

# **Window Terminal Programmer's Guide for Asynchronous Communications**

Stratus Technologies  
R194-02

---

# Notice

The information contained in this document is subject to change without notice.

UNLESS EXPRESSLY SET FORTH IN A WRITTEN AGREEMENT SIGNED BY AN AUTHORIZED REPRESENTATIVE OF STRATUS TECHNOLOGIES, STRATUS MAKES NO WARRANTY OR REPRESENTATION OF ANY KIND WITH RESPECT TO THE INFORMATION CONTAINED HEREIN, INCLUDING WARRANTY OF MERCHANTABILITY AND FITNESS FOR A PURPOSE. Stratus Technologies assumes no responsibility or obligation of any kind for any errors contained herein or in connection with the furnishing, performance, or use of this document.

Software described in Stratus documents (a) is the property of Stratus Technologies International, S.à r.l. or the third party, (b) is furnished only under license, and (c) may be copied or used only as expressly permitted under the terms of the license.

Stratus documentation describes all supported features of the user interfaces and the application programming interfaces (API) developed by Stratus. Any undocumented features of these interfaces are intended solely for use by Stratus personnel and are subject to change without warning.

This document is protected by copyright. All rights are reserved. No part of this document may be copied, reproduced, or translated, either mechanically or electronically, without the prior written consent of Stratus Technologies.

Stratus, the Stratus logo, Continuum, Continuous Processing, StrataLINK, and StrataNET are registered trademarks of Stratus Technologies International, S.à r.l.

ftServer, ftServer with design, Stratus 24 x 7 with design, The World's Most Reliable Server, Selectable Availability, XA/R, SQL/2000, and The Availability Company are trademarks of Stratus Technologies International, S.à r.l.

RSN is a trademark of Lucent Technologies, Inc.  
All other trademarks are the property of their respective owners.

Manual Name: *Window Terminal Programmer's Guide for Asynchronous Communications*

Part Number: R194  
Revision Number: 02  
VOS Release Number: 14.4.0  
Printing Date: May 2001

Stratus Technologies, Inc.  
111 Powdermill Road  
Maynard, Massachusetts 01754-3409

© 2001 Stratus Technologies International, S.à r.l. All rights reserved.

---

# Contents

---

---

## Preface

xi

---

<b>1. Overview of Window Terminal Communications</b>	<b>1-1</b>
Window Terminal and Old Asynchronous Drivers	1-1
Setup Procedure for the Window Terminal Driver	1-2
Features of the Window Terminal Driver	1-3
Architecture of the Window Terminal Driver	1-3
Terminal-Interface Layers	1-5
Access Layers	1-5
Asynchronous Access Layer	1-6
Console Controller Access Layer	1-6
OS TELNET Access Layer	1-7
The STCP Access Layer	1-8
Application Program Interface	1-9
Application Design Considerations	1-10
Selecting a Programming Approach	1-10
Simple Sequential I/O	1-12
Unprocessed Raw I/O	1-12
Application-Managed I/O	1-12
Working with Primary Windows and Subwindows	1-13
Using National Language Support	1-17
Using Flow Control	1-18
Using Wait Mode or No-Wait Mode	1-20
Using Hardware Features	1-21
Using an Interrupt Table	1-21
Using Automatic Frame Detection	1-22
Using Break Handling	1-22
Using the Standard Include Files	1-23
Compiling, Binding, and Debugging Considerations	1-24
RS-232-C Asynchronous Device Interface	1-24
Types of Asynchronous Connections	1-25
Direct Connections	1-25
Modem Connections	1-26
RS-232-C Communications I/O Hardware	1-28
The K-series I/O Subsystem	1-29

K101 Full-Modem MultiCommunications I/O Adapter	1-31
K111 Null-Modem MultiCommunications I/O Adapter	1-31
K110 Twinaxial Printer Interface I/O Adapter	1-32
K118 Asynchronous I/O Adapter	1-32
RS-232-C Configurations	1-33
Asynchronous Channel or Subchannel Characteristics	1-36
Full-Duplex Transmission	1-37
Baud-Rate Configuration	1-37
Login/Slave Configuration	1-37
ASCII or Baudot Communication	1-38
Parity Configuration	1-38
Bits-Per-Character Configuration	1-39
Stop-Bits Configuration	1-39
Force-Listen Configuration	1-39
Privileged-Terminal Configuration	1-39

---

<b>2. Simple Sequential I/O</b>	2-1
Characteristics of Simple Sequential I/O	2-1
Subwindow Organization	2-3
Pause Processing	2-7
Retrieving Input from a Simple Sequential Subwindow	2-9
Generic Input Request Handling in a Simple Sequential Subwindow	2-10
Sending Output to a Simple Sequential Subwindow	2-13
Issuing Generic Output Requests in the Simple Sequential Subwindow	2-13
Issuing Generic Output Requests That Switch the Subwindow to Formatted I/O Mode	2-15

---

<b>3. Unprocessed Raw I/O</b>	3-1
Characteristics of Unprocessed Raw I/O	3-1
Retrieving Unprocessed Raw Input	3-3
Selecting an Unprocessed Raw Input Mode	3-3
Using Normal Raw Mode	3-5
Using Raw Table Mode	3-5
Using Raw Record Mode	3-6
Creating an Interrupt Table	3-7
Sending Unprocessed Raw Output	3-9

---

<b>4. Application-Managed I/O</b>	4-1
Characteristics of Application-Managed I/O	4-1
Using a Formatted I/O Subwindow	4-2
Creating a Formatted I/O Subwindow	4-3
Enabling and Disabling Formatted I/O Mode in a Simple Sequential Subwindow	4-3
Sending Formatted Sequential Output	4-4
Issuing Generic Output Requests Available with Formatted Sequential Output	4-5
Generic Output Requests That Affect Subwindow Updates	4-7
Retrieving Formatted Sequential Input	4-8
Input Handling with Formatted Sequential Input	4-9
Generic Input Requests Available with Formatted Sequential Input	4-10
Retrieving Raw Input from a Formatted I/O Subwindow	4-12
Controlling Input Mapping with Processed Raw Input	4-12
Decoding the Input Returned in Binary Data	
Introducer (BDI) Sequences	4-16
Using Translated Input Mode	4-19
Using Function-Key Input Mode	4-20
Using Generic Input Mode	4-23

---

<b>5. Generic Input Requests and Generic Output Requests</b>	5-1
Handling Generic Input Requests	5-1
Handling Generic Output Requests and Other Translated Output	5-6
Handling Generic Output Requests	5-7
Compatibility Issues Concerning Generic Output Requests	5-9
Output Requests for Performing Cursor Operations	5-13
Output Requests for Performing Clearing Operations	5-18
Output Requests for Performing Line Operations	5-19
Output Requests for Setting Visual Attributes	5-22
Output Requests for Setting Modes	5-26
Miscellaneous Generic Output Requests	5-28
Handling National Language Support (NLS) Characters	5-30
Defining the Handling of Standard ASCII Control Characters	5-31

---

<b>6. Attaching and Opening a Port</b>	6-1
s\$attach_port	6-2
s\$open	6-6

---

s\$seq_open	6-11
-------------	------

---

<b>7. Performing Input and Output Operations</b>	<b>7-1</b>
s\$read_raw	7-3
s\$seq_read	7-14
s\$write_raw	7-22
s\$seq_write	7-25
s\$seq_write_partial	7-31

---

<b>8. Performing Control Operations</b>	<b>8-1</b>
s\$set_io_time_limit	8-2
s\$set_wait_mode	8-5
s\$set_no_wait_mode	8-7
s\$read_device_event	8-12
s\$control	8-15
Global Control Opcodes	8-24
Opcodes Affecting the RS-232-C Communications Medium	8-26
Opcodes Affecting All Communications Media	8-37
Opcodes Affecting the Terminal	8-40
Opcodes Affecting the Primary Window and Subwindow	8-46
s\$control_device	8-80

---

<b>9. Closing and Detaching a Port</b>	<b>9-1</b>
s\$close	9-2
s\$detach_port	9-5

---

<b>Appendix A. Compatibility Issues</b>	<b>A-1</b>
8-Bit Support and Parity Options	A-1
Flow-Control Options	A-2
Function of Simple Sequential I/O	A-3
The Function of Unprocessed Raw I/O	A-4
The Function of Processed Raw Input	A-6
Generic Input and Generic Output Requests	A-8
Requests That Apply Only to the Window Terminal Driver	A-8
Requests That Do Not Apply to the Window Terminal Driver	A-9
Requests That Should Be Replaced by Opcodes, TTP Subroutines, or Other Requests	A-9
Window Terminal Requests That Have Aliases	A-12
Handling of Attributes	A-12

---

Handling of Control Characters	A-13
Handling of <code>s\$control</code> Opcodes	A-14
Handling of the Complete-Write Option	A-21
Asynchronous Device Configuration	A-21

---

<b>Appendix B. Internal Character Coding System</b>	B-1
Left-hand Control Character Set	B-3
Control Characters 7 to 13 Decimal (07 to 0D Hexadecimal)	B-3
SUB Character	B-3
ESC Character	B-3
Left-hand Graphic Character Set	B-3
Right-hand Control Character Set	B-3
Single-Shift <code>x</code> Characters	B-4
Locking Shift Introducer	B-4
Word Processing Introducer	B-4
Binary Data Introducer	B-5
Right-hand Supplementary Graphic Character Set	B-5

---

<b>Appendix C. Terminal-Type Subroutines</b>	C-1
--	-----

---

<b>Appendix D. Sample Program</b>	D-1
-----------------------------------	-----

---

<b>Appendix E. Tools and Commands</b>	E-1
Commands That Set Up and Configure the Device	E-3
Commands That Display Device Information	E-4
System Analysis Tool	E-5

---

<b>Appendix F. Default Internal Character Sets</b>	F-1
--	-----

---

<b>Glossary</b>	Glossary-1
-----------------	------------

---

<b>Index</b>	Index-1
--------------	---------

---

---

## Figures

Figure 1-1.	Architecture of the Window Terminal Driver	1-4
Figure 1-2.	Sample Primary Window and Subwindows	1-16
Figure 1-3.	A Direct Terminal Connection	1-26
Figure 1-4.	A Terminal Connection Using Modems	1-27
Figure 1-5.	Multicom munications I/O Adapter Subchannels	1-31
Figure 1-6.	Null-Modem Adapter Signals	1-36
Figure 2-1.	Subwindow Organization for Simple Sequential I/O	2-5
Figure 4-1.	Levels of Input Processing	4-14
Figure B-1.	Internal Character Coding System	B-2



