects the first Programmed Symbols (PS) character set defined for the terminal.
PSB	Selects the second PS.
PSC	Selects the third PS.
PSD	Selects the fourth PS.
PSE	Selects the fifth PS.
PSF	Selects the sixth PS.
'*yy*'x	Selects the Programmed Symbols character set named by the value for *yy*, where *yy* is a 2-digit hexadecimal string constant that has a value of "40"x through "EF"x.

Examples

```
charset 'apl'     /* Select the APL character set.        */
charset '45'x     /* Select PS character set named '45'x. */
```

CMACCP — Accept_Conversation

```
CMACCP (conversation_ID, return_code) [LOGGING log_byte]
```

Where

conversation_ID (character output) is the conversation identifier assigned to the conversation. CPI Communications supplies and maintains the *conversation_ID*. When the *return_code* is set equal to CM_OK, the value returned in this parameter is used by the program on all subsequent statements issued for this conversation.

return_code (numeric output) specifies the result of the statement execution.

log_byte is a 1-byte string constant or a string expression that specifies a byte of user data to be associated with this CPI-C statement during logging. *log_byte* is also associated with the FMH-5 data received when a CMACCP successfully completes. The *log_byte* remains active until another CPI-C statement is issued. This byte gives users of the Response Time Utility a way to identify transactions from transmitted or received data when gathering statistics by the various user-defined "log_byte" categories. Only the first character or first two hexadecimal digits of the string expression are used. The default *log_byte* is X'00'.

Function

A program uses the CMACCP statement to simulate the CPI-C CMACCP call, which is usedto accept an incoming conversation. Like CMINIT, the CMACCPstatement initializes values for various conversation characteristics. The difference between the two statements is that the program that will later allocate the conversation issues the CMINIT statement, and the partner program that will accept the conversation after it is allocated issues the CMACCP statement.

For more information on CMACCP and its parameters, refer to the Accept_Conversation (CMACCP) call description in the "Call Reference" chapter of *Systems Application Architecture Common Programming Interface Communications Reference*.

CMALLC — Allocate

```
CMALLC (conversation_ID, return_code ) [LOGGING log_byte]
```

Where

conversation_ID (character input) is the conversation identifier of a conversation that has been initialized with a CMINIT statement.

return_code (numeric output) specifies the result of the statement execution.

log_byte is a 1-byte string constant or a string expression that specifies a byte of user data to be associated with this CPI-C statement during logging. *log_byte* is also associated with the FMH-5 data transmitted when a CMALLC successfully completes. The *log_byte* remains active until another CPI-C statement is issued. This byte gives users of the Response Time Utility a way to identify transactions from transmitted or received data when gathering statistics by the various user-defined "log_byte" categories. Only the first character or first two hexadecimal digits of the string expression are used. The default *log_byte* is X'00'.

Function

A program uses the CMALLC statement to simulate the CPI-C CMALLC call, which is used to start a conversation with its partner program.

Before issuing the CMALLC statement, a program has the option of issuing one or more of the following statements to set allocation parameters:

CMSCT — Set_Conversation_Type

CMSMN — Set_Mode_Name

CMSPLN — Set_Partner_LU_Name

CMSRC — Set_Return_Code

CMSSL — Set_Sync_Level

CMSTPN — Set_TP_Name

For more information on CMALLC and its parameters, refer to the Allocate (CMALLC) call description in the "Call Reference" chapter of *Systems Application Architecture Common Programming Interface Communications Reference*.

CMCFM — Confirm

```
CMCFM (conversation_ID, request_to_send_received, return_code)
     [LOGGING log_byte]
```

Where

conversation_ID (character input) is the conversation identifier.

request_to_send_received (numeric_output) specifies the variable containing an indication of whether the remote program issued a CMRTS statement.

Note: If *return_code* is set to CM_PROGRAM_PARAMETER_CHECK or CM_PROGRAM_STATE_CHECK, the value contained in *request_to_send_received* is meaningless.

return_code (numeric output) specifies the result of the statement execution.

log_byte is a 1-byte string constant or a string expression that specifies a byte of user data to be associated with this CPI-C statement during logging. The *log_byte* remains active until another CPI-C statement is issued. Only the first character or first two hexadecimal digits of the string expression are used. The default *log_byte* is X'00'.

Function

A local program uses the CMCFM statement to simulate the CPI-C CMCFM call, which is used to send a confirmation request to the remote program and then wait for a reply. The remote program replies with a CMCFMD statement. The local and remote programs use the CMCFM and CMCFMD statements to synchronize their processing of the data.

Note: The *sync_level* conversation characteristic for the *conversation_ID* specified must be set to CM_CONFIRM to use this statement. Use the CMSSL (Set_Sync_Level) statement to set the conversation's synchronization level.

For more information on CMCFM and its parameters, refer to the Confirm (CMCFM) call description in the “Call Reference” chapter of *Systems Application Architecture Common Programming Interface Communications Reference*.

CMCFMD — Confirmed

```
CMCFMD (conversation_ID, return_code) [LOGGING log_byte]
```

Where

conversation_ID (character input) specifies the conversation identifier for the conversation on which CMCFM has been received.

return_code (numeric output) specifies the result of the statement execution.

log_byte is a 1-byte string constant or a string expression that specifies a byte of user data to be associated with this CPI-C statement during logging. The *log_byte* remains active until another CPI-C statement is issued. Only the first character or first two hexadecimal digits of the string expression are used. The default *log_byte* is X'00'.

Function

A program uses the CMCFMD statement to simulate the CPI-C CMCFMD call, which is usedto send a confirmation reply to the remote program. The local and remote programs can use the CMCFM and CMCFMD to synchronize their processing.

For more information on CMCFMD and its parameters, refer to the Confirmed (CMCFMD) call description in the “Call Reference” chapter of *Systems Application Architecture Common Programming Interface Communications Reference*.

CMDEAL — Deallocate

```
CMDEAL (conversation_ID, return_code) [LOGGING log_byte]
```

Where

conversation_ID (character input) specifies the conversation identifier of the conversation to be deallocated.

return_code (numeric output) specifies the result of the statement execution.

log_byte is a 1-byte string constant or a string expression that specifies a byte of user data to be associated with this CPI-C statement during logging. The *log_byte* remains active until another CPI-C statement is issued. Only the first character or first two hexadecimal digits of the string expression are used. The default *log_byte* is X'00'.

Function

A program uses the CMDEAL statement to simulate the CPI-C CMDEAL call, which is used to end a conversation. The CMDEAL statement can include the function of the CMFLUS or CMCFM statement, depending on the value of the *deallocate_type* conversation characteristic. The *conversation_ID* is no longer assigned when the conversation is deallocated as part of this statement.

Before issuing the CMDEAL statement, a program has the option of issuing one or both of the following statements to set deallocation parameters:

CMSDT — Set_Deallocate_Type

CMSLD — Set_Log_Data

For more information on CMDEAL and its parameters, refer to the Deallocate (CMDEAL) call description in the "Call Reference" chapter of *Systems Application Architecture Common Programming Interface Communications Reference*.

CMECS — Extract_Conversation_State

```
CMECS (conversation_ID, conversation_state, return_code)
     [LOGGING log_byte]
```

Where

conversation_ID (character input) specifies the conversation identifier of the desired conversation.

conversation_state (numeric output) specifies the conversation state that is returned to the local program. The *conversation_state* can be one of the following:

CM_INITIALIZE_STATE

CM_SEND_STATE
CM_RECEIVE_STATE
CM_SEND_PENDING_STATE
CM_CONFIRM_STATE
CM_CONFIRM_SEND_STATE
CM_CONFIRM_DEALLOCATE_STATE

return_code (numeric output) specifies the result of the statement execution.

log_byte is a 1-byte string constant or a string expression that specifies a byte of user data to be associated with this CPI-C statement during logging. The *log_byte* remains active until another CPI-C statement is issued. Only the first character or first two hexadecimal digits of the string expression are used. The default *log_byte* is X'00'.

Function

A program uses the CMECS statement to simulate the CPI-C CMECS call, which is usedto extract the conversation state for a given conversation. The value is returned in the *conversation_state* parameter.

For more information on CMECS and its parameters, refer to the Extract_Conversation_State (CMECS) call description in the “Call Reference” chapter of *Systems Application Architecture Common Programming Interface Communications Reference*.

CMECT — Extract_Conversation_Type

```
CMECT (conversation_ID, conversation_type, return_code)
     [LOGGING log_byte]
```

Where

conversation_ID (character input) specifies the conversation identifier of the desired conversation.

conversation_type (numeric output) specifies the conversation type that is returned to the local program. The *conversation_type* can be one of the following:
CM_BASIC_CONVERSATION
CM_MAPPED_CONVERSATION

return_code (numeric output) specifies the result of the statement execution.

log_byte is a 1-byte string constant or a string expression that specifies a byte of user data to be associated with this CPI-C statement during logging. The *log_byte* remains active until another CPI-C statement is issued. Only the first character or first two hexadecimal digits of the string expression are used. The default log_byte is X'00'.

Function

A program uses the CMECT statement to simulate the CPI-C CMECT call, which is used to extract the *conversation_type* characteristic's value for a given conversation. The value is returned in the *conversation_type* parameter.

For more information on CMECT and its parameters, refer to the Extract_Conversation_Type (CMECT) call description in the "Call Reference" chapter of *Systems Application Architecture Common Programming Interface Communications Reference*.

CMEMN — Extract_Mode_Name

```
CMEMN (conversation_ID, mode_name, mode_name_length, return_code),
     [LOGGING log_byte]
```

Where

conversation_ID (character input) specifies the conversation identifier of the desired conversation.

mode_name (character output) specifies the variable containing the mode name. The mode name designates the network properties for the session allocated, or to be allocated, which will carry the conversation specified by the *conversation_ID*.

mode_name_length (numeric output) specifies the variable containing the length of the returned *mode_name* parameter.

return_code (numeric output) specifies the result of the statement execution.

log_byte is a 1-byte string constant or a string expression that specifies a byte of user data to be associated with this CPI-C statement during logging. The *log_byte* remains active until another CPI-C statement is issued. Only the first character or first two hexadecimal digits of the string expression are used. The default log_byte is X'00'.

Function

A program uses the CMEMN statement to simulate the CPI-C CMEMN call, which is usedto extract the *mode_name* and *mode_name_length*characteristics' values for a given conversation. The values are returned in the *mode_name* and *mode_name_length*parameters.

For more information on CMEMN and its parameters, refer to the Extract_Mode_Name (CMEMN) call description in the "Call Reference" chapter of *Systems Application Architecture Common Programming Interface Communications Reference*.

CMEPLN — Extract_Partner_LU_Name

```
CMEPLN (conversation_ID, partner_LU_name, partner_LU_name_length,,return_code)
      [LOGGING log_byte]
```

Where

conversation_ID (character input) specifies the conversation identifier of the desired conversation.

partner_LU_name (character output) specifies the variable containing the name of the LU where the remote program is located.

partner_LU_name_length (numeric output) specifies the variable containing the length of the returned *partner_LU_name* parameter.

log_byte is a 1-byte string constant or a string expression that specifies a byte of user data to be associated with this CPI-C statement during logging. The *log_byte* remains active until another CPI-C statement is issued. Only the first character or first two hexadecimal digits of the string expression are used. The default log_byte is X'00'.

Function

A program uses the CMEPLN statement to simulate the CPI-C CMEPLN call, which is usedto extract the *partner_LU_name* and *partner_LU_name_length*characteristics' values for a given conversation. The values are returned in the *partner_LU_name* and *partner_LU_name_length*parameters.

For more information on CMEPLN and its parameters, refer to the Extract_Partner_LU_Name (CMEPLN) call description in the "Call Reference" chapter of *Systems Application Architecture Common Programming Interface Communications Reference*.

CMESL — Extract_Sync_Level

```
CMESL (conversation_ID, sync_level, return_code) [LOGGING log_byte]
```

Where

conversation_ID (character input) specifies the conversation identifier of the desired conversation.

sync_level (numeric output) specifies the variable containing the *sync_level* characteristic of the conversation. The *sync_level* variable can have one of the following values:

CM_NONE

CM_CONFIRM

return_code (numeric output) specifies the result of the statement execution.

log_byte is a 1-byte string constant or a string expression that specifies a byte of user data to be associated with this CPI-C statement during logging. The *log_byte* remains active until another CPI-C statement is issued. Only the first character or first two hexadecimal digits of the string expression are used. The default log_byte is X'00'.

Function

A program uses the CMESL statement to simulate the CPI-C CMESL call, which is usedto extract the *sync_level* characteristic's value for a given conversation. The value is returned in the *sync_level* parameter.

For more information on CMESL and its parameters, refer to the Extract_Sync_Level (CMESL) call description in the "Call Reference" chapter of *Systems Application Architecture Common Programming Interface Communications Reference*.

CMFLUS — Flush

```
CMFLUS (conversation_ID, return_code) [LOGGING log_byte]
```

Where

conversation_ID (character input) specifies the conversation identifier of the desired conversation.

return_code (numeric output) specifies the result of the statement execution.

log_byte is a 1-byte string constant or a string expression that specifies a byte of user data to be associated with this CPI-C statement during logging. The *log_byte* remains active until another CPI-C statement is issued. Only the first character or first two hexadecimal digits of the string expression are used. The default log_byte is X'00'.

Function

A program uses the CMFLUS statement to simulate the CPI-C CMFLUS call, which is usedto empty the local logical unit's send buffer for a given conversation. When notified by CPI Communications that a CMFLUS has been issued, the LU sends any information it has buffered to the remote LU. The information that can be buffered comes from the CMALLC, CMSEND, or CMSERR statement. For more details of when and how buffering occurs, refer to the call descriptions of those calls in the "Call Reference" chapter of *Systems Application Architecture Common Programming Interface Communications Reference*.

Also, for more information on CMFLUS and its parameters, refer to the Flush (CMFLUS) call description in the "Call Reference" chapter of *Systems Application*

Architecture Common Programming Interface Communications Reference.

CMINIT — Initialize_Conversation

```
CMINIT (conversation_ID, sym_dest_name, return_code) [LOGGING log_byte]
```

Where

conversation_ID (character output) specifies the conversation identifier assigned to the conversation. CPI Communications supplies and maintains the *conversation_ID*. If the CMINIT statement is successful (*return_code* is set to CM_OK), the local program uses the identifier returned in this variable for the rest of the conversation.

sym_dest_name (character input) specifies the symbolic destination name. The symbolic destination name is provided by the program and points to an entry in the side information table.[7] The appropriate entry in the side information is retrieved and used to initialize the conversation's characteristics (such as partner LU name, partner TP name, and mode name to be used for the session). Alternatively, a blank *sym_dest_name* (one composed of eight space characters) may be specified. When this is done, the program is responsible for setting up the appropriate destination information, using Set statements, before issuing the CMALLC statement for this conversation.

return_code (numeric output) specifies the result of the statement execution.

log_byte is a 1-byte string constant or a string expression that specifies a byte of user data to be associated with this CPI-C statement during logging. The *log_byte* remains active until another CPI-C statement is issued. Only the first character or first two hexadecimal digits of the string expression are used. The default log_byte is X'00'.

Function

A program uses the CMINIT statement to simulate the CPI-C CMINIT call, which is used to initialize values for various conversation characteristics before the conversation is allocated (by a CMALLC statement). The remote partner program uses the CMACCP statement to initialize values for the conversation characteristics on its end of the conversation.

A program can override the values that are initialized by this statement using the appropriate Set statements, such as CMSSL to set the synchronization level. Once the value is changed, it remains changed until the end of the conversation or until changed again by a Set statement.

For more information on CMINIT and its parameters, refer to the Initialize_Conversation (CMINIT) call description in the “Call Reference” chapter

7. You can find a discussion of the side information table in the “Simulating CPI-C Transaction Programs” chapter of *Creating WSim Scripts*. Further, you can find descriptions of the SIDEINFO operand of the APPCLU statement, and of the SIDEENT, SIDEINFO, and SIDEEND statements that make up the SIDEINFO group, in Part 1, “WSim language statements,” on page 1.

of *Systems Application Architecture Common Programming Interface Communications Reference*.

CMPTR — Prepare_To_Receive

```
CMPTR (conversation_ID, return_code) [LOGGING log_byte]
```

Where

conversation_ID (character input) specifies the conversation identifier of the desired conversation.

return_code (numeric output) specifies the result of the statement execution.

log_byte is a 1-byte string constant or a string expression that specifies a byte of user data to be associated with this CPI-C statement during logging. The *log_byte* remains active until another CPI-C statement is issued. Only the first character or first two hexadecimal digits of the string expression are used. The default log_byte is X'00'.

Function

A program uses the CMPTR statement to simulate the CPI-C CMPTR call, which is used to change the conversation from **Send** to **Receive** state in preparation for receiving data. The change to **Receive** state can be either completed as part of this statement or deferred until the program issues a CMFLUS or CMCFM statement. When the change to **Receive** state is completed as part of this statement, it may include the function of the CMFLUS or CMCFM statement. This statement's function is determined by the value of the *prepare_to_receive_type* conversation characteristic. See "CMSPTR — Set_Prepare_To_Receive_Type" on page 390 for more information about the *prepare_to_receive_type* conversation characteristic.

Before issuing the CMPTR statement, a program has the option of issuing the following statement which affects the function of the CMPTR statement:

CMSPTR — Set_Prepare_To_Receive_Type

For more information on CMPTR and its parameters, refer to the Prepare_To_Receive (CMPTR) call description in the "Call Reference" chapter of *Systems Application Architecture Common Programming Interface Communications Reference*.

CMRCV — Receive

```
CMRCV (conversation_ID, receive_buffer, requested_length, data_received,,
      received_length, status_received, request_to_send_received,,
      return_code) [LOGGING log_byte]
```

Where

conversation_ID (character input) specifies the conversation identifier of the desired conversation.

receive_buffer (character output) specifies the variable in which the program is to receive data. This parameter contains data only if *return_code* is set to CM_OK or CM_DEALLOCATED_NORMAL and *data_received* is not set to CM_NO_DATA_RECEIVED.

requested_length (numeric input) specifies the maximum amount of data the program is to receive. Valid *requested_length* values range from 0 to 32767.

data_received (numeric output) specifies whether the program received data. Unless *return_code* is set to CM_OK or CM_DEALLOCATED_NORMAL, the value contained in *data_received* is meaningless. The *data_received* variable can have one of the following values:

- CM_NO_DATA_RECEIVED (basic and mapped conversations)
- CM_DATA_RECEIVED (basic conversations only)
- CM_COMPLETE_DATA_RECEIVED (basic and mapped conversations)
- CM_INCOMPLETE_DATA_RECEIVED (basic and mapped conversations)

received_length (numeric output) specifies the variable containing the amount of data the program received, up to the maximum. If the program does not receive data on this statement, the value contained in *received_length* is meaningless.

status_received (numeric output) specifies the variable containing an indication of the conversation status. The *status_received* variable can have one of the following values:

- CM_NO_STATUS_RECEIVED
- CM_SEND_RECEIVED
- CM_CONFIRM_RECEIVED
- CM_CONFIRM_SEND_RECEIVED
- CM_CONFIRM_DEALLOC_RECEIVED

request_to_send_received (numeric_output) specifies the variable containing an indication of whether the remote program issued a CMRTS statement. If *return_code* is set to CM_PROGRAM_PARAMETER_CHECK or CM_PROGRAM_STATE_CHECK, the value contained in *request_to_send_received* is meaningless. The *request_to_send_received* variable can have one of the following values:

- CM_REQ_TO_SEND_RECEIVED
- CM_REQ_TO_SEND_NOT_RECEIVED

return_code (numeric output) specifies the result of the statement execution. The return codes that can be returned depend on the state and characteristics of the conversation at the time this statement is issued.

log_byte is a 1-byte string constant or a string expression that specifies a byte of user data to be associated with this CPI-C statement during logging. *log_byte* is also associated with the data which may be received when a CMRCV successfully completes. The *log_byte* remains active until another CPI-C statement is issued. This byte gives users of the Response Time Utility a way to identify transactions from transmitted or received data when gathering statistics by the various user-defined "log_byte" categories. Only the first character or first two hexadecimal digits of the string expression are used. The default *log_byte* is X'00'.

Function

A program uses the CMRCV statement to simulate the CPI-C CMRCV call, which is used to receive information from a given conversation. The information received can be a data record (on a mapped conversation), data (on a basic conversation), conversation status, or a request for confirmation.

Before issuing the CMRCV statement, a program has the option of issuing one or both of the following statements which affect the function of the CMRCV Statement:

CMSF — Set_Fill

CMSRT — Set_Receive_Type

For more information on CMRCV and its parameters, refer to the Receive (CMRCV) call description in the "Call Reference" chapter of *Systems Application Architecture Common Programming Interface Communications Reference*.

CMRTS — Request_To_Send

```
CMRTS (conversation_ID, return_code) [LOGGING log_byte]
```

Where

conversation_ID (character input) specifies the conversation identifier of the desired conversation.

return_code (numeric output) specifies the result of the statement execution.

log_byte is a 1-byte string constant or a string expression that specifies a byte of user data to be associated with this CPI-C statement during logging. The *log_byte* remains active until another CPI-C statement is issued. Only the first character or first two hexadecimal digits of the string expression are used. The default log_byte is X'00'.

Function

A local program uses the CMRTS statement to simulate the CPI-C CMRTS call, which is used to notify the remote program that the local program would like to enter **Send** state for a given conversation.

For more information on CMRTS and its parameters, refer to the Request_To_Send (CMRTS) call description in the "Call Reference" chapter of *Systems Application Architecture Common Programming Interface Communications Reference*.

CMSCT — Set_Conversation_Type

```
CMSCT (conversation_ID, conversation_type, return_code)
     [LOGGING log_byte]
```

Where

conversation_ID (character input) specifies the conversation identifier of the desired conversation.

conversation_type (numeric input) specifies the type of conversation to be allocated when CMALLC is issued. The *conversation_type* variable can have one of the following values:

CM_BASIC_CONVERSATION

CM_MAPPED_CONVERSATION

return_code (numeric output) specifies the result of the statement execution.

log_byte is a 1-byte string constant or a string expression that specifies a byte of user data to be associated with this CPI-C statement during logging. The *log_byte* remains active until another CPI-C statement is issued. Only the first character or first two hexadecimal digits of the string expression are used. The default log_byte is X'00'.

Function

A program uses the CMSCT statement to simulate the CPI-C CMSCT call, which is used to set the *conversation_type* characteristic for a given conversation. The CMSCT statement overrides the value assigned when the CMINIT statement was issued.

Note: The CMSCT statement can be issued only after the CMINIT for the conversation has completed, and before the CMALLC is issued.

For more information on CMSCT and its parameters, refer to the Set_Conversation_Type (CMSCT) call description in the “Call Reference” chapter of *Systems Application Architecture Common Programming Interface Communications Reference*.

CMSDT — Set_Deallocate_Type

```
CMSDT (conversation_ID, deallocate_type, return_code) [LOGGING log_byte]
```

Where

conversation_ID (character input) specifies the conversation identifier of the desired conversation.

deallocate_type (numeric input) specifies the type of deallocation to be performed. The *deallocate_type* variable can have one of the following values:

CM_DEALLOCATE_SYNC_LEVEL
CM_DEALLOCATE_FLUSH
CM_DEALLOCATE_CONFIRM
CM_DEALLOCATE_ABEND

return_code (numeric output) specifies the result of the statement execution.

log_byte is a 1-byte string constant or a string expression that specifies a byte of user data to be associated with this CPI-C statement during logging. The *log_byte* remains active until another CPI-C statement is issued. Only the first character or first two hexadecimal digits of the string expression are used. The default log_byte is X'00'.

Function

A program uses the CMSDT statement to simulate the CPI-C CMSDT call, which is usedto set the *deallocate_type* characteristic for a given conversation. The CMSDT statement overrides the value assigned when the CMINIT or CMACCP statement was issued.

For more information on CMSDT and its parameters, refer to the Set_Deallocate_Type (CMSDT) call description in the "Call Reference" chapter of *Systems Application Architecture Common Programming Interface Communications Reference*.

CMSED — Set_Error_Direction

```
CMSED (conversation_ID, error_direction, return_code) [LOGGING log_byte]
```

Where

conversation_ID (character input) specifies the conversation identifier of the desired conversation.

error_direction (numeric input) specifies the direction of the data flow in which the program detected an error. The parameter is significant only if CMSERR is issued in the **Send_Pending** state (that is, immediately after a CMRCV on which both data and a conversation status of CM_SEND_RECEIVED are received). Otherwise, the *error_direction* value is ignored when the program issues CMSERR.

The *error_direction* variable can have one of the following values:

CM_RECEIVE_ERROR
CM_SEND_ERROR

return_code (numeric output) specifies the result of the statement execution.

log_byte is a 1-byte string constant or a string expression that specifies a byte of user data to be associated with this CPI-C statement during logging. The *log_byte* remains active until another CPI-C statement is issued. Only the first character or first two hexadecimal digits of the string expression are used. The default log_byte is X'00'.

Function

A program uses the CMSED statement to simulate the CPI-C CMSED call, which is usedto set the *error_direction* characteristic for a given conversation. The CMSED statementoverrides the value assigned when the CMINIT or CMACCP statement was issued.

For more information on CMSED and its parameters, refer to the Set_Error_Direction (CMSED) call description in the "Call Reference" chapter of *Systems Application Architecture Common Programming Interface Communications Reference*.

CMSEND — Send_Data

```
CMSEND (conversation_ID, send_buffer, send_length,,
      request_to_send_received, return_code) [LOGGING log_byte]
```

Where

conversation_ID (character input) specifies the conversation identifier of the desired conversation.

send_buffer (character input) specifies the information to be sent. The form of the information depends on the conversation type.

For a basic conversation, *send_buffer* specifies the data to be sent. The data consists of logical records, each containing a 2-byte length field followed by the data field. The length of the data field can range from 0 to 32765 bytes. The length of the record equals the length of the data field plus the 2-byte length field.

For a mapped conversation, *send_buffer* specifies the data record to be sent. The length of the data record is given by the *send_length* parameter.

send_length (numeric input) specifies the size of the *send_buffer* parameter and the number of bytes to be sent on the conversation. The meaning of *send_length* depends on the conversation type:

For basic conversations, the *send_length* ranges in value from 0 to 32767. When a program issues a CMSEND statement during a basic conversation, *send_length* specifies the size of the *send_buffer* parameter and is **not** related to the length of a logical record.

For mapped conversations, the *send_length* ranges in value from 0 to 32763. When a program issues a CMSEND statement during a mapped conversation, *send_length* specifies the length of a data record.

request_to_send_received (numeric output) specifies the variable containing an indication of whether or not a request-to-send notification has been received. The *request_to_send_received* variable can have one of the following values:

CM_REQ_TO_SEND_RECEIVED

CM_REQ_TO_SEND_NOT_RECEIVED

return_code (numeric output) specifies the result of the statement execution.

log_byte is a 1-byte string constant or a string expression that specifies a byte of user data to be associated with this CPI-C statement during logging. *log_byte* is also associated with the data which may be transmitted when a CMSEND successfully completes. The *log_byte* remains active until another CPI-C statement is issued. This byte gives users of the Response Time Utility a way to identify transactions from transmitted or received data when gathering statistics by the various user-defined “log_byte” categories. Only the first character or first two hexadecimal digits of the string expression are used. The default *log_byte* is X'00'.

Function

A local program uses the CMSEND statement to simulate the CPI-C CMSEND call, which is used to send data to the remote program.

When issued during a mapped conversation, the CMSEND statement sends one data record to the remote program. The data record consists entirely of data and is not examined by the LU for possible logical records.

When issued during a basic conversation, this statement sends data to the remote program. The data consists of logical records. The amount of data is specified independently of the data format.

Before issuing the CMSEND statement, a program has the option of issuing one or more of the following statements which affect the function of the CMSEND statement:

CMSST — Set_Send_Type

If *send_type* = CM_SEND_AND_PREP_TO_RECEIVE, optional setup may include: CMSPTR — Set_Prepare_To_Receive_Type

If *send_type* = CM_SEND_AND_DEALLOCATE, optional setup may include: CMSDT — Set_Deallocate_Type

For more information on CMSEND and its parameters, refer to the Send_Data (CMSEND) call description in the “Call Reference” chapter of *Systems Application Architecture Common Programming Interface Communications Reference*.

CMSERR — Send_Error

```
CMSERR (conversation_ID, request_to_send_received, return_code),
       [LOGGING log_byte]
```

Where

conversation_ID (character input) specifies the conversation identifier of the desired conversation.

request_to_send_received (numeric output) specifies the variable containing an indication of whether or not a request-to-send notification has been received. The *request_to_send_received* variable can have one of the following values:

CM_REQ_TO_SEND_RECEIVED
CM_REQ_TO_SEND_NOT_RECEIVED

return_code (numeric output) specifies the result of the statement execution.

log_byte is a 1-byte string constant or a string expression that specifies a byte of user data to be associated with this CPI-C statement during logging. *log_byte* is also associated with the error log_data which may be transmitted when a CMSERR successfully completes. The *log_byte* remains active until another CPI-C statement is issued. This byte gives users of the Response Time Utility a way to identify transactions from transmitted or received data when gathering statistics by the various user-defined "log_byte" categories. Only the first character or first two hexadecimal digits of the string expression are used. The default *log_byte* is X'00'.

Function

A local program uses the CMSERR statement to simulate the CPI-C CMSERR call, which is used to inform the remote program that the local program detected an error during the conversation. If the conversation is in **Send** state, the CMSERR statement forces the LU to flush its send buffer.

When this call completes successfully, the local program's end of the conversation is in **Send** state and the remote program's end is in **Receive** state. Further action is defined by program logic.

Beforeissuing the CMSERR statement, a program has the option of issuing one or more of the following statements which affect the function of the CMSERR statement:

CMSED — Set_Error_Direction
CMSLD — Set_Log_Data

For more information on CMSERR and its parameters, refer to the Send_Error (CMSERR) call description in the "Call Reference" chapter of *Systems Application Architecture Common Programming Interface Communications Reference*.

CMSF — Set_Fill

```
CMSF (conversation_ID, fill, return_code) [LOGGING log_byte]
```

Where

conversation_ID (character input) specifies the conversation identifier of the desired conversation.

fill (numeric input) specifies whether the program is to receive data in terms of the logical-record format of the data or independent of the logical-record format. The *fill* variable can have one of the following values:

CM_FILL_LL
CM_FILL_BUFFER

return_code (numeric output) specifies the result of the statement execution.

log_byte is a 1-byte string constant or a string expression that specifies a byte of user data to be associated with this CPI-C statement during logging. The *log_byte* remains active until another CPI-C statement is issued. Only the first character or first two hexadecimal digits of the string expression are used. The default log_byte is X'00'.

Function

A program uses the CMSF statement to simulate the CPI-C CMSF call, which is used to set the *fill* characteristic for a given conversation. The CMSF statement overrides the value that was assigned when the CMINIT or CMACCP statement was issued.

Note: This statement applies only to basic conversations. The fill characteristic is ignored for mapped conversations.

For more information on CMSF and its parameters, refer to the Set_Fill (CMSF) call description in the "Call Reference" chapter of *Systems Application Architecture Common Programming Interface Communications Reference*.

CMSFM5 — Set_FM_Header_5_Extension

```
CMSFM5 (conversation_ID, FMH5_extension, FMH5_extension_length,,
       return_code) [LOGGING log_byte]
```

Where

conversation_ID (character input) specifies the conversation identifier of the desired conversation.

FMH5_extension (character input) specifies the information beyond the base FMH-5. All of the extension data for the conversation must appear on a single CMSFM5 statement. Subsequent CMSFM5 statements for this conversation overwrite any extension information previously supplied.

The format is a character string from two to 255 characters long and must appear exactly as it would appear in an FMH-5.

FMH5_extension_length (numeric input) specifies the total length of the extension. The format is an integer value from two to 255. If an incorrect length is specified, the statement completes with a parameter check return code.

return_code (numeric output) specifies the result of the statement execution.

log_byte is a 1-byte string constant or a string expression that specifies a byte of user data to be associated with this CPI-C statement during logging. The *log_byte* remains active until another CPI-C statement is issued. Only the first character or first two hexadecimal digits of the string expression are used. The default log_byte is X'00'.

Function

The CMSFM5 statement provides FMH-5 extension data to WSim during a CPI-C simulation. The CMSFM5 statement is an extension to the CPI-C architecture. A program uses the CMSFM5 statement to provide additional information beyond the base FMH-5 that may be required for an Attach request. This additional information is called FMH-5 extension data, and can include security subfields, logical unit of work, conversation correlator, and PIP data.

For more information on the format of the base FMH-5, refer to *VTAM Programming for LU 6.2*. The CMSFM5 statement is used to supply all data that is required beyond the base FMH-5 (defined by the ISTFM5 DSECT). This data is specified by the *FMH5_extension* parameter. The format of this parameter is a character string from 2 to 255 bytes long. The extension data must be specified exactly as it would appear in an FMH-5. For example, if both access security information and a conversation correlator are required; specify the data defined by the FM5ASI DSECT, followed by the data defined by the FM5CVCOR DSECT. The *FMH5_extension_length* parameter must reflect the total length of the FMH-5 extension data. The format of this parameter is an integer from 2 to 255. If an incorrect length is specified by this parameter, or within the extension data, the CMSFM5 verb will complete with a parameter check return code.

CMSLD — Set_Log_Data

```
CMSLD (conversation_ID, log_data, log_data_length, return_code),
       [LOGGING log_byte]
```

Where

conversation_ID (character input) specifies the conversation identifier of the desired conversation.

log_data (character input) specifies the program-unique error information that is to be logged. The data supplied by the program is any data that the program wants logged. The data must be from character set 00640.

log_data_length (numeric input) specifies the length of the program-unique error information. The length can be from 0 to 512 bytes. If zero, the *log_data_length* characteristic is set to zero (effectively setting the *log_data* characteristic to the null string), and the *log_data* parameter of this statement is ignored.

return_code (numeric output) specifies the result of the statement execution.

log_byte is a 1-byte string constant or a string expression that specifies a byte of user data to be associated with this CPI-C statement during logging. The *log_byte* remains active until another CPI-C statement is issued. The default log_byte is X'00'. Only the first character or first two hexadecimal digits of the string expression are used.

Function

A program uses the CMSLD statement to simulate the CPI-C CMSLD call, which is used to set the *log_data* and *log_data_length* characteristics for a given conversation. The CMSLD statement overrides the values that were assigned when the CMINIT or CMACCP statement was issued.

Note: This statement applies only to basic conversations. The *log_data* characteristic is ignored for mapped conversations.

For more information on CMSLD and its parameters, refer to the Set_Log_Data (CMSLD) call description in the "Call Reference" chapter of *Systems Application Architecture Common Programming Interface Communications Reference*.

CMSMN — Set_Mode_Name

```
CMSMN (conversation_ID, mode_name, mode_name_length, return_code),
      [LOGGING log_byte]
```

Where

conversation_ID (character input) specifies the conversation identifier of the desired conversation.

mode_name (character input) specifies the mode name designating the network properties for the session to be allocated to the conversation. The network properties include, for example, the class of service to be used, and whether data is encrypted. The mode name must be from character set 01134. (A program must have special authority to specify a mode name that is used by SNA service transaction programs only, such as SNASVCMG.)

mode_name_length (numeric input) specifies the length of the mode name. The length can be from zero to eight bytes. If zero, the mode name for this conversation is set to null and the *mode_name* parameter included with this statement is not significant.

return_code (numeric output) specifies the result of the statement execution.

log_byte is a 1-byte string constant or a string expression that specifies a byte of user data to be associated with this CPI-C statement during logging. The *log_byte* remains active until another CPI-C statement is issued. Only the first character or first two hexadecimal digits of the string expression are used. The default log_byte is X'00'.

Function

A program uses the CMSMN statement to simulate the CPI-C CMSMN call, which is used to set the *mode_name* and *mode_name_length* characteristics for a conversation. The CMSMN statement overrides the current values, which were originally acquired from the side information using the *sym_dest_name*. Issuing this statement does not change the values in the side information. It only changes the *mode_name* and *mode_name_length* for this conversation.

Note: The CMSMN statement can be issued onlyafter the CMINIT for the conversation has completed, and before the CMALLC is issued.

For more information on CMSMN and its parameters, refer to the Set_Mode_Name (CMSMN) call description in the "Call Reference" chapter of *Systems Application Architecture Common Programming Interface Communications Reference*.

CMSPLN — Set_Partner_LU_Name

```
CMSPLN (conversation_ID, partner_LU_name,partner_LU_name_length,,
       return_code) [LOGGING log_byte]
```

Where

conversation_ID (character input) specifies the conversation identifier of the desired conversation.

partner_LU_name (character input) specifies the name of the remote LU at which the remote program is located. This LU name is any name by which the local LU knows the remote LU for purposes of allocating a conversation. The partner LU name must be from character set 01134.

partner_LU_name_length (numeric input) specifies the length of the partner LU name. This value can be from 1 to 17.

return_code (numeric output) specifies the result of the statement execution.

log_byte is a 1-byte string constant or a string expression that specifies a byte of user data to be associated with this CPI-C statement during logging. The *log_byte* remains active until another CPI-C statement is issued. Only the first character or first two hexadecimal digits of the string expression are used. The default log_byte is X'00'.

Function

A program uses the CMSPLN statement to simulate the CPI-C CMSPLN call, which is used to set the *partner_LU_name* and *partner_LU_name_length* characteristics for a given conversation. The CMSPLN statement overrides the current values, which were originally acquired from the side information using the *sym_dest_name*. Issuing this statement does not change the values in the side information. It only changes the *partner_LU_name* and *partner_LU_name_length* for this conversation.

Note: The CMSPLN statement can be issued only after the CMINIT for the conversation has completed,and before the CMALLC is issued.

For more information on CMSPLN and its parameters, refer to the Set_Partner_LU_Name (CMSPLN) call description in the "Call Reference" chapter of *Systems Application Architecture Common Programming Interface Communications Reference*.

CMSPTR — Set_Prepare_To_Receive_Type

```
CMSPTR (conversation_ID, prepare_to_receive_type, return_code),
       [LOGGING log_byte]
```

Where

conversation_ID (character input) specifies the conversation identifier of the desired conversation.

prepare_to_receive_type (numeric input) specifies the type of prepare-to-receive processing to be performed for this conversation. The *prepare_to_receive_type* variable can have one of the following values:

CM_PREP_TO_RECEIVE_SYNC_LEVEL
CM_PREP_TO_RECEIVE_FLUSH
CM_PREP_TO_RECEIVE_CONFIRM

return_code (numeric output) specifies the result of the statement execution.

log_byte is a 1-byte string constant or a string expression that specifies a byte of user data to be associated with this CPI-C statement during logging. The *log_byte* remains active until another CPI-C statement is issued. Only the first character or first two hexadecimal digits of the string expression are used. The default log_byte is X'00'.

Function

A program uses the CMSPTR statement to simulate the CPI-C CMSPTR call, which is usedto set the *prepare_to_receive_type*characteristic for a given conversation. The CMSPTR statement overrides the value that was assigned when the CMINIT or CMACCP statement was issued.

For more information on CMSPTR and its parameters, refer to the Set_Prepare_To_Receive_Type (CMSPTR) call description in the "Call Reference" chapter of *Systems Application Architecture Common Programming Interface Communications Reference*.

CMSRC — Set_Return_Control

```
CMSRC (conversation_ID, return_control, return_code) [LOGGING log_byte]
```

Where

conversation_ID (character input) specifies the conversation identifier of the desired conversation.

return_control (numeric input) specifies when a program receives control back after issuing a CMALLC statement. The *return_control* variable can have one of the following values:

CM_WHEN_SESSION_ALLOCATED

CM_IMMEDIATE

return_code (numeric output) specifies the result of the statement execution.

log_byte is a 1-byte string constant or a string expression that specifies a byte of user data to be associated with this CPI-C statement during logging. The *log_byte* remains active until another CPI-C statement is issued. Only the first character or first two hexadecimal digits of the string expression are used. The default log_byte is X'00'.

Function

A program uses the CMSRC statement to simulate the CPI-C CMSRC call, which is used to set the *return_control* characteristic for a given conversation. The CMSRC statement overrides the value that was assigned when the CMINIT statement was issued.

Note: The CMSRC statement can be issued only after the CMINIT for the conversation has completed,and before the CMALLC is issued.

For more information on CMSRC and its parameters, refer to the Set_Return_Control (CMSRC) call description in the "Call Reference" chapter of *Systems Application Architecture Common Programming Interface Communications Reference*.

CMSRT — Set_Receive_Type

```
CMSRT (conversation_ID, receive_type, return_code) [LOGGING log_byte]
```

Where

conversation_ID (character input) specifies the conversation identifier of the desired conversation.

receive_type (numeric input) specifies the type of receive to be performed. The *receive_type* variable can have one of the following values:

CM_RECEIVE_AND_WAIT

CM_RECEIVE_IMMEDIATE

return_code (numeric output) specifies the result of the statement execution.

log_byte is a 1-byte string constant or a string expression that specifies a byte of user data to be associated with this CPI-C statement during logging. The *log_byte* remains active until another CPI-C statement is issued. Only the first character or first two hexadecimal digits of the string expression are used. The default log_byte is X'00'.

Function

A program uses the CMSRT statement to simulate the CPI-C CMSRT call, which is usedto set the *receive_type*characteristic for a conversation. The CMSRT statement overrides the value that was assigned when the CMINIT or CMACCP statement was issued.

For more information on CMSRT and its parameters, refer to the Set_Receive_Type (CMSRT) call description in the “Call Reference” chapter of *Systems Application Architecture Common Programming Interface Communications Reference*.

CMSSL — Set_Sync_Level

```
CMSSL (conversation_ID, sync_level, return_code) [LOGGING log_byte]
```

Where

conversation_ID (character input) specifies the conversation identifier of the desired conversation.

sync_level (numeric input) specifies the synchronization level that the local and remote programs can use on this conversation. The *sync_level* can have one of the following values:

CM_NONE
CM_CONFIRM

Note: WSim does not support the sync-point synchronization level.

return_code (numeric output) specifies the result of the statement execution.

log_byte is a 1-byte string constant or a string expression that specifies a byte of user data to be associated with this CPI-C statement during logging. The *log_byte* remains active until another CPI-C statement is issued. Only the first character or first two hexadecimal digits of the string expression are used. The default log_byte is X'00'.

Function

A program uses the CMSSL statement to simulate the CPI-C CMSSL call, which is used to set the *sync_level* characteristic for a given conversation. The *sync_level* characteristic is used to specify the level of synchronization processing between the two programs. WSim supports either no synchronization or confirmation-level synchronization (using the CMCFM or CMCFMD statements). The CMSSL statement overrides the value that was assigned when the CMINIT statement was issued.

Note: The CMSSL statement can be issued only after theCMINIT for the conversation has completed, and before the CMALLC is issued.

For more information on CMSSL and its parameters, refer to the Set_Sync_Level (CMSSL) call description in the "Call Reference" chapter of *Systems Application Architecture Common Programming Interface Communications Reference*.

CMSST — Set_Send_Type

```
CMSST (conversation_ID, send_type, return_code) [LOGGING log_byte]
```

Where

conversation_ID (character input) specifies the conversation identifier of the desired conversation.

send_type (numeric input) specifies what, if any, information is to be sent to the remote program in addition to any data supplied on the CMSEND call, and whether the data is to be sent immediately or buffered.

The *send_type* variable can have one of the following values:

CM_BUFFER_DATA
CM_SEND_AND_FLUSH
CM_SEND_AND_CONFIRM
CM_SEND_AND_PREP_TO_RECEIVE
CM_SEND_AND_DEALLOCATE

return_code (numeric output) specifies the result of the statement execution.

log_byte is a 1-byte string constant or a string expression that specifies a byte of user data to be associated with this CPI-C statement during logging. The *log_byte* remains active until another CPI-C statement is issued. Only the first character or first two hexadecimal digits of the string expression are used. The default log_byte is X'00'.

Function

A program uses the CMSST statement to simulate the CPI-C CMSST call, which is usedto set the *send_type*characteristic for a conversation. The CMSST statement overrides the value that was assigned when the CMINIT or CMACCP statement was issued.

For more information on CMSST and its parameters, refer to the Set_Send_Type (CMSST) call description in the "Call Reference" chapter of *Systems Application Architecture Common Programming Interface Communications Reference*.

CMSTPN — Set_TP_Name

```
CMSTPN (conversation_ID, TP_name, TP_name_length, return_code),
       [LOGGING log_byte]
```

Where

conversation_ID (character input) specifies the conversation identifier of the desired conversation.

TP_name (character input) specifies the name of the remote program. A program with the appropriate privilege can specify the name of an SNA service transaction program. The TP name must be from character set 00640.

TP_name_length (numeric input) specifies the length of the *TP_name*. The value can be from 1 to 64.

return_code (numeric output) specifies the result of the statement execution.

log_byte is a 1-byte string constant or a string expression that specifies a byte of user data to be associated with this CPI-C statement during logging. The *log_byte* remains active until another CPI-C statement is issued. Only the first character or first two hexadecimal digits of the string expression are used. The default log_byte is X'00'.

Function

A program uses the CMSTPN statement to simulate the CPI-C CMSTPN call, which is used to set the *TP_name* and *TP_name_length* characteristics for a given conversation. The CMSTPN statement overrides the current value that was originally acquired from the side information using the *sym_dest_name*. Issuing this statement does not change the value of *TP_name* in the side information. It only changes the *TP_name* for this conversation.

Note: The CMSTPN statement can be issued onlyafter the CMINIT for the conversation has completed, and before the CMALLC is issued.

For more information on CMSTPN and its parameters, refer to the Set_TP_Name (CMSTPN) call description in the "Call Reference" chapter of *Systems Application Architecture Common Programming Interface Communications Reference*.

CMTRTS — Test_Request_To_Send_Received

```
CMTRTS (conversation_ID, request_to_send_received, return_code),
       [LOGGING log_byte]
```

Where

conversation_ID (character input) specifies the conversation identifier of the desired conversation.

request_to_send_received (numeric output) specifies the variable containing an indication of whether or not a request-to-send notification has been received. The *request_to_send_received* variable can have one of the following values:

CM_REQUEST_TO_SEND_RECEIVED

CM_REQUEST_TO_SEND_NOT_RECEIVED

Unless *return_code* is set to CM_OK, the value of *request_to_send_received* is not meaningful.

return_code (numeric output) specifies the result of the statement execution.

log_byte is a 1-byte string constant or a string expression that specifies a byte of user data to be associated with this CPI-C statement during logging. The *log_byte* remains active until another CPI-C statement is issued. Only the first character or first two hexadecimal digits of the string expression are used. The default log_byte is X'00'.

Function

A program uses the CMTRTS statement to simulate the CPI-C CMTRTS call, which is usedto determine whether a request-to-send notification has been received from the remote program for the specified conversation.

For more information on CMTRTS and its parameters, refer to the Test_Request_To_Send_Received (CMTRTS) call description in the “Call Reference” chapter of *Systems Application Architecture Common Programming Interface Communications Reference*.

COLOR

```
COLOR color_specification
```

Where

color_specification is a string constant taken from the following list:
- BLUE
- FIELD
- GREEN
- PINK
- RED
- TURQUOISE
- WHITE
- YELLOW

You can use uppercase or lowercase letters. You must enclose the string constant in single or double quotation marks.

Function

The COLOR statement simulates the action of the 3270 key that selects the color for displaying data. This statement is valid only for 3270 or LU Type 2 simulations.

If you do not code the COLOR statement, the extended field attribute byte value determines the color.

Use the “FIELD” string constant to select the color defined by the extended field attribute byte.

Examples

```
type 'Hello'      /* The color of "Hello" is determined by the */
                  /* extended field attribute.                 */
color 'red'       /* Display the following data in red.        */
type 'Goodbye'    /* "Goodbye" will be red.                    */
```

CONSTANT

```
CONSTANT name {integer_constant_expression}
              {string_constant_expression}
```

Where

name is a 1-character to 32-character name conforming to the rules for STL variable names. (See “Using variables and constants” on page 237 for more information about variable naming rules.)

integer_constant_expression is an integer constant or an expression involving only integer constants.

string_constant_expression is a string constant or an expression involving only string constants.

Function

The CONSTANT statement declares a named integer or string constant to the STL Translator. Once declared, the translator substitutes the value of the specified constant expression whenever it encounters *name* in the program.

Examples

```
constant a 1          /* "A" is a named constant with an */
                      /* integer value of 1.             */
constant x a+5        /* "X" is a named constant with an */
                      /* integer value of 6.             */
constant s 'Hello'    /* "S" is a named constant with a  */
                      /* string value of 'Hello'.        */
constant t 'there'    /* "T" is a named constant with a  */
                      /* string value of 'there'.        */
constant y s' 't      /* "Y" is a named constant with a  */
                      /* string value of 'Hello there'.  */
&#38;#8942;
b = a             /* Equivalent to: b = 1.             */
b = 5*x           /* Equivalent to: b = 30.            */
c = s             /* Equivalent to: c = 'Hello'.       */
c = s t           /* Equivalent to: c = 'Hello' 'there'. */
c = y             /* Equivalent to: c = 'Hello there'.   */
```

Notes

- Use named constants instead of string or integer variables whenever possible because they do not require counters, save areas, or switches.
- The CONSTANT statement is a declarative statement. You can code it only **outside** an STL procedure.

CTAB

```
CTAB
```

Function

The CTAB statement conditionally tabs to the next field if the cursor is not currently at the beginning of a field. This statement is valid for simulation of 3270 terminals only.

Examples

```
type '1234'
ctab         /* If the previous entry did not fill the input */
             /* field, tab to the next input field.          */
```

CURSOR

```
CURSOR({integer_expression}[,integer_expression])
       {direction}
```

Where

integer expression is an integer expression.

direction is a string constant taken from the following list: UP, DOWN, LEFT, RIGHT. You can use uppercase or lowercase letters. You must enclose the string constant in single or double quotation marks.

Function

The CURSOR statement positions the cursor at an absolute screen location or moves the cursor up, down, left, or right relative to its current location. This statement has no effect for nondisplay terminals.

If you specify a single integer expression, the argument is interpreted as an absolute screen offset (offset zero is the first screen position).

If you specify two integer expressions, the first is interpreted as an absolute row number and the second as an absolute column number (row 1, column 1 is the first position on the screen).

Valid row and column designations must be from 1 to 255. Valid offset designations must be from 0 to 32766.

If you specify a direction (“UP”, “DOWN”, “LEFT”, or “RIGHT”), the cursor will be moved in the indicated direction. If you specify an integer expression as a second argument, the cursor will be moved that number of spaces. If you do not specify a second argument, the cursor will be moved one space.

Examples

```
cursor(1,1)             /* Position the cursor at row 1,    */
                        /* column 1.                        */
cursor(1000)            /* Position the cursor at offset    */
                        /* 1000 on the screen.              */
row_number = 12
cursor(row_number,1)    /* Position the cursor at row 12,   */
                        /* column 1.                        */
cursor("up")            /* Move the cursor up one row.      */
count = 20
cursor("left",count+1)  /* Move the cursor left 21 columns. */
```

Note

For display terminals with multiple partitions defined, the CURSOR statement moves the cursor within the currently active partition. When a row and column are specified, however, the cursor is moved to that location on the entire screen (not just within the currently active partition). The partition that owns the area of the display where the cursor was moved becomes the currently active partition.

CURSRSEL

```
CURSRSEL
```

Function

The CURSRSEL statement simulates the action of the Cursor Select key on the 3270 display terminals. It is valid for simulation of these terminals only.

Examples

```
cursor(index(screen,'XXXX'))    /* Position cursor on first */
                                /* occurrence of XXXX.      */
btab                            /* Back up to input field.  */
cursrsel                        /* Select this item.        */
cursor(index(screen,'YYYY'))    /* Position cursor on first */
                                /* occurrence of YYYY.      */
btab                            /* Back up to input field.  */
cursrsel                        /* Select this item.        */
```

Note

When selected using the Cursor Select key or a light pen, some display fields transmit accumulated message data automatically. Your application determines which fields cause such transmission.

Thus, it is possible that an STL program might be interrupted by a CURSRSEL statement. Select these fields using TRANSMIT USING CURSRSEL rather than CURSRSEL. The TRANSMIT USING CURSRSEL statement allows you to specify a WAIT condition, enabling you to control the execution of your STL program better.

DEACT

```
DEACT  {onin_onout_labels}
       {event_names}
       {ALL IO ONS}
       {ALL EVENT ONS}
```

Where

*onin_onout_labels*are one or more labels coded on previous ONIN or ONOUT statements. Multiple labels must be separated by commas.

*event_names*are one or more string expressions that specify the names of events. Multiple event names must be separated by commas. String constant expressions must be 1 to 8 alphanumeric characters and enclosed in single or double quotes. (Strings containing hexadecimal constants are exempt from this restriction.) If you specify a nonconstant string expression, the first 8 characters of the string or substring are used. If the string is shorter than 8 characters, the available characters are used. To satisfy conditions using event names, the first 8 characters must match exactly. If you specify a string variable, it cannot be the name of one of the reserved variables (for example, BUFFER).

Function

The DEACT statement deactivates outstanding ONIN, ONOUT, or ON SIGNALED statement conditions. Unless explicitly deactivated using this statement, these conditions will remain active for the duration of the current STL program (ONIN and ONOUT), or until the specified event is signaled (ON SIGNALED).

If you specify ONIN or ONOUT labels, only the conditions on the statement identified by those labels are deactivated. If ALL IO ONS is specified, all currently active ONIN and ONOUT conditions are deactivated.

If you specify event names, all ON SIGNALED conditions referencing the specified names are deactivated. If you specify ALL EVENT ONS, all currently active ON SIGNALED conditions are deactivated.

Examples

Example 1

```
getout: onin index(data,'ABEND') > 0 then deact getout
                           /* Don't look for ABEND anymore.      */
```

Example 2

```
on signaled('ALLOK') then say 'Everything went well with this test'
&#38;#8942;
if index(screen,'ERROR!!!') > 0 then
   deact 'ALLOK'           /* Deactivate all ON conditions for   */
                           /* event ALLOK if an error occurs.    */
&#38;#8942;
signal 'ALLOK'             /* Write ALL OK message if there were */
                           /* no errors.                         */
```

Example 3

```
red:   onin index(data,'RED') > 0 then color = 'RED'
blue:  onin index(data,'BLUE') > 0 then color = 'BLUE'
green: onin index(data,'GREEN') > 0 then color = 'GREEN'
&#38;#8942;
if index(screen,'GRAPHIC CARD ERROR') > 0 then
   deact red, blue, green
```

Example 4

```
on signaled('INITCARD') then say 'Graphic Card Initialized'
on siganled('INITSCRN') then say 'Screen Initialized'
on signaled('INITMENU') then say 'Please Make a Selection'
&#38;#8942;
if index(screen,'ERROR') > 0 then
   deact 'INITCARD', 'INITSCRN', 'INITMENU'
```

Notes

- If you use an ONIN or ONOUT label in the DEACT statement, you must have coded the label on an ONIN or ONOUT statement above the DEACT statement in the current STL program. Attempts to deactivate specific ONIN or ONOUT conditions that have not yet been processed by the STL Translator in the current program are flagged as errors.
- When multiple labels or names are entered on several lines, two commas are necessary to continue the DEACT statement. For example:

```
DEACT 'EVENT1', 'EVENT2',,
      'EVENT3'
```

DELAY

```
DELAY({'RATE',rate_table_number}[,uti_name])
      {fixed_time}
      {random_function}
```

Where

rate_table_number is either an integer constant or an arithmetic expression involving **only** integer constants. It must be from 0 to 255. *fixed_time* is an integer expression with a value from 0 to 2147483647.

random_function is a RANDOM function. See "RANDOM" on page 481 for the syntax of this function.

uti_name is a string constant expression that is the name of a UTI statement in the network definition.

Note: The name of the network-level UTI is NTWRKUTI.

Function

The DELAY statement specifies the delay after the next transmit by a simulated terminal. The value specified in this statement will override the default delay for the terminal (specified with the DELAY operand for the terminal in the network definition).

The actual delay (in hundredths of seconds) will be the delay value specified in this statement multiplied by the terminal's UTI value or by the value of the named UTI that you specify as an argument. This value applies to the RATE, fixed time, and random number specifications.

If you specify a single integer constant expression, that integer will be the delay value.

If the first argument is 'RATE', the delay value is chosen randomly from the rate table on the RATE statement referenced by the following integer constant. The RATE statement is a network definition statement.

If you use a RANDOM function as an argument, the resulting random number is used as the delay value.

Examples

```
exdelay: msgtxt
say 'setting delay 1' tod()
delay(5)                    /* Set a delay of 5 times             */
                            /* the UTI value for message 2.       */
type 'Message 1'
transmit
say 'setting delay 2' tod()
delay(random(1,10))        /* Set a delay of a random number from  */
type 'Message 2'           /* 1 to 10 multiplied by the UTI value  */
                           /* for message 3.                       */
transmit
type 'Message 3'
transmit
endtxt
```

Note

This statement affects only the delay immediately following the next interruption of STL processing (normally a Transmit Interrupt). In other words, message 1 is sent with the default delay. Then, the five second delay takes effect before resuming program execution, causing message 2 to be sent. Subsequent delays revert to the default delay for the terminal until another DELAY statement is coded.

DELETE

```
DELETE [delete_count]
```

Where

*delete_count*must be an integer expression with a value from 1 to 255. If *delete_count* is not specified, a value of 1 is assumed.

Function

The DELETE statement simulates the action of the Delete key on 3270 and 5250 display terminals. The statement is valid for simulation of these terminals only. *delete_count* is the number of characters to be deleted beginning with the character at the cursor's current position.