---

# Tables

Table 1-1.	Subroutines That Communicate with the Window Terminal Driver	
	1-9	
Table 1-2.	Available Include Files	1-23
Table 1-3.	Active Pins on a Full-Modem	
	Asynchronous Channel/Subchannel	1-34
Table 1-4.	Active Pins on a Null-Modem Asynchronous	
	Channel/Subchannel	1-35
Table 2-1.	Generic Input Requests Available with Simple Sequential I/O	
	2-11	
Table 2-2.	Generic Output Requests Available with Simple	
	Sequential I/O	2-14
Table 2-3.	Generic Output Requests That Switch the	
	Subwindow to Formatted I/O Mode	2-15
Table 4-1.	Generic Output Requests Available with	
	Formatted Sequential Output	4-6
Table 4-2.	Generic Input Requests Available with Formatted Sequential	
	Input	4-10
Table 4-3.	Basic Function Keys	4-20
Table 5-1.	Generic Input Requests	5-2
Table 5-2.	Generic Output Requests	5-9
Table 5-3.	Display Attributes for the Window	
	Terminal Driver	5-25
Table 5-4.	Line Graphics Characters	5-27
Table 7-1.	Wait Mode and <code>s\$read_raw</code>	7-5
Table 7-2.	No-Wait Mode and <code>s\$read_raw</code>	7-7
Table 7-3.	Input-Stream Metacharacters	7-10
Table 7-4.	<code>s\$read_raw</code> Error Codes	7-11
Table 8-1.	Window Terminal Opcodes	8-19
Table 8-2.	Connection States	8-37
Table 8-3.	Input Modes and Values	8-55
Table A-1.	Differences in Line-Editing Options	A-3
Table A-2.	Differences in the Unprocessed Raw Input Modes	A-5
Table A-3.	Methods of Returning Processed Raw Input	A-6
Table A-4.	Replacing Generic Output Requests Used by the Old	
	Asynchronous	
	Driver	A-10
Table A-5.	Differences in the Handling of Control Characters	A-13
Table A-6.	Comparison of Standard Asynchronous and Window Terminal	

	Opcodes	A-15	
Table A-7.	Differences in Asynchronous Device Configuration		A-22
Table C-1.	TTP Configuration Subroutines		C-2
Table C-2.	TTP Keyboard Subroutines		C-2
Table C-3.	TTP Generic Input Subroutines		C-3
Table C-4.	TTP Character-Translation Subroutines		C-4
Table C-5.	TTP Generic Output Subroutines		C-5
Table C-6.	TTP Output-Capability Subroutines		C-6
Table C-7.	TTP Display-Attribute Subroutines		C-7
Table E-1.	Tools and Commands Supporting Asynchronous Device Communications	E-2	
Table F-1.	VOS Internal Character Code Set		F-1

---

# Preface

The *Window Terminal Programmer's Guide for Asynchronous Communications* (R194) documents the application interface for the window terminal software. The window terminal driver that supports the window-based environment is bound into the kernel of VOS.

This manual is intended for application programmers who are designing applications that will run in the window terminal environment. The window terminal environment offers support for communications protocols such as the RS-232-C asynchronous communications protocol and the Stratus implementation of the TELNET communications protocol, which is part of the STCP product set or the OS TCP/IP product set.

Before using the *Window Terminal Programmer's Guide for Asynchronous Communications* (R194), you should be familiar with the following Stratus manuals.

- *VOS Communications Software: Defining a Terminal Type* (R096)
- VOS Subroutines manuals
- VOS System Administration manuals

## Manual Version

This manual is a revision. Change bars, which appear in the margin, note the specific changes to text since the previous publication of this manual.

This revision incorporates the following changes.

- The manual contains information about POSIX support for window terminal applications.
- The manual introduces the following window terminal opcodes.
  - `TERM_OPEN_EXISTING_WINDOW_OPCODE` (2107)
  - `TERM_ENABLE_CACHED_IO_OPCODE` (2108)
  - `TERM_DISABLE_CACHED_IO_OPCODE` (2109)
  - `TERM_POSIX_OPEN_OPCODE` (2110)
  - `TERM_POSIX_CLOSE_OPCODE` (2111)
  - `TERM_GET_SUBWIN_BORDER_OPCODE` (2112)

- `TERM_SET_SUBWIN_BORDER_OPCODE` (2113)
  - `TERM_GET_SESSION_ID_OPCODE` (2114)
  - `TERM_GET_POSIX_GETATTR_OPCODE` (2115)
  - `TERM_SET_POSIX_GETATTR_OPCODE` (2116)
- The manual contains updated information on the following:
    - a new `-rts_cts_flow_control` option supported for K-Series I/O adaptor and Console Controller hardware
    - use of comments in C and PL/I structures to indicate a version number for those structures requiring version numbers
    - to correct how a break is signalled when the `RETURN_ERROR` break action is used with the `TERM_SET_BREAK_ACTION` (2051) opcode is used

## Manual Organization

This manual has nine chapters, six appendixes, and a glossary.

[Chapter 1](#) provides an overview of window terminal communications. It describes the architecture of the window terminal driver, the application program interface (focusing on programming considerations), and the asynchronous device interface (focusing on hardware options and the characteristics of asynchronous connections).

[Chapter 2](#) describes the simple sequential I/O programming approach.

[Chapter 3](#) describes the unprocessed raw I/O programming approach.

[Chapter 4](#) describes the application-managed I/O programming approach.

[Chapter 5](#) discusses generic input requests and generic output requests.

[Chapter 6](#) describes the subroutines that attach and/or open a port to a window terminal device.

[Chapter 7](#) describes the subroutines that either retrieve input from or send output to a window terminal device.

[Chapter 8](#) describes the subroutines that perform the following control operations: setting a time limit, enabling wait mode or no-wait mode, reading a device event, and handling `s$control` opcodes specified by your application program.

[Chapter 9](#) describes the subroutines that either close or detach the port from the window terminal device.

[Appendix A](#) describes the differences between the window terminal driver and the old asynchronous driver, which is the original Stratus asynchronous driver.

[Appendix B](#) discusses the Stratus internal character coding system.

[Appendix C](#) describes the terminal-type subroutines that enable your application program to access the terminal-specific information in the terminal-type databases.

[Appendix D](#) provides a sample program.

[Appendix E](#) provides an overview of the tools and commands that apply to asynchronous devices.

[Appendix F](#) presents a table that lists the characters in the default internal character sets.

## Related Manuals

Refer to the following Stratus manuals for related documentation.

- *VOS Communications Software: Defining a Terminal Type* (R096)
- *Window Terminal User's Guide* (R256)
- *VOS Commands Reference Manual* (R098)
- *VOS System Analysis Manual* (R073)
- *VOS Communications Software: Asynchronous Communications* (R025)
- *National Language Support User's Guide* (R212)
- *Product Configuration Bulletin: Asynchronous Devices* (R289)
- *Site Planning Guide* (R003)
- *VOS Communications Software: X.25 and X.29 Programming* (R028)
- *Migrating VOS Applications to the Stratus RISC Architecture* (R288)
- *VOS Continuum 600 and 1200 Series with PA-8000: Operation and Maintenance Guide* (R445)
- *Continuum Series 600 and 1200: Site Planning Guide* (R391)
- *Site Planning Guide: K114 I/O Adapter* (R319)
- *Site Planning Guide: K118 Asynchronous I/O Adapter* (R325)
- STCP Manuals (for STCP TELNET information):
  - VOS STREAMS TCP/IP Administrator's Guide* (R419)
  - VOS STREAMS TCP/IP Programmer's Guide* (R420)
  - VOS STREAMS TCP/IP User's Guide* (R421)
  - VOS STREAMS TCP/IP Migration Guide* (R418)

- *Software Release Bulletin: VOS Release 14.3.0 (R914)* (for information about POSIX) and the online doc file >system>doc>posix>posix.doc (for more information about building VOS POSIX.1 applications)
- OS TCP/IP manuals (for OS TELNET information):
  - VOS Communications Software: OS TCP/IP User's Guide (R222)*
  - VOS OS TCP/IP Administrator's Guide (R223)*
  - VOS Communications Software: OS TCP/IP Programmer's Manual (R224)*
- VOS Language manuals:
  - VOS PL/I Language Manual (R009)*
  - VOS BASIC Language Manual (R011)*
  - VOS COBOL Language Manual (R010)*
  - VOS FORTRAN Language Manual (R013)*
  - VOS Pascal Language Manual (R014)*
  - VOS Standard C Reference Manual (R363)*
  - VOS C Language Manual (R040)*
- VOS Subroutines manuals:
  - VOS PL/I Subroutines Manual (R005)*
  - VOS BASIC Subroutines Manual (R018)*
  - VOS COBOL Subroutines Manual (R019)*
  - VOS FORTRAN Subroutines Manual (R020)*
  - VOS Pascal Subroutines Manual (R021)*
  - VOS C Subroutines Manual (R068)*
- VOS System Administration manuals:
  - VOS System Administration: Administering and Customizing a System (R281)*
  - VOS System Administration: Starting Up and Shutting Down a Module or System (R282)*
  - VOS System Administration: Registration and Security (R283)*
  - VOS System Administration: Disk and Tape Administration (R284)*
  - VOS System Administration: Backing Up and Restoring Data (R285)*
  - VOS System Administration: Administering the Spooler Facility (R286)*
  - VOS System Administration: Configuring a System (R287)*

- VOS Forms Management System manuals:

*VOS PL/I Forms Management System (R016)*

*VOS BASIC Forms Management System (R033)*

*VOS COBOL Forms Management System (R035)*

*VOS FORTRAN Forms Management System (R037)*

*VOS Pascal Forms Management System (R039)*

*VOS C Forms Management System (R070)*

Refer also to the following third-party documentation.

POSIX.1 Standard ISO/IEC 9945-1:1996(E) ANSI/IEEE Std 1003.1, 1996 Edition (for POSIX information)

## Notation Conventions

This manual uses the following notation conventions.

- Italics introduces or defines new terms. For example:

The *master disk* is the name of the member disk from which the module was booted.

- Boldface emphasizes words in text. For example:

Every module **must** have a copy of the `module_start_up.cm` file.

- Monospace represents text that would appear on your terminal's screen (such as commands, subroutines, code fragments, and names of files and directories). For example:

```
change_current_dir (master_disk)>system>doc
```

- Monospace italic represents terms that are to be replaced by literal values. In the following example, the user must replace the monospace-italic term with a literal value.

```
list_users -module module_name
```

- Monospace bold represents user input in examples and figures that contain both user input and system output (which appears in monospace). For example:

```
display_access_list system_default
```

```
%dev#m1>system>acl>system_default
```

```
w *.*
```

## Key Mappings for VOS Functions

VOS provides several command-line and display-form functions. Each function is mapped to a particular key or combination of keys on the terminal keyboard. To perform a function, you press the appropriate key(s) from the command-line or display form. For an explanation of the command-line and display-form functions, see the manual *Introduction to VOS* (R001).

The keys that perform specific VOS functions vary depending on the terminal. For example, on a V103 ASCII terminal, you press the **[Shift]** and **[F20]** keys simultaneously to perform the `INTERRUPT` function; on a V105 PC/+ 106 terminal, you press the **[1]** key on the numeric keypad to perform the `INTERRUPT` function.

### NOTE

Certain applications may define these keys differently. Refer to the documentation for the application for the specific key mappings.

The following table lists several VOS functions and the keys to which they are mapped on commonly used Stratus terminals and on an IBM PC® or compatible PC that is running the Stratus PC/Connect-2 software. (If your PC is running another type of software to connect to a Stratus host computer, the key mappings may be different.) For information about the key mappings for a terminal that is not listed in this table, refer to the documentation for that terminal.