Examples

```
text = 'This text must be deleted!!!'
cursor(index(screen,text))

/* Either one of the following DELETE statements can be used to */
/* delete this string from the screen.                          */

delete 28

/* The following statement has the same effect.                 */

delete length(text)
```

Note

The DELETE statement is ignored if the cursor is not currently positioned at an input field. WSim writes an informational message to the log data set if a DELETE statement is ignored.

DO statements

The four DO statements in STL are the following:

- Simple DO groups
- DO WHILE loops
- DO FOREVER loops
- Iterative DO loops.

Each of these DO statements is described separately in the following sections.

Simple DO groups

```
DO
   statement
   .
   .
   .
END
```

Where

statement is any valid STL statement or statement group.

Function

The statements in a simple DO group are executed once, as though they were a single statement.

When coded after a THEN, ELSE, or WHEN statement, simple DO groups enable you to execute multiple statements conditionally.

Examples

```
if a = 1 then
   do
     b = 0   /* Both statements in this DO group will */
     c = 0   /* be executed if "a" has a value of 1.  */
   end
```

DO WHILE loops

```
DO WHILE condition
  statement
     .
     .
     .
END
```

Where

condition is a valid STL condition. See “Using conditions and relational operators” on page 261 for more information about conditions.

statement is any valid STL statement or statement group.

Function

WSim executes the statements contained in a DO WHILE group repetitively while the specified condition remains true. When WSim first encounters the DO WHILE statement, it evaluates the condition. If the condition is true, WSim executes the group of statements between the DO WHILE and the END statements. Control returns to the DO WHILE statement where the condition is reevaluated. If the condition is false, control passes to the statement following the END statement, and the DO WHILE group of statements is not executed.

Examples

```
a = 0
do while a < 100              /* Execute the following statements */
                              /* while a < 100.                   */
   say 'The value of A is' char(a)
   a = a + 1
end
```

DO FOREVER loops

```
DO FOREVER
   statement
   .
   .
   .
END
```

Where

statement is any valid STL statement or statement group.

Function

A DO FOREVER loop is like a DO WHILE loop whose condition is **always** true. WSim executes the statements in a DO FOREVER group repeatedly until a LEAVE statement is executed or the simulated network is canceled. (You can use various operator commands to cancel a network. See , SC31-8948 for details.)

Use this statement with care, since it can result in a “program execution loop,” in which STL execution for the simulated terminal is never interrupted.

Examples

```
do forever
  type '1234567890'
  transmit and wait until onin    /* Wait until something is */
                                  /* received.               */
end
```

Iterative DO loops

```
DO control_variable = initial_value TO exit_value [BY increment]
  statement
  .
  .
  .
END
```

Where

control_variable is an integer variable.

exit_value is an integer expression.

initial_value is an integer expression.

increment is an integer constant or an integer variable.

statement is any valid STL statement or statement group.

Function

WSim executes an iterative DO group of statements in a loop until the value of the control variable exceeds the exit value. When WSim first encounters the iterative DO statement, it assigns the initial value to the control variable. On subsequent iterations, WSim increments the control variable by the specified increment value. If you do not specify an increment, WSim uses a default increment of 1.

After the control variable is incremented (or initialized on the first iteration of the loop), WSim compares its value with the exit value. If the control variable's value is less than or equal to the exit value, the statements between the DO and END statements are executed. If the value of the control variable exceeds the exit value, control passes to the statement following the END.

Examples

The following iterative DO loop performs the same function as the DO WHILE loop that appears on “DO WHILE loops” on page 403. Note that the default increment of one is used.

```
do a = 0 to 99                     /* Execute the following statements */
                                   /* 100 times.                       */
   say 'The value of A is 'char(a)
end
```

Note

The execution of DO WHILE, DO FOREVER, and iterative DO loops can be modified by the ITERATE and LEAVE statements. See “ITERATE” on page 413 and “LEAVE” on page 414 for descriptions of these statements.

DUP

```
DUP
```

Function

The DUP statement simulates the action of the duplicate key on 3270 and 5250 display terminals. The statement is valid for simulation of these terminals only.

Examples

```
dup
```

ENDTXT

```
ENDTXT
```

Function

The ENDTXT statement ends an STL procedure.

Examples

```
myproc: msgtxt
&#38;#8942;
endtxt    /* End of MYPROC. */
```

ENDUTBL

```
ENDUTBL
```

Function

The ENDUTBL statement ends an STL user table.

Examples

```
mytabl: msgutbl
&#38;#8942;
endutbl   /* End of MYTABL. */
```

EREOF

```
EREOF
```

Function

The EREOF statement simulates the action of the Erase to End of Field key on 3270 display terminals. The statement is valid for simulation of these terminals only.

Examples

```
type 'Now is the time for all good men'
ereof        /* Make sure the rest of this input field is erased. */
tab          /* Move to the next input field.                     */
```

ERIN

```
ERIN
```

Function

The ERIN statement simulates the Erase Input key on 3270 and 5250 display terminals. It erases all input fields on a panel. This statement is valid for simulation of these terminals only.

Examples

```
erin              /* Erase all input fields on a panel. */
```

EXECUTE

```
EXECUTE execute_procedure_name
```

Where

execute_procedure_name is the name on an STL statement that defines an STL execute procedure.

Function

The EXECUTE statement is an asynchronous subset statement. When encountered in an ONIN, ONOUT, or ON SIGNALED statement, EXECUTE causes the named procedure to be executed immediately by WSim. This execution is asynchronous to normal STL program execution.

Examples

```
mainproc: msgtxt
onin substr(ru,1,5) = 'ERROR' then execute hiterror
onin substr(ru,200,5) = 'ERROR' then execute hiterror
&#38;#8942;
```

```
endtxt                         /* End of Mainproc.               */
hiterror: msgtxt               /* This procedure is executed when */
                               /* 'ERROR' is received.           */
say 'An error was found in the received data.'
endtxt
```

Notes

- Execute procedures can contain only a limited subset of STL statements and expressions. The STL Translator enforces these limitations when processing an STL procedure that has been previously referenced in an EXECUTE statement.
- Use of a procedure name in an EXECUTE statement defines that procedure as an execute procedure. Use of a procedure name in a CALL statement defines that procedure as a called procedure. When the STL Translator encounters an STL statement that has not been defined as an execute procedure, it defines it as a called procedure. The name of an execute procedure cannot be used in a CALL statement, nor can a called procedure name be used in an EXECUTE statement.
- This statement **must** be coded directly following the THEN keyword on the ONIN, ONOUT, or ON SIGNALED statement.
- For more information about the EXECUTE asynchronous subset statement and the statements that you can use in an execute procedure, see “Using asynchronous subset statements” on page 294.

FLDADV

```
FLDADV
```

Function

The FLDADV statement simulates the action of the Field Advance key on a 5250 display terminal. It moves the cursor to the next input field. The statement is valid for simulation of this terminal only.

Examples

```
cursor(10,20)   /* Position cursor on row 10, column 20. */
fldadv          /* Advance to the next input field.      */
```

FLDBKSP

```
FLDBKSP
```

Function

The FLDBKSP statement simulates the action of the Field Backspace key on a 5250 display terminal. The statement is valid for simulation of this terminal only.

Examples

```
cursor(10,20)   /* Position cursor on row 10, column 20. */
fldbksp         /* Back up to previous input field.      */
```

FLDMINUS

```
FLDMINUS
```

Function

The FLDMINUS statement simulates the action of the Field Minus (F-) key on a 5250 display terminal. The statement is valid for simulation of this terminal only.

Examples

```
cursor(10,20)  /* Position cursor on row 10, column 20. */
fldminus       /* Simulate action of Field Minus key.   */
```

FLDPLUS

```
FLDPLUS
```

Function

The FLDPLUS statement simulates the action of the Field Exit or Field Plus (F+) key on a 5250 display terminal.

Examples

```
cursor(10,20)           /* Position cursor on row 10, column 20. */
type 'Hello'
fldplus                 /* Clear the rest of this field and      */
                        /* advance to the next.                  */
```

FM

```
FM
```

Function

The FM statement simulates the Field Mark key on 3270 display terminals. The statement is valid for simulation of these terminals only.

Examples

```
cursor(10,20)        /* Position cursor on row 10, column 20. */
fm                   /* Mark this field.                      */
```

Note

You can also simulate the FM key by including the FM function in a string on a TYPE statement. For example:

```
type 'some data'fm()  /* Enter data and include a field mark.   */
```

HIGHLITE

```
HIGHLITE highlight_specification
```

Where

highlight_specification is a string constant from the following list:
- BLINK
- FIELD
- REVERSE
- UNDERLINE.

You can use uppercase or lowercase letters. You must enclose the string constant in single or double quotation marks.

Function

The HIGHLITE statement simulates a 3270 key that selects the highlighting option for displaying data input.

highlight_specification specifies the highlighting option to be used for displaying subsequent data input from this terminal. You may choose one of the following highlighting options:

BLINK	The display of the input data will alternate between display and nondisplay modes.
FIELD	The highlighting option defined by the extended field attribute byte will be selected.
REVERSE	The input data will be displayed as reversed image characters.
UNDERLINE	The displayed input data will be underlined.

Examples

```
type 'Hello'          /* "Hello" is entered as normal text.     */
highlite "reverse"
type 'Goodbye'        /* "Goodbye" is entered as reverse video. */
```

Note

If you do not code the HIGHLITE statement, the highlighting option is selected by the extended field attribute byte value.

HOME

```
HOME
```

Function

The HOME statement simulates the Home key on 3270 and 5250 display terminals. The statement is valid for simulation of these terminals only.

Examples

```
home           /* Move cursor to first input field on screen. */
type 'Hello'
```

IF

```
IF condition THEN[;] statement[;]
            [ELSE[;] statement]
```

Where

condition is any valid STL condition.

statement is a single statement or a statement group, such as a DO or SELECT group. It may also be another IF statement.

Function

The IF statement is used to execute a statement or group of statements conditionally. If the condition is true, the statement (or statement group) following the THEN keyword is executed. If the condition is false and an ELSE has been coded, the statement following the ELSE keyword is executed. If the condition is false and an ELSE has not been coded, execution continues with the statement following the IF/THEN statement; the statement following the THEN keyword is not executed.

Examples

```
if a = 1 then
  say 'A has a value of 1'
else
  say 'A does not have a value of 1'

if mydata = 'Hello' then
 do
  say 'MYDATA has a value of "Hello"'
  a = 1
  b = off
 end
else                                   /* MYDATA is not Hello. */
 if mydata = 'Goodbye' then
  say 'MYDATA has a value of "Goodbye"'
 else
  nop
```

Note

A semicolon (;) is not required between a THEN or ELSE keyword and the following statement, as shown in the following example.

```
if a = 1 then say 'A has a value of 1'
else say 'A does not have a value of 1'
```

INITSELF

```
INITSELF(resource[,[mode][,[user_data][,log_byte]]])
```

Where

resource is a string expression. If *resource* is a string constant expression, it must be 1 to 8 alphanumeric characters and must not contain blanks. The characters must be enclosed in single or double quotation marks. String constant expressions containing hexadecimal strings are exempt from this restriction. The STL Translator will translate all lowercase characters to uppercase in these constants.

mode is a string expression. If *mode* is a string constant expression, it must be 1 to 8 alphanumeric characters and must not contain blanks. The STL Translator will translate all lowercase characters to uppercase in these constants. These constants must be enclosed in single or double quotation marks. String constant expressions containing hexadecimal strings are exempt from this restriction.

user_data is a string expression. The length of *user_data* should not exceed 255 characters although the STL Translator permits longer strings. A maximum of 255 characters will be sent.

log_byte is a 1-byte string constant or a string expression that specifies a byte of user data to be associated with all data transmitted and received. The *log_byte* remains active until data is "typed" and transmitted with a TRANSMIT statement or until an INITSELF, TERMSELF, or SNACMND statement is issued. This byte gives users of the Response Time Utility a way to identify transactions when gathering statistics by the various user-defined "log_byte" categories. Only the first character or first two hexadecimal digits of the string expression are used. The default *log_byte* is X'00'.

Function

The INITSELF statement sends an INITIATE SELF format 0 RU to initiate sessions when simulating SNA terminals or logical units (LUs).

resource specifies the name of the partner LU. The *mode* is the name of a mode entry table, which you may need to use when simulating SNA terminals or logical units. The *user_data* is typically information such as a password, which is unique to a user.

For additional examples of the INITSELF statement and a complete description of its use, see "Logging on and off an application" on page 278.

Examples

```
initself('MYAPPL','MYMODE','MYPASSWD')    /* Includes mode entry */
                                          /* and user data.      */

initself('MYAPPL',,'MYPASSWD')            /* Includes user data, */
                                          /* but no mode entry.  */

initself('MYAPPL','MYMODE',,'A5'x)        /* Includes mode entry */
                                          /* and log byte.       */
log_byte = 'm'
initself('MYAPPL',,,log_byte)             /* Includes a variable */
                                          /* log byte            */
```

Notes

- The INITSELF statement causes a Transmit Interrupt. Thus, all accumulated message data will be sent.
- The INITSELF statement is identical to specifying a type of INITSELF on the SNACMND statement.

INSERT

```
INSERT
```

Function

The INSERT statement simulates the Insert key on 3270 display terminals. The statement is valid for simulation of these terminals only.

Examples

```
type 'Now is time'   /* Enter text into current field on screen. */
cursor('left',4)     /* Move back to the 't' in 'time'.         */
insert               /* Get into insert mode.                   */
type 'the '          /* Insert 'the ' before 'time'.            */
                     /* Current field should now contain 'Now is */
                     /* the time'.                              */
reset                /* Get out of insert mode.                 */
```

INTEGER

INTEGER [{SHARED|UNSHARED}] *variable_list*

Where

variable_list is any number of valid STL variable names not previously declared, separated by commas.

Function

The INTEGER statement explicitly declares one or more INTEGER variables, which can be specified as SHARED or UNSHARED. The default class is UNSHARED.

Examples

```
                      /* Declares the unshared integer variables, */
                      /* "amount" and "balance".                  */
integer unshared amount, balance

integer mycount       /* Declares the unshared integer variable,  */
                      /* "mycount".                               */

integer shared total  /* Declares the shared integer variable,    */
                      /* "total".                                 */
```

Note

The INTEGER statement is a declarative statement. You can code it only **outside** an STL procedure.

ITERATE

```
ITERATE
```

Function

The ITERATE statement causes an immediate iteration of the innermost DO WHILE, DO FOREVER, or iterative DO loop.

When WSim encounters this statement, it stops execution of the statements within the loop, and passes control back to the DO statement. When it is a DO WHILE statement WSim evaluates the condition and executes the DO WHILE group of statements again if the condition is true. If it is a DO FOREVER statement, the statement group is executed again. When it is an iterative DO statement, the value of the control variable is incremented, and the DO statement group is executed again if the new value does not exceed the exit value.

Examples

The following code transmits the numbers 0-19 and 21-99. The number 20 will not be transmitted.

```
b = 20
do a = 0 to 99
   if b = a then
      iterate
   type char(a)
   transmit
end
```

Note

You can code the ITERATE statement only inside a repetitive DO loop.

JUMP

```
JUMP [pid_number]
```

Where

pid_number is an integer constant expression from 0 to 15.

Function

The JUMP statement simulates the Jump key on a 3270 display terminal. The statement is valid for simulation of this terminal only.

pid_number specifies the partition identification number (PID) of the partition to be made active with the JUMP statement. When you code the JUMP statement without the PID operand, the next sequential partition becomes active.

Examples

The following example enters the same data in partitions 0, 1, 2, and 3.

```
jump 0            /* Go to partition 0.                  */
do i = 1 to 4     /* Do the following 4 times.           */
   type 'hello'   /* Enter data in the current partition. */
   jump           /* Jump to the next partition.         */
end
```

LCLEAR

```
LCLEAR
```

Function

The LCLEAR statement simulates the Local Clear key on 3270 display terminals. The statement is valid for simulation of these terminals only.

Examples

```
home      /* Move to first input field on screen. */
lclear    /* Clear the screen (local clear).      */
```

LEAVE

```
LEAVE
```

Function

The LEAVE statement stops execution of the innermost DO WHILE, DO FOREVER, or iterative DO loop. Control passes to the statement following the END statement for the loop.

Examples

The following code writes the string 'Hello' to the operator three times.

```
a = 1
do forever
   if a = 4 then
      leave
   else
      nop
   say 'Hello'
   a = a + 1
end
```

Note

You can code the LEAVE statement only inside a repetitive DO loop.

LIGHTPEN

```
LIGHTPEN(integer_expression1[,integer_expression2])
```

Where

integer_expression1 is an integer expression with a value from 0 to 32766 if only one argument is supplied or from 1 to 255 if two arguments are supplied.

integer_expression2 is an integer expression with a value from 1 to 255.

Function

The LIGHTPEN statement simulates a Selector Light Pen on 3270 and 5250 display terminals.

If you specify a single argument, it is interpreted as a screen offset (offset 0 is the first screen position).

If you specify two arguments, WSim interprets the first as a row number and the second as a column number (row 1, column 1 is the first position on the screen.)

Valid row and column designations must be from 1 to 255. Valid offset designations must be from 0 to 32766.

Examples

```
lightpen(index(screen,'XXXX'))    /* Select item XXXX. */
```

Note

When selected using the Cursor Select key or a light pen, some display fields transmit accumulated message data automatically. Your application determines which fields cause such transmission.

Thus, it is possible that an STL program might be interrupted by a LIGHTPEN statement. Select these fields using TRANSMIT USING CURSRSEL rather than LIGHTPEN. The TRANSMIT USING CURSRSEL statement allows you to specify a WAIT condition, enabling you to better control the execution of your STL program.

LOG

```
LOG {string_expression}
   {DISPLAY}
```

Where

string_expression is any valid STL string expression.

Function

The LOG statement writes the user-specified string expression or the 3270 or 5250 display image to the log data set for formatting by the Loglist Utility. If *string_expression* is specified, the expression will be evaluated and written to the log data set with the LOG record type. If DISPLAY is specified, the screen image buffer and the screen attribute table will be written to the log data set with the DSPY record type.

Examples

```
log 'I am going to log the display image for 'devid()
log display
```

MONITOR

```
MONITOR
```

Function

The MONITOR statement causes the Display Monitor Facility to display the simulated 3270 display image as it exists at this point in the message generation process. To use this statement, the UPDATE=MONITOR operand on the M (Display Monitor Facility) operator command must be in effect when starting the monitor. See , SC31-8948 for more information about the Display Monitor Facility. This statement is valid for simulation of 3270 terminals only.

Examples

```
/* Cause each new screen image to be displayed by the           */
/* Display Monitor Facility.                                    */

do forever
   transmit and wait until onin    /* Send data and wait for new */
                                   /* screen.                    */
   monitor                         /* Display this screen.       */

   &#38;#8942;
end
```

Note

If you execute the MONITOR statement for a terminal that is not being monitored by the Display Monitor Facility or do not start the Display Monitor Facility with the UPDATE=MONITOR operand, the statement is ignored.

MSGTXT

```
procedure_name: MSGTXT [string_constant]
```

Where

procedure_name is a valid STL name.

string_constant is a 1- to 37-character string constant that specifies MSGTXT operand values.

Function

The MSGTXT statement defines the start of an STL procedure.

If a *string_constant* is coded, it is included on the MSGTXT message generation statement created by the STL Translator. The string constant must be enclosed in single or double quotation marks. It can be uppercase or lowercase.

The operand values that you can specify for *string_constant* are listed in Part 1, "WSim language statements," on page 1 in the description of the MSGTXT message generation statement.

Note: STL does not validate the operand values that you specify for *string_constant*.

Examples

```
myproc: msgtxt             /* This is the beginning of procedure */
                           /* MYPROC.                            */

yourproc: msgtxt 'text=blk' /* In procedure YOURPROC, send text as*/
                           /* a block.                           */
```

MSGUTBL

```
msgutbl_name: MSGUTBL
              utbl_entry
              ┌            ┐
              │ utbl_entry │
              │   .        │
              │   .        │
              │   .        │
              └            ┘
              ENDUTBL
```

Where

msgutbl_name is a valid STL name.

utbl_entry is a string constant expression. At least one *utbl_entry* is required. If multiple entries are coded, each must start on a separate line (or be separated from the previous entry by a semicolon). String constants must be enclosed in single or double quotation marks. You can have up to 2147483647 entries in the table.

Function

The MSGUTBL statement is used to declare a user table.

For a description of user tables and how they can be used, see "Using user tables" on page 275.

Examples

```
myutbl: msgutbl    /* This is the beginning of user table MYUTBL. */
   'entry0'
   'entry1'
endutbl
```

Notes

- The MSGUTBL statement is a declarative statement. You can code it only **outside** an STL procedure.
- You may continue entries.

NL

```
NL
```

Function

The NL statement simulates the New Line key on 3270 and 5250 display terminals. The statement is valid for simulation of these terminals only.

Examples

```
cursor(1,1)       /* Move to row 1, column 1 on screen. */
type 'Hello'      /* Enter data on line 1.              */
nl                /* Move to next line on screen.       */
type 'Goodbye'    /* Enter data on line 2.              */
```

Note

You can also simulate the action of the NL key by including the NL function on a TYPE statement.

NOP

```
NOP
```

Function

NOP is a dummy statement that has no effect. It can be useful as the target of a THEN, ELSE, or OTHERWISE statement.

Examples

```
if index(screen,'Hello') > 0 then
   nop
else                              /*  Hello not on screen. */
   error_flag = on
```

NORESP

```
NORESP
```

Function

The NORESP statement is an asynchronous subset statement. When executed, it suppresses the automatic generation of SNA responses by WSim. Instead, WSim will interpret the next message transmitted by the terminal as the user-provided SNA response.

Examples

```
onin substr(ru,1,33) = 'Please send me a strange response' then error = on
onin substr(ru,1,33) = 'Please send me a strange response' then noresp
&#38;#8942;
transmit and wait until ...
if error = on then        /* Need to send exception sense response. */
  do
    type '012345678'
    setrh on(exc,sni)
    suspend()             /* Required to send response without      */
                          /* sending an AID.                        */
  end
```

Notes

- A TRANSMIT statement should not be used to send the response. This causes an AID byte to be set. Instead, code SUSPEND(). This causes program execution to be interrupted and the response to be sent.
- The NORESP statement ensures that an SNA response is always sent.
- The NORESP asynchronous subset statement is ignored for non-SNA terminals. If the NORESP asynchronous subset statement is not coded, WSim will automatically build the SNA response.
- This statement **must** be coded directly following the THEN keyword on the ONIN or ONOUT statement.

ON SIGNALED

```
ON SIGNALED(event_name) THEN[;] asynchronous_subset_statement
```

Where

event_name is a string variable, a substring (SUBSTR) of a string variable, or a string constant expression that specifies the name of an event. If it is a string constant expression, *event_name* must be between 1 and 8 alphanumeric characters and enclosed in single or double quotes. If you specify a string variable or a substring of a string variable, the first 8 characters of the string or substring will be used. If the string or substring is shorter than 8 characters, the available characters will be used. To satisfy conditions using event names, the first 8 characters must match exactly. If you specify a string variable, it cannot be the name of one of the reserved variables (for example, BUFFER).

asynchronous_subset_statement is one of the following STL statements permitted on asynchronous statements:

- ABORT statement
- CALL statement

- CANCEL DELAY statement
- CANCEL SUSPEND statement
- EXECUTE statement
- Any statement that can be coded within an EXECUTE procedure, except RETURN.

Function

The ON SIGNALED statement specifies a statement or group of statements to be executed when the *event_name* is signaled. For descriptions of asynchronous subset statements, see “Using asynchronous subset statements” on page 294.

Examples

```
/* Tell the operator when MYEVENT is signaled. */

on signaled('MYEVENT') then
   say 'MYEVENT has been signaled.'

/* Set some bits */

on signaled('MYEVENT2') then do
   timer = off
   begin_phase2 = on
end
```

Note

An event can be signaled by the execution of a SIGNAL or QSIGNAL statement or by the issuing of the “A *network*, SIGNAL=*event_name*” command by the operator.

ONIN and ONOUT

```
[label:] {ONIN} [asynchronous_condition] THEN[;] asynchronous_subset_statement
         {ONOUT}
```

Where

label is a valid STL label.

asynchronous_condition is a valid asynchronous condition. See “Setting up asynchronous conditions” on page 299 for a description of asynchronous conditions.

asynchronous_subset_statement is any valid asynchronous subset statement. See “Using asynchronous subset statements” on page 294 for descriptions of asynchronous subset statements.

Function

Use the ONIN and ONOUT statements to establish asynchronous conditions that will be tested when data is received (ONIN) or transmitted (ONOUT) by the terminal. These conditions, and their associated actions, are called asynchronous because they are executed outside of normal synchronous STL program execution.

Once activated (by the processing of the ONIN or ONOUT statement), these conditions remain active for the duration of the current STL program, or until explicitly deactivated by a DEACT statement.

Note: CPI-C simulations do not support testing of asynchronous input and output conditions. If ONIN or ONOUT statements are specified in a CPI-C simulation they are ignored.

Examples

```
asyrsp: onin rh &= '80'x then do /* Test under mask for SNA response? */
   sna_response_received = on    /* Set bit for later test.          */
   say 'An SNA response was received.' /* Notify the user.           */
   end
asyexc: onin substr(rh,2) &= '10'x then do /* Exception response      */
   exception_response = on                /* Set bit for later test  */
   say 'An exception response was received.' /* Notify the user.     */
   end
&#38;#8942;
transmit and wait until onin     /* Transmit and wait for response    */
/**********************************************/
/* See if an exception response was received. */
/**********************************************/
if sna_response_received & exception_response then
   call erpproc
else
   call goodproc
```

An ONIN or ONOUT statement does not have to include a condition. The THEN keyword can immediately follow the ONIN or ONOUT keyword. All incoming (ONIN) or outgoing (ONOUT) messages satisfy such a null condition.

```
onin then something_received = on  /* Set bit on when anything */
                                   /* is received.             */
```

Note

Outstanding asynchronous conditions are tested in the order of their appearance in an STL program (after network-level IFs). Hence, the condition labeled “asyrsp” in the preceding example will be tested before the condition labeled “asyexc”. See Part 1, “WSim language statements,” on page 1 for more information on network IFs.

OPCMND

```
OPCMND operator_command
```

Where

operator_command is a string expression.

Function

The OPCMND statement specifies an operator control command to be processed as if it were entered from the operator console. *operator_command* defines the operator command to be entered when the STL program is executing. See , SC31-8948 for details of the various operator commands.

Examples

```
opcmnd 'A 'netid()',U=100'  /* Alter UTI to 100. */
/****************************************************/
/* The following code has the same effect as the     */
/* preceding statement.  In the following example,   */
/* the operator command is "built" and stored in a   */
/* string variable before being issued.              */
/****************************************************/
uti_value = 100
operator_command = 'A 'netid()',U='char(uti_value)
opcmnd operator_command
```

Notes

- You can execute all operator commands with the OPCMND statement, with the exception of console recovery subcommands, which must be entered by the system console operator. Although *operator_command* can be any length, a maximum of 120 characters will actually be passed to the operator command processor.
- STL does not validate the operator command.
- The *operator_command* does not take effect immediately. It will be queued for execution with any other operator commands and be executed by WSim at the earliest when program execution is suspended.

POST

POST *event_name* **[AFTER** *time* **[TAG** *event_tag***]]**

Where

event_name is a string expression that specifies the name of an event. If it is a string constant expression, *event_name* must be 1 to 8 alphanumeric characters and enclosed in single or double quotes. (Strings containing hexadecimal constants are exempt from this restriction.) If you specify a nonconstant string expression, the first 8 characters of the string are used. If the string is shorter than 8 characters, the available characters are used. To satisfy conditions using event names, the first 8 characters must match exactly. If you specify a string variable, it cannot be the name of one of the reserved variables (for example, BUFFER).

time is an integer value that indicates the number of seconds that the posting of the event is to be delayed. It must be an integer expression with a value from 1 to 21474836.

event_tag specifies a “tag” to be assigned to the POST statement. This tag is only valid if the *time* value is also coded. The rules for specifying the *event_tag* are the same as those for specifying *event_name*.

Function

The POST statement posts the named event. Once posted, all POSTED functions that specify *event_name* will return a value of true and all WAIT UNTIL POSTED conditions specifying *event_name* will be satisfied.

Note: See “Specifying variable event names with a time delay” on page 306 for information about specifying variable event names with a time delay.

When you specify *time* and *event_tag*, you can cancel events that have been assigned the same tag before the specified time has elapsed. You can assign the same tag to events that are specified with a POST, QSIGNAL, RESET, or SIGNAL statement. Thus, you can cancel several events by assigning them the same tag.

If an event tag is not specified, a default tag is assigned that has a value that is the same as the event name.

See "CANCEL" on page 366 for more information about how to use a tag to cancel events.

Examples

```
if message = 'Hello' then
   post 'HELLO'
else
   post 'OTHER'
```

The following statements show how variable event names can be posted.

```
event_name = substr(message,43,8)
post event_name
```

Note

The event *event_name* will remain posted until the end of the program that posted it or until it is reset by the RESET statement. The operator can post an event by issuing the "A *network*, POST=*event_name*" command.

PUSH

```
PUSH string [TO queue_name]
```

Where

string is a string expression.

queue_name is a string expression consisting of 1 to 8 alphanumeric characters. This is optional.

Function

The PUSH statement places *string* on *queue_name* on a last in first out (LIFO) basis.

If the specified *queue_name* is a nonconstant expression, the first 8 characters of the string are used. If the string is shorter than 8 characters, the available characters are used. When specifying *queue_name* with the PUSH statement, the test/string item will be placed on a queue exactly matching *queue_name*. If you specify a string variable, it cannot be the name of one of the STL reserved variables (for example, BUFFER). If not specified, *queue_name* defaults to a unique value assigned to each device.

Examples

```
Qname = 'QUEUE1'               /* Assigns 'QUEUE1' to "Qname"        */
push 'ABCD'                    /* Places 'ABCD' on unique device Q   */
push '1234567'  TO Qname       /* Places '1234567' on queue 'QUEUE1' */
push '7654321'  TO 'QUEUE1'    /* Places '7654321' on queue 'QUEUE1' */
a = PULL(Qname)                /* Assigns '7654321' to "a"           */
b = PULL('QUEUE1')             /* Assigns '1234567' to "b"           */
c = PULL()                     /* Assigns 'ABCD' to "c"              */
```

Note: The named queue structure and text/string data items are allocated dynamically by WSim and deleted as the queue is emptied.

QSIGNAL

QSIGNAL *event_name* **[AFTER** *time* **[TAG** *event_tag***]]**

Where

event_name is a string expression that specifies the name of an event. If it is a string constant expression, *event_name* must be 1 to 8 alphanumeric characters and enclosed in single or double quotes. (Strings containing hexadecimal constants are exempt from these restrictions.) If you specify a nonconstant string expression, the first 8 characters of the string are used. If the string is shorter than 8 characters, the available characters are used. To satisfy conditions using event names, the first 8 characters must match exactly. If you specify a string variable, it cannot be the name of one of the reserved variables (for example, BUFFER).

time is an integer value that indicates the number of seconds that the posting of the event is to be delayed. It must be an integer expression with a value from 1 to 21474836.

event_tag specifies a "tag" to be assigned to the QSIGNAL statement. This tag is only valid if the *time* value is also coded. The rules for specifying the *event_tag* are the same as those for specifying *event_name*.

Function

The QSIGNAL statement causes a qualified signaling of the named event. A qualified signal affects only the terminal that issues the QSIGNAL; therefore, only that terminal's outstanding ON SIGNALED conditions will be satisfied.

Note: See "Specifying variable event names with a time delay" on page 306 for information about specifying variable event names with a time delay.

When you specify *time* and *event_tag*, you can cancel events that have been assigned the same tag before the specified time has elapsed. You can assign the same tag to events that are specified with a POST, QSIGNAL, RESET, or SIGNAL statement. Thus, you can cancel several events by assigning them the same tag.

If an event tag is not specified, a default tag will be assigned that has a value that is the same as the event name.

See "CANCEL" on page 366 for more information about how to use a tag to cancel events.

Examples

```
qsignal 'MYEVENT'
```

QUEUE

```
QUEUE string [TO queue_name]
```

Where

string is a string expression.

queue_name is a string expression consisting of 1 to 8 alphanumeric characters. This is optional.

Function

The QUEUE statement adds *string* to *queue_name* on a first in, first out (FIFO) basis.

If the specified *queue_name* is a nonconstant expression, the first 8 characters of the string are used. If the string is shorter than 8 characters, the available characters are used. When specifying *queue_name* with the QUEUE statement, the test/string item will be placed in a queue by the exact name as *queue_name*. If you specify a string variable, it cannot be the name of one of the STL reserved variables (for example, BUFFER). If not specified, *queue_name* defaults to a unique value assigned to each device.

Examples

```
queue 'AAAA' TO 'Queue1'        /* Places 'AAAA'  in queue 'Queue1'   */
queue 'BBBB' TO 'QUEUE1'        /* Places 'BBBB'  in queue 'QUEUE1'   */
a = PULL('QUEUE1')              /* Assigns 'BBBB' to "a"              */
```

Note: The named queue structure and text/string data items are allocated dynamically by WSim and deleted as the queue is emptied.

QUIESCE

```
QUIESCE ┌ UNTIL {ONIN [asynchronous_condition]}  ┐
        │       {ONOUT [asynchronous_condition]} │
        │       {SIGNALED(event_name)}           │
        ≪                                        ┘
```

Where

asynchronous_condition is a valid asynchronous condition. See “Testing asynchronous conditions” on page 291 for a description of asynchronous conditions.

event_name is a string expression that specifies the name of an event. If it is a string constant expression, *event_name* must be 1 to 8 alphanumeric characters and enclosed in single or double quotes. (Strings containing hexadecimal constants are exempt from this restriction.) If you specify a nonconstant string expression, the first 8 characters of the string are used. If the string is shorter than 8 characters, the available characters are used. To satisfy conditions using event names, the first 8 characters must match exactly. If you specify a string variable, it cannot be the name of one of the reserved variables (for example, BUFFER).

Function

The QUIESCE statement stops program execution and quiesces the terminal. You can optionally specify the conditions under which the terminal is to be released from its quiesced state.

Examples

```
quiesce                 /* Go to sleep.  Never wake up.       */

quiesce until onin index(screen,'WAKE UP') > 0
                        /* Quiesce until the string 'WAKE UP' */
                        /* is received.                       */
```

Notes

- A quiesced terminal can also be released by the "A *resource*, RELEASE" operator command.
- The statement causes a Transmit Interrupt. Thus, all accumulated message data will be sent.
- If you are in the middle of generating elements of an SNA chain, quiescing the terminal will not prevent the program from being reentered to generate subsequent chain elements.
- A simulated TCP/IP device that has been quiesced will not attempt to reconnect to the specified server after disconnection.

RESET event

RESET *event_name* **[AFTER** *time* **[TAG** *event_tag***]]**

Where

event_name is a string expression that specifies the name of an event. If it is a string constant expression, *event_name* must be 1 to 8 alphanumeric characters and enclosed in single or double quotes. (Strings containing hexadecimal constants are exempt from these restrictions.) If you specify a nonconstant string expression, the first 8 characters of the string are used. If the string is shorter than 8 characters, the available characters are used. To satisfy conditions using event names, the first 8 characters must match exactly. If you specify a string variable, it cannot be the name of one of the reserved variables (for example, BUFFER).

time is an integer value that indicates the number of seconds that the posting of the event is to be delayed. It must be an integer expression with a value from 1 to 21474836.

event_tag specifies a "tag" to be assigned to the RESET statement. This tag is only valid if the *time* value is also coded. The rules for specifying the *event_tag* are the same as those for specifying *event_name*.

Function

The RESET statement is used to mark the specified event incomplete so that it can be posted again.

When you specify *time* and *event_tag*, you can cancel events that have been assigned the same tag before the specified time has elapsed. You can assign the same tag to events that are specified with a POST, QSIGNAL, RESET, or SIGNAL statement. Thus, you can cancel several events by assigning them the same tag.

Note: See "Specifying variable event names with a time delay" on page 306 for information about specifying variable event names with a time delay.

If an event tag is not specified, a default tag is assigned that has a value that is the same as the event name.

See "CANCEL" on page 366 for more information about how to use a tag to cancel events.

Examples

```
if posted('MYEVENT') then       /* MYEVENT has been posted.       */
   do
      say 'MYEVENT has been posted again'
      reset 'MYEVENT'           /* Reset MYEVENT for next posting. */
   end
```

Note

When the STL Translator encounters the RESET keyword by itself in a statement, it is interpreted as a Reset key simulation statement. When followed by an expression of any type, the RESET keyword is interpreted as a ***Reset Event*** statement and is processed as such. See "RESET key" for a description of the Reset key simulation statement.

RESET key

```
RESET
```

Function

The RESET statement simulates the action of the Reset key on 3270 display terminals. The statement is valid for simulation of these terminals only.

Examples

```
type 'Now is time' /* Enter text into current field on screen. */
cursor('left',4)   /* Move back to the 't' in 'time'.         */
insert             /* Get into insert mode.                   */
type 'the '        /* Insert 'the ' before 'time'.            */
                   /* Current field should now contain 'Now is */
                   /* the time'.                              */
reset              /* Get out of insert mode.                 */
```

Note

When the STL Translator encounters the RESET keyword by itself in a statement, it is interpreted as a Reset key simulation statement. When followed by an expression of any type, the RESET keyword is interpreted as a ***Reset Event*** statement and is processed as such. See "RESET event" on page 426 for a description of the Reset Event statement.

RETURN

```
RETURN
```

Function

The RETURN statement returns control to the point where the last CALL statement was issued. It is ignored if no CALL is outstanding.

An ENDTXT statement functions as a RETURN statement if WSim encounters it during execution while a CALL or EXECUTE is outstanding.

If the CALL was issued as an asynchronous subset statement, control returns to the statement that would have been executed next if the CALL had not been issued. In Example 2 below, for instance, the ONIN condition will be activated when “proc1” is processed and will be active when “proc2” is called. If the ONIN condition is satisfied during the Transmit Interrupt of “proc2,” the procedure “newhello” will be called. When the Transmit Interrupt ends, “newhello” will be executed. When the RETURN statement in “newhello” is executed, control returns to the statement labeled “next” in “proc2” since this statement would have been executed after the Transmit Interrupt if the asynchronous CALL had not been encountered.

Examples

Example 1

```
if no_more_data then       /* More data to process?       */
   return                  /* Return to calling procedure. */
else                       /* More data to process.       */
&#38;#8942;
```

Example 2

```
proc1: msgtxt
onin index(data,'Hello') > 0 then call newhello
&#38;#8942;
call proc2
endtxt

proc2: msgtxt
type 'Are you there?'
transmit and wait until onin
next: say 'The Transmit Interrupt has ended.'
endtxt

newhello: msgtxt
number_of_hellos = number_of_hellos + 1
return
endtxt
```

SAY

```
SAY message [TYPE 'ABRHD']
```

Where

message is a string expression.

Function

The SAY statement writes *message* to the operator console. The string that you specify can be any length; however, a maximum of 100 characters will be displayed.

If TYPE 'ABRHD' is specified, *message* will be written with an abbreviated header containing only a message number preceding the data. If omitted, *message* is written with network and device or LU names included in the header and using message number ITP113I or ITP137I.

Examples

```
say 'Beginning to execute procedure' msgtxtid()
say 'Beginnning to execute procedure' msgtxtid() type 'ABRHD'
```

Note: When using the SAY statement, make sure that the system console is not overloaded with these messages.

SCROLL

```
SCROLL {UP}
       {DOWN}
```

Function

The SCROLL statement simulates the Scroll key on 3270 display terminals. The statement is valid for simulation of these terminals only.

The UP or DOWN keyword specifies whether the displayed data is to be scrolled up or down in relation to the current viewport.

Examples

```
scroll up  /* Scroll up within the current presentation space. */
```

SELECT

```
SELECT
    WHEN condition THEN[;] statement
  ┌                                  ┐
  │ WHEN condition THEN[;] statement │
  │  .                               │
  │  .                               │
  │  .                               │
  └                                  ┘
    OTHERWISE[;] statement
END
```

Where

condition is any valid STL condition.

statement is a single statement or a statement group, such as DO, SELECT, or IF/THEN/ELSE.

Function

The SELECT statement is used conditionally to execute one of several alternative statements.

WSim evaluates each condition following a WHEN in turn. If a condition is true, WSim executes the statement (or statement group) following the THEN and control passes to the END statement. If a condition is false, control passes to the next WHEN statement.

If none of the WHEN conditions are true, control passes to the statement (or statement group) following the OTHERWISE.

Examples

```
data_length = length(message)
select
   when data_length > 100 then
      say 'Data is longer than 100 characters'
   when data_length > 50 then
      say 'Data is longer than 50 characters and shorter than 101 characters'
   when data_length > 10 then
      say 'Data is longer than 10 characters and shorter than 51 characters'
   otherwise
      say 'Data is shorter than 11 characters'
end
```

Note

You must include at least one WHEN statement and an OTHERWISE statement in a SELECT group.

SETRH

SETRH [TYPE *request_type***] [CHAIN** *chain_position***] [ON(***rh_settings***)] [OFF(***rh_settings***)]**

Note: You must code at lease one of the operands (TYPE, CHAIN, ON, or OFF).

Where

request_type is a string constant that can have the following values:

DFC	Data flow control
FM	FM data
NC	Network control
SC	Session control.

This is used to specify the type of request being built. There is no default value for *request_type*. This parameter is optional. You can use uppercase or lowercase letters. The string constant must be enclosed in single or double quotation marks.

chain_position is a string constant that can have the following values:

FIRST	First RU of chain
MIDDLE	Middle RU of chain
LAST	Last RU of chain
ONLY	Only RU of chain.

If 'FIRST' is specified, ON(EXC) and OFF(CDI) are set to default values. There is no default value for *chain_position*. This parameter is optional. You can use uppercase or lowercase letters. The string constant must be enclosed in single or double quotation marks.

rh_settings can be one or more RH keywords, separated by commas. The following list defines the valid RH keywords:

BB	Set begin bracket.
CDI	Set change direction.
CEB	Set conditional end bracket.
DR1	Definite response 1 requested.
DR2	Definite response 2 requested.
EB	Set end bracket.
EXC	Exception response requested.
FMI	This RU is formatted. This is ignored for a VTAMAPPL LU.
QRI	Set queued response indicator.
RESP	The data is a response.
SNI	This RU contains sense data.

This is how the STL SETRH statement maps over to *SNA Formats* bits:

WSim	SNA Terminology
BB	BBI
CDI	CDI
CEB	CEBI
DR1	DR1I
DR2	DR2I
EB	EBI
EXC	ERI/RTI [8]
FMI	FI
QRI	QRI
RESP	RRI
SNI	SDI

Function

The SETRH statement performs the following functions:

8. The ERI and RTI bits are the same bit.

- Modifies the SNA request/response header built for the next message sent based on the “on” or “off” settings. For example, the bits in the RH represented by the “on” RH settings will be set on and those represented by the “off” RH settings will be set off.
- Optionally specifies chaining of transmitted messages.

This statement only affects the next transmitted message, except in the case where chaining is specified. If chaining is specified, then all messages will be considered a part of the chain until the last RU in the chain is sent.

Examples

```
/* Set begin bracket                                      */
/* and specify that this is a data flow control request   */

setrh type 'DFC' on(bb)
type ...                          /* Enter the data to send */
transmit                          /* Send the data with the */
                                  /* modified RH values     */

/* Indicate that this is the first RU in a chain of RUs   */

setrh chain 'FIRST'
type ...                          /* Enter the data to send */
transmit                          /* Send the data with the */
                                  /* modified RH values     */
```

Notes

- This statement is valid only for SNA simulations.
- The SETRH is overridden for all SNACMNDs except for LUSTAT. For LUSTAT, the following RH settings cannot be changed: TYPE is DFC, CHAIN is ONLY, FMI is ON, RESP is OFF, and SNI is OFF.

SETTH

```
SETTH SEQNO sequence_number
```

Note: The SEQNO operand is required.

Where

sequence_number is an integer constant that specifies the value of the transmission header sequence number field.

Note: Changing the sequence number field can cause an error in the access method of the system under test.

Function

The SETTH statement modifies the SNA request/response header built for the next message sent based on the “on” and “off” settings.

This statement only affects the next transmitted message. It is ignored for 3270 SNA and 5250 terminals, and for CPI-C transaction programs.

Examples

```
/* Set transaction header sequence number               */

setth seqno 20
type ...                          /* Enter the data to send */
transmit                          /* Send the data with the */
                                  /* modified TH values     */
```

Note

This statement is valid only for SNA simulation.

SIGNAL

SIGNAL *event_name* **[AFTER** *time* **[TAG** *event_tag***]]**

Where

event_name is a string expression that specifies the name of an event. If it is a string constant expression, *event_name* must be 1 to 8 alphanumeric characters and enclosed in single or double quotes. (Strings containing hexadecimal constants are exempt from this restriction.) If you specify a nonconstant string expression, the first 8 characters of the string are used. If the string is shorter than 8 characters, the available characters are used. To satisfy conditions using event names, the first 8 characters must match exactly. If you specify a string variable, it cannot be the name of one of the reserved variables (for example, BUFFER).

time is an integer value that indicates the number of seconds that the posting of the event is to be delayed. It must be an integer expression with a value from 1 to 21474836.

event_tag specifies a "tag" to be assigned to the SIGNAL statement. This tag is only valid if the *time* value is also coded. The rules for specifying the *event_tag* are the same as those for specifying *event_name*.

Function

The SIGNAL statement signals the named event. A signal affects all terminals in the network; therefore, all outstanding ON SIGNALED conditions for the named event will be satisfied.

Note: See "Specifying variable event names with a time delay" on page 306 for information about specifying variable event names with a time delay.

When you specify *time* and *event_tag*, you can cancel events that have been assigned the same tag before the specified time has elapsed. You can assign the same tag to events that are specified with a POST, QSIGNAL, RESET, or SIGNAL statement. Thus, you can cancel several events by assigning them the same tag.

If an event tag is not specified, a default tag will be assigned that has a value that is the same as the event name.

See "CANCEL" on page 366 for more information about how to use a tag to cancel events.

Examples

```
signal 'GOODTEST'
```

SNACMND

```
SNACMND(type,[arg1...arg10])
```

Where

type is one of the keywords as follows:

BID, BIS, CANCEL, CHASE, CLEAR, INITSELF, LUSTAT, QUEC, RELQ, RSHUTD, RTR, SBI, SDT, SHUTD, SIGNAL, STSN, TERMSELF, or UNBIND.

This is used to specify the type of SNA command to be simulated.

Note: Coding a *type* of INITSELF or TERMSELF on the SNACMND produces the same results as coding the INITSELF or TERMSELF statement, respectively.

arg1,...,arg5 depends on the *type* specified. The table below tells what arguments each type of SNA command expects - required arguments are asterisked. The list following the table shows the values that are expected for each argument.

Table 15. Argument List for SNACMND Statement Based on Command Type

Type	1st Argument	2nd Argument	3rd Argument	4th Argument	5th Argument
BID					
BIS					
CANCEL					
CHASE					
CLEAR					
INITSELF	*resource**	*mode*	*user_data*	*log_byte*	
LUSTAT	*sense*	*log_byte*			
QUEC					
RELQ					
RIDISC					
RNDISC					
RSHUTD					
RTR					
SBI					
SDT					
SHUTD					
SIGNAL	*sense*	*log_byte*			
STSN	*pseqact*	*pseqval*	*sseqact*	*sseqval*	*log_byte*
TERMSELF	*resource*	*log_byte*			
UNBIND	*sense*	*son*			

Argument	Format
resource	is a string expression. If *resource* is a string constant expression, it must be 1 to 8 alphanumeric characters and must not contain blanks. The characters must be enclosed in single or double quotation marks. String constant expressions containing hexadecimal strings are exempt from this restriction. The STL Translator will translate all lowercase characters to uppercase in these constants.
mode	is a string expression. If *mode* is a string constant expression, it must be 1 to 8 alphanumeric characters and must not contain blanks. The characters must be enclosed in single or double quotation marks. String constant expressions containing hexadecimal strings are exempt from this restriction. The STL Translator will translate all lowercase characters to uppercase in these constants.
user_data	is a string expression. The length of *user_data* should not exceed 255 characters although the STL Translator permits longer strings. A maximum of 255 characters will be sent.
log_byte	is a 1-byte string constant or a string expression. **Note:** Only the first character or first two hexadecimal digits of the string expression are used.
pseqact	Is a string constant that can have the following values: **IGNORE** Specifies that this STSN command is to be ignored. **SET** Specifies that the primary-to-secondary sequence number of the secondary end user is to be set to *pseqval* operand value. **TEST** Specifies that the secondary end user must return its primary-to-secondary sequence number in the response RU. **TESTSET** Specifies that the primary-to-secondary sequence number of the control program (CP) manager is to be set to the *pseqval* operand value, and the secondary end user is to compare that value against its own and respond accordingly.
pseqval	Is an integer constant with a value from 0 to 65535.
sseqact	Is a string constant that can have the following values: **IGNORE** Specifies that this STSN command is to be ignored. **SET** Specifies that the primary-to-secondary sequence number of the secondary end user is to be set to *sseqval* operand value. **TEST** Specifies that the secondary end user must return its primary-to-secondary sequence number in the response RU. **TESTSET** Specifies that the primary-to-secondary sequence number of the control program (CP) manager is to be set to the *sseqval* operand value, and the secondary end user is to compare that value against its own and respond accordingly.
sseqval	Is an integer constant with a value from 0 to 65535.
log_byte	Is a 1-byte string constant or a string expression.
sense	is a 8-digit hexadecimal string constant.
son	is a 2-digit hexadecimal string constant.

Function

The SNACMND statement builds an SNA command to be sent by the simulated logical unit and sets the SNA headers in the message to indicate the presence of the command.

Each *type* of SNA command has a function, that is, it corresponds to a particular SNA command. The function of each *type* of SNA command is as follows:

BID	Bid
BIS	Bracket initiation stopped
CANCEL	Cancel chain
CHASE	Chase responses
CLEAR	Clear
INITSELF	Initiate self, format 0
LUSTAT	LU status
QUEC	Quiesce at end of chain
RELQ	Release quiesce
RSHUTD	Request shutdown
RTR	Ready to receive
SBI	Stop bracket initiation
SDT	Start data traffic
SHUTD	Shutdown
SIGNAL	Signal
STSN	Set and test sequence numbers
TERMSELF	Terminate self, format 0
UNBIND	Unbind session.

The arguments have a unique function for each of the SNA command types. The function of the arguments are as follows:

resource	Specifies the name of the partner LU with which the session is to be established or terminated.
mode	Specifies the name of a mode entry table to be used when simulating SNA terminals or logical resources.
user_data	Specifies unique user information, such as passwords.
log_byte	specifies a byte of user data to be associated with all data transmitted and received. The *log_byte* remains active until data is "typed" and transmitted with a TRANSMIT statement or until an INITSELF, TERMSELF, or another SNACMND statement is issued. This byte gives users of the Response Time Utility a way to identify transactions when gathering statistics by the various "log byte" categories. The default *log_byte* is X'00'.
sense	Specifies the sense information to be sent with the SNA command being simulated.
son	Specifies the session outage notification (SON) code to be sent with the command being simulated.
pseqact	Specifies the action to be executed by the STSN receiver for the primary-to-secondary sequence number. The default *pseqact* value is SET.

pseqval	Specifies the primary-to-secondary sequence number value to be sent with the STSN. The default *pseqval* is 0.
sseqval	Specifies the secondary-to-primary sequence number value to be sent with STSN. The default *sseqval* is 0.
sseqact	Specifies the action to be executed by the STSN receiver for the secondary-to-primary sequence number. The default *sseqact* value is SET.
log_byte	Specifies the byte of user data to be associated with all data transmitted and received. The *log_byte* remains active until data is "typed" and transmitted with a TRANSMIT statement or until an INITSELF, TERMSELF, or another SNACMND statement is issued. This byte gives users of the Response Time Utility a way to identify transactions when gathering statistics by the various "log_byte" categories. The default *log_byte* is X'00'.

Examples

```
/* Send an Unbind SNA command to Unbind the session.        */
/* Send a sense code of 08010002.                            */

snacmnd(unbind,'08010002'x)

/* Send a command to start the data traffic.                */

snacmnd(sdt)

/* Send an SNA attention key.                               */

snacmnd(signal,'00010000'x)

log_byte = 'm'

/* Send an SNA attention key, setting the log               */
/* byte to 'm' using the variable log_byte.                 */

snacmnd(signal,'00010000'x,log_byte)
```

Notes

- This statement is valid only for SNA simulation.
- The SNACMND statement causes a Transmit Interrupt. Thus, all accumulated message data will be sent.
- The SETRH is overridden for all SNACMNDs except for LUSTAT. For LUSTAT, the following RH settings cannot be changed: TYPE is DFC, CHAIN is ONLY, FMI is ON, RESP is OFF, and SNI is OFF.

STRING

```
STRING [{SHARED|UNSHARED}] variable_list
```

Where

variable_list is any number of valid STL variable names not previously declared, separated by commas.

Function

The STRING statement explicitly declares one or more STRING variables, which may be specified as SHARED or UNSHARED. If the class designation is not included, the variable will be assigned the UNSHARED class.

Examples

```
                       /* Declares the unshared string variables, */
                       /* "password" and "logon_mode".            */
string unshared password, logon_mode

string mydata          /* Declares the unshared string variable,  */
                       /* "mydata".                               */

string shared netdata  /* Declares the shared string variable,    */
                       /* "netdata".                              */
```

Note

The STRING statement is a declarative statement. You can code it only **outside** an STL procedure.

STRIPE

```
STRIPE stripe_data
```

Where

stripe_data is a string expression.

Function

The STRIPE statement defines message data to be transmitted to the system under test by a magnetic stripe reader. This statement is valid for 3270 terminal simulation only. WSim ignores the STRIPE statement if you use it when simulating nondisplay terminals.

Examples

```
stripe '123456789'    /*  Send this string to the host. */
```

Note

The STRIPE statement causes a Transmit Interrupt. Thus, all accumulated message data will be sent. See “Interrupting program execution” on page 286 for information about a Transmit Interrupt.

SUSPEND

```
SUSPEND( ┌ {'RATE',rate_table_number}[,uti_value]  ┐)
         │ {fixed_time}                             │
         │ {random_function}                        │
         └                                          ┘
```

Where

rate_table_number is an integer constant expression with a value from 0 to 255.

uti_value is a string constant expression that names a UTI statement in the network definition.

Note: The name of the network-level UTI is NTWRKUTI.

fixed_time is an integer expression with a value from 0 to 2147483647.

random_function is an instance of the RANDOM function as described in "RANDOM" on page 481.

Function

The SUSPEND statement suspends normal execution for a simulated terminal for the specified amount of time. Although normal execution of the procedure is suspended, all outstanding asynchronous conditions remain active and are tested as needed. After the suspension interval is complete, WSim continues normal synchronous execution with the statement following the SUSPEND statement (unless control has been altered by a CALL asynchronous subset statement).

The value of the SUSPEND argument is multiplied by the terminal's UTI value or by the value of the UTI specified as an argument to arrive at the suspension interval (in hundredths of seconds). This value applies to the RATE, fixed time and random number specifications. If you do not specify an argument, the default DELAY value for the terminal is used to calculate the suspension interval. If you specify *fixed_time*, the value of the integer constant expression is used. If you specify 'RATE',*rate_table_number*, a value chosen randomly from the designated rate table is used. If you specify *random_function*, the value of the random number returned by the RANDOM function is used.

Examples

```
NET1    NTWRK
        .
        .
        .
UTISEC  UTI  100
        .
        .
Deck1:  msgtxt
        .
        .
        .
suspend(5,'UTISEC')    /* Suspend execution for 5 seconds         */
                       /* (UTI=UTI value of "UTISEC").            */

suspend(5)             /* Suspend execution for 5 seconds         */
                       /* (UTI=100).                              */

suspend('rate',1)      /* Select the next value from rate         */
                       /* table 1 and multiply it by the UTI      */
                       /* to arrive at the suspension interval.   */

suspend(random(5,10))  /* Suspend for random interval between 5   */
                       /* and 10 seconds (UTI=100).               */

suspend(random('rn',4))/* Suspend for random interval based on    */
                       /* RN statement number 4.                  */
```

The SUSPEND statement can be useful when waiting for data to arrive at a display terminal before taking further action, as shown in the following example.

```
type 'Hello'
transmit    /* Notice: There is no WAIT clause on this TRANSMIT.  */
            /* The normal Transmit Interrupt delay will be taken; */
            /* then, normal STL execution will resume.            */

/**********************************************************/
/* The following DO WHILE loop will continue to be executed */
/* while 'Goodbye' is not on the screen.  The screen is     */
/* checked every 5 seconds for the string.  If it has not   */
/* yet been received, execution is suspended for another 5  */
/* seconds.                                                 */
/**********************************************************/
do while index(screen,'Goodbye') = 0
   suspend(5)     /* Suspend execution for 5 seconds (UTI=100). */
end

/**********************************************************/
/* 'Goodbye' was received.  Go on to next activity.         */
/**********************************************************/
&#38;#8942;
```

Notes

- You can cancel a suspension interval with the CANCEL SUSPEND asynchronous subset statement. For details, see "CANCEL" on page 366.
- This statement causes a Transmit Interrupt. Thus, all accumulated message data will be sent.