VOS Function	V103 ASCII	V103 EPC	IBM PC or Compatible PC	V105 PC/+ 106	V105 ANSI
CANCEL	<b>[F18]</b>	* †	* †	<b>[5]</b> † or * †	<b>[F18]</b>
CYCLE	<b>[F17]</b>	<b>[F12]</b>	<b>[Alt]-[C]</b>	<b>[4]</b> †	<b>[F17]</b>
CYCLE BACK	<b>[Shift]-[F17]</b>	<b>[Shift]-[F12]</b>	<b>[Alt]-[B]</b>	<b>[7]</b> †	<b>[Shift]-[F17]</b>
DISPLAY FORM	<b>[F19]</b>	_ †	_ †	<b>[6]</b> † or _ †	<b>[F19]</b> or <b>[Shift]-[Help]</b>
HELP	<b>[Shift]-[F8]</b>	<b>[Shift]-[F2]</b>	<b>[Shift]-[F2]</b>	<b>[Shift]-[F8]</b>	<b>[Help]</b>
INSERT DEFAULT	<b>[Shift]-[F11]</b>	<b>[Shift]-[F10]</b>	<b>[Shift]-[F10]</b>	<b>[Shift]-[F11]</b>	<b>[F11]</b>
INSERT SAVED	<b>[F11]</b>	<b>[F10]</b>	<b>[F10]</b>	<b>[F11]</b>	<b>[Insert Here]</b>
INTERRUPT	<b>[Shift]-[F20]</b>	<b>[Shift]-[Delete]</b>	<b>[Alt]-[I]</b>	<b>[1]</b> †	<b>[Shift]-[F20]</b>
NO PAUSE	<b>[Shift]-[F18]</b>	<b>[Shift]-* †</b>	<b>[Alt]-[P]</b>	<b>[8]</b> †	<b>[Shift]-[F18]</b>

† Numeric-keypad key



## Format for Subroutines

Stratus manuals use the following format conventions for documenting subroutines. Note that the subroutine descriptions do not necessarily include each of the following sections.

### **subroutine\_name**

The name of the subroutine is at the top of the first page of the subroutine description.

### **Purpose**

Explains briefly what the subroutine does.

### **Usage**

Shows how to declare the variables passed as arguments to the subroutine, declare the subroutine entry in a program, and call the subroutine.

### **Arguments**

Describes the subroutine arguments.

### **Explanation**

Provides information about how to use the subroutine.

### **Error Codes**

Explains some error codes that the subroutine can return.

### **Examples**

Illustrates uses of the subroutine or provides sample input to and output from the subroutine.

### **Related Information**

Refers you to other subroutines and commands similar to or useful with this subroutine.

## Online Documentation

Stratus provides the following types of online documentation.

- The directory `>system>doc` provides supplemental online documentation, including updates and corrections to Stratus manuals and a glossary of terms.
- The VOS StrataDOC Web site is an online-documentation service provided by Stratus. It enables Stratus customers to view, search, download, print, and comment on VOS technical manuals via a common Web browser. It also provides the latest updates and corrections available on the VOS document set.

If you are a Stratus customer with a current support contract, you can access the VOS StrataDOC Web site, at no charge, at <http://stratadoc.stratus.com>. You can also order the VOS StrataDOC CD-ROM from Stratus.

This manual is available on the VOS StrataDOC Web site.

For information about accessing the VOS StrataDOC Web site, contact the Stratus Customer Assistance Center (CAC). For information about ordering the VOS StrataDOC CD-ROM, see the next section, "Ordering Manuals."

## Ordering Manuals

You can order manuals in the following ways.

- If your system is connected to the Remote Service Network (RSN™), issue the `maint_request` command at the system prompt. Complete the on-screen form with all of the information necessary to process your manual order.
- Customers in North America can call the Stratus Customer Assistance Center (CAC) at (800) 221-6588 or (800) 828-8513, 24 hours a day, 7 days a week. All other customers can contact their nearest Stratus sales office, CAC office, or distributor; see the file `cac_phones.doc` in the directory `>system>doc` for CAC phone numbers outside the U.S.

Manual orders will be forwarded to Order Administration.

## Commenting on This Manual

You can comment on this manual by using the command `comment_on_manual` or by completing the customer survey that appears at the end of this manual. To use the `comment_on_manual` command, your system must be connected to the RSN. If your system is **not** connected to the RSN, you must use the customer survey to comment on this manual.

The `comment_on_manual` command is documented in the manual *VOS System Administration: Administering and Customizing a System* (R281) and the *VOS Commands Reference Manual* (R098). There are two ways you can use this command to send your comments.

- If your comments are brief, type `comment_on_manual`, press `[Enter]` or `[Return]`, and complete the data-entry form that appears on your screen. When you have completed the form, press `[Enter]`.
- If your comments are lengthy, save them in a file before you issue the command. Type `comment_on_manual` followed by `-form`, then press `[Enter]` or `[Return]`. Enter this manual's part number, R194, then enter the name of your comments file in the `-comments_path` field. Press the key that performs the `CYCLE` function to change the value of `-use_form` to `no` and then press `[Enter]`.

### NOTE \_\_\_\_\_

If `comment_on_manual` does not accept the part number of this manual (which may occur if the manual is not yet

registered in the `manual_info.table` file), you can use the `mail request` of the `maint_request` command to send your comments.

Your comments (along with your name) are sent to Stratus over the RSN.

Stratus welcomes any corrections and suggestions for improving this manual.



---

# Chapter 1

## Overview of Window Terminal Communications

This chapter describes various aspects of window terminal asynchronous communications. It covers the following topics.

- Window terminal driver and the standard asynchronous driver
- Setup procedure for the window terminal driver
- Features of the window terminal driver
- Architecture of the window terminal driver
- Application program interface
- Application design considerations
- Compiling, binding, and debugging considerations
- RS-232-C asynchronous device interface

Note that the standard asynchronous driver was the original driver for the VOS windows implementation. It has been superseded by the window terminal driver. Hereafter in this manual the standard asynchronous driver is referred to as the “old asynchronous driver.”

### Window Terminal and Old Asynchronous Drivers

Stratus offers two device drivers for handling asynchronous communications: the window terminal driver and the old asynchronous driver. The *window terminal driver* is bound in the VOS kernel and offers a layered architecture and establishes a window environment for terminal devices communicating with a Stratus module. The VOS spooler facility recognizes printers configured as window terminal devices, which means that the window terminal driver can support printer connections as well as terminal connections.

In its basic configuration, the window terminal driver supports character-mode terminals (for example, video display terminals) and printers that are connected to Stratus modules using the *RS-232-C interface*, which is the communications medium established by the Electronic Industries Association (EIA) for physical asynchronous devices. However, the driver’s layered architecture also enables it to support other

types of communications media that perform terminal I/O (such as Stratus OS TELNET communications) while maintaining consistent user and programming interfaces.

- Unlike the window terminal driver, the *old asynchronous driver* is the original Stratus asynchronous driver and handles basic RS-232-C asynchronous communications only. It accommodates the needs of character-mode terminals, printers, and other asynchronous devices that are directly connected to Stratus modules using the RS-232-C interface. The old asynchronous driver lacks many of the features of the window terminal driver, and its single-layer architecture combines the code associated with character-mode terminals with the code associated with the RS-232-C communications medium. The old asynchronous driver does not support the Console Controller access layer. For information about the Console Controller, see the *VOS Continuum 600 and 1200 Series with PA-8000: Operation and Maintenance Guide* (R445).

## Setup Procedure for the Window Terminal Driver

The VOS tape included with each release contains both the RS-232-C-based window terminal driver and the old asynchronous driver. The window terminal driver, device-interface driver, the Console Controller access layer, and the old asynchronous driver are bound into the operating-system kernel.

As part of the window terminal driver setup procedure, the system administrator must also edit the device configuration file (`devices.tin`) to include entries for every terminal, printer, and connection that will use the window terminal driver. The *Product Configuration Bulletin: Asynchronous Devices* (R289) describes how to create window terminal entries for terminals and printers, discusses the various software and hardware requirements for the window terminal driver, and explains how to control the allocation of internal memory. The manual *VOS System Administration: Configuring a System* (R287) provides an overview of device configuration and other types of system configuration.

The remainder of this manual describes the window terminal driver. For detailed information about the old asynchronous driver, see the manual *VOS Communications Software: Asynchronous Communications* (R025). See [Appendix A](#) for information about the compatibility issues involved when migrating an old asynchronous driver application program to the window terminal environment.

## Features of the Window Terminal Driver

The window terminal driver offers the following features:

- a layered architecture that separates the application program interface and terminal code from the code associated with the communications medium. This enables the driver to support different types of communications media without changing the user interface or the application programming interface.
- a flexible application programming interface that supports various types of application design, such as simple sequential I/O (line-mode I/O) and raw I/O.
- windowing, which allows one terminal to be multiplexed among several processes.
- the processes may be cooperating or independent.
- various levels of terminal independence, as supported by the enhanced Stratus terminal-type database. This database consists of compiled and installed terminal types (TTPs), where each TTP is derived from terminal-specific information in a terminal-type definition (`.ttp`) file. By relying on the information in an installed TTP, your application program is device independent and is therefore able to work with various types of terminals.
- full National Language Support (NLS), which enables the use of multiple character sets in any text string for both sequential I/O and forms I/O.
- input flow control, output flow control, and bidirectional flow control for devices connected to the module through the RS-232-C asynchronous interface.
- support for K-Series hardware through the RS-232-C asynchronous interface. (For information about K-Series hardware, see the section “[RS-232-C Communications I/O Hardware](#)” later in this chapter.)
- support for direct, dial-up, and dial-out RS-232-C connections.

## Architecture of the Window Terminal Driver

The window terminal driver consists of terminal-interface layers and access layers. This layered architecture separates the code associated with character-mode terminals from the code associated with the communications medium. With this architecture, each layer can work and be maintained independently.

[Figure 1-1](#) shows the architecture of the window terminal driver.