SYSREQ

```
SYSREQ
```

Function

The SYSREQ statement simulates the action of the SYSREQ key on an SNA device. This statement is only valid for Telnet 3270E and Telnet 5250 simulation.

Examples

```
sysreq         /* Simulate the pressing of the SYSREQ key    */
```

TAB

```
TAB
```

Function

The TAB statement simulates the Tab key on 3270 display terminals. The statement is valid for simulation of these terminals only.

Examples

```
home              /* Move to first input field on screen. */
do i = 2 to 5     /* Tab to the 5th input field.         */
   tab
end
```

Note

You can also simulate the action of the TAB key by using the TAB function on a TYPE statement.

TERMSELF

```
TERMSELF([resource][,log_byte])
```

Where

resource is a string expression. If *resource* is a string constant expression, it must be from 1 to 8 alphanumeric characters, must not contain blanks, and must be enclosed in single or double quotation marks. String constant expressions containing hexadecimal strings are exempt from these restrictions. The STL Translator will translate lowercase letters to uppercase.

log_byte is a 1-byte string constant or a string expression that specifies a byte of user data to be associated with all data transmitted and received. The *log_byte* remains active until data is “typed” and transmitted with a TRANSMIT statement again or until an INITSELF, TERMSELF, or SNACMND statement is issued. This byte gives users of the Response Time Utility a way to identify transactions when gathering statistics by the various user-defined “log_byte” categories. Only the first character or first two hexadecimal digits of the string expression are used. The default *log_byte* is X'00'.

Function

The TERMSELF statement sends TERMINATE SELF format 0 RU terminate sessions when simulating SNA terminals or logical units (LUs).

resource specifies the name of the partner LU. If *resource* is not specified, the name of the LU's current partner is used.

Examples

```
log_byte = 'm'
if SNA_sense_received then     /* Something went wrong. */
   do
      say luid()' is terminating session - SENSE received'
      termself('MYAPPL',log_byte)
      quiesce                  /* Go to sleep.          */
   end
else
   call goodproc
```

Notes

- The TERMSELF statement causes a Transmit Interrupt. Thus, all accumulated message data will be sent.

- The TERMSELF statement is identical to specifying a type of TERMSELF on an SNACMND statement.

TRANSMIT

```
TRANSMIT [USING aid_key] [LOGGING log_byte]
         [AND wait_statement]
```

Where

aid_key is an Attention Identifier (AID) key name taken from the following list:

CLEAR	HELP	ROLLUP
CLEARPTN	PA1-3	SEND
CMD*nn*	PF*nn*	SENDLINE
CURSRSEL	PRINT	SENDMSG
ENTER	ROLLDOWN	SYSREQ

Note: To simulate the action of the SYSREQ key on a Telnet 3270E or Telnet 5250 device, use the SYSREQ device key statement.

nn is an integer from 1 to 24 used to identify command and PF keys.

log_byte is a 1-byte string constant or a string expression that specifies a byte of user data to be associated with all data transmitted and received. The *log_byte* remains active until data is "typed" and transmitted with a TRANSMIT statement again or until an INITSELF, TERMSELF, or SNACMND statement is issued. This byte gives users of the Response Time Utility a way to identify transactions when gathering statistics by the various user-defined "log_byte" categories. Only the first character or first two hexadecimal digits of the string expression are used. The default *log_byte* is X'00'.

wait_statement is a valid STL WAIT statement as described in "WAIT" on page 446.

Function

The TRANSMIT statement directs WSim to transmit data previously "typed" at a simulated terminal with a TYPE statement. Optionally, the TRANSMIT statement can specify the AID key to be associated with the transmitted data and an asynchronous condition that must be satisfied before normal execution resumes.

Note: Do not use the TRANSMIT statement in CPI-C transaction program simulations. This statement will cause execution of the CPI-C transaction program to be interrupted. CPI-C transaction programs send messages using the CMSEND statement.

aid_key specifies the AID generating key to be associated with the transmitted data. If the "USING aid_key" clause is not included, the default AID of ENTER is used.

wait_statement, when coded, directs WSim not to resume normal STL execution until the specified wait condition is met. If the "AND wait_statement" clause is not included, WSim resumes normal STL execution after the Transmit Interrupt for the terminal expires. For more information about the TRANSMIT statement and the Transmit Interrupt, see Chapter 20, “Transmitting and receiving messages from an STL program,” on page 285.

Examples

```
type 'Hello'

/*********************************************************/
/* Press PF1.  This transmits the "Hello" just entered on  */
/* the TYPE statement.  Wait until the screen is refreshed */
/* by the application with an ERASEWRITE ('F5'x) command.  */
/*********************************************************/

transmit using PF1 and wait until onin substr(ru,1,1) = 'f5'x

/*********************************************************/
/* Press PF3 and set the log byte to be the character 'f'. */
/*********************************************************/

transmit using PF3 logging 'f'

type 'Goodbye'

log_byte = 'm'
type 'Transmit this message using a variable log byte.'

/*********************************************************/
/* This transmits the message on the previous type        */
/* statement and sets the log byte to 'm' using the       */
/* variable log_byte.                                     */
/*********************************************************/
transmit logging log_byte
```

Note

If the TRANSMIT statement is encountered and no data has been “typed” with a TYPE statement, only the AID indicator will be transmitted.

TYPE

TYPE *data*

Where

data is a string expression.

Function

The TYPE statement defines message data to be transmitted to the system under test by the simulated terminal. Multiple TYPE statements can be used to “build” complex messages. For information about defining messages, see Chapter 19, “Generating messages for an STL program,” on page 267.

Examples

```
type 'Hello, my name is 'luid()'.'
if a = 1 then
   do
      a = a + 1
      type '  What is your name?'
   end
else
   type '  How are you?'
transmit and wait until onin substr(ru,1,5) = 'Hello'
```

When A is 1, the preceding example will transmit the message, “Hello, my name is LU1. What is your name?” When A is not 1, the following message will be transmitted: “Hello, my name is LU1. How are you?”

Notes:
- Data typed for display devices represents data being entered by an operator into a display buffer. The actual data transmitted is then constructed by WSim for the specific display simulation type.
- Data typed for TCP/IP FTP Client devices represents a local FTP command. The data to be transmitted to the server, if any, is then constructed by WSim.
- Do not use the TYPE statement for CPI-C transaction program simulations. Transaction program message data is defined by setting up a save area that represents the send buffer.

USEREXIT

```
USEREXIT(module_name[,parm_list])
```

Where

module_name is a string constant expression. It must be 1 to 8 alphanumeric characters and must conform to standard JCL member naming conventions. That is, each character must be one of: A-Z, a-z, #, $, @, 0-9, with the first character alphabetic. The characters must be enclosed in single or double quotation marks.

parm_list is a string constant expression. The length of *parm_list* should not exceed 100 characters.

Function

The USEREXIT statement invokes a user exit routine during the execution of the program. It allows the exit routine to produce messages and control execution of the program by setting return codes.

The *module_name* specifies the member (user exit load module) in the load library that was loaded during initialization and is to gain control when this statement is encountered during program execution.

Note: You should concatenate the user exit data sets to the STEPLIB DD JCL statement.

parm_list specifies the parameters to be passed to the user exit when it is called.

The user exit routine must set one of the following return codes in register 15 to indicate the action to be taken upon completion of the user exit:

Code Meaning

0 Continue executing the program as if the user exit had not been called.

4 Continue executing the program as if the user exit had not been called.

Note: Unless you included Scripting Language statements directly using @GENERATE, codes 0 and 4 result in the same action for STL programs.

8 A message was created by the user exit. Continue processing as if the message had been generated by a TYPE statement.

Note: STL statements that cause a Transmit Interrupt or another TYPE statement following the user exit will cause this message to be transmitted.

12 Put the terminal in the wait state. If a message was created before calling the exit routine but has not yet been transmitted, transmit it now.

16 Do not put the terminal in the wait state. If a message was created before calling the exit routine but has not yet been transmitted, transmit it now.

Note: Refer to , SC31-8950 for more information on user exit routines.

Examples

```
userexit('MYEXIT','NOWAIT')
```

UTI

```
UTI uti_name
```

Where

uti_name is the 1- to 8-character string constant expression corresponding to the name coded on a UTI network definition statement. You can use uppercase or lowercase letters. The *uti_name* must be enclosed in single or double quotation marks.

Note: The name of the network-level UTI is NTWRKUTI.

Function

You use the UTI statement to change the individual user time interval (IUTI) for your terminal. The new IUTI value for the terminal will be the current value of the UTI identified by *uti_name*.

Examples

```
uti 'UTI01'        /* Make UTI01 the terminal's current UTI. */
uti 'NTWRKUTI'     /* Change back to the network UTI value.  */
```

VERIFY

```
VERIFY simple_condition [FOR description]
```

Where

simple_condition is of the form:

actual_expression relational_operator expected_expression

The *actual_expression* represents the actual value to be verified and *expected_expression* represents the value that the actual value is expected to contain.*relational_operator* is a valid STL relational operator. The record is written only if the condition is true. See "Using conditions and relational operators" on page 261 for more information on simple conditions and*relational_operator. description* is a string expression. This gives an identifier to VRFY (VERIFY) records in the log data set. The Loglist Utility can then use this description to group common records in the loglist output. The description you specify can be any length; however, only a maximum of 50 characters are logged.

Function

The VERIFY statement creates a verify record in the log data set, which is used by the Loglist Utility to produce Verification Reports. See , SC31-8947 for more information on the purpose and format of these reports.

Note: Verification Reports reference resources, counters, save areas, and switches. Therefore, refer to the variable dictionary to determine how STL variables map to their associated resources.

Examples

```
/* Serial numbers, which are found at offset 30 on the screen   */
/* cannot contain a "9" as the first digit.  If this occurs     */
/* then log a verify record.  The description "Serial number    */
/* in error" can be used when looking at the Verification       */
/* Summary Report to determine how many errors of this nature   */
/* occurred.                                                    */

verify substr(screen,30,1) = '9' for 'Serial number in error'

verify amount = 0 /* Log verify record if "amount" equals zero. */
```

WAIT

```
WAIT ┌                                      ┐
     │ UNTIL {ONIN [asynchronous_condition]} │
     │       {ONOUT [asynchronous_condition]}│
     │       {POSTED(event_name)}            │
     │       {SIGNALED(event_name)}          │
     └                                      ┘
```

Where

asynchronous_condition is a valid asynchronous condition. See "Setting up asynchronous conditions" on page 299 for a description of asynchronous conditions. *event_name* is a string expression that specifies the name of an event. If it is a string constant expression, *event_name* must be 1 to 8 alphanumeric characters and enclosed in single or double quotes. (Strings containing hexadecimal constants are exempt from this restriction.) If you specify a nonconstant string expression, the first 8 characters of the string are used. If the string is shorter than 8 characters, the available characters are used. If you specify a string variable, it cannot be the name of one of the reserved variables (for example, BUFFER).

Function

The WAIT statement stops program execution and places the terminal in a wait state. You may optionally specify those asynchronous conditions under which WSim releases the terminal from its wait state. This statement enables you to simulate the action of a terminal operator waiting for a reply before entering the next message.

The terminal is released from the wait state under the following conditions:
- If WAIT UNTIL ONIN│ONOUT *asynchronous_condition is used, the terminal will be released from its wait state when the specifiedasynchronous_condition is satisfied.*
- If WAIT UNTIL POSTED(*event_name) is used, the terminal will be released from its wait state when the specified event is posted.*
- If WAIT UNTIL SIGNALED(*event_name) is used, the terminal will be released from its wait state when the specified event is signaled or qsignaled.*
- A BIND SNA RU is received.
- Another procedure is called as the result of an ONIN, ONOUT, or ON SIGNALED condition being satisfied.
- Automatic terminal recovery is entered.
- The operator enters an S (Start) command.
- The operator enters an F (Console Recovery) command, causing the terminal to enter terminal recovery.

An ONIN or ONOUT portion of the WAIT statement does not have to include a condition. The ONIN or ONOUT keyword may end the statement. Any incoming (ONIN) or outgoing (ONOUT) message will satisfy such a null condition.

Notes:
- The WAIT statement can also be coded as a clause on the TRANSMIT statement.
- This statement causes a Transmit Interrupt. Thus, all accumulated message data will be sent.

Examples

```
wait until onin index(ru,'WELCOME') > 0
                                   /* Wait for WELCOME message.  */

wait until posted('MYEVENT')       /* Wait until MYEVENT is      */
                                   /* posted.                    */

transmit and wait until signaled('MYEVENT')
                                   /* Transmit data and wait     */
                                   /* until MYEVENT is signaled. */

wait until onin                    /* Wait until the next        */
                                   /* message is received.       */

transmit and wait until onin       /* Transmit data and wait     */
                                   /* until the next message is  */
                                   /* received.                  */
```

Note

A WAIT statement without an UNTIL clause places the terminal in a wait state without any obvious means of getting out. Any of the conditions listed above will reset the state. For this reason, you should normally include an UNTIL clause on a WAIT statement unless you have ONIN, ONOUT, ON SIGNALED, or other ways to reset the state.

Chapter 26. Reference to STL functions

This chapter describes the syntax and usage of the STL functions. For a general explanation of STL functions, see “Using functions” on page 249.

APPCLUID

```
APPCLUID()
```

Results

String

Function

The APPCLUID function returns the name of the APPC LU on which the transaction program is defined. The name of the APPC LU is the name field of the APPCLU statement with trailing blanks removed.

Examples

```
say 'The name of the APPCLU is: 'appcluid()'.'
```

Note

This function is intended for use with CPI-C simulations; results will be the null string if used in other simulations.

ATTR3270

```
ATTR3270(screen_location[,length])
```

Where

screen_location is one of several methods of identifying a location on a simulated screen.

length is an integer constant expression with a value between 1 and 11.

Results

String

Function

The ATTR3270 function returns a string containing information about the standard field, extended field, and character attribute values associated with a particular screen location. The length of the returned string will be from 1 to 11 characters, depending on the value specified by the *length* argument. If *length* is not specified, 9 characters will be returned.

The *screen_location* can be specified in one of the following ways:

- The *screen_position* is an integer expression with a value from 1 to 32767. This number is the position on the screen being queried (the first screen position is 1).

  For example, you can use the position in this way:

  ```
  attributes = attr3270(159,9)       /* Assign the attributes of   */
                                     /* screen position 159 to     */
                                     /* variable "attributes".     */

  a = 159                            /* Use a variable as a screen */
  attributes = attr3270(a,9)         /* position.                  */

  a = 2                              /* Use an integer expression  */
  attributes = attr3270(a*11,1)      /* as a screen position.      */
  ```

- Use COFF()[±{*offset*|(*offset*)}] to specify a screen location relative to the current cursor offset. If an increment or decrement is not specified, the position to be queried is the current cursor position. If an increment or decrement is specified, a screen position beyond (+) or before (-) the cursor position will be queried.

  If *offset* is not enclosed in parentheses, it must be either an integer constant or an integer variable. If it is enclosed in parentheses, it can be any integer expression. Its value must be from 0 to 32766.

  The following examples show how to use this function to specify the screen location.

  ```
  attributes = attr3270(coff())       /* Assign the attributes of  */
                                      /* the current cursor's      */
                                      /* location to "attributes". */

  cursor(100)
  attributes = attr3270(coff()+8)     /* Assign the attributes     */
                                      /* of screen offset 108 to   */
                                      /* "attributes".             */

  cursor(100)
  attributes = attr3270(coff()+(6+2),3)
                                      /* Assign the attributes     */
                                      /* of screen offset 108 to   */
                                      /* "attributes".  Only 3     */
                                      /* characters of attribute   */
                                      /* information will be       */
                                      /* returned.                 */

  cursor(100)
  offset = 102
  attributes = attr3270(coff()+offset) /* Set the attributes of    */
                                      /* screen offset 202 to      */
                                      /* "attributes".             */
  ```

- Use LENGTH(SCREEN)[-{*decrement*|(*decrement*)}] to specify a screen location relative to the end of the screen. If *decrement* is not specified, the last screen position will be queried. If *decrement* is specified, the screen location *decrement* number of positions before the end of the screen will be queried.

  If the *decrement* is not enclosed in parentheses, it must be either an integer constant or an integer variable. If it is enclosed in parentheses, it can be any integer expression. Its value must be from 0 to 32766.

  The following examples show how to use the end of the screen to define the screen location.

```
attributes = attr3270(length(screen))
                                     /* Assign the attributes     */
                                     /* of the last screen        */
                                     /* location to "attributes". */

attributes = attr3270(length(screen)-40)
                                     /* Assign the attributes     */
                                     /* of the screen location    */
                                     /* 40 characters from the    */
                                     /* end of the screen to      */
                                     /* "attributes".             */

offset = 35
attributes = attr3270(length(screen)-(offset*2))
                                     /* Set the attributes of the */
                                     /* screen location that is   */
                                     /* 70 characters from the    */
                                     /* end of the screen to      */
                                     /* "attributes".             */
```

- Use ROWCOL(*row_number,col_number*) to specify the row and column numbers of the screen location to be queried. *row_number* and *col_number* must be integer expressions with a value from 1 to 255.

  The following examples show how to specify the screen location using a row and column location.

```
attributes = attr3270(rowcol(1,1))  /* Assign the attributes     */
                                    /* of the first screen       */
                                    /* location to "attributes". */

a = 12; b = 40
attributes = attr3270(rowcol(a,b))  /* Assign the attributes of  */
                                    /* the screen location at    */
                                    /* row 12, column 40 to      */
                                    /* "attributes".             */
```

The ATTR3270 function returns up to eleven EBCDIC characters. Table 16 explains the meanings of these characters.

Table 16. Definitions of EBCDIC characters returned by ATTR3270

Byte	Defines	Values	
1	Attribute Information	**E =**	location specification error
		0 =	unformatted screen, default field and actual character attribute values will be generated
		1 =	formatted screen, no field attribute defined at specified location, actual field and character attribute values will be generated
		2 =	formatted screen, field attribute defined at specified location, actual field and default character attribute values will be generated
2	Standard Field Attribute	**x =**	EBCDIC translated standard field attribute character when byte 1 is set to 1 or 2. **Note:** A blank will be generated when byte 1 is set to E or 0.

Table 16. Definitions of EBCDIC characters returned by ATTR3270 (continued)

Byte	Defines	Values	
3	Highlighting Field Attribute	**N =**	no extended attribute buffer or byte 1 set to E
		0 =	default, normal highlighting
		1 =	blinking
		2 =	reverse image
		3 =	underlined
4	Highlighting Attribute Character	**N =**	no extended attribute buffer or byte 1 set to E
		0 =	default, field defined
		1 =	blinking
		2 =	reverse image
		3 =	underlined
5	Color Field Attribute	**N =**	no extended attribute buffer or byte 1 set to E
		0 =	default, normal color
		1 =	blue
		2 =	red
		3 =	pink
		4 =	green
		5 =	turquoise
		6 =	yellow
		7 =	white
6	Color Character Attribute	**N =**	no extended attribute buffer or byte 1 set to E
		0 =	default, field defined
		1 =	blue
		2 =	red
		3 =	pink
		4 =	green
		5 =	turquoise
		6 =	yellow
		7 =	white

Table 16. Definitions of EBCDIC characters returned by ATTR3270 (continued)

Byte	Defines	Values	
7	Character Set Field Attribute	**N =**	no extended attribute buffer or byte 1 set to E
		0 =	default, base character set
		1 =	APL
		2 =	PSA
		3 =	PSB
		4 =	PSC
		5 =	PSD
		6 =	PSE
		7 =	PSF
		8 =	DBCS
8	Character Set Character Attribute	**N =**	no extended attribute buffer or byte 1 set to E
		0 =	default, field defined
		1 =	APL
		2 =	PSA
		3 =	PSB
		4 =	PSC
		5 =	PSD
		6 =	PSE
		7 =	PSF
		8 =	DBCS
9	Field Validation Field Attribute	**N =**	field validation not supported or byte 1 set to E
		0 =	default, no field validation specified
		1 =	trigger
		2 =	mandatory enter
		3 =	trigger and mandatory enter
		4 =	mandatory fill
		5 =	mandatory fill and trigger
		6 =	mandatory fill and mandatory enter
		7 =	mandatory fill, mandatory enter, and trigger
10	Field Outlining Definition	**N =**	field outlining not supported or byte 1 set to E
		0–9, A–F =	field outlining bit settings

Table 16. Definitions of EBCDIC characters returned by ATTR3270 (continued)

Byte	Defines	Values	
11	SO/SI Operator Creation Attribute	**N =**	DBCS not supported or byte 1 set to E
		0 =	SO/SI creation by operator not enabled
		1 =	SO/SI creation by operator enabled

Examples

```
attributes = attr3270(coff())            /* Get attributes of     */
                                         /* cursor location.      */
if substr(attributes,3,1) = '1' & ,      /* If cursor is blinking */
  substr(attributes,5,1) = '2' then      /* and red, say so.      */
  say 'The cursor is currently in a BLINKING RED field.'
else                                     /* Otherwise, the cursor */
                                         /* is not in a blinking, */
                                         /* red field.            */
&#38;#8942;
```

Restrictions on use of ATTR3270 function in asynchronous conditions

The ATTR3270 function returns a string constant that can usually be used wherever string constants are valid. There are, however, some restrictions on the use of this function within asynchronous conditions. They are the following:

- If you use the COFF(), LENGTH(SCREEN), or ROWCOL(*row,column*) form of *screen_location*, you cannot specify an integer expression that contains one or more variables.
- If you use the *screen_position* form of *screen_location*, you can specify only integer constant expressions.

The following examples show valid and invalid uses of the ATTR3270 function in asynchronous conditions:

```
offset = 45
row =    23
column = 67

/*****************************************************/
/* VALID uses of ATTR3270 in an asynchronous condition. */
/*****************************************************/

onin attributes = attr3270(108,4) then found = on
onin attributes = attr3270(coff()+offset) then found = on
onin attributes = attr3270(length(screen)-(13*45)) then found = on
onin attributes = attr3270(rowcol(row,column),7) then found = on

/******************************************************/
/* INVALID uses of ATTR3270 in an asynchronous condition. */
/******************************************************/

onin attributes = attr3270(offset) then found = on
               /* The specification for screen_location must be an */
               /* integer constant expression.                     */

onin attributes = attr3270(coff()-(offset+4),2) then found = on
onin attributes = attr3270(rowcol(row+8,column-10)) then found = on
               /* The COFF(), LENGTH(SCREEN), and ROWCOL(row,col)  */
               /* specification for screen_location cannot         */
               /* represent an integer expression containing one   */
```

```
                                   /* or more variables.  In the above examples, the   */
                                   /* variables offset, row, and column are used as    */
                                   /* part of an expression.                           */
```

BITAND

```
BITAND (string1[,[string2] [,pad]])
```

Where

string1 is a string expression

string2 is a string expression. This is optional.

pad is a 1-character string constand or a 2-digit hexadecimal constant used for padding. This is optional.

Results

String

Function

The BITAND function returns a string composed of the *string1* and *string2* input strings logically AND'ed together, bit by bit. The length of the result is the length of the longer of the two strings. The shorter of the two strings is extended with the pad character on the right before carrying out the logical operation. The default for *string2* is the null string and the default for *pad* is X'FF'.

Examples

```
a = BITAND('55AA'x,'FF88'x)           /* Assigns '5588'x to "a"      */
b = BITAND('COLORADO','FF'x,'BF'x)    /* Assigns 'Colorado' to "b"   */
```

Note: The BITAND function cannot be used in asynchronous conditions.

BITOR

```
BITOR(string1[,[string2][,pad]])
```

Where

string1 is a string expression.

string2 is a string expression. This is optional.

pad is a 1-character string constant or 2-digit hexadecimal constant used for padding. This is optional.

Results

String

Function

The BITOR function returns a string composed of the *string1* and *string2* input strings logically inclusive-OR'ed together, bit by bit. The length of the result is the length of the longer of the two strings. The shorter of the two strings is extended with the pad character on the right before carrying out the logical operation. The default for *string2* is the null string and the default for *pad* is X'00'.

Examples

```
a = BITOR('152535'x,'22'x)          /* Assigns '372535'x to "a"       */
b = BITOR('Barney',,'40'x)          /* Assigns 'BARNEY' to "b"        */
c = BITOR('112233'x,'66'x,'88'x)    /* Assigns '77AABB'x to "c"       */
```

Note: The BITOR function cannot be used in asynchronous conditions.

BITXOR

BITXOR(*string1***[,[***string2***][,***pad***]])**

Where

string1 is a string expression.

string2 is a string expression. This is optional.

pad is a 1-character string constant or 2-digit hexadecimal constant used for padding. This is optional.

Results

String

Function

The BITXOR function returns a string composed of the *string1* and *string2* input strings logically eXclusive-ORed together, bit by bit. The length of the result is the length of the longer of the two strings. The shorter of the two strings is extended with the pad character on the right before carrying out the logical operation. The default for *string2* is the null string and the default for *pad* is X'00'.

Examples

```
a = BITXOR('1211'x,'22'x)             /* Assigns '3011'x to "a"     */
b = BITXOR('1111'x,'444444'x,'40'x)    /* Assigns '555504'x to "b"   */
c = BITXOR('AAAA'x,,'FF'x)            /* Assigns '5555'x to "c"     */
```

Note: The BITXOR function cannot be used in asynchronous conditions.

B2X

```
B2X(binary_string)
```

Where

binary_string is a string expression containing only '0's or '1's.

Results

String

Function

The B2X function returns a string in character format, that represents *binary_string* converted to hexadecimal. It can be any length. You can optionally include blanks in *binary_string* (at four-digit boundaries only, not leading or trailing) to aid readability; they are ignored. The returned string uses uppercase alphabetics for the values A-F, and does not include blanks.

If *binary_string* is the null string or a string formatted other than as described above, B2X returns a null string. If the number of binary digits in *binary_string* is not a multiple of four, then up to three 0 digits are added on the left before the conversion to make a total that is a multiple of four.

Examples

```
a = B2X('10101010')                    /* Assigns 'AA' to "a"      */
b = B2X('1   0011     1100')           /* Assigns '13C' to "b"     */
c = B2X('10001110    0111')            /* Assigns '8E7' to "c"     */
```

Note: The B2X function cannot be used in asynchronous conditions.

CCOL

```
CCOL()
```

Results

Integer

Function

The CCOL function returns the column number of the current cursor position for display terminals. The first column on the screen is column number 1.

Examples

```
a = ccol() + 5   /* Set "a" to the cursor column plus 5. */
```

Note

This function returns unpredictable results for nondisplay terminals.

CENTER

```
CENTER(string,length[,pad])
```

Where

string is a string expression.

length is an integer expression with a value from 1 to 32767.

pad is a 1-character string constant or 2-digit hexadecimal constant used for padding. This is optional.

Results

String

Function

The CENTER function returns a string of length *length* with *string* centered in it, with *pad* characters added as necessary on both ends. If *string* is longer than *length*, it is truncated at both ends to fit. If an odd number of characters are truncated or added, the right-hand end loses or gains one more character than the left-hand end. The default value for *pad* is a blank (X'40').

Examples

```
a = CENTER('abc',5)                /* Assigns ' abc ' to "a"      */
b = CENTER('abcdef',10,'C1'x)      /* Assigns 'AAabcdefAA' to "b" */
c = CENTER('abcdefg',4)            /* Assigns 'bcde' to "c"       */
```

Note: The CENTER function cannot be used in asynchronous conditions.

CHAR

```
CHAR(number[,length])
```

Where

number is an integer expression.

length is an integer constant expression with a value from 1 to 10.

Results

String

Function

The CHAR function converts an integer value into its EBCDIC character representation and returns the converted string. The length of the returned string is determined by *length*.

If the converted number is longer than *length*, only the rightmost digits will be returned. If the converted number is shorter than *length*, the returned value will be padded with zeros on the left.

If *length* is not specified, the converted value will be returned without leading zeros.

Examples

```
i = 57
say 'Variable I is equal to 'char(i)'.'
```

The resulting message will be “Variable I is equal to 57.”

Note

Integer values **must** be converted using the CHAR function before they can be used as part of a string expression.

CMONTH

```
CMONTH()
```

Results

String

Function

The CMONTH function returns the name of the current month in mixed case letters (January, February, ..., December).

Examples

```
say 'The current month is 'cmonth()'.'
```

COFF

```
COFF()
```

Results

Integer

Function

The COFF function returns the offset of the current cursor position for display terminals. The offset is relative to the beginning of the screen (or of the currently active partition for terminals that support partitioning). The first position of a screen (partition) is offset 1, except when the COFF function is explicitly invoked from the CURSOR function, in which case it is offset 0.

Examples

```
a = coff()         /* Set variable "a" to the cursor's offset.     */
cursor(coff()+5)   /* Move the cursor five positions to the right. */
```

Note

This function returns unpredictable results for nondisplay terminals.

COPIES

```
COPIES(string,n)
```

Where

string is a string expression.

n is an integer expression with a value from 1 to 32767.

Results

String

Function

The COPIES function returns *n* concatenated copies of *string*.

Examples

```
a = COPIES('abc',3)             /* Assigns 'abcabcabc' to "a"       */
b = COPIES(' ab',3)             /* Assigns ' ab ab ab' to "b"       */
```

Note: The COPIES function cannot be used in asynchronous conditions.

CPOS

```
CPOS(row_number,column_number)
```

Where

row_number is an integer expression that will be compared to the current row of the cursor. If an integer constant is used, it must be from 1 to 255.

column_number is an integer expression that will be compared to the current column of the cursor. If an integer constant is used, it must be from 1 to 255.

Results

Bit

Function

The CPOS function returns a bit value: ON if the current cursor position matches the row and column values specified as function arguments; OFF otherwise.

Examples

```
if cpos(13,25) then ...    /* Condition will be met if      */
                           /* cursor is on row 13, column   */
                           /* 25.                           */

if cpos(13,colx) then ...  /* Condition will be met if the  */
                           /* cursor row is 13, and the     */
                           /* cursor column is the same as  */
                           /* the value of variable "colx." */

if cpos(1,1) & substr(screen,coff(),5) = 'Hello' then ...
                           /* Condition will be met if      */
                           /* cursor is on row 1, column 1, */
                           /* and the screen contains       */
                           /* 'Hello' at the cursor         */
                           /* location.                     */

on_target = cpos(12,40)    /* Bit variable "on_target" will */
                           /* be set ON if the cursor is at */
                           /* row 12, column 40; otherwise, */
                           /* it will be set OFF.           */
```

CROW

```
CROW()
```

Results

Integer

Function

The CROW function returns the row number of the current cursor position for display terminals. The first row on the screen is row number 1.

Examples

```
a = crow() + 5      /* Set "a" to the cursor row plus 5. */
```

Note

This function returns unpredictable results for nondisplay terminals.

C2D

```
C2D(string[,length])
```

Where

string is any valid string expression.

length is an integer constant expression with a value of 1 to 4.

Results

Integer

Function

The C2D function returns the integer value of the binary representation of *string*. *length* specifies the number of characters of *string* to be converted. If *length* is not specified, all characters will be converted.

Examples

```
i = '50'x       /* "i" is a string variable.                 */
a =  c2d(i)     /* "a" is an integer variable with a value of */
                /* 80 ('50'x).                               */
a = c2d('A')    /* "a" is an integer variable with a value of */
                /* 193 ('C1'x).                              */
```

Note

If the specified string is the null string, the result of this function is undefined.

C2X

```
C2X(string)
```

Where

string is any valid string expression.

Results

String

Function

The C2X function returns a character string containing the hexadecimal representation (unpacked) of *string*.

Examples

```
c = 'Mark'      /* "c" is a string variable.              */
say c2x(c)      /* You will see "D4819992".               */
say c2x('7D'x)  /* You will see "7D".                     */
```

DATE

```
DATE([date_format][,days_offset][,offset_direction])
```

Where

date_format is a single character string constant from the following list: D, E, H, J, M, N, O, P, S, T, U, W. The default value is T. You can use uppercase or lowercase letters. If specified directly, the character must be enclosed in single or double quotation marks. It can also be a named single-byte string constant.

days_offset is an integer expression. The default is 0.

offset_direction is an indicator of whether the *days_offset* is into the future ('+'), or into the past ('-'), relative to the current date. The format is a single byte string constant with the value '+' or '-'. The default is '+'.

Results

String

Function

The DATE function returns the current date in the following format depending on the *date_format* requested:

D (Days); returns number of days so far this year in the format: ddd (with leading zeros).

E (European); returns date in the format: dd/mm/yy.

H (Julian-packed with hex F padded); returns date in the packed format: yydddF.

J (Julian); returns date in the format: yyddd.

M (Month); returns full name of the current month in mixedcase. For example, May.

N (Normal); returns date in the format: dd mon yyyy. For example, 06 Feb 2002.

O (Ordered); returns date in the format: yy/mm/dd.

P (Packed WSim standard); returns date in the packed format: mmddyy.

S (Sorted); returns date in the format: yyyymmdd.

T (WSim standard-default); returns date in the format: mmddyy.

U (USA); returns date in the format: mm/dd/yy

W (Weekday); returns the day of the week in mixed case. For example, Friday.

If *days_offset* is not specified or is 0, the current date will be returned. If a value other than 0 is specified, the date *days_offset* days into the future or into the past (depending on *offset_direction*) will be returned.

Examples

```
say date('U') /* Display date in USA format, for example 02/18/02  */
say date()    /* Display date in standard WSim format,             */
              /* for example, 021802                               */
say date(,10) /* Display date 10 days from today in standard WSim  */
              /* format, that is, if today is 02/18/02 then        */
              /* 022802 would be returned.                         */
say date('N',10,'-')
              /* Display date 10 days ago, in Normal format.       */
              /* That is, if today is 02/18/02 then 08 Feb 2002    */
              /* is returned.                                      */
```

DAY

```
DAY()
```

Results

String

Function

The DAY function returns the number of the current day of the month (a 2-character string).

Examples

```
today = day()
if today = '01' then       /* First day of the month. */
 do                        /* Change all passwords.   */
&#38;#8942;
```

DBCSADD

```
DBCSADD(string)
```

Results

String

Function

The DBCSADD function adds SO/SI characters to a string expression. For more information on DBCS, see “Simulating DBCS terminals” on page 271.

Examples

```
                                 /* Before: '.A.B.C'           */
a = dbcsadd('42C142C242C3'x)     /* a = '0E42C142C242C30F'x    */
                                 /* After:  '<.A.B.C>'         */
```

DBCSADJ

```
DBCSADJ(string)
```

Results

String

Function

The DBCSADJ function deletes SI/SO character pairs from a string expression. For more information on DBCS, see “Simulating DBCS terminals” on page 271.

Examples

```
                                     /* Before: '<.A><.B.C>'       */
a = dbcsadj('0E42C10F0E42C242C30F'x) /* a = '0E42C142C242C30F'x    */
                                     /* After:  '<.A.B.C>'         */
```

DBCSDEL

```
DBCSDEL(string)
```

Results

String

Function

The DBCSDEL function deletes SO/SI characters from a string expression. For more information on DBCS, see “Simulating DBCS terminals” on page 271.

Examples

```
                                  /* Before: '0E42C142C242C30F'x  */
a = dbcsdel('<.A.B.C>')           /* a = '.A.B.C'                 */
                                  /* After:  '42C142C242C3'x      */
```

DBCS2SB

```
DBCS2SB(string)
```

Results

String

Function

The DBCS2SB function converts ward 42 (EBCDIC) DBCS string expression data to SBCS data. For more information on DBCS, see “Simulating DBCS terminals” on page 271.

Examples

```
a = dbcs2sb('<.A.B.C>')            /* a = 'C1C2C3'x or 'ABC'      */

a = dbcs2sb('0E42C142C242C30F'x)    /* a = 'C1C2C3'x or 'ABC'      */

a = dbcs2sb('42C142C242C3'x)        /* a = 'C1C2C3'x or 'ABC'      */
```

DELSTR

```
DELSTR(string,n[,length])
```

Where

string is a string expression.

n is the position of the first character in *string* to be deleted. *n* is an integer expression with a value from 1 to 32767.

length is the number of characters to be deleted. *length* is an integer expression with a value from 0 to 32767. This is optional.

Results

String

Function

The DELSTR function deletes the substring beginning at character *n* for length *length*. If *length* is not specified, the rest of the string is deleted (including character *n*). If *n* is greater than the length of *string*, the string is returned unchanged.

Examples

```
a = delstr('ABCD',3)              /* Assigns 'AB' to "a"          */
b = delstr('ABCDE',3,2)           /* Assigns 'ABE' to "b"         */
c = delstr('ABCDE',6)             /* Assigns 'ABCDE' to "c"       */
d = delstr('F5C3D9FF'x,2,2)       /* Assigns 'F5FF' to "d"        */
```

Note

The DELSTR function cannot be used in asynchronous conditions.

DELWORD

```
DELWORD(string,n[,length])
```

Where

string is a string expression.

n is an integer expression with a value from 1 to 32767.

length is an integer expression with a value from 1 to 32767. This is optional.

Results

String

Function

The DELWORD function returns *string* after deleting the substring that starts at the *n*th word and is of *length* blank-delimited words. If you omit *length*, or if *length* is greater than the number of words from *n* to the end of *string*, the function deletes the remaining words in *string* (including the *n*th word). If *n* is greater than the number of words in *string*, the function returns *string* unchanged. The string deleted includes any blanks following the final word involved but none of the blanks preceding the first word involved.

Examples

```
a = DELWORD('Now is the time',2,2)    /* Assigns 'Now time' to "a"   */
b = DELWORD('Now is the time',3)      /* Assigns 'Now is' to "b"     */
c = DELWORD('Now is the time',1,2)    /* Assigns 'the time' to "c"   */
```

Note: The DELWORD function cannot be used in asynchronous conditions.

DEVID

```
DEVID()
```

Results

String

Function

The DEVID function returns the name of the simulated device executing the STL program. This is the 1- to 8-character name coded on the DEV or LU definition statement.

Examples

```
say devid() 'has logged on'
```

D2C

```
D2C(number[,n])
```

Where

number is an integer expression.

n is an integer constant expression with a value of 1 to 4. This is optional.

Results

String

Function

The D2C function converts a decimal integer value into its equivalent hexadecimal string value. *n* specifies the number of characters to be returned. If *n* is specified as 1 and the hexadecimal string has a length greater than 1, then the rightmost byte is returned. If *n* is not specified, all significant bytes of the number will be returned; a leading zero will not be returned.

Examples

```
a =  D2C(15)               /* Assigns '0F'x to "a"            */
b =  D2C(32767)            /* Assigns '7FFF'x to "b"          */
c =  D2C(32767,1)          /* Assigns 'FF'x to "c"            */
```

Notes:

- The D2C function cannot be used in asynchronous conditions.
- The D2C function is equivalent to the HEX function and was added solely to accommodate those familiar with the REXX programming language.

E2D

```
E2D(numeric_string_expression[,length])
```

Where

numeric_string_expression is a string expression whose value is the EBCDIC representation of a decimal number.

length is an integer constant expression with a value from 1 to 10; the default is 10.

Results

Integer

Function

The E2D function converts the first *length* characters of *numeric_string_expression* from an EBCDIC representation of a number to a decimal number. Leading non-numeric characters are ignored. Conversion continues until the first *length* characters of *number_string_expression* are processed or until a non-numeric character is found after a numeric character.

If the value of *numeric_string_expression* is greater than 2147483647 or if the first *length* number of characters does not contain any numeric characters, E2D will return unpredictable results.

Examples

```
a = 'This is message number 100.'
count = e2d(substr(a,24,3),3)   /* "count" is an integer        */
                                /* variable.  After this        */
                                /* statement, "count" will have */
                                /* a value of 100.              */

dollars = e2d('$5',2)           /* "dollars" now has a value    */
                                /* of five. The leading dollar  */
                                /* sign ($) is ignored.         */

dollars = e2d('$5.25',5)        /* "dollars" will be assigned a */
                                /* value of five.  The decimal  */
                                /* point (.) terminates         */
                                /* conversion, even though a    */
                                /* length of five is specified. */
```

FM

```
FM()
```

Results

String

Function

The FM function simulates the action of the Field Mark key on a display device. This function is valid only on TYPE statements and should be used only when simulating 3270 terminals; WSim will ignore FM if you try to simulate terminals not listed here.

Examples

```
cursor(10,20)  /* Position cursor on row 10, column 20. */
type fm()      /* Mark this field.                      */
```

Note

The function of the Field Mark key can also be simulated using the FM statement.

HEX

```
HEX(number[,length])
```

Where

number is an integer expression.

length is an integer constant expression with a value of 1 to 4.

Results

String

Function

The HEX function converts a decimal integer value into its equivalent hexadecimal string value. *length* specifies the number of characters to be returned. If *length* is not specified, all significant bytes of the *number* will be returned; a leading zero will not be returned.

If a *length* of 1 is specified and the hexadecimal string has a length greater than 1, then the right-most byte is returned.

Examples

```
a = 15          /* "a" is an integer variable with a value of 15. */
hex_a = hex(a)  /* "hex_a" is a string variable with a value of   */
                /*  '0F'x.                                        */
hex_b = hex(4,2) /* "hex_b" is a string variable with a value of  */
                /* '0004'x.                                       */
```

ID

```
ID([length])
```

Where

length is an integer constant expression with a value from 1 to 8.

Results

String

Function

The ID function returns all or part of the name of a terminal. If *length* is not specified, an 8-character string will be returned. If *length* is longer than the terminal's name, the returned string will be padded with blanks on the right. If *length* is shorter than the terminal's name, the name will be truncated on the right.

Examples

```
/* This terminal's name is LUXYZ.                        */

say id()     /* Result: 'LUXYZ   '.                      */
say id(5)    /* Result: 'LUXYZ'.                         */
say id(6)    /* Result: 'LUXYZ ' (notice blank padding). */
say id(3)    /* Result: 'LUX' (notice truncated name).   */
```

INDEX

```
INDEX(source,target)
```

Where

source is a string expression.

target is a string expression.

Results

Integer

Function

The INDEX function identifies the position within a *source* string of a *target* string. If the source string does not contain the target string, a value of 0 is returned.

Examples

```
a = index('Hello','H')              /* "a" will be set to 1.   */
a = index('Hello','t')              /* "a" will be set to 0.   */

if index(screen,'Welcome') > 0 then ...
                                    /* True if WELCOME is on   */
                                    /* the screen.             */

onin index(data,'***') > 0 then ...   /* True if *** is in the   */
                                    /* data stream.            */

message = substr(screen,index(screen,'ERROR'),80)
                                    /* Set variable "message"  */
                                    /* to the 80 characters of */
                                    /* screen data beginning   */
                                    /* with ERROR.             */
```

Note

Be careful when using INDEX to find a string on a screen with spaces in it; they may actually be nulls on the screen.

INSERT

```
INSERT(new,target[,[n][,[length][,pad]]])
```

Where

new is the string expression to be inserted.

target is the string expression where *new* is to be inserted.

n is the position in *target* after which *new* is to be inserted. *n* is an integer expression with a value from 0 to 32766. This is optional.

length is the number of characters to which *new* will be padded as it is inserted. *length* is an integer expression with a value from 0 to 32767. This is optional.

pad is the 1-character string constant or 2-digit hexadecimal constant used for padding.

Results

String

Function

The INSERT function inserts the string *new*, padded to length *length*, into the string *target* after character *n*. If *n* is greater than the length of the target string, padding is added between the two strings. The default value for *n* is 0, which means insert before the beginning of the string. The default value for *length* is the length of *new*. The default *pad* character is blank (X'40').

Examples

```
a = insert(' ','ABCDEF',3)        /* Assigns 'ABC DEF' to "a"     */
b = insert('123','ABC',5,6)       /* Assigns 'ABC  123   ' to "b" */
c = insert('123','ABC',5,6,'+')   /* Assigns 'ABC++123+++' to "c" */
d = insert('123','ABC')           /* Assigns '123ABC' to "d"      */
e = insert('123','ABC',,5,'-')    /* Assigns '123--ABC' to "e"    */
f = insert('1234'x,'ABCD'x,1,3)   /* Assigns 'AB123440CD'x to "f" */
g = insert('12'x,'AB'x,1,3,'FF'x) /* Assigns 'AB12FFFF'x to "g"   */
```

Note

The INSERT function cannot be used in asynchronous conditions.

LASTPOS

LASTPOS(*needle***,***haystack***[,***start***])**

Where

needle is a string expression.

haystack is a string expression.

start is an integer expression with a value from 1 to 32767. This is optional.

Results

Integer

Function

The LASTPOS function returns the position of the last occurrence of one string, *needle*, in another, *haystack*. (See also the POS function.) If *needle* is the null string or is not found then the function returns a 0. By default the search starts at the last character of *haystack* and scans backward. You can override this by specifying *start*, the point at which the backward scan starts. The default for *start* is set equal to LENGTH(*haystack*) if the value specified is larger than LENGTH(*haystack*) or if it is omitted.

Examples

```
a =  LASTPOS(' ','ab cd ef gh')           /* Assigns 9 to "a"        */
b =  LASTPOS(' ','ab cd ef gh',7)         /* Assigns 6 to "b"        */
c =  LASTPOS('25','12352425352522',10)    /* Assigns 7 to "c"        */
```

Note: The LASTPOS function cannot be used in asynchronous conditions.

LASTVERB

```
LASTVERB()
```

Results

String

Function

The LASTVERB function returns the name of the last CPI-C statement (verb) that was issued for this CPI-C STL program.

Examples

```
say 'The name of the last CPI-C statement is: 'lastverb()'.'
```

Note: This function is intended for use with CPI-C simulations; results will be the null string if used in other simulations.

LEFT

```
LEFT(string,length[,pad])
```

Where

string is a string expression.

length is the final length of the string. *length* is an integer expression with a value from 0 to 32767.

pad is the 1-character string constant or 2-digit hexadecimal constant used for padding. This is optional.

Results

String

Function

The LEFT function returns a string of length *length* containing the leftmost *length* characters of *string*. The string is padded with *pad* characters (or truncated) on the right as needed. The default *pad* character is a blank (X'40').

Examples

```
a = left('ABC D',8)             /* Assigns 'ABC D   ' to "a"    */
b = left('ABC D',8,'.')         /* Assigns 'ABC D...' to "b"    */
c = left('ABCDEF',3)            /* Assigns 'ABC' to "c"         */
d = left('F5C4D9'x,5)           /* Assigns 'F5C4D94040'x to "d" */
e = left('F5C4D9'x,5,'AA'x)     /* Assigns 'F5C4D9AAAA'x to "e" */
```

Note

The LEFT function cannot be used in asynchronous conditions.

LENGTH

```
LENGTH({integer_expression})
       {string}
```

Where

integer_expression is an integer expression.

string is a string expression.

Results

Integer

Function

The LENGTH function returns the length of the specified string expression or integer expression.

Examples

```
a = length('Now is the time')  /* "a" is assigned a value of 15. */

b = 'Now is the time'
a = length(b)                  /* "a" is assigned a value of 15. */

a = length(255)                /* "a" is assigned a value of 3.  */
```

Note

Special instances of the LENGTH function are permitted on SUBSTR and ATTR3270 function references. See “ATTR3270” on page 449 and “SUBSTR” on page 488 for more information.

LUID

```
LUID()
```

Results

String

Function

The LUID function returns the name of the simulated LU executing the STL program. This is the 1- to 8-character name coded on the LU network definition statement.

Examples

```
say 'My LU name is' luid()
```

MONTH

```
MONTH()
```

Results

String

Function

The MONTH function returns a 2-character string that is the number of the current month.

Examples

```
this_month = month()
today = day()
if this_month = '04' & ,    /* It must be April.               */
   today = '15' then        /* It is the 15th day of the month. */
   do                       /* Calculate income taxes.          */

   &#38;#8942;
```

MSGTXTID

```
MSGTXTID()
```

Results

String

Function

The MSGTXTID function returns the name of the current STL procedure. This is the 1- to 8-character name coded on the MSGTXT statement.

Examples

```
say 'Executing procedure: 'msgtxtid()
```

NETID

```
NETID()
```

Results

String

Function

The NETID function returns the name of the simulated terminal's network. This is the 1- to 8-character name coded on the NTWRK network definition statement.

Examples

```
say 'My network is 'netid()
```

NL

```
NL()
```

Results

String

Function

The NL function simulates the function of the New Line key on a 3270 (LU Type 2) terminal. The cursor is set to the first unprotected character location of the next line on the screen. If no unprotected fields exist, the cursor is set to character location 0 (top left corner of screen or partition). If the screen contains no fields, the cursor is set to the first position on the next line.

Examples

```
type 'This is the first line'nl()'And this is the second line'
```

Note

This function is only valid on a TYPE statement.

NUMCOLS

```
NUMCOLS()
```

Results

Integer

Function

The NUMCOLS function returns the number of columns on a simulated terminal's screen.

Examples

```
a = numcols()    /* Set variable "a" to number of columns on screen. */
```

Note

Use this function for display terminals only. It returns unpredictable results for nondisplay terminals.

NUMROWS

```
NUMROWS()
```

Results

Integer

Function

The NUMROWS function returns the number of rows on a simulated terminal's screen.

Examples

```
a = numrows()    /* Set variable "a" to number of rows on screen. */
```

Note

Use this function for display terminals only. It returns unpredictable results for nondisplay terminals.

OVERLAY

```
OVERLAY(new,target[,[n][,[length][,pad]]])
```

Where

new is a string expression.

target is a string expression.

n is an integer expression. This is optional.

length is an integer expression. This is optional.

pad is a 1-character string constant or 2-digit hexadecimal constant used for padding. This is optional.

Results

String

Function

The OVERLAY function returns the string *target*, which, starting at the *n*th character, is overlaid with the string *new*, padded or truncated to the value of *length*. Overlays may also extend beyond the end of the optional target string. If *n* is greater than the length of the target string, padding is added before the new string. The default for *n* is 1. If *length* is not specified, it defaults to a value equal to the length of the string expression *new*. The default value for *pad* is a blank (X'40').

Examples

```
a = OVERLAY(' ','abcdef',3)          /* Assigns 'ab def' to  "a"      */
b = OVERLAY('.','abcdef',3,2)        /* Assigns 'ab. ef'  to "b"      */
c = OVERLAY('qq','abcd')             /* Assigns 'qqcd'  to "c"        */
d = OVERLAY('qq','abcd',4)           /* Assigns 'abcqq'  to "d"       */
e = OVERLAY('123','abc',5,6,'+')     /* Assigns 'abc+123+++'  to "e"  */
```

Note: The OVERLAY function cannot be used in asynchronous conditions.

PATHID

```
PATHID()
```

Results

String

Function

The PATHID function returns the name of the PATH statement currently being executed. This is the 1- to 8-character name coded on the PATH statement.

Examples

```
say 'The current path is' pathid()
```

Note: The PATHID function can be used in asynchronous conditions.

POS

```
POS(needle,haystack[,start])
```

Where

needle is a string expression.

haystack is a string expression.

start is an integer expression with a value from 1 to 32767. This is optional.

Results

Integer

Function

The POS function returns the position of one string, *needle*, in another, *haystack*. (See also the LASTPOS function.) If *needle* is the null string or is not found or if *start* is greater than the length of *haystack* then the function returns a 0. By default the search starts at the first character of *haystack*.

Examples

```
a =  POS(' ','ab cd ef gh')             /* Assigns 3 to "a"        */
b =  POS(' ','ab cd ef gh',7)           /* Assigns 9 to "b"        */
c =  POS('25','12352425352522',10)      /* Assigns 11 to "c"       */
```

Note: The POS function cannot be used in asynchronous conditions.

POSTED

```
POSTED(event_name)
```

Where

event_name is a string expression that specifies the name of an event. If it is a string constant expression, *event_name* must be 1 to 8 alphanumeric characters and enclosed in single or double quotation marks. (Strings containing hexadecimal constants are exempt from this restriction.) If you specify a nonconstant string expression, the first 8 characters of the string are used. If the string is shorter than 8 characters, the available characters are used. To satisfy conditions using event names, the first 8 characters must match exactly. If you specify a string variable, it cannot be the name of one of the reserved variables (for example, BUFFER).

Results

Bit

Function

The POSTED function returns a value of true (ON) if the specified event has been posted. Otherwise, it returns a value of false (OFF).

Examples

```
if posted('MYEVENT') then          /* Test posting of MYEVENT. */
   post_flag = posted('MYEVENT')    /* Set "post_flag" ON if    */
                                    /* MYEVENT is posted.       */
```

PULL

```
PULL([queue_name])
```

Where

queue_name is a string expression consisting of 1 to 8 alphanumeric characters. This is optional.

Results

String

Function

The PULL function returns the next text/string item from *queue_name*.

If the specified *queue_name* is a nonconstant expression, the first 8 characters of the string are used. If the string is shorter than 8 characters, the available characters are used. When specifying *queue_name* within the PULL function, the specified name must exactly match the name initially used to QUEUE or PUSH the text/string item.

If you specify a string variable, it cannot be the name of one of the STL reserved variables (for example, BUFFER). If not specified, *queue_name* defaults to a unique value assigned to each device.

Examples

```
user_queue  = 'UseridQ'         /* Assigns 'UseridQ' to "user_queue"  */
queue '111111' TO user_queue    /* Places '111111' on queue 'UseridQ' */
queue '222222' TO 'UseridQ'     /* Places '222222' on queue 'UseridQ' */
queue 'ABCD'                    /* Places 'ABCD' on unique device Q   */
a = PULL('UseridQ')             /* Assigns '111111' to "a"            */
b = PULL(user_queue)            /* Assigns '222222' to "b"            */
c = PULL()                      /* Assigns 'ABCD' to "c"              */
```

Notes:

- The PULL function can be used in asynchronous conditions.
- The named queue structure and text/string data items are allocated dynamically by WSim and deleted as the queue is emptied.

QUEUED

```
QUEUED([queue_name])
```

Where

queue_name is a string expression consisting of 1 to 8 alphanumeric characters. This is optional.

Results

Integer

Function

The QUEUED function returns the number of text/string items on *queue_name*.

If the specified *queue_name* is a nonconstant expression, the first 8 characters of the string are used. If the string is shorter than 8 characters, the available characters are used. When specifying *queue_name* within the QUEUED function, the specified name must exactly match the name initially used to QUEUE or PUSH the text/string item. If you specify a string variable, it cannot be the name of one of the STL reserved variables (for example, BUFFER). If not specified, *queue_name* defaults to a unique value assigned to each device.

Examples

```
Q_name = 'I'||devid()                /* This example will place   */
Do i = 1 to 5                        /* five entries on a queue.  */
   Queue char(i) to Q_name           /* Then the queue will be    */
End                                  /* read until it is empty.   */
Do while queued(Q_name) > 0
   Say 'Queue item' pull(Q_name)
End
```

Notes:
- The QUEUED function cannot be used in asynchronous conditions.
- The named queue structure and text/string data items are allocated dynamically by WSim and deleted as the queue is emptied.

RANDOM

```
RANDOM({'RN',rn_number})
       {low,high}
```

Where

rn_number is an integer constant expression with a value from 0 to 255.

low is an integer expression with a value from 0 to 2147483646.

high is an integer expression with a value from 1 to 2147483647. The value for *high* must be greater than the value for *low*.

Results

Integer

Function

The RANDOM function returns a random number.

If (*low*,*high*) is specified, the number returned will be in the range of numbers between *low* and *high*.

If ("RN",*rn_number*) is specified, the number returned will be taken from the range of numbers specified on the RN network definition statement with label *rn_number*.

Examples

```
a = random(1,200)   /* "a" is assigned a random number between 1 */
                    /* and 200.                                  */

a = random("RN",3)  /* "a" is assigned a random number in the    */
                    /* range specified by RN statement 3.        */
```

REPEAT

```
REPEAT(character,count)
```

Where

character is a single character string constant.

count is an integer expression with a value from 1 to 32767.

Results

String

Function

The REPEAT function returns a string consisting of *count* occurrences of *character*.

Examples

```
say repeat('A',5)      /* Result will be "AAAAA".       */

say repeat('FF'x,5)    /* Result will be "FFFFFFFFFF"x. */

a = 50
say repeat('X',a)      /* X is repeated 50 times.       */
```

REVERSE

```
REVERSE(string)
```

Where

string is a string expression.

Results

String

Function

The REVERSE function returns a string with the order of the characters reversed.

Examples

```
a =  REVERSE('abcdefghi')          /* Assigns 'ihgfedcba' to "a"     */
b =  REVERSE('517C'x)              /* Assigns '7C51'x to "b"         */
```

Note: The REVERSE function cannot be used in asynchronous conditions.

RIGHT

```
RIGHT(string,length[,pad])
```

Where

string is a string expression.

length is the final length of the string. *length* is an integer expression with a value from 0 to 32767.

pad is the 1-character string constant or 2-digit hexadecimal constant used for padding. This is optional.

Results

String

Function

The RIGHT function returns a string of length *length* containing the rightmost *length* characters of *string*. The string is padded with *pad* characters (or truncated) on the left as needed. The default *pad* character is a blank (X'40').

Examples

```
a = right('ABC  D',8)          /* Assigns '  ABC  D' to "a"    */
b = right('ABC DEF',5)         /* Assigns 'C DEF' to "b"       */
c = right('12',5,'0')          /* Assigns '00012' to "c"       */
d = right('F5C4D9'x,5)         /* Assigns '4040F5C4D9'x to "d" */
e = right('F5C4D9'x,5,'AA'x)   /* Assigns 'AAAAF5C4D9'x to "e" */
```

Note

The RIGHT function cannot be used in asynchronous conditions.