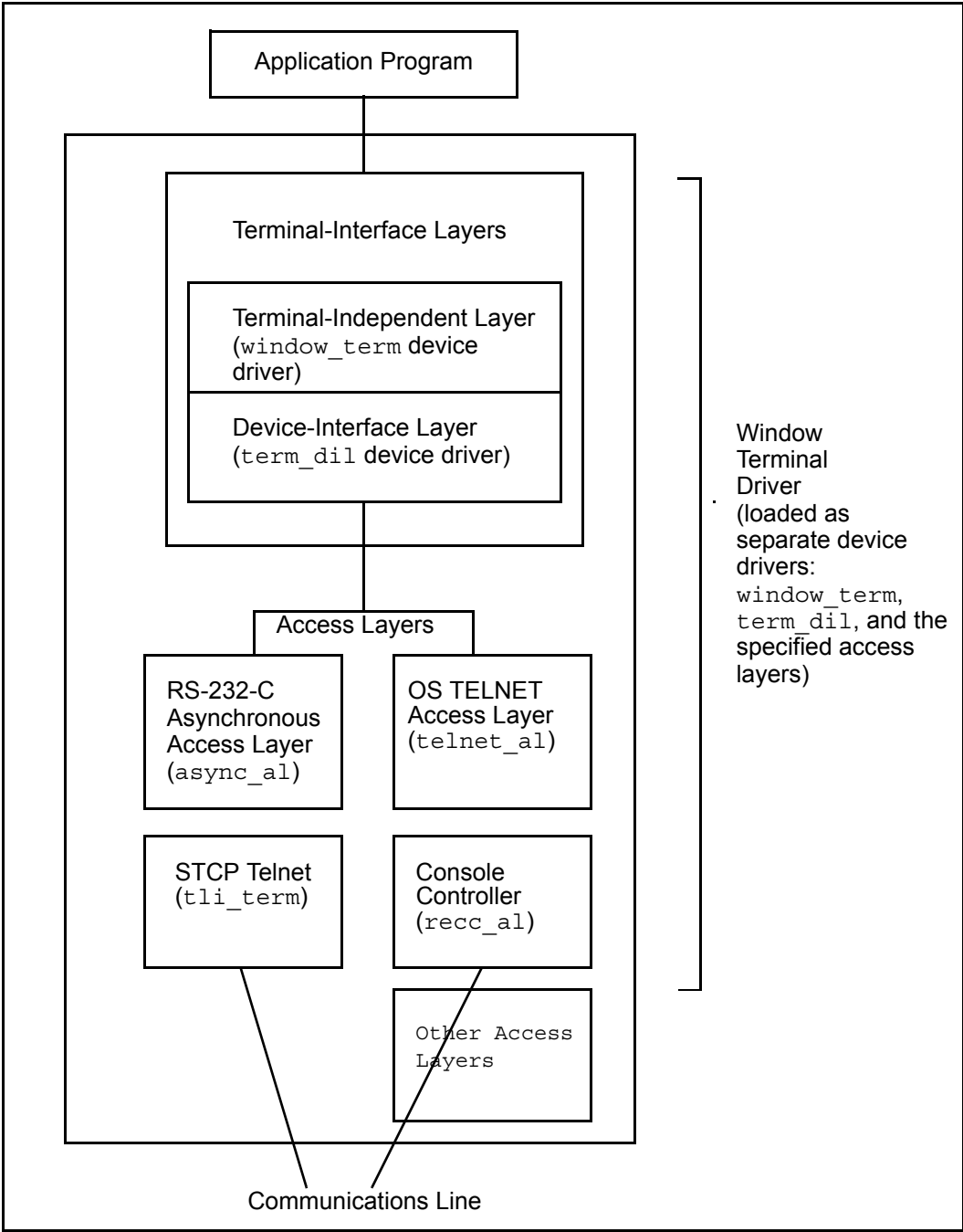


Figure 1-1. Architecture of the Window Terminal Driver



## Terminal-Interface Layers

The terminal-interface layers of the window terminal driver provide the application program interface. They support all character-mode terminals, regardless of the physical connection used, both as login and slave devices.

To separate application-specific tasks from terminal-specific tasks, there are two terminal-interface layers: the terminal-independent layer and the device-interface layer.

The terminal-independent layer handles requests from the application program without concern for the type of terminal being used. This layer is the `window_term` device driver.

The device-interface layer, on the other hand, requires information about the terminal or printer in order to perform input and output translation. The device-interface layer relies on information provided in the TTP for the device, which describes the input and output sequences used by the terminal or printer. This layer is the `term_dil` device driver.

## Access Layers

Each access layer of the window terminal driver provides the interface to a particular communications medium. The access layers supported by the window terminal driver use the same basic user and application program interface, which allows window terminal users to take advantage of various types of communications media without observing differences in user- or application-level functions.

In general, each access layer frees the terminal-interface layers from the requirements of a particular communications medium, and contains all medium-specific code as well as the operating system's call-side and interrupt code. The access layer also incorporates the communications I/O hardware subsystem, which establishes the physical connection between the module and a terminal, printer, or network device associated with a particular communications medium.

With the exception of Console Controller access layer, an access layer must be explicitly loaded and each device must be configured to specify the chosen access layer. When writing your application program, keep in mind that each access layer has its own set of medium-specific characteristics. This includes support for 8-bit data transmission and parity options.

For example, most of the `s$control` opcodes that affect the RS-232-C communications medium (including the opcodes for parity, bits per character, and flow control, as documented in [Chapter 8](#)) apply only to the asynchronous access layer and are ignored by access layers such as OS TELNET. Access layers such as OS TELNET have different requirements (for example, Ethernet network and OS TCP/IP software requirements) and use a different hardware interface (an Ethernet adapter) to provide

the connection to the network. The opcodes that generally apply to all communications media are documented in the section “Opcodes Affecting All Communications Media” in [Chapter 8](#).

In this manual, all information on the device interface, operating characteristics, and operations affecting the communications medium applies to the RS-232-C interface, as governed by the asynchronous access layer. Unless otherwise noted, all information on the window environment and the application program’s interface to the window environment applies to all window terminal connections, regardless of the access layer used.

## Asynchronous Access Layer

To handle asynchronous communications, the basic window terminal driver configuration includes the *asynchronous access layer*. This layer (called `async_al`) provides a standard interface to the RS-232-C communications medium. The asynchronous access layer therefore contains all code that is specific to RS-232-C.

The asynchronous access layer is designed to provide generic communications services that map specific events into generic terms. For example, the loss of the data set ready (DSR) signal from a modem is mapped into a generic disconnect event.

The asynchronous access layer offers several programming benefits.

- **Protocol support.** The asynchronous access layer enables an application program to read packets of information using automatic frame detection. Because the application program can specify how frames end, some protocols using accepted framing techniques (for example, bisynchronous-like protocols) can be supported.
- **A full 8-bit data path with parity options.** The asynchronous access layer supports a full 8-bit data path between the application program and the device. Devices transmitting characters of eight data bits can use even parity, odd parity, or no parity. Note that this feature is available with K-series communications hardware, which is described in the section “[RS-232-C Communications I/O Hardware](#)” later in this chapter.
- **Input, output, and bidirectional flow control.** The asynchronous access layer allows an application program to use either XON/XOFF flow control or data set lead (DSL) flow control to control the flow of input to the module, output to the device, or both. (For more information about flow control, see the section “[Using Flow Control](#)” later in this chapter and the description of the `s$control` subroutine in [Chapter 8](#).)

## Console Controller Access Layer

The Console Controller access layer supports the same set of basic window terminal features as the *asynchronous access layer* with the exception that the line cannot be configured on the Console port. This layer (called `recc_al`) provides a standard

interface to the RS-232-C communications medium and contains all code that is specific to RS-232-C.

The Console Controller access layer is designed to provide generic communications services that map specific events into generic terms. For example, the loss of the data set ready (DSR) signal from a modem is mapped into a generic disconnect event.

The Console Controller access layer offers several programming benefits.

- **Protocol support.** The asynchronous access layer enables an application program to read packets of information using automatic frame detection. Because the application program can specify how frames end, some protocols using accepted framing techniques (for example, bisynchronous-like protocols) can be supported.
- **A full 8-bit data path with parity options.** The asynchronous access layer supports a full 8-bit data path between the application program and the device. Devices transmitting characters of eight data bits can use even parity, odd parity, or no parity. Note that this feature is also available with K-series communications hardware, which is described in the section “[RS-232-C Communications I/O Hardware](#)” later in this chapter.
- **Input, output, and bidirectional flow control.** The asynchronous access layer allows an application program to use either XON/XOFF flow control or data set lead (DSL) flow control to control the flow of input to the module, output to the device, or both. (For more information about flow control, see the section “[Using Flow Control](#)” later in this chapter and the description of the `s$control` subroutine in [Chapter 8](#).)

For detailed information about the Console Controller, see the *VOS Continuum 600 and 1200 Series with PA-8000: Operation and Maintenance Guide* (R445).

## OS TELNET Access Layer

The OS TCP/IP product set supports the OS TELNET access layer, which works with the window terminal driver to provide TELNET protocol connections on an Ethernet-based local area network (LAN). The OS TELNET access layer supports three types of connections:

- incoming remote login connections, to enable remote users on the network to log in to a system running Stratus OS TCP/IP software
- incoming slave connections, in which an active application program is waiting for a slave connection
- outgoing (VOS-initiated) slave connections, in which the access layer for the application program actively requests a connection to a specified address on the network.

The OS TELNET access layer observes the TELNET protocol, a standard TCP/IP utility that allows remote users on the network to log in to and use an OS TCP/IP system as if they were directly connected to that system. This access layer provides remote TELNET users with the same window terminal user interface used by local window terminal users, including the standard input-line editing functions. The OS TELNET access layer is called `telnet_al`.

For information about the software, hardware, and configuration requirements of the OS TELNET access layer and the OS TCP/IP software, see the *VOS OS TCP/IP Administrator's Guide* (R223). The OS TCP/IP product set also includes the *VOS Communications Software: OS TCP/IP User's Guide* (R222) and the *VOS Communications Software: OS TCP/IP Programmer's Manual* (R224).

## The STCP Access Layer

The STCP TELNET access layer works with the window terminal driver to provide TELNET protocol connections on an Ethernet-based LAN. STCP TELNET provides the same programming interface characteristics as OS TELNET access layer; however, it is administered differently.

The STCP TELNET access layer supports three types of connections:

- incoming remote login connections, to enable remote users on the network to log in to a system running Stratus STCP software
- incoming slave connections, in which an active application program is waiting for a slave connection
- outgoing (VOS-initiated) slave connections, in which the access layer for the application program actively requests a connection to a specified address on the network.

The STCP TELNET access layer observes the TELNET protocol, which allows remote users on the network to log in to and use an STCP/IP system as if they were directly connected to that system. This access layer provides remote TELNET users with the same window terminal user interface used by local window terminal users, including the standard input-line editing functions. The STCP TELNET access layer is called `tli_term`.

For information about the software, hardware, and configuration requirements of the STCP TELNET access layer and the STCP software, see the *VOS STREAMS TCP/IP Administrator's Guide* (R419). The STCP product set also includes the *VOS STREAMS TCP/IP Programmer's Guide* (R420), *VOS STREAMS TCP/IP User's Guide* (R421), and the *VOS STREAMS TCP/IP Migration Guide* (R418).

The remainder of this chapter provides an overview of the application program interface and the RS-232-C asynchronous device interface.

## Application Program Interface

All window terminal application programs designed to run on a Stratus module must use a set of VOS subroutines in order to communicate with the terminal-interface layers of the window terminal driver. These subroutines can be grouped according to function, as shown in [Table 1-1](#).

**Table 1-1. Subroutines That Communicate with the Window Terminal Driver**

Function	Subroutines
Attaching and opening a port (Chapter 6)	<code>s\$attach_port</code> , <code>s\$open</code> , <code>s\$seq_open</code>
Performing input and output operations <sup>†</sup> (Chapter 7)	<code>s\$read_raw</code> , <code>s\$seq_read</code> , <code>s\$write_raw</code> , <code>s\$seq_write</code> , <code>s\$seq_write_partial</code>
Performing control operations (Chapter 8)	<code>s\$set_io_time_limit</code> , <code>s\$set_wait_mode</code> , <code>s\$set_no_wait_mode</code> , <code>s\$read_device_event</code> , <code>s\$control</code> , <code>s\$control_device</code>
Closing and detaching a port (Chapter 9)	<code>s\$close</code> , <code>s\$detach_port</code>

<sup>†</sup> Input and output operations can also be performed by subroutines such as `s$read`, `s$read_code`, `s$write`, and `s$write_code`. See Chapter 7 for more information about using these subroutines.

As indicated by the grouping of the subroutines, you typically design your application program to perform the following tasks.

- Attach and open a port to establish communication with the device.
- Specify control information such as the use of wait mode or no-wait mode, time limits, and basic operating information (for example, the use of an input mode, TTP input section or configuration section, or input and output flow control).
- Retrieve input using one of the read subroutines.
- Send output using one of the write subroutines.
- Close the port and detach the port from the device.

Chapters 2 through 5 provide detailed information about the different ways your application program and the window terminal driver handle input and output. Chapters 6 through 9 provide the subroutine-reference information that you will need to perform the various tasks.

## Application Design Considerations

When designing your application program to work with the window terminal driver, you must determine how and when to use the subroutines. How you use the subroutines depends on the following criteria:

- your choice of a programming approach that best describes the functions you want your application program to perform, how much input translation you need when retrieving input, and how much you want to control the contents of the screen display
- how you want to organize the window environment (primary windows and subwindows)
- whether you want to take advantage of National Language Support (NLS)
- whether you want to use input flow control, output flow control, or both (if you are using the RS-232-C asynchronous interface)
- whether you want to use wait mode or no-wait mode
- which hardware features you want to use (for example, a K-series interrupt table)
- whether you want to take advantage of the driver include files shipped with the operating-system software.

This section addresses each of these design considerations.

## Selecting a Programming Approach

Before you can begin writing your application program, you must determine which type of input and which type of output will best utilize the functions you want the application program to perform. Often, the combination of input and output characterizes a particular programming style. This section introduces the types of input and output supported by the window terminal driver and then describes some commonly used input/output combinations.

The window terminal driver supports four types of input.

- **Simple sequential input.** This type of input is in effect at operating-system command level. It provides terminal-independent line editing, line wrapping, typeahead processing, automatic character echoing, and NLS support.
- **Formatted sequential input.** This type of input provides automatic character echoing and line editing when a read operation is active, but does not process any

typeahead (input entered before a read operation). It also allows your application program to position the input line anywhere in the display of the subwindow (described later in this chapter). Formatted sequential input is available when the subwindow is in a window mode called *formatted I/O mode*.

- **Unprocessed raw input.** This type of input allows your application program to retrieve untranslated input. Your application program can control the flow of input by specifying interrupt characters or by imposing a record format on the input.
- **Processed raw input.** This type of input allows your application program to choose one of three levels of input mapping and to perform its own character echoing. Emacs is an example of an application program that uses processed raw input.

The window terminal driver supports three types of output.

- **Simple sequential output.** This type of output performs automatic pause processing and display management on behalf of your application program. Your application program can perform a subset of generic screen-display functions and can send NLS characters.
- **Formatted sequential output.** This type of output is available in formatted I/O mode and allows your application program to control the screen display. Your application program can perform a large set of generic screen-display functions and send NLS characters. Since pause processing is not performed automatically, your application program can design and control its own pause processing. The `list_users` command (a VOS application program) uses formatted sequential output.
- **Raw output.** This type of output allows your application program to send untranslated and unformatted output. It is not suitable for application programs that want to take advantage of the window environment.

#### NOTE \_\_\_\_\_

Although it is permitted, you should avoid designing your application program to process either type of sequential input with raw output. This combination of input and output can cause problems with the display of data. If you decide to retrieve sequential input (with `s$seq_read`) and send raw output (with `s$write_raw`), be aware that `s$write_raw` invalidates the state of the screen display while `s$seq_read` maintains the screen display and attempts to restore it after it has been invalidated by `s$write_raw`. In switching between raw output and sequential input, your application program may prevent the window terminal driver from displaying the appropriate

line-editing operation. In general, raw output is best combined with raw input.

The terminal-interface layers of the window terminal driver support several common application programming approaches that combine the various types of input and output. These approaches offer different ways of handling devices that are connected to the module over an asynchronous communications line.

The following sections briefly describe the common programming approaches. See Chapters 2 through 4 for a detailed description of each approach.

### Simple Sequential I/O

This approach is suitable for an interactive application program that simply reads/writes records of text to a terminal while taking advantage of input-character echoing, input-line editing, typeahead processing (where unread input is visible), and the terminal's screen-display functions. This approach, sometimes referred to as line-mode I/O, is used at operating-system command level. To use this approach, your application program must simply retrieve input with `s$seq_read` and send simple sequential output with `s$seq_write` or `s$seq_write_partial`.

### Unprocessed Raw I/O

This approach is suitable for a noninteractive application program that requires binary I/O processing and controls the communications line as a simple communications device. With this approach, raw bytes are transmitted without being translated. Your application program can pace (control) the flow of input and/or group input by specifying which bytes cause interrupts and which bytes terminate records. To use this approach, your application program must specify an unprocessed raw input mode with the `s$control` opcode `TERM_SET_INPUT_MODE (2057)` before calling `s$read_raw`, and must send raw output using `s$write_raw`.

### Application-Managed I/O

This approach is suitable for an interactive application program that manages the contents and organization of the subwindow display. With this approach, your application program must use formatted I/O mode to control the processing of sequential output or input. (All output in this mode is formatted sequential output.)

Since this approach does not provide automatic line wrapping or pause processing, your application program has more flexibility and can perform its own line wrapping or pause processing if desired. Your application program can use formatted sequential output with formatted sequential input or any type of raw input. To give your application program maximum control, retrieve **processed** raw input and send formatted sequential output. Processed raw input performs different levels of input mapping (for example, the handling of function keys and generic input requests) and requires your application program to perform its own character echoing.



The formatted sequential input/output combination provides echoing and line-editing support when a read operation is active, but does not process any typeahead. (It also allows your application program to position the input line anywhere in the display of the subwindow.) To use application-managed I/O, your application program must activate formatted I/O mode, retrieve input with `s$read_raw` or `s$seq_read`, and send output with `s$seq_write` or `s$seq_write_partial`.

Application programs can also be designed to handle forms I/O. Forms I/O is suitable for an interactive application program that allows the terminal user to enter input in a display form. Forms I/O is part of the VOS FMS software; it requires you to use the `accept` or `screen` statement in your application program. (See the VOS Forms Management System manuals for more information about using forms I/O.)

In designing your application program, you should also consider certain features of the terminal-interface layers. Three major features of the terminal-interface layers are the implementation of primary windows/subwindows, the support of international character sets, and the support of input, output, and bidirectional flow control for RS-232-C device connections. The next three sections describe these features.

## Working with Primary Windows and Subwindows

The Stratus window terminal driver creates a window environment for terminal users. A *primary window* is a section of the terminal screen that displays the output of a single process or program. When the terminal user logs in, the window terminal driver creates a primary window that comprises the entire terminal screen. Additional independent primary windows can be created by the terminal user or by the application program, and users can move among these windows to provide input to the respective processes or programs.

Terminal users manipulate primary windows by using the *window manager*, which is described in the *Window Terminal User's Guide* (R256). Application programs manipulate primary windows by using the `s$open` and `s$control` subroutines. (See [Chapter 6](#) for a description of `s$open`; see [Chapter 8](#) for a description of `s$control`.)

By default, each port attached to a process has its own primary window. Although a process typically attaches to a terminal only once, it can open the same terminal again to create an additional primary window. The primary window is created when a process opens the terminal, but remains invisible until the first I/O operation is performed. The primary window becomes invisible when it is closed. More than one port can become attached to a primary window as the result of one of the POSIX system calls `fork`, `dup`, `dup2` or the `TERM_OPEN_EXISTING_WINDOW_OPCODE (2107)`. Note that in the case of `fork` the ports have the same port ID, but they are distinct ports because they are in different processes. When more than one port refers to same primary window, the processes owning the ports must keep the screen updated properly. (For information about the opcode `TERM_OPEN_EXISTING_WINDOW_OPCODE (2107)`, see the description of opcode in [Chapter 8](#). For information about the POSIX system

calls `fork`, `dup`, or `dup2`, see the online doc file  
>system>doc>posix>posix.doc.)

Primary windows overlap completely; you can think of them as being ordered from top to bottom, where one primary window hides all other primary windows. The primary window that appears on the terminal screen is on top of the other primary windows and is the *current primary window*. Terminal input is sent to the process associated with the current primary window. The terminal user and the `s$open` and `s$control` subroutine calls can change which primary window is the current (top) primary window.

In addition to primary windows, the window terminal driver supports subwindows. A *subwindow* is a section of a primary window. Subwindows are controlled by the application program executing in the primary window.

The application program can create and arrange subwindows in many ways within a primary window. For example, it can create subwindows of various sizes that may or may not overlap. After creating its subwindows, the application program can rearrange them by changing their position within the primary window or by moving them above or below the other subwindows. (Note that moving a particular subwindow to the top or to the bottom is visually significant **only** if the subwindows overlap. If the subwindows do not overlap, moving one subwindow above or below the others does not affect the appearance of the subwindows.)

Regardless of how an application program arranges its subwindows, I/O is always sent to the subwindow that the application program designates as the *current I/O subwindow*. Note that if any part of the current I/O subwindow is overlapped when the application program issues a read, the window terminal driver automatically moves the current I/O subwindow on top of the overlapping subwindows.

A primary window can have three types of subwindows: simple sequential I/O, formatted sequential I/O, and forms. When the port is opened, the first subwindow created is always a simple sequential I/O subwindow. This subwindow is the *original subwindow*. Each primary window can have only one simple sequential I/O subwindow.

The application program can explicitly create formatted I/O subwindows by using an `s$control` opcode, or it can use another `s$control` opcode to switch the original subwindow to formatted I/O mode. The application program can use the FMS software to implicitly create forms subwindows. In addition, the application program can use `s$control` opcodes to resize the subwindows, delete the subwindows (except the original subwindow), change which subwindow is the current I/O subwindow, and change which subwindow is the top or bottom subwindow.

When the application program manipulates a subwindow, it must provide a *subwindow ID*. All subwindows have a subwindow ID. The original subwindow always has a subwindow ID of 0. Each time the application program creates a subwindow using `s$control`, the window terminal driver returns that subwindow's ID.

Figure 1-2 illustrates a sample primary window and subwindows. In this figure, assume that the terminal user has created four primary windows. The primary window that is logically on top of the other primary windows is the current primary window. As indicated, this primary window contains four subwindows. Since these subwindows overlap, the top subwindow appears to be above the other subwindows. In this arrangement, assume that the application program executing in the primary window has designated the top subwindow as the current I/O subwindow. In addition, note that the bottom subwindow is the same size as the primary window.

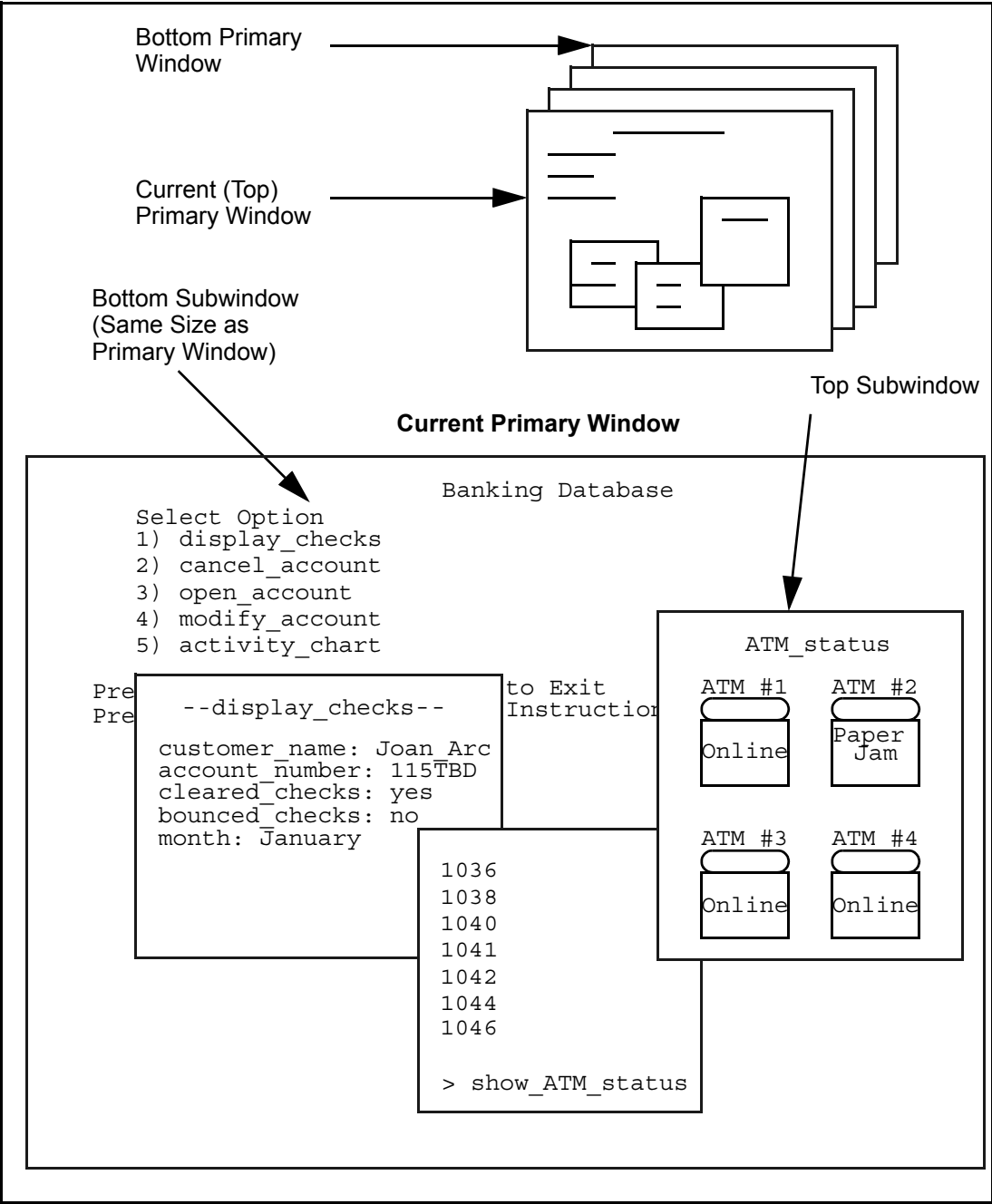


Figure 1-2. Sample Primary Window and Subwindows

When multiple ports from different processes are attached to the same primary window, it can be difficult to coordinate screen updates between the various processes. Subwindows make this much easier. The application program can assign subwindows to specific ports and the window terminal driver will manage the screen updates using the subwindows. Subwindows are owned by the ports that created them and may not be modified by the other ports. The current I/O subwindow is per port (and not per primary window).

Subwindows also allow sharing of a primary window between FMS and non-FMS applications (from the same process or from multiple cooperating processes). FMS creates its own subwindows to run in. Therefore, the sharing of a primary window between FMS and non-FMS applications is more or less automatic. The subroutine `s$fms_set_master_bounds` allows the location and the size of the FMS master form to be set. The FMS master form uses the entire screen by default.

## Using National Language Support

National Language Support (NLS) enables you to develop application programs that handle several different languages and alphabets simultaneously, as well as display text on terminals in the native language of the terminal user. This feature enables application programs to handle two Japanese character sets (kanji and katakana), a Korean character set (hangul), the simplified Chinese character set, parts 1 and 2 of the Chinese graphic character set, a character set from western Europe (Latin alphabet No. 1), and the standard ASCII character set.

Application programs can manipulate text in the appropriate language and perform operations on both single-byte and double-byte character sets. The kanji and hangul character sets, for example, require two bytes per character instead of one byte per character.

All character sets are defined by the operating system's *internal character coding system* as a matrix of hexadecimal codes. Each hexadecimal code represents a character. The left half of this matrix is the ASCII character set; the right half is a supplementary graphic set that can be Latin alphabet No. 1, kanji, katakana, hangul, or any of the Chinese character sets. ([Appendix B](#) presents the internal character coding system. [Appendix F](#) identifies the default internal character sets supported by the internal character coding system.)

The internal character coding system uses a *character-translation database* to translate input and output according to the native character set of the application program and terminal device. This database is created from information supplied in the character-translation section of the terminal type (TTP) associated with the device. A TTP defines a particular type of terminal to the system and defines the appropriate input keystroke mappings. Different sections of the TTP perform different mappings. The character-translation section of the TTP maps the character set used by the terminal to the internal character coding scheme used by the operating system.

For more information about TTPs, see the manual *VOS Communications Software: Defining a Terminal Type* (R096). For more information about the internal character coding system, see [Appendix B](#).

## Using Flow Control

Although flow control is just one of the operating parameters affecting RS-232-C communications between the terminal and the module, it is an important consideration for application programs using the asynchronous access layer. *Flow control* regulates the flow of data between the device and its communications partner (for example, the module). Regulating the flow of data with flow control prevents the communications partner from being overloaded with data.

The application program and the operating parameters set at the terminal determine the use of flow control. Unlike other operating parameters, flow control is not defined in the `devices.tin` file, which is an input file used to generate the device configuration table recognized by the operating system.

The window terminal driver supports the following types of flow control.

- **Input flow control.** This type of flow control regulates the speed at which input is sent from the device to the module by temporarily stopping and restarting the flow of input. Input flow control is used when a device (such as a personal computer) is sending the module more data than it can handle (for example, during a file transfer from the personal computer's asynchronous communications port). The window terminal driver supports input flow control for devices connected to the module using K-series hardware.
- **Output flow control.** This type of flow control regulates the speed at which output is sent from the module to the device by temporarily stopping and restarting the flow of output sent from the module. Output flow control is used when the module is sending more data than the attached device (for example, a V103 terminal) can handle. The window terminal driver supports output flow control for devices connected to the module using K-series hardware.
- **Bidirectional flow control.** This type of flow control is a combination of both input and output flow control. To use bidirectional flow control properly, you must follow the rules for both output and input flow control. The window terminal driver supports bidirectional flow control for devices connected to the module using K-series hardware.

### NOTE

The old asynchronous driver offers more limited flow-control support. The old asynchronous driver supports output flow control for terminals and printers connected to the module using K-series hardware, but it does **not** support input flow control for physical terminals

and printers. With the old asynchronous driver, input flow control is available only for virtual device connections (virtual terminals and virtual printers), which are governed by the X.25 and X.29 communications protocols. (See the manual *VOS Communications Software: X.25 and X.29 Programming* (R028) for more information about using virtual terminals and virtual printers with the X.25 protocol.)

These types of flow control are governed by certain flow-control protocols, often referred to as handshaking protocols. Of these protocols, the most common are *XON/XOFF flow control* and *data set lead (DSL) flow control*. DSL flow control is sometimes referred to as data terminal ready (DTR) flow control.

The window terminal driver allows your application program to use either XON/XOFF flow control, DSL flow control, or no flow control. Note that since flow control affects the communications line between the module and the device, it affects all of the windows associated with the line, not just the current window.

If your application program uses XON/XOFF flow control, two special characters, XOFF and XON, can temporarily stop and restart the flow of output and/or input. When the communications partner receives the XOFF character (usually ASCII `DC3`, which is often `CTRL-S`), it suspends output or input. The flow of output or input resumes upon receipt of the XON character (usually ASCII `DC1`, which is often `CTRL-Q`). (Although there are standard XON/XOFF characters, your application program can assign different XON/XOFF characters.)

If your application program uses DSL flow control (for **local** connections only), the window terminal driver automatically reconfigures the RS-232-C channel or subchannel associated with the device for a force-listen arrangement. In this arrangement, the loss of the DSR signal (the device drops the DTR signal) is interpreted as XOFF and the return of the DSR signal (the device raises the DTR signal) is interpreted as XON. (For example, when a modem drops the DSR signal on a dial-up line, the process is terminated.) DSL flow control is often used by printers that cannot manage the flow of output, although printers can use XON/XOFF flow control. (The VOS spooler facility controls only XON/XOFF flow control, not DSL flow control.)

XON/XOFF flow control is the more common flow-control protocol. To activate an input flow-control protocol and specify flow-control characters, your application program must use the `s$control` opcode `TERM_SET_INPUT_FLOW_INFO` (2008). To activate an output flow-control protocol and specify flow-control characters, your application program must use the opcode `TERM_SET_OUTPUT_FLOW_INFO` (2016).

**NOTE**

If you are using bidirectional flow control (both input and output flow control), you must specify the same flow-control information when using the `s$control` flow-control opcodes. Note also that you cannot mix DSL input flow control with XON/XOFF flow control. (For instructions on checking and specifying flow-control information, see the descriptions of the opcodes

`TERM_GET_INPUT_FLOW_INFO (2007)`,  
`TERM_SET_INPUT_FLOW_INFO (2008)`,  
`TERM_GET_OUTPUT_FLOW_INFO (2015)`, and  
`TERM_SET_OUTPUT_FLOW_INFO (2016)` in the section “Opcodes Affecting the RS-232-C Communications Medium” in [Chapter 8](#).)

The asynchronous and Console Controller access layers for the window terminal driver recognize a new `-rts_cts_flow_control` option. This option is specified in the device table “parameters” string. The `-rts_cts_flow_control` option overrides the XON/XOFF or DSL flow control mechanisms specified with the `s$control` calls. This option is for use with high-speed modems that require flow control to do reliable binary transfers. This option requires that an RTS/CTS cable be used to connect the modem.

## Using Wait Mode or No-Wait Mode

Another programming consideration is the use of wait mode or no-wait mode. By default, all ports initially operate in wait mode.

In *wait mode*, the operating system controls how and when calls return to your application program. Since your application program is not controlling the “waiting,” it cannot expect read or write calls to return immediately. The operating system waits (when necessary) and determines when the call can return. The operating system has basic rules for determining when a call can return. Typically, the particular subroutine (and/or the input mode in effect during read calls) determines how the operating system handles the call.

In order for your application program to explicitly control waiting, you must design it to specify no-wait mode with the subroutine `s$set_no_wait_mode`. As an alternative, you can design it to specify an I/O time limit with the subroutine `s$set_io_time_limit`. (The subroutine `s$set_io_time_limit` allows you to specify how long the operating system waits for an I/O subroutine call to complete before returning to your application program.)

With no-wait mode, your application program does not have to wait for the I/O operation to complete. Instead, the call returns to your application program immediately. Your



application program may then proceed to perform other tasks. When your application program receives the error code `e$caller_must_wait` and only when it receives the error code `e$caller_must_wait`, the event associated with the port will be notified after the operation completes. Your application program should wait on the event and reissue the same call to complete (or in some cases continue) the operation. For example, if your application program receives the error code `e$caller_must_wait` (1277) from a call to `s$read_raw`, the next call to the port must be from `s$read_raw`.

The device independent VOS no-wait mode model only allows one operation to be pending in no-wait mode. In other words, if you receive `e$caller_must_wait` on a device, you may only issue the I/O call that returned `e$caller_must_wait` or call `s$control` with the `ABORT (3)` opcode. If you make any other I/O call, the driver may reject it. If you are running in no-wait mode, you should issue the `ABORT (3)` opcode before closing. The window terminal driver relaxes the VOS rules by allowing one input and one output operation to be pending at the same time; therefore, you can do no-wait mode reads and writes at the same time.

The events for window terminal connections may be port specific. If you obtain a new port to a window terminal connection, you need to read the event (even if you already have the event for another port on the same primary window, do not assume it will be the same). Some window terminal ports do share the same event ID. This is done to maintain compatibility with older programs that predate the POSIX port changes but the rules are too complicated for applications to depend on.

In general, application programs using no-wait mode to retrieve any type of raw input or to send either type of output must be prepared to handle partial reads or writes and to loop until the entire read or write is complete. The recommended procedure is described in the `s$set_no_wait_mode` section in [Chapter 8](#).

## Using Hardware Features

You can also design your application program to take advantage of certain hardware features. This section briefly describes the hardware features that can influence the design of your application program.

### Using an Interrupt Table

In general, the K-series adapters supporting asynchronous devices process and then forward each character they receive, where each character generates an operating-system interrupt. (With RS-232-C communications through the asynchronous access layer, *interrupts* are control signals that cause the operating system to suspend its current processing temporarily.) In general, this mode of operation is not very efficient in terms of the module's processing time. One way that your application program can reduce the module's interrupt processing overhead is to implement its own interrupt table. An *interrupt table* reduces the number of interrupts handled by the module's processor and increases the module's performance by

specifying which characters will cause interrupts and which will not. However, note that using an interrupt table may increase the processing overhead on some adapters (for example, the K118 I/O adapter). Also note that other access layers may handle interrupts differently (for example, the OS TELNET access layer handles interrupts as event notifications to the user process, not as interrupts to the operating system).

The K-series line of hardware offers a full 8-bit data path between the application program and the device. When the line is configured to transmit and receive 8-bit characters, your application program can specify a 256-byte interrupt table that supports four types of interrupt characters. This gives your application program more flexibility in specifying which characters cause interrupts and how. (Two of the four types of interrupt characters allow you to delay the interrupt until the arrival of one or two checksum characters, which are referred to as trailer bytes because they trail the interrupt character. A *checksum character* is used by some communications protocols to verify the integrity of a packet or block of data transmitted over the communications line.)

To verify interrupt-table information, you must use the `s$control` opcode `TERM_GET_INTERRUPT_TABLE (2060)` in your application program. To have the access layer use a specific interrupt table, you must use the `s$control` opcode `TERM_SET_INTERRUPT_TABLE (2061)`. See [Chapter 3](#) for more information about using interrupt tables. See the section “Opcodes Affecting the Primary Window and Subwindow” in [Chapter 8](#) for a description of the opcodes `TERM_GET_INTERRUPT_TABLE (2060)` and `TERM_SET_INTERRUPT_TABLE (2061)`.

## Using Automatic Frame Detection

Another hardware feature available to your application program is automatic frame detection. *Automatic frame detection* is the recognition of blocks or packets of data that are packaged or “framed” with control information such as the packet origin and destination. This feature is suitable for RS-232-C device connections. K-series hardware supports automatic frame detection through raw record mode, which is described in [Chapter 3](#). Automatic frame detection supports protocols that use standard framing techniques, and is suitable for block-oriented protocols that terminate each block or packet with a unique character that does not appear elsewhere in the input stream.

## Using Break Handling

K-series hardware allows your application program to use *break handling*, the processing of special characters in the input stream to identify common RS-232-C asynchronous errors. The window terminal driver supports break handling for 7-bit characters and 8-bit characters. Break handling is supported for 7-bit or 8-bit character transmission by the K-series hardware. (The old asynchronous driver does not support break handling for 8-bit characters, regardless of the type of hardware.)

When break handling is in effect, characters in the range 80 through 86 hexadecimal are processed as metacharacters; some of these metacharacters return error codes to your application program to indicate common asynchronous errors, such as parity and frame errors. A *metacharacter* is a type of marker that the adapter places in the input stream to notify the driver of certain conditions (usually, error conditions). A metacharacter is not actual data. (For more information about the handling of metacharacters and asynchronous errors, see the error-code section of the `s$read_raw` and `s$seq_read` descriptions in [Chapter 7](#).)

## Using the Standard Include Files

Each release of the operating-system software provides various include files, some of which are specific to asynchronous communications and to the window terminal driver. In general, you should provide the appropriate definition for these include files in your application program instead of typing (and hardcoding) the definitions and structures from the include files directly in your application program. Using the most recently shipped include files ensures that your application program has access to the most recent changes to the window terminal driver.

[Table 1-2](#) lists and describes the include files used by the window terminal driver.

**Table 1-2. Available Include Files**

Include File	Description
<code>window_control_opcodes.incl.pl1</code> , <code>window_control_opcodes.incl.c</code>	These files list the available <code>s\$control</code> opcodes (see <a href="#">Chapter 8</a> ).
<code>window_term_info.incl.pl1</code> , <code>window_term_info.incl.c</code>	These files define the structures that your application program uses to pass control information (see <a href="#">Chapter 8</a> ).
<code>function_key_codes.incl.pl1</code> , <code>function_key_codes.incl.c</code>	These files define the function keys supported by the window terminal driver (see <a href="#">Chapter 2</a> ).
<code>video_request_defs.incl.pl1</code> , <code>video_request_defs.incl.c</code>	These files define the generic input requests supported by the window terminal driver (see <a href="#">Chapter 5</a> ).
<code>output_sequence_codes.incl.pl1</code> , <code>output_sequence_codes.incl.c</code>	These files define the generic output requests supported by the window terminal driver (see <a href="#">Chapter 5</a> ).

## Compiling, Binding, and Debugging Considerations

Once you finish designing and writing your application program, you must perform the following steps to ensure that the program can run on a Stratus module.

1. Compile the application program using the appropriate VOS language compiler. For information on how to use a compiler, see the appropriate language manual (for example, the *VOS PL/I Language Manual* (R009) or the *VOS Standard C Reference Manual* (R363)).
2. Bind the object module using the `bind` command, as described in the *VOS Commands Reference Manual* (R098).

Keep in mind that there are various options available to you when compiling (for example, specifying longmap rather than shortmap mapping rules). The window terminal driver generally uses longmap structures. The only shortmap structures that the window terminal driver uses are associated with the opcodes

`TERM_GET_PWIN_DEFAULTS` (2034), `TERM_SET_PWIN_DEFAULTS` (2035), `TERM_GET_PWIN_PARAMS` (2082), and `TERM_SET_PWIN_PARAMS` (2083). If you use these opcodes in your application program, their structures must contain shortmap directives. (These structures are defined in the system include files `window_term_info.incl.pl1` and `window_term_info.incl.c`, which are presented in [Chapter 8](#).)

As long as you design your application program to use the include file (or if you enter the structures **exactly** as they are shown in [Chapter 8](#)), it makes no difference which mapping rules you specify when compiling, even if you have an XA/R reduced instruction set computer (RISC) system. (For general information about porting an existing application program to the Stratus RISC environment, see the manual *Migrating VOS Applications to the Stratus RISC Architecture* (R288).) However, if you do not provide the appropriate definition for the include file associated with the structures, make sure that the structures are defined as shortmap structures in the application program.

You use the VOS debugger to debug your application program and to control the execution of your application program. You can also use the debugger to set breakpoints, view the contents of memory locations and registers, and execute the code one step at a time. See the *VOS Commands Reference Manual* (R098) for a description of the VOS debugger.

## RS-232-C Asynchronous Device Interface

Your application program uses the asynchronous access layer of the window terminal driver to establish RS-232-C asynchronous communication with a device.

Asynchronous communication is characterized by a start/stop transmission mode.

Each character transmitted in start/stop mode is preceded by a start bit and is followed by one or more stop bits. Since there is no clocking mechanism, the interval between

characters can vary. This method of transmitting data is designed to accommodate irregular transmissions (for example, from terminals).

The window terminal driver uses the asynchronous access layer to incorporate the standard asynchronous (start/stop) protocol and to provide the interface between system processes or application programs and asynchronous devices. While the terminal-interface layers of the window terminal driver handle communication with the application program, the asynchronous access layer handles the mechanics of asynchronous communication on behalf of the application program and enables the connection of asynchronous devices (such as character-mode terminals) to a module. The asynchronous access layer provides the interface to the device's native communications medium (RS-232-C). Therefore, the asynchronous access layer is responsible for handling the requirements of the medium as well as the transmission characteristics of the device associated with the medium.

The asynchronous access layer encompasses the communications I/O hardware subsystem, which provides the actual RS-232-C connections to asynchronous devices. The remainder of this chapter describes the options, characteristics, and requirements for making RS-232-C connections to asynchronous devices through the asynchronous access layer.

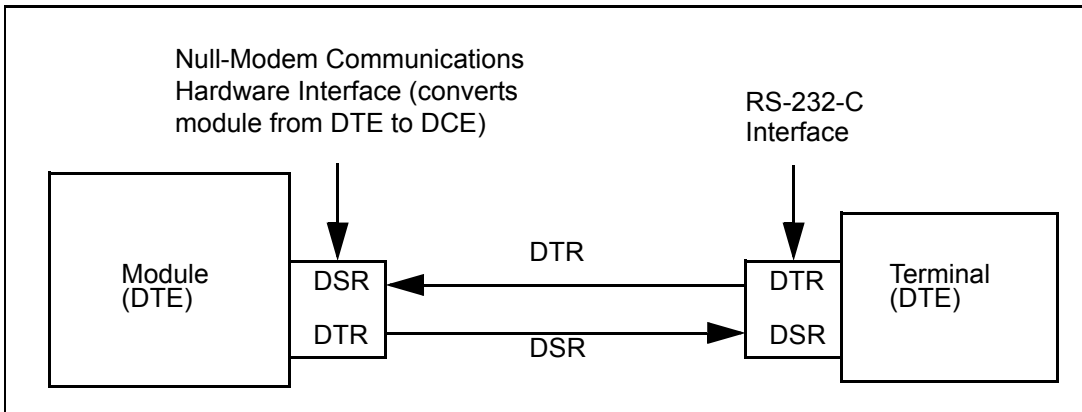
## Types of Asynchronous Connections

An asynchronous device such as a character-mode terminal connects to a Stratus module via an asynchronous communications line and the appropriate Stratus communications I/O hardware. The asynchronous device uses either a *direct connection* or a *modem connection*. The type of connection determines which hardware is used and affects how the device is configured for the module.

### Direct Connections

Direct connections are established by attaching the asynchronous device directly to the module; that is, the asynchronous line goes directly from a local terminal to the module (without a modem). In this arrangement, the terminal is often referred to as a hardwired terminal and the line associated with it is referred to as a direct-connect line. A direct-connect line requires the use of null-modem (direct-connect) hardware, which is built into the null-modem adapters. (See the section “[RS-232-C Communications I/O Hardware](#)” later in this chapter for information about these adapters.)

[Figure 1-3](#) illustrates a direct connection between a module and a terminal. As shown in this figure, the module and the terminal both function as data terminal equipment (DTE) devices. The switching between these DTE devices is handled by the module's null-modem hardware, which functions as a data communications equipment (DCE) device.



**Figure 1-3. A Direct Terminal Connection**

The module and the terminal establish communications by means of input and output signals that travel over a cable. While the module's communications hardware continually asserts (sends out) the data terminal ready (DTR) output signal, the terminal asserts its DTR output signal when powered up. To make the connection, the module's DTR output signal activates the terminal's data set ready (DSR) input signal, and the terminal's DTR output signal activates the module's DSR input signal.

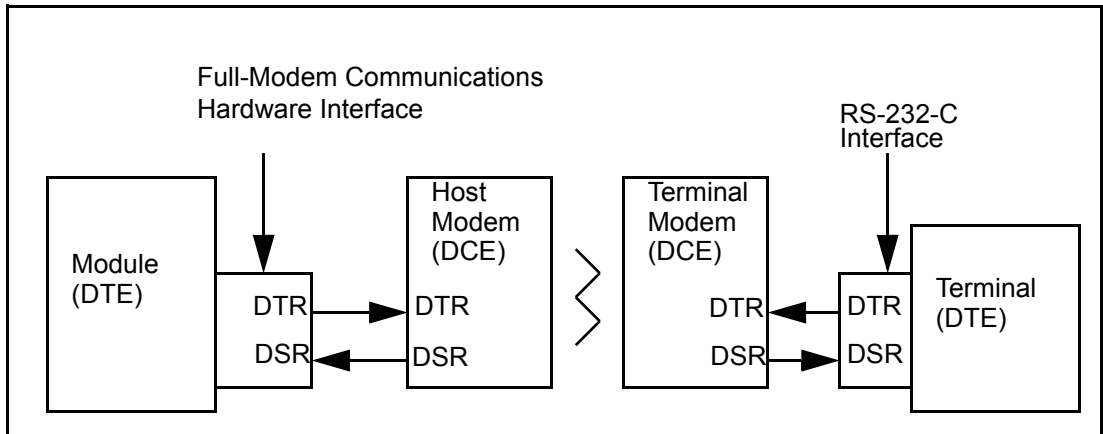
Note that the module's communications hardware asserts the DTR signal for either a login terminal or an open slave terminal. Although the data carrier detect (DCD) signal is not actually transmitted, it is always present.

## Modem Connections

Modem connections are established by attaching terminals (typically, remote terminals) to the module via modems. The asynchronous line between the module and the terminal uses two modems. One modem is on the module's end (the *host modem*) and one modem is on the terminal's end (the *terminal modem*). [Figure 1-4](#) illustrates this configuration. Note that the module and the terminal function as DTE devices, while the modems function as DCE devices. Modem connections require the use of full-modem communications hardware, with full-duplex modems only.

The module and the terminal establish communications via modems by means of the DTR, DSR, and DCD signals. While the module and the powered-up terminal assert the DTR signals, the modems assert the DSR signals. How the signals are asserted depends on the configuration of the line. Modem lines can be configured for dial-up, dial-out, or both. When configured for dial-up, the module's end of the line can accept calls from the terminal modem through a dial-in (auto-answer) modem. A drop in the DSR signal on a dial-up line breaks the connection immediately. A dial-up configuration typically exists between a login terminal and the module. When configured for dial-out,

the line can support more-advanced modem protocols that allow for both the dial-up capability and the dial-out capability. The dial-out capability is based on the CCITT V.25bis standard. Note that the line can be configured for dial-out when it is connected to a K-series I/O adapter.



**Figure 1-4. A Terminal Connection Using Modems**

When a line is configured for dial-up, a connection can be established through the following sequence of events. Note that this sequence of events may not apply to all types of modems.

1. The module asserts the DTR signal, which travels to the host modem.
2. When the terminal is powered up, the terminal's DTR signal travels to the terminal modem.
3. The terminal modem dials the host modem.
4. When the host modem receives the call, it asserts the DSR signal.
5. When both modems detect the presence of the carrier signal and assert the DCD signal, the connection is established.

The configuration of the terminal modem determines how the modem asserts the DSR signal. Typically, the terminal modem asserts the DSR signal when it is powered up or when the host modem answers the phone.

#### NOTE \_\_\_\_\_

This sequence of events does not address the request to send (RTS) and clear to send (CTS) signals, since these signals are not used to establish the connection.

However, the module must assert the RTS signal during Step 1, and the host modem must assert the CTS signal before Step 5.

When a dial-up line is disconnected, the sequence of events is the reverse of the login sequence of events.

1. The connection terminates (that is, the terminal is powered off). The terminal drops the DTR signal.
2. The terminal modem hangs up the line and drops the DCD signal. (Depending on how the terminal modem is configured, it may or may not drop the DSR signal.)
3. The host modem drops the DSR signal and the DCD signal.
4. The module drops the DTR signal.
5. The module reasserts the DTR signal after a minimum delay of .5 second. This occurs after the overseer process assigns a new prelogin process to the port, or after the application program hangs up the line and performs a listen operation. (As described in [Chapter 8](#), an application program hangs up the line using the `s$control opcode TERM_HANGUP (2001)`, and listens to the line using the `s$control opcode TERM_LISTEN (2023)`.)

The module does **not** reassert the DTR signal until the host modem drops the DSR and DCD signals to indicate that the connection has been terminated. Once the module reasserts the DTR signal, it is ready to receive another connection request from the host modem.

#### NOTE

If the terminal user logs out instead of powering off the terminal, Step 4 will occur first, then Step 3, then Step 2, and so on. In general, when the host modem drops the DSR and DCD signals, it may also drop the CTS signal. The module drops the RTS signal when it drops the DTR signal. The host modem **must** drop its DSR signal within three seconds after the module drops the DTR signal.

## RS-232-C Communications I/O Hardware

The K-series type of I/O hardware resides in Stratus modules and handles RS-232-C device connections. This type of hardware contains a controller board that resides in the main chassis of the module's equipment cabinet, an adapter chassis that connects to the controller board, and adapters that reside in the adapter chassis. (The adapters provide the device connections.)



The K-series handles disk and tape I/O as well as communications I/O. In general, the K-series offers increased performance. If your application program performs file transfers or message passing, you should use a multicom munications K-series adapter (such as the K101 I/O adapter). See the manual *VOS System Administration: Configuring a System* (R287) for configuration guidelines and detailed information about the components of this hardware line. This section briefly describes some of these components.

### **The K-series I/O Subsystem**

The K-series hardware line offers a complete I/O subsystem managed by an I/O processor. The I/O processor is a powerful controller that resides in the main chassis of the equipment cabinet with other full-size boards. K-series I/O processors typically operate in duplex (as a pair), which means that each pair must be installed in adjacent main-chassis slots in an even-odd sequence. (The first I/O processor in the pair must be in an even slot and the second I/O processor must be in the next higher-numbered odd slot.)

The I/O processor pair connects to a 16-slot I/O adapter chassis, which houses tape, disk, terminator, and communications I/O adapters. The I/O adapter chassis slots are numbered from right to left. The terminator adapters represent the end of the I/O adapter chassis and must reside in slots 14 and 15.

The communications I/O adapters provide a specified number of connections to communications lines, which connect to a specified number of devices (for example, asynchronous, synchronous, or Ethernet devices). The communications I/O adapters providing asynchronous device connections are called multicom munications I/O adapters.

There are four multicom munications I/O adapters.

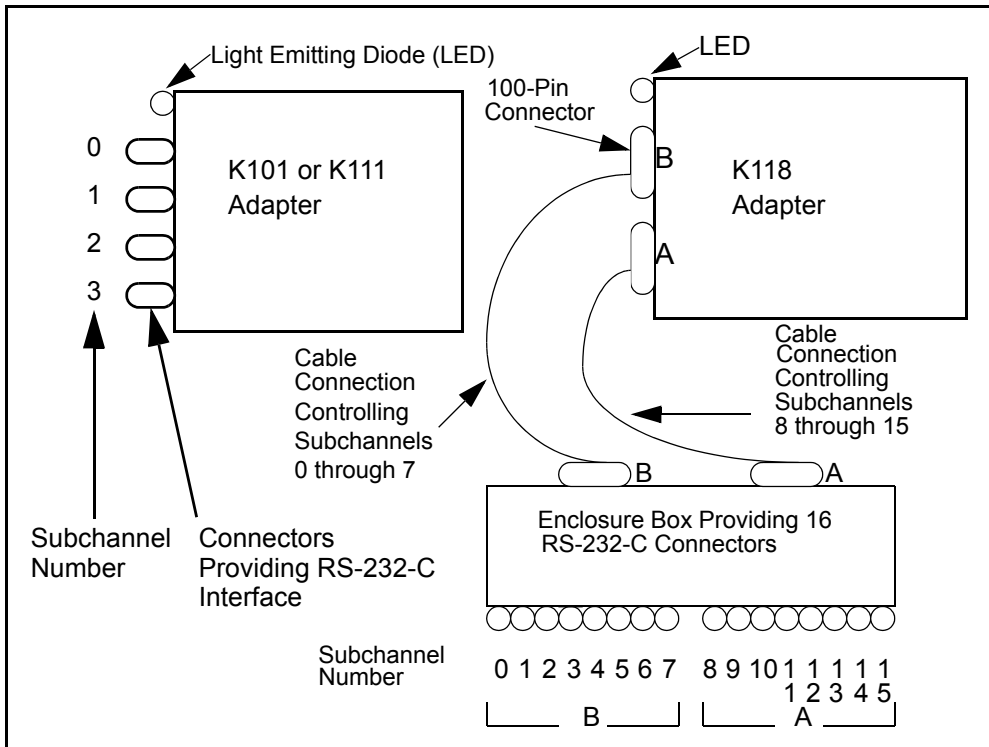
- K101 Full-Modem MultiCommunications I/O Adapter
- K111 Null-Modem MultiCommunications I/O Adapter
- K110 Twinaxial Printer Interface I/O Adapter
- K118 Asynchronous I/O Adapter

In addition to providing standard end-user device connections, either the K111 or K118 adapter can provide the monitor-terminal connection in each module. Modules requiring a calendar/clock and the Remote Service Network (RSN) connection can use either the K118 adapter or the K103 Remote Service Communications I/O Adapter. (The K103 adapter is a special-purpose adapter that provides a calendar/clock and handles both a monitor-terminal connection and an RSN connection. For information about monitor-terminal connections, RSN connections, and the K103 adapter, see the manual *VOS System Administration: Configuring a System* (R287).)

Each of the multicomcommunications I/O adapters provides a specific number of subchannels. An I/O adapter *subchannel* is a communications path generally associated with a particular connector (typically, DB-25), which is in turn associated with a particular electrical interface (in this case, EIA RS-232-C).

The system administrator (or privileged user) who configures asynchronous devices on the system must specify both the slot in which the I/O adapter resides and each subchannel used on that adapter. When configuring K118 subchannels, the system administrator should also specify whether each subchannel uses a full- or null-modem (direct-connect) configuration. (The system administrator configures asynchronous devices by creating device entries in the system configuration file `devices.tin`. See the *Product Configuration Bulletin: Asynchronous Devices* (R289) for more information about creating entries for asynchronous devices.)

The K101 and K111 adapters provide four subchannels; the K118 adapter provides 16 subchannels; the K110 adapter provides one subchannel. [Figure 1-5](#) illustrates how the connectors on the K101, K111, and K118 correspond to a number of subchannels and support the RS-232-C interface. A description of each of the adapters follows [Figure 1-5](#).



**Figure 1-5. Multicomunications I/O Adapter Subchannels**

### **K101 Full-Modem MultiCommunications I/O Adapter**

This full-modem I/O adapter supports four asynchronous lines and uses an RS-232-C interface. It can connect remote terminals to a module using a modem and a full-modem cable. It