RNUM

```
RNUM({'RN',rn_number[,length]})
     {low,high[,length]}
```

Where

rn_number is an integer constant expression with a value from 0 to 255.

length is an integer constant expression with a value from 1 to 10.

low is an integer expression with a value from 0 to 2147483646.

high is an integer expression with a value from 1 to 2147483647. The value for *high* must be greater than the value for *low*.

Results

String

Function

The RNUM function returns the string (EBCDIC) representation of a random number. If (*low,high*) is specified, the number returned will be in the range of numbers between *low* and *high*.

If ('RN',*rn_number*) is specified, the number returned will be taken from the range of numbers specified on the RN network definition statement with label *rn_number*.

The length of the returned string is specified by *length*. If *length* is not coded, a default length of ten will be used. The random number generated by this function will be padded with leading zeros if it is shorter than *length*.

Examples

```
a = rnum(1,200,3)    /* Generate a 3-digit random number (string) */
                     /* between 1 and 200 and assign it to string */
                     /* variable "a".                             */

a = rnum('RN',5,4)   /* Generate a 4-digit random number (string) */
                     /* in the range specified on RN statement 5  */
                     /* and assign it to string variable "a".     */
```

ROWCOL

ROWCOL(*row*,*column***)**

Where

row is an integer expression. If an integer constant expression, *row* must be from 1 to 255.

column is an integer expression. If an integer constant expression, *column* must be from 1 to 255.

Results

Integer

Function

The ROWCOL function is valid only when coded as the offset on a SUBSTR or ATTR3270 function. It returns the screen index of the position referenced by *row* and *column*.

When coding ROWCOL as the offset on a SUBSTR function, the source of the substring must be either BUFFER or SCREEN.

Examples

```
a = substr(screen,rowcol(1,1),5)  /* Assigns to string variable  */
                                  /* "a" the 5-character string  */
                                  /* found at row 1, column 1    */
                                  /* on the screen.              */

x = 20
y = 40
a = substr(screen,rowcol(x,y+5),10)
                                  /* Assigns to string variable  */
                                  /* "a" the 10-character string */
                                  /* found at row 20, column 45  */
                                  /* on the screen.              */

if substr(screen,rowcol(x,y+5),5) = 'Hello' then
                                  /* Tests the screen location   */
                                  /* row 20, column 45 for a     */
                                  /* value of Hello.             */
```

SB2DBCS

SB2DBCS(*string***)**

Results

String

Function

The SB2DBCS function converts SBCS string expression data to ward 42 (EBCDIC) DBCS data. For more information on DBCS, see “Simulating DBCS terminals” on page 271.

Examples

```
a = sb2dbcs('ABC')                /* a = '.A.B.C'              */
                                  /* a = '42C142C242C3'x       */
```

SB2MDBCS

```
SB2MDBCS(string)
```

Results

String

Function

The SB2MDBCS function converts SBCS string expression data to ward 42 (EBCDIC) DBCS data and wraps SO/SI characters around the DBCS data to make it a mixed string. For more information on DBCS, see “Simulating DBCS terminals” on page 271.

Examples

```
a = sb2mdbcs('ABC')               /* a = '<.A.B.C>'            */
                                  /* a = '0E42C142C242C30F'x   */

type sb2mdbcs('HELLO THERE IN WARD 42 DBCS DATA')
```

SESSNO

```
SESSNO()
```

Results

String

Function

The SESSNO function returns the session number of the simulated LU executing the STL program. The format of this session number is *-nnnnn*, where *nnnnn* is the number of the session, zero-padded on the left.

Examples

```
say 'My session number is 'sessno()
```

Note

This function returns the null string for simulated terminals without session numbers.

SPACE

SPACE(*string***,***n***[,***pad***])**

Where

string is a string expression.

n is an integer expression with a value from 1 to 32767. This is optional.

pad is a 1-character string constant or 2-digit hexadecimal constant used for padding. This is optional.

Results

String

Function

The SPACE function returns the blank-delimited words in *string* with *n pad* characters between each word. If *n* is 0, all blanks are removed. Leading and trailing blanks are always removed. The default for *n* is 1, and the default value for *pad* is a blank (X'40').

Examples

```
a = SPACE('aa   bb    cc',1,'+')     /* Assigns 'aa+bb+cc' to "a"      */
b = SPACE(' ab  cd  ',2)             /* Assigns 'ab  cd' to "b"        */
c = SPACE('  a  a  ',9,'#')          /* Assigns 'a#########a' to "c"   */
```

Note: The SPACE function cannot be used in asynchronous conditions.

STRIP

STRIP(*string***[,[***option***][,***char***]])**

Where

string is a string expression.

option is a single character constant with one of the following values:

B or b removes both leading and trailing characters from *string*

L or l removes leading characters from *string*

T or t removes trailing characters from *string* *char* is a single character constant. This is optional.

Results

String

Function

The STRIP function returns *string* with leading or trailing characters or both removed, based on the *option* you specify. The third argument, *char*, specifies the character to be removed. The default for *option* is B and the default for *char* is a blank (X'40').

Examples

```
a = STRIP('   abc da   ')             /* Assigns 'abc da' to "a"       */
b = STRIP('aaaabcdefaaa','T','a' )    /* Assigns 'aaaabcdef '  to "b"  */
c = STRIP('aaaabcdefaaa','b','a' )    /* Assigns 'bcdef' to "c"        */
```

Note: The STRIP function cannot be used in asynchronous conditions.

SUBSTR

```
SUBSTR(source,starting_position[,length])
```

Where

source is a string expression.

starting_position is an integer expression with a value from 1 to 32767.

length is an integer expression with a value from 1 to 32767.

Results

String

Function

The SUBSTR function returns the substring of *source* that begins at the *starting_position* and is of length *length*. If you are simulating a 3270 terminal, you can use the options available on the ATTR3270 function that identify screen location as your *starting_position*. If *length* is omitted, the default is the rest of the string.

If the *length* is greater than the length of the *source*, then the string is not padded with blanks to *length*.

Examples

```
a = substr('ABCDEFG',2,3)        /* Assigns 'BCD' to variable "a".   */

b = 'ABCDEFG'
a = substr(b,3)                  /* Assigns 'CDEFG' to variable "a". */

c = 4
```

```
a = substr(b,c)                /* Assigns 'DEFG' to variable "a".  */

d = 2
if substr(b,c,d) = 'DE' then ... /* Condition will be satisfied.     */

a = substr(b,4,8)              /* Assigns 'DEFG' to variable "a".  */
```

Restrictions on use of the SUBSTR function in asynchronous conditions

The SUBSTR function returns a string expression that can generally be used wherever string expressions are valid. There are, however, some restrictions on the use of the SUBSTR function within asynchronous conditions.

1. The *source* expression must be a string variable or STL reserved variable; it cannot be any other type of string expression such as a string constant, string concatenation, or function that returns a string.
2. Only integer constants and integer constant expressions can be used as *starting_position* arguments.
3. Only integer constants, integer constant expressions, or single integer variables can be used as *length* arguments. You cannot use expressions that include variables.

The following are some examples of invalid and valid SUBSTR functions used on asynchronous conditions.

```
/* Invalid (Variable position specified). */

name_position = 43
onin substr(screen,name_position,6) = 'MYNAME' then ...

/* Valid (Constant position specified).   */

onin substr(screen,43,6) = 'MYNAME' then ...

/* Invalid (Variable expression specified for length). */

full_name = 'John Doe'
blank_spot = index(full_name,' ')
onin substr(screen,150,blank_spot-1) = substr(full_name,1,blank_spot-1) then ...

/* Valid (Simple variable specified for length). */

full_name = 'John Doe'
first_name_len = index(full_name,' ') - 1
onin substr(screen,150,first_name_len) = substr(full_name,1,first_name_len) then ...
```

SUBWORD

SUBWORD(*string*,*n*[,*length*]**)**

Where

string is a string expression.

n is an integer expression with a value from 1 to 32767.

length is an integer expression with a value from 1 to 32767. This is optional.

Results

String

Function

The SUBWORD function returns the substring of *string* that starts at the *n*th word and is up to *length* blank-delimited words. If you omit *length*, it defaults to the number of remaining words in *string*. The returned string never has leading or trailing blanks, but includes all blanks between the selected words.

Examples

```
a = SUBWORD('Now is the time',2,2)     /* Assigns 'is the' to "a"      */
b = SUBWORD('Now is the time',3)       /* Assigns 'the time' to "b"    */
c = SUBWORD('Now is the time',5)       /* Assigns ''  to "c"           */
```

Note: The SUBWORD function cannot be used in asynchronous conditions.

TAB

```
TAB()
```

Results

String

Function

The TAB function simulates the function of the TAB key on a 3270 (LU Type 2) terminal. The cursor is set to the first character location of the next unprotected field on the screen.

Examples

```
type 'This is the first field'tab()'and this is the second field'
```

Notes

- The TAB function is only valid on a TYPE statement. It will be ignored for terminal types other than 3270 (LU2) terminals.
- The function of the TAB key can also be simulated using the TAB statement.

TCPIPID

```
TCPIPID()
```

Results

String

Function

The TCPIPID function returns the name of the TCP/IP connection associated with a simulated device. This is the 1- to 8-character name coded on the TCPIP network definition statement.

Examples

```
say 'My TCP/IP connection is 'tcpipid()
```

Note

If the simulated device is not associated with a TCPIP network definition statement, this function returns a null value.

TOD

```
TOD([length])
```

Where

length is an integer constant expression with a value from 1 to 8.

Results

String

Function

The TOD function returns a string representing the current time of day in the form *HHMMSSTH* (hours, minutes, seconds, tenths, and hundredths of seconds). If *length* is not specified, an 8-character string will be returned. If *length* is specified and it is less than 8, the left-most portion of the string will be returned.

Examples

```
a = tod(4)        /* Get only hours and minutes. */
say 'The current time is 'substr(a,3,2)' minutes past 'substr(a,1,2)
```

TPID

```
TPID()
```

Results

String

Function

The TPID function returns the name of the simulated transaction program executing the STL program.

Examples

```
say 'The transaction program name is: 'tpid()'.'
```

Note: This function is intended for use with CPI-C simulations; results will be the null string if used in other simulations.

TPINSTNO

```
TPINSTNO()
```

Results

String

Function

The TPINSTNO function returns the 5-digit instance number of the simulated transaction program executing the STL program. The format for this number is *-nnnnn*, where *nnnnn* is the instance number of the transaction program, zero-padded on the left. The very first instance is *-00001*.

Examples

```
say 'The transaction program instance number is: 'tpinstno()'.'
```

Note: This function is intended for use with CPI-C simulations; results will be the null string if used in other simulations.

TRANSLATE

TRANSLATE(*string***[,[***tableo***][,[***tablei***][,***pad***]]])**

Where

string is the string expression to be translated.

tableo is the output table string expression. *tableo* defaults to the null string and is padded with pad as necessary.

tablei is the input translation table string expression. *tablei* defaults to the hexadecimal range of characters X'00' through X'FF.

pad is the 1-character string constant or 2-digit hexadecimal constant used for padding the output table when necessary. The default pad is a blank.

Results

String

Function

The TRANSLATE function returns a string of the same length as the input string with characters translated according to the tables specified. You can also use this function to reorder the characters in the input string.

TRANSLATE searches *tablei* for each character in *string*. If the character is found, then the corresponding character in *tableo* is used in the result string; if there are duplicates in *tablei*, the first (leftmost) occurrence is used. If the character is not found, the original character in *string* is used. The result string is always the same length as *string*.

The tables can be of any length. If you omit *tableo*, *tablei*, and *pad*, WSim simply translates *string* to uppercase (that is, lowercase a-z to uppercase A-Z). But if you include *pad* without specifying *tablei* or *tableo*,WSim translates the entire string to pad characters.

You can translate from one code to another (such as EBCDIC to ASCII or ASCII to EBCDIC) by specifying a 256-byte standard translation table for *tableo* and omitting *tablei* and *pad*.

Examples

```
Statement executed                    Results
TRANSLATE('abcdef')                   'ABCDEF'
TRANSLATE('abbc','&','b')             'a&&c'
TRANSLATE('abcdef','12','ec')         'ab2d1f'
TRANSLATE('abcdef','12','abcd','.')   '12..ef'
TRANSLATE('APQRV',,'PR')              'A Q V'
TRANSLATE('4123','abcd','1234')       'dabc'
```

Note: The last example shows how you can use the TRANSLATE function to reorder the characters in a string. In the example, the last character of any four-character string specified as the second argument would be moved to the beginning of the string.

UTBL

```
UTBL(utbl_name,{index_number})
               {'R'}
               {'Rn'}
```

Where

utbl_name is the name of a user table. This name may be specified in one of the following formats:

- *msgutbl_name* is the name coded on an MSGUTBL statement in this or another STL program. This can also be the name coded on a message generation MSGUTBL statement.
- *utbl_number* is the number coded as a label on a UTBL statement within a network definition. The number must be from 0 to 255.

index_number is an integer expression.

n is a number from 0 to 255 and specifies a UDIST network definition statement.

Results

String

Function

The UTBL function returns an entry from the user table specified by *utbl_name*.

The second argument determines which entry is to be returned.

If *index_number* is used, the integer expression is evaluated and its value is used to index into the user table. User table entries begin with an index of 0. If *index_number* is greater than the maximum index for the user table, a null string is returned.

If R is used, a randomly selected entry will be returned.

If R*n* is used, an entry is chosen from the table randomly using the distribution defined by the UDIST statement referenced by *n*.

Examples

```
mynames: msgutbl
   "Mary"     /* This is entry number 0. */
   "Joe"      /* This is entry number 1. */
   "John"     /* This is entry number 2. */
   "Sue"      /* This is entry number 3. */
endutbl
&#38;#8942;
a = 1
b = 2
name = utbl(mynames,0)      /* "Mary" is returned.         */
name = utbl(mynames,a+b)    /* "Sue" is returned.          */
name = utbl(mynames,"R")    /* A random entry is returned. */
name = utbl(mynames,"R3")   /* A random entry with the     */
                            /* distribution defined by     */
                            /* UDIST number 3 is returned. */
```

UTBLMAX

```
UTBLMAX(user_table)
```

Where

user_table is one of the following:
- The name of an STL MSGUTBL defined in this or another STL source data set.
- A number (integer constant expression) from 0 to 255 corresponding to the number coded on a UTBL network definition statement.
- The name coded on a MSGUTBL definition statement.

Results

Integer

Function

The UTBLMAX function returns the index of the last entry in the specified user table.

Examples

```
do i = 0 to utblmax(myutbl)  /* Loop through MYUTBL. */
&#38;#8942;
end
```

UTBLSCAN

```
UTBLSCAN(source,utbl_name[,integer_variable])
```

Where

source is the string expression for which the specified UTBL or MSGUTBL is to be searched.

utbl_name is the name of a user table. This name can be specified in one of the following formats:
- *msgutbl_name* is the name coded on an MSGUTBL statement in this or another STL program. This can also be the name coded on a message generation MSGUTBL statement.
- *utbl_number* is the number coded as a label on a UTBL statement in a network definition. The number must be from 0 to 255.

integer_variable is the name of an integer variable.

Results

Bit (and optionally, integer)

Function

The UTBLSCAN function scans the user table specified by *utbl_name* for an entry that matches *source*.

The UTBLSCAN function will return an ON (or “true”) setting if an entry in the UTBL is found that matches the source string. If a match is not found, the function will return an OFF (or “false”) setting. If you specify an integer variable as a third argument and if a match is found, the UTBLSCAN function will also assign the matching UTBL entry number to the specified integer variable. If a match is not found, the specified integer variable's value will not be changed.

Examples

```
sample: msgutbl
        'Hello'
        'Goodbye'
        endutbl
&#38;#8942;
if utblscan('Hello',sample) = on then ...  /* This condition will  */
                                           /* be met.             */
```

```
greeting = 'Good Morning'
greeting_found = utblscan(greeting,sample) /* UTBLSCAN function    */
                                           /* can be used in bit   */
                                           /* assignment.  Here,   */
                                           /* "greeting_found" is  */
                                           /* set to OFF.          */
```

VTAMAPID

```
VTAMAPID()
```

Results

String

Function

The VTAMAPID function returns the name of the VTAMAPPL associated with a simulated LU. This is the 1- to 8-character name coded on the VTAMAPPL network definition statement.

Examples

```
say 'My VTAMAPPL is 'vtamapid()'.'
```

Note

If the simulated LU is not associated with a VTAMAPPL, this function will return the null string.

WORD

WORD(*string***,***n***)**

Where

string is a string expression.

n is an integer expression with a value from 1 to 32767.

Results

String

Function

The WORD function returns the *n*th blank-delimited word in *string* or returns the null string if fewer than *n* words are in *string*. This function is exactly equivalent to SUBWORD(*string*,*n*,1).

Examples

```
a = WORD('Now is       the time',3)     /* Assigns 'the' to "a"     */
b = WORD('Now is the time',5)           /* Assigns '' to "b"        */
```

Note: The WORD function cannot be used in asynchronous conditions.

WORDINDEX

```
WORDINDEX(string,n)
```

Where

string is a string expression.

n is an integer expression with a value from 1 to 32767.

Results

Integer

Function

The WORDINDEX function returns the position of the first character in the *n*th blank-delimited word in *string* or returns 0 if fewer than *n* words are in *string*.

Examples

```
a = WORDINDEX('Now is the time',2)        /* Assigns 5 to "a"       */
b = WORDINDEX('Now is the time',3)        /* Assigns 8 to "b"       */
c = WORDINDEX('Now is the time',5)        /* Assigns 0 to "c"       */
```

Note: The WORDINDEX function cannot be used in asynchronous conditions.

WORDPOS

```
WORDPOS(phrase,string[,start])
```

Where

phrase is a string expression.

string is a string expression.

start is an integer expression with a value from 1 to 32767. This is optional.

Results

Integer

Function

The WORDPOS function returns the word number of the first word of *phrase* found in *string* or returns 0 if *phrase* contains no words or if *phrase* is not found in *string*. Multiple blanks between words in either *phrase* or *string* are treated as a single blank for the comparison, but otherwise the words must match exactly. By default the search starts at the first word in *string*. You can override this by specifying *start*, the word at which to start the search.

Examples

```
a = WORDPOS(' is what','It is      what it is')  /* Assigns 2 to "a"    */
b = WORDPOS('it     is','It is what it is')      /* Assigns 4 to "b"    */
c = WORDPOS(' is ','It is what it is',3)         /* Assigns 5 to "b"    */
d = WORDPOS(' Is ','It is what it is')           /* Assigns 0 to "d"    */
```

Note: The WORDPOS function cannot be used in asynchronous conditions.

WORDS

```
WORDS(string)
```

Where

string is a string expression.

Results

Integer

Function

The WORDS function returns the number of blank-delimited words in *string*.

Examples

```
a = WORDS('Now is the time ')             /* Assigns 4 to "a"      */
b = WORDS('a b d  cd  e    f')            /* Assigns 6 to "b"      */
```

Note: The WORDS function cannot be used in asynchronous conditions.

X2B

```
X2B(hexstring)
```

Where

hexstring is a string expression containing hexadecimal characters.

Results

String

Function

The X2B function returns a string, in character format, that represents *hexstring* converted to a binary string. The *hexstring* is a string of hexadecimal characters. It can be of any length. Each hexadecimal character is converted to a string of four binary digits. You can optionally include blanks in *hexstring* (at byte boundaries only, not leading or trailing) to aid readability; they are ignored. The returned string has a length that is a multiple of four, and does not include any blanks. If *hexstring* is null, the function returns a null string.

Examples

```
a = X2B('F1C2')                  /* Assigns '1111000111000010' to "a"  */
b = X2B('5A  A5')                /* Assigns '0101101010100101' to "b"  */
```

Note: The X2B function cannot be used in asynchronous conditions.

X2C

```
X2C(hexstring)
```

Where

hexstring is a string expression containing hexadecimal characters.

Results

String

Function

The X2C function returns a string, in character format, that represents *hexstring* converted to a character string. The returned string is half as many bytes as the original *hexstring*. The string *hexstring* can be any length. If necessary, it is padded with a leading 0 to make an even number of hexadecimal digits. You can optionally include blanks in *hexstring* (at byte boundaries only, not leading or trailing) to aid readability; they are ignored. If *hexstring* is null, the function returns a null string.

Examples

```
a = X2C('F1F2  F3')                       /* Assigns '123' to "a"      */
b = X2C('C1 C2 C3')                       /* Assigns 'ABC' to "b"      */
c = X2C('8140')                           /* Assigns 'a '  to "b"      */
```

Note: The X2C function cannot be used in asynchronous conditions.

YEAR

```
YEAR()
```

Results

String

Function

The YEAR function returns a 2-character string that represents the last two characters of the current year.

Examples

```
if year() = '02' then   /* It must be 2002. */
   say 'The current year is 2002.'
```

Chapter 27. Keys valid for particular devices

The following list indicates the device key statements and AID keys that can be used for particular devices.

3270 Simulation

BTAB

CHARSET

CLEAR

CLEARPTN

COLOR

CTAB

CURSOR

CURSRSEL

DELETE

DUP

ENTER

EREOF

ERIN

FM

HIGHLITE

HOME

INSERT

JUMP

LCLEAR

LIGHTPEN

NL

PA*n*

PF*nn*

RESET

SCROLL

TAB

Chapter 28. Expressions not allowed in asynchronous conditions

STL expressions requiring WSim counters or save areas for the computation of intermediate results are not allowed in asynchronous conditions on ONIN, ONOUT, or ON SIGNALED statements. The following types of expressions are not allowed:

- Any SUBSTR function with a string constant as the first argument.

```
/***************/
/* NOT ALLOWED */
/***************/
onin substr('hello',1,1) = 'h' then ...

/***********/
/* ALLOWED */
/***********/
onin substr(c,1,1) = 'h' then ...    /* "c" is a string variable.    */
```

- Any SUBSTR function with a nonconstant integer as the second argument.

  Note: There are two exceptions to this rule. The COFF() and LENGTH(SCREEN) functions can be used to specify an offset.

```
/***************/
/* NOT ALLOWED */
/***************/
onin substr(ru,a,1) = 'h' then ...   /* "a" is an integer variable.  */
onin substr(ru,index(ru,'hello'),1) = 'h' then ...
onin substr(ru,a+1,1) = 'h' then ... /* "a" is an integer variable.  */

/***********/
/* ALLOWED */
/***********/
onin substr(ru,7,5) = 'Hello' then ...
onin substr(screen,length(screen)-15,8) = 'MORE ...' then ...
onin substr(screen,coff()+3,7) = 'WELCOME' then ...
```

- Any integer comparison involving the following functions: CCOL, COFF (except when used in SUBSTR as discussed previously), CROW, C2D, E2D, NUMCOLS, NUMROWS, RANDOM, and RNUM.

```
/***************/
/* NOT ALLOWED */
/***************/
onin crow() = 5 then ...

/***********/
/* ALLOWED */
/***********/
onin a = index(data,'test') then ... /* "a" is an integer variable.  */
```

- Any integer expression involving arithmetic where one or more operands are not integer constants.

```
/***************/
/* NOT ALLOWED */
/***************/
onin count + 5 = 20 then ...
onin count = i + 20 then ...         /* "i" is an integer variable.  */

/***********/
/* ALLOWED */
```

```
/***********/
onin count = 1 + 5 then ...
onin count = 1 + d then ...          /* "d" is an integer constant.  */
```

- Any POSTED function or ON SIGNALED statement that specifies an event name composed of hexadecimal string constants, a function reference, or concatenation involving a string variable.

```
/***************/
/* NOT ALLOWED */
/***************/
onin posted('ffff'x) then ...
onin posted(month()) then ...
on signaled('hello'||mydata) then .../* "mydata" is a string         */
                                     /* variable.                    */

/***********/
/* ALLOWED */
/***********/
onin posted('MARK') then ...
on signaled('MARK'||'IT') then ...
```

- Any UTBLMAX function that references a user table that has not been previously processed (in the same program) by the translator.
- Any USEREXIT statement.
- Any LENGTH function that specifies one of the following:
  - A nonconstant string expression
  - A string expression containing a hexadecimal string constant
  - An integer variable.

```
/***************/
/* NOT ALLOWED */
/***************/
onin a = length(mydata) then ...     /* "mydata" is a nonconstant    */
                                     /* string expression.           */
onin a = length('ffff'x) then ...
onin a = length(count) then ...      /* "count" is an integer        */
                                     /* variable.                    */

/***********/
/* ALLOWED */
/***********/
onin a = length(d) then ...          /* "d" is a string constant.    */
onin a = length(5) then ...
```

- Any QUEUED function.
- Any string comparison involving the BITAND, BITOR, BITXOR, B2X, CENTER, COPIES, C2X, DELWORD, D2C, LASTPOS, LEFT, OVERLAY, POS, REVERSE, RIGHT, SPACE, STRIP, SUBWORD, WORD, WORDINDEX, WORDPOS, WORDS, X2B, X2C, INSERT, DELSTR, DBCSADD, DBCSADJ, DBCSDEL, DBCS2SB, SB2DBCS, SB2MDBCS, and TRANSLATE functions.

```
/***************/
/* NOT ALLOWED */
/***************/
onin c = c2x('dfdf') then ...        /* "c" is a string variable.    */

/***********/
/* ALLOWED */
/***********/
onin c = substr(buffer,1,1) then...  /* "c" is a string variable.    */
```

- ATTR3270 with integer expression that contains one or more variables.

```
/***************/
/* NOT ALLOWED */
/***************/
```

```
onin atts = attr3270(offset) then ...
onin atts = attr3270(coff()-(offset+4)) then ...
onin atts = attr3270(rowcol(row+8,12)) then ...

/***********/
/* ALLOWED */
/***********/
onin atts = attr3270(50) then ...
```

Chapter 29. STL reserved words

This chapter contains a comprehensive list of STL reserved words. These words are STL keywords, functions, and reserved variable names that may not be used as user-defined variable or constant names.

@EJECT
@ENDGEN
@ENDGENERATE
@ENDNET
@ENDNETWORK
@ENDPROGRAM
@GEN
@GENERATE
@IFNUM
@INCLUDE
@NET
@NETWORK
@PROGRAM
ABORT
AFTER
ALL
ALLOCATE
AND
APPCLUID
ATTR
ATTR3270
BADGE
BB
BID
BIS
BIT
BITAND
BITOR
BITXOR
BTAB
BUFFER
BY
B2X
CALL
CANCEL
CARD
CCOL
CDI
CEB
CENTER
CHAIN
CHAR
CHARSET
CHASE
CLEAR
CLEARPTN
CMD1-24
CMSEND
CMSERR
CMSF
CMSFM5
CMSLD
CMSMN
CMSPLN
CMSPTR
CMSRC
CMSRT
CMSSL
CMSST
CMSTPN
CMTRTS
CNTLRID
COFF
COLOR
CONSTANT
COPIES
CPOS
CR
CROW
CTAB
CURSOR
CURSRSEL
C2D
C2X
DATA
DATE
DAY
DBCSADD
DBCSADJ
DBCSDEL
DBCS2SB
DEACT
DELAY
DELETE
DELSTR
DELWORD
DEVID
DIDO
DISPLAY
DO
DOWN
DR1
DR2
DUP
EXECUTE
E2D
FLDADV
FLDBKSP
FLDMINUS
FLDPLUS
FM
FMI
FOR
FOREVER
FPORTID
HELP
HEX
HIGHLITE
HOME
ID
IF
INDEX
INITOTH1
INITOTH2
INITSELF
INITSLF1
INITSLF2
INSERT
INTEGER
IO
ITERATE
JUMP
LASTPOS
LASTVERB
LCHID
LCLEAR
LEAVE
LEFT
LENGTH
LIGHTPEN
LINID
LINNAME
LLSID
LOG
LOGGING
LPORTID
LUID
LUSTAT
MAG10
MAG10S
MAG63
OPCMND
OTHERWISE
OVERLAY
PACE
PA1-3
PF1-24
POS
POST
POSTED
PRINT
PUID
PULL
PUSH
PU21ID
QRI
QSIGNAL
QUEC
QUEUE
QUEUED
QUIESCE
RANDOM
RELQ
REPEAT
RESET
RESP
RETURN
REVERSE
RH
RIDISC
RIGHT
RNDISC
RNUM
ROLLDOWN
ROLLUP
ROWCOL
RSHUTD
RTR
RU
SAY
SBI
SB2DBCS
SB2MDBCS
SCREEN
SCROLL
SDT
SELECT
SEND

CMACCP
CMALLC
CMCFM
CMCFMD
CMDEAL
CMECS
CMECT
CMEMN
CMEPLN
CMESL
CMFLUS
CMINIT
CMND
CMONTH
CMPTR
CMRCV
CMRTS
CMSCT
CMSDT
CMSED
D2C
EB
EDI
EFI
ELSE
END
ENDTXT
ENDUTBL
ENTER
EOMC
EOMI
EOMPB
EOMS
EOT
EREOF
EREOS
ERIN
ESC
EVENT
EXC
MAG63S
MCHID
MONITOR
MONTH
MSGTXT
MSGTXTID
MSGUTBL
NETID
NL
NONE
NOP
NORESP
NTWKPRTY
NUMCOLS
NUMROWS
OFF
ON
ONIN
ONOUT
ONS
SENDLINE
SENDMSG
SEQNO
SESSNO
SETRH
SETTH
SHARED
SHOW
SHUTD
SIGNAL
SIGNALED
SNACMND
SNI
SPACE
SSCPID
STRING
STRIP
STRIPE
SUBSTR
SUBWORD
SUSPEND
SYSREQ
TAB
TAG
TCPIPID
TERMID
TERMSELF
TERMSLF1
TGNFIFO
TGSWEEP
TH
THEN
TO
TOD
TPID
TPINSTNO
TRANSLATE
TRANSMIT
TYPE
UNBIND
UNSHARED
UNTIL
UP
USEREXIT
USING
UTBL
UTBLMAX
UTBLSCAN
UTI
VERIFY
VRCWI
VRCWRI
VRPACCNT
VRPRQ
VRPRS

VRRWI
VTAMAPID
WAIT
WHEN
WHILE
WORD
WORDINDEX
WORDPOS
WORDS
WRU
XOFF
XON
X2B
X2C
YEAR

Chapter 30. STL Variable and Named Constant Declarations for CPI-C Verb Parameters

This chapter contains a comprehensive list of STL variable declarations for CPI Communications (CPI-C) verb parameters. The declarations are contained in the CPICVAR and CPICCON members of the samples dataset.

STL variable declarations for CPI-C verb parameters

```
/********************************************************************/
/* STL variable declarations for CPI-C verb parameters              */
/********************************************************************/
/* String parameters                                                */
string conversation_ID                       /* Conversation ID      */
string mode_name                             /* Mode name            */
string partner_LU_name                       /* Partner LU name      */
string sym_dest_name                         /* Symbolic dest name   */
string TP_name                               /* TP name              */
string log_data                              /* log data             */
string send_buffer                           /* send buffer          */
string receive_buffer                        /* receive buffer       */
string fmh5_extension                        /* fmh5_extension       */
/********************************************************************/
/* Integer parameters                                               */
integer conversation_state                   /* Conversation state   */
integer conversation_type                    /* Conversation type    */
integer data_received                        /* Data received        */
integer deallocate_type                      /* Deallocate type      */
integer error_direction                      /* Error direction      */
integer fill                                 /* Fill value           */
integer log_data_length                      /* Log data length      */
integer mode_name_length                     /* Mode name length     */
integer partner_LU_name_length               /* Partner LU name length*/
integer prepare_to_receive_type              /* Prepare to RCV type  */
integer receive_type                         /* Receive type         */
integer received_length                      /* Received length      */
integer request_to_send_received             /* Request-to-send rcvd */
integer requested_length                     /* Requested length     */
integer return_code                          /* Return code          */
integer return_control                       /* Return control       */
integer send_length                          /* Send length          */
integer send_type                            /* Send type            */
integer status_received                      /* Status received      */
integer sync_level                           /* Sync-level           */
integer TP_name_length                       /* TP name length       */
integer fmh5_extension_length                /* fmh5_extension_length */
/********************************************************************/
```

STL named constant declarations for CPI-C verb parameters

```
/********************************************************************/
/* STL named constant declarations for CPI-C verb parameters        */
/********************************************************************/
/* Conversation states                                              */
constant cm_initialize_state          2  /* Initialize state     */
constant cm_send_state                3  /* Send state           */
constant cm_receive_state             4  /* Receive state        */
constant cm_send_pending_state        5  /* Send-pending state   */
constant cm_confirm_state             6  /* Confirm state        */
constant cm_confirm_send_state        7  /* Confirm send state   */
constant cm_confirm_deallocate_state  8  /* Confirm deallocate   */
constant cm_defer_receive_state        9  /* Defer receive state    */
constant cm_defer_deallocate_state     10  /* Defer deallocate state */
/*******************************************************************/
/* Conversation types                                              */
constant cm_basic_conversation         0  /* Basic conversation     */
constant cm_mapped_conversation        1  /* Mapped conversation    */
/*******************************************************************/
/* Data received values                                            */
constant cm_no_data_received           0  /* No data received       */
constant cm_data_received              1  /* Data received          */
constant cm_complete_data_received     2  /* Completed data received*/
constant cm_incomplete_data_received   3  /* Incomplete data rcvd   */
/*******************************************************************/
/* Deallocate types                                                */
constant cm_deallocate_sync_level      0  /* Deallocate sync-level  */
constant cm_deallocate_flush           1  /* Deallocate flush       */
constant cm_deallocate_confirm         2  /* Deallocate confirm     */
constant cm_deallocate_abend           3  /* Deallocate abend       */
/*******************************************************************/
/* Error direction                                                 */
constant cm_send_error                 0  /* Send error             */
constant cm_receive_error              1  /* Receive error          */
/*******************************************************************/
/* Fill values                                                     */
constant cm_fill_ll                    0  /* Fill LL                */
constant cm_fill_buffer                1  /* Fill buffer            */
/*******************************************************************/
/* Prepare-to-receive types                                        */
constant cm_prep_to_receive_sync_level  0  /* Sync-level             */
constant cm_prep_to_receive_flush      1  /* Flush                  */
constant cm_prep_to_receive_confirm    2  /* Confirm                */
/*******************************************************************/
/* Receive types                                                   */
constant cm_receive_and_wait           0  /* Receive and wait       */
constant cm_receive_immediate          1  /* Receive immediate      */
/*******************************************************************/
/* Request-to-send received values                                 */
constant cm_req_to_send_not_received   0  /* Req-to-send not rcvd   */
constant cm_req_to_send_received       1  /* Req-to-send rcvd       */
/*******************************************************************/
/* Return codes                                                    */
constant cm_ok                         0  /* OK                     */
constant cm_allocate_failure_no_retry   1  /* Alloc failure- no retry*/
constant cm_allocate_failure_retry     2  /* Alloc failure- retry   */
constant cm_conversation_type_mismatch  3  /* Conv type mismatch     */
constant cm_pip_not_specified_correctly 5  /* PIP not spec correctly */
constant cm_security_not_valid         6  /* Security not valid     */
constant cm_sync_lvl_not_supported_lu   7  /* Sync-level not supp-LU */
constant cm_sync_lvl_not_supported_pgm  8  /* Sync-level not supp-pgm*/
constant cm_tpn_not_recognized         9  /* TP name not recognized */
constant cm_tp_not_available_no_retry  10  /* TP not avail- no retry */
constant cm_tp_not_available_retry     11  /* TP not avail- retry    */
constant cm_deallocated_abend          17  /* Deallocate abend       */
constant cm_deallocated_normal         18  /* Deallocate normal      */
constant cm_parameter_error            19  /* Parameter error        */
constant cm_product_specific_error     20  /* Product specific error */
constant cm_program_error_no_trunc     21  /* Program error- no trunc*/
constant cm_program_error_purging      22  /* Program error- purging */
constant cm_program_error_trunc        23  /* Program error- trunc   */
constant cm_program_parameter_check    24  /* Program error- parm chk*/
constant cm_program_state_check        25  /* Program state check    */
constant cm_resource_failure_no_retry  26  /* Rsrce failure- no retry*/
constant cm_resource_failure_retry     27  /* Resource failure- retry*/
constant cm_unsuccessful               28  /* Unsuccessful           */
constant cm_deallocated_abend_svc      30  /* Deallocated abend SVC  */
constant cm_deallocated_abend_timer    31  /* Deallocated abend timer*/
constant cm_svc_error_no_trunc         32  /* SVC error- no truncate */
constant cm_svc_error_purging          33  /* SVC error- purging     */
constant cm_svc_error_trunc            34  /* SVC error- truncate    */
/*******************************************************************/
/* Return control values                                           */
constant cm_when_session_allocated      0  /* When session allocated */
constant cm_immediate                   1  /* Immediate              */
/*******************************************************************/
/* Send types                                                      */
constant cm_buffer_data                 0  /* Buffer data            */
constant cm_send_and_flush              1  /* Send and flush         */
constant cm_send_and_confirm            2  /* Send and confirm       */
constant cm_send_and_prep_to_receive    3  /* Send and prep-to-rcv   */
constant cm_send_and_deallocate         4  /* Send and deallocate    */
/*******************************************************************/
/* Status received values                                          */
constant cm_no_status_received          0  /* No status received     */
constant cm_send_received               1  /* Send received          */
constant cm_confirm_received            2  /* Confirm received       */
constant cm_confirm_send_received       3  /* Confirm send received  */
constant cm_confirm_dealloc_received    4  /* Confirm deallocate rcvd*/
/*******************************************************************/
/* Sync levels                                                     */
constant cm_none                        0  /* None                   */
constant cm_confirm                     1  /* Confirm                */
/*******************************************************************/
```

Part 4. Appendixes

Notices

This information was developed for products and services that are offered in the USA.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM® Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
United States of America

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Trademarks and service marks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

IBM	IMS™	MVS™
MVS/SP	OS/390®	SAA
Series/1	SP	System/370
VTAM®	Systems Application Architecture®	

Other company, product, and service names, which may be denoted by a double asterisk (**), may be trademarks or service marks of others.

Glossary

This glossary includes terms and definitions from the *IBM Vocabulary for Data Processing, Telecommunications, and Office Systems*, GC20-1699-6. Further definitions are from the following volumes and reports. The symbols follow the definitions to which they refer.

- The *American National Standard Dictionary for Information Systems*, ANSI X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI). Copies may be purchased from the American National Standards Institute, 11 West 42nd Street, New York, New York 10036. Definitions are identified by the symbol (A) after the definition.
- Definitions from draft proposals and working papers under development by the International Standards Organization, Technical Committee 97, Subcommittee 1 are identified by the symbol (TC97).
- Definitions from draft international standards, draft proposals, and working papers in development by the ISO/TC97/SC1 are identified by the symbol (T), indicating final agreement has not yet been reached among participating members.
- Definitions from the *CCITT Sixth Plenary Assembly Orange Book, Terms and Definitions* and working documents published by the International Consultative Committee on Telegraph and Telephone of the International Telecommunication Union, Geneva, 1980 are identified by the symbol (CCITT/ITU).
- Definitions from published sections of the *ISO Vocabulary of Data Processing*, developed by the International Standards Organization, Technical Committee 97, Subcommittee 1 and from published sections of the *ISO Vocabulary of Office Machines*, developed by subcommittees of ISO Technical Committee 95, are indicated by the symbol (ISO).

A

AID Attention identifier.

American National Standard Code for Information Interchange (ASCII)
The standard code, using a coded character set consisting of 7-bit coded characters (8 bits including parity check), used for information interchange among data processing systems, data communication systems, and associated equipment. The ASCII set consists of control characters and graphic characters. (A)

American National Standards Institute (ANSI)
An organization consisting of producers, consumers, and general interest groups, that establishes the procedures by which accredited organizations create and maintain voluntary industry standards in the United States. (A)

ANSI American National Standards Institute.

Application Programming Interface (API)
(1) The formally defined programming language interface between an IBM system control program or licensed program and its user. (2) The interface through which an application program interacts with an access method. In VTAM, it is the language structure used in control blocks so that application programs can reference them and be identified to VTAM.

arithmetic operator
An operator that involves addition (+), subtraction (-), multiplication (*), division (/), or remainder division (//).

ASCII American National Standard Code for Information Interchange.

asynchronous condition
A condition that is tested outside the flow of normal program execution.

asynchronous subset statement
An asynchronous statement that can be executed when an asynchronous condition is satisfied on an ONIN, ONOUT, or ON SIGNALED statement. Asynchronous subset statements are executed during the transmit interrupt.

asynchronous statement
A statement whose actions occur outside the flow of normal program execution.

attention identifier (AID)
A code that the terminal sends in the inbound data stream to identify the

operator action or structured field function that caused the data stream to be sent to the application program. An AID is always sent as the first byte of the inbound data stream. Structured fields in the data stream may also contain an AID.

available
In VTAM, pertaining to a logical unit that is active, connected, enabled, and not at its session limit.

B

bind In SNA, a request to activate a session between two logical units (LUs).

bit condition
A condition that tests for the equality or inequality of a bit variable.

bit constant
A constant whose value can be either ON or OFF.

bit expression
An expression that, when evaluated, can have the values ON or OFF.

bit variable
A variable that can take on one of two possible values: ON or OFF.

C

carriage return (CR)
The operation that prepares for the next character to be printed or displayed at the specified first position on the same line. (A)

CCC Copy control character.

CD Change direction.

chain A group of logically linked records, for example, an SNA message.

character set
(1) A defined collection of characters in a loadable or nonloadable set selected by means of a local character set identifier. (2) An attribute type in the extended field and character attributes. (3) An attribute passed between session partners in the Start Field Extended, Modify Field, and Set Attribute orders.

Common Programming Interface for Communications (CPI-C)
In WSim, an application programming interface (API) used to perform program-to-program communications using LU type 6.2 communication protocols. An evolving application programming interface (API), embracing functions to meet the growing demands from different application environments and to achieve openness as an industry standard for communications programming. CPI-C provides access to interprogram services such as (a) sending and receiving data, (b) synchronizing processing between programs, and (c) notifying a partner of errors in the communication.

complex condition
A condition involving two or more relational expressions joined by logical operators.

condition
One or more relational expressions that are evaluated to be either true or false.

constant
A value that does not change in the course of program execution.

CPI-C Common programming interface for communications.

CR Carriage return.

D

data flow control (DFC)
In SNA, a request/response unit (RU) category used for requests and responses exchanged between the data flow control layer in one half-session and the data flow control layer in the session partner.

data set
The major unit of data storage and retrieval, consisting of a collection of data in one of several prescribed arrangements and described by control information to which the system has access.

data set members
Members of partitioned data sets that are individually named elements of a larger file that can be retrieved by name.

data terminal equipment (DTE)
That part of a data station that serves as a data source, data link, or both, and provides for the data communications control function according to protocols. (TC97)

DBCS Double-byte character set.

DBCS subfield
DBCS data identified with SO and SI characters.

ddname
Data definition name.

destination logical unit (DLU)
The logical unit to which data is to be sent. Contrast with origin logical unit (OLU).

DFC Data flow control.

DLU Destination logical unit.

double-byte character set (DBCS)
A set of characters in which each character is represented by two bytes. Languages such as Japanese, Chinese, and Korean, which contain more symbols that can be represented by 256 code points, require double-byte character sets. Because each character requires two bytes, the typing, display, and printing of DBCS characters requires hardware and programs that support DBCS.

E

EB End bracket.

EBCDIC
Extended binary-coded decimal interchange code.

EDI Enciphered data indicator.

enciphered data indicator (EDI)
Information to be supplied.

end bracket (EB)
In SNA, the value (binary 1) of the end bracket indicator in the request header (RH) of the first request of the last chain of a bracket; the value denotes the end of the bracket.

event (1) An occurrence of significance to a task; typically, the completion of an asynchronous operation, such as an input/output operation. (2) In WSim, a named indicator/flag which can be used for communications among terminal scripts.

event dictionary
Information included at the end of the printed listing produced by the STL Translator that lists all event names as well as the type of event and where it is referenced.

execute procedure
A procedure containing a limited subset of STL statements that is executed outside the flow of normal program execution.

expression
A variable, constant, function, or any combination of these joined by arithmetic operators (for integers) or string operators (for strings).

extended binary-coded decimal interchange code (EBCDIC)
A coded character set of 256 8-bit characters.

extended field attribute
Additional field definition to the field attribute that controls defining additional properties such as color, highlighting, character set, and field validation. The extended field attribute is altered by information passed in the Start Field Extended and Modify Field orders.

F

facility
(1) An operational capability, or the means for providing such a capability. (T) (2) A service provided by an operating system for a particular purpose; for example, the checkpoint/restart facility.

FID SNA format identification.

File Transfer Protocol (FTP)
In the Internet suite of protocols, an application layer protocol that uses TCP and Telnet services to transfer bulk-data files between machines or hosts.

FM Function management.

FMD Function management data.

format identification (FID) field
In SNA, a field in each transmission header (TH) that indicates the format of the TH; that is, the presence or absence of certain fields. TH formats differ in accordance with the types of nodes between which they pass.

FTP File transfer protocol.

function management data (FMD)
In SNA, a request unit (RU) category

used for end-user data exchanged between logical units (LUs) and for requests and responses exchanged between network services components of LUs, physical units (PUs), and system services control points (SSCPs).

H

hexadecimal string constant
A string constant that contains hexadecimal characters.

I

ILU Initiating logical unit.

IMS/VS
Information Management System/Virtual Storage.

Information Management System/Virtual Storage (IMS/VS)
A general purpose system that enhances the capabilities of OS/VS for batch processing and telecommunication and allows users to access a computer-maintained data base through remote terminals.

initiating logical unit (ILU)
The logical unit that initiates a session with another logical unit or between two other logical units.

input/output (I/O)
(1) Pertaining to a device whose parts can perform an input process and an output process at the same time. (2) Pertaining to a functional unit or channel involved in an input process, output process, or both, concurrently or not, and to the data involved in such a process. ***Note:*** *The phrase input/output may be used in place of input/output data, input/output signals, and input/output process when such a usage is clear in context.* (3) Pertaining to input, output, or both.

instance
A copy of a transaction program that is operating on a given logical unit. If multiple instances are supported on a logical unit, multiple copies of the same transaction program can operate simultaneously.

integer condition
A condition that tests an integer variable.

integer constant
A constant whose value can be a positive decimal integer from 0 to 65535.

integer constant expression
An integer expression involving only integer constants and integer operators.

integer expression
An expression composed of integer variables or constants that can be joined by arithmetic operations.

integer variable
A variable that can take on any positive integer value from 0 to 65535.

intermessage delay
The elapsed time between receipt of a system response at a terminal and the time when a new transaction is entered. It is synonymous with the time a real operator requires to think about what to do next.

I/O Input/output.

J

JCL Job control language.

job control language (JCL)
A problem-oriented language designed to express statements in a job that are used to identify the job or describe its requirements to an operating system. (A)

L

label A statement identifier that enables the referencing of that statement from elsewhere in a program.

line feed (LF)
The incremental relative movement between the paper carrier and the type carrier in a direction perpendicular to the writing line.

literal text DBCS data
DBCS data with SO and SI characters wrapped around it.

log data set
The data set WSim uses to record activities that occur during a simulation.

logical operator
An operator that establishes a relationship between two or more simple conditions, forming a complex condition.

logical unit (LU)
(1) A port through which a user gains access to the services of a network. (2) In SNA, a port through which an end user accesses the SNA network and the functions provided by system services control points (SSCPs). An LU can support at least two sessions—one with an SSCP and one with another LU—and may be capable of supporting many sessions with other logical units.

logic test
In WSim, a conditional test on an input or output message, a counter, or other item using the ¬IF statement. The IF actions can be used to control the message generation process.

Loglist Utility
A utility that enables WSim to produce a formatted report of the log data set.

LU Logical unit.

LUSTAT
Logical unit status.

M

message generation
In WSim, the process of executing statements that generate messages from the resources being simulated by WSim.

message generation deck
A collection of message generation statements, beginning with a MSGTXT statement and ending with an ENDTXT statement.

message generation statements
The collection of statements that define the actions to be performed by WSim, including message generation and logic testing.

module
A program unit that is discrete and identifiable with respect to compiling, combining with other units, and loading; for example, the input to, or output from, an assembler, compiler, linkage editor, or executive routine. (A)

MTRC
Message generation trace record.

Multiple Virtual Storage (MVS)
An IBM licensed program whose full name is the Operating System/Virtual Storage (OS/VS) with Multiple Virtual Storage/System Product for System/370*. It is a software operating system controlling the execution of programs.

MVS Multiple Virtual Storage.

N

named constant
An integer or string constant that is defined using the INTEGER or STRING statement.

NC Network control.

NCB Network control block.

network control block (NCB)
A WSim control block containing information about simulated networks.

network definition
A collection of network definition statements that defines the terminals being simulated and the various options used for different lines, terminals, and devices that make up the simulated system.

network definition statements
A collection of statements that defines the network configuration WSim simulates when processing the message generation source statements.

network services (NS)
In SNA, the services within network addressable units (NAUs) that control network operation through SSCP-SSCP, SSCP-PU, and SSCP-LU sessions.

O

operating system (OS)
Software that controls the execution of programs. An operating system may provide services such as resource allocation, scheduling, input/output control, and data management. ***Note:*** *Although operating systems are predominantly software, partial or complete hardware implementations are possible.* (A)

operator
A symbol representing an action to be performed on two items.

OS Operating system.

P

PA Program attention.

partitioned data set (PDS)
A data set in direct access storage that is divided into partitions, called members, each of which can contain a program, part of a program, or data.

path information unit (PIU)
In SNA, a message unit consisting of a transmission header (TH) alone, or of a TH followed by a basic information unit (BIU) or a BIU segment.

PF Program function.

PIU Path information unit.

PLU Primary logical unit.

posting
The act of indicating that a named event has occurred. The notification is available to any terminal that inquires about the event and is in effect until explicitly reset.

primary logical unit (PLU)
In SNA, the logical unit (LU) that contains the primary half-session for a particular LU-LU session. Each session must have a PLU and secondary logical unit (SLU). The PLU is the unit responsible for the bind and is the controlling LU for the session. A particular LU may contain both primary and secondary half-sessions for different active LU-LU sessions. Contrast with secondary logical unit (SLU).

printed listing
A listing created by the STL Translator that contains the message generation statements the translator produces, the STL input source lines, the variable dictionary, the event dictionary, and the WSim Preprocessor output.

procedure
A group of statements, beginning with a MSGTXT statement and ending with and ENDTXT statement, that performs a specific task.

program
A group of procedures and declarative statements that contain all the instructions required to accomplish a specific task.

programmed symbols (PS)
In the 3270 Information Display System, an optional feature that stores up to six user-definable, program-loadable character sets of 190 characters each in terminal read/write storage for display or printing by the terminal.

PS Programmed symbols.

PSID Product Set ID.

PU Physical unit.

R

record (1) A set of data treated as a unit (TC97); for example, in stock control, each invoice could constitute one record. (2) In VTAM, the unit of data transmission for record-mode. A record represents whatever amount of data the transmitting node chooses to send. (3) In Series/1*, a portion of a data set accessed at the logical level (GET/PUT).

relational expression
An expression involving one or more relational operators.

relational operator
An operator that establishes a relationship between expressions.

request/response header (RH)
In SNA, control information preceding a request/response unit (RU), that specifies the type of RU (request unit or response unit) and contains control information associated with that RU.

request/response unit (RU)
In SNA, a generic term for a request unit or a response unit.

request unit (RU)
(1) In SNA, a message unit that contains control information, such as a request code, or function management (FM) headers, end-user data, or both. (2) In DPCX, the smallest unit of data or control information.

reserved word
A word that has a special meaning in STL and cannot be used in a different context. Reserved words include keywords, function names, special reserved variables, and bit values.

resource
(1) Any facility of the computing system or operating system required by a job or

task, and including main storage, input/output devices, the processing unit, data sets, and control or processing programs. (2) In the NetView program, any hardware or software that provides function to the network.

response unit (RU)
In SNA, a message unit that acknowledges a request unit; it may contain prefix information received in a request unit. If positive, the response unit may contain additional information (such as session parameters in response to BIND session), or if negative, contains sense data defining the exception condition.

return code
A code used to influence the execution of succeeding instructions. (A)

RH Request header or response header.

RNR Receive not ready.

RR Receive ready.

RU Request unit or response unit.

S

SA Set attribute.

SAP Service access point.

SBCS Single-byte character set.

SBI Stop Bracket Initiation.

SC Session Control.

script See WSim script.

SDT Start data traffic.

secondary logical unit (SLU)
In SNA, the logical unit (LU) that contains the secondary half-session for a particular LU-LU session. An LU may contain secondary and primary half-sessions for different active LU-LU sessions. Contrast with primary logical unit (PLU).

session control (SC)
In SNA, (1) One of the components of transmission control. Session control is used to purge data flowing in a session after an unrecoverable error occurs, to resynchronize the data flow after such an error, and to perform cryptographic verification. (2) A request unit (RU) category used for requests and responses exchanged between the session control components of a session and for session activation and deactivation requests and responses.

shared variable
A variable that can be used by all terminals in a network.

SI Shift In. Used with DBCS. This is the X'0F' character that ends DBCS data.

signaling
The act of indicating that a named event has occurred. Unlike posting, the notice cannot be consulted later to determine if the event occurred.

simple condition
A condition involving a single relational expression.

SLU Secondary logical unit.

SNA Systems Network Architecture.

SO Shift Out. Used with DBCS. This is the X'0E' character that begins DBCS data.

statement
An object in STL made up of variables, constants, keywords, function names, operators, punctuation, labels, MSGTXT names, and MSGUTBL names. A statement can declare variable types and classes, assign a value to a variable, or control the execution of the program.

STL Structured Translator Language.

STL Translator
In WSim, a utility that acts as the STL translator and translates STL statements into message generation source statements.

STRC STL trace record.

string condition
A condition involving string expressions.

string constant
A constant whose value can be any set of characters enclosed in single or double quotation marks.

string constant expression
An expression composed of string constants joined by string operators.

string expression
An expression composed of string

variables, string constants, or hexadecimal string constants that can be joined by string operators.

string operator
An operator that involves concatenation of two strings. || concatenates two strings as is and a blank (X'40') joins them with an intervening blank.

string variable
A variable that can only contain characters.

Structured Translator Language (STL)
A set of conventions and rules for writing syntactically allowable statements that will create message generation source statements.

Systems Network Architecture (SNA)
The description of the logical structure, formats, protocols, and operational sequences for transmitting information units through and controlling the configuration and operation of networks.

T

TCP/IP
Transmission Control Protocol/Internet Protocol. A set of communication protocols that support peer-to-peer connectivity functions for both local and wide area networks.

TH Transmission header.

think time
The elapsed time between receipt of a system response at a terminal and the time when a new transaction is entered. Synonym for intermessage delay.

time sharing option (TSO)
An optional configuration of the operating system that provides conversational time sharing from remote stations in a network using VTAM.

TP Transaction program.

transaction program (TP)
In WSim, any program that uses LU type 6.2 communication protocols to communicate with another program. Transaction programs are implemented in WSim using the CPI-C application program interface.

transmission header (TH)
In SNA, control information, optionally followed by a basic information unit (BIU) or a BIU segment, that is created and used by path control to route message units and to control their flow within the network.

transmit interrupt
An interruption of program execution that occurs when accumulated message data is sent to the host system.

TSO Time sharing option.

U

unshared variable
A variable that can be used only by individual terminals in a network. Each terminal has its own private copy of an unshared variable.

user exit
A specialized routine, written by the user, to customize WSim to meet a unique requirement.

user table
In WSim, one or more text data entries contained in a table format which may be referenced for logic testing and message generation.

UTI User time interval.

V

variable
A data item that is used in a program in a certain way, but whose value can vary. In a program, each variable has a unique symbolic name.

variable dictionary
Information included at the end of the printed listing produced by the STL Translator that lists the names of all variables as well as the variable's type, class, to which WSim resource it maps, where it is defined, and where it is referenced.

Virtual Telecommunications Access Method (VTAM)
An IBM licensed program that controls communication and the flow of data in an SNA network. It provides single-domain, multiple-domain, and interconnected network capability.

VTAM
Virtual Telecommunications Access Method.

W

ward 42
The portion of DBCS codes that corresponds to those of SBCS. The first byte is X'42'. The second byte is the hexadecimal value of the corresponding single-byte EBCDIC code.

ward byte
The first byte of each DBCS character.

work variable
A WSim save area, counter, or switch used by the STL Translator when translating an STL program that does not map to an STL variable name.

write-to-operator (WTO)
An optional user-coded service that enables the writing of a message to the system console operator that informs the operator of errors and unusual system conditions that may need correcting.

WSF Write structured field.

WSim Display Monitor Facility
A VTAM application program within WSim that displays simulated 3270 screen images on a monitor. It is used to monitor a simulation dynamically, enabling a user to debug scripts and view interactions with host applications.

WSim network
The set of statements defining an entire network, including both the network definition statements and the message generation source statements. Should not be confused with a packet switching network.

WSim script
The set of statements defining an entire network, including both the network definition statements and the message generation source statements.

WTO Write-to-operator.

X

XID Exchange identification.

Bibliography

The following manuals provide additional information about the definition and operation of networks simulated by WSim:

WSim Library

WSim User's Guide, SC31-8948

WSim Messages and Codes, SC31-8951

WSim Test Manager User's Guide and Reference, SC31-8949

Creating WSim Scripts, SC31-8945

WSim Script Guide and Reference, SC31-8946

WSim Utilities Guide, SC31-8947

WSim User Exits, SC31-8950

Related publications

Systems Application Architecture Common Programming Interface Communications Reference, SC26-4399-06. (WSim does not support CPI-C functions that have been added in later releases of this document.)

VTAM Programming for LU 6.2, SC31-6437.

Index

Printed in USA

SC31-8946-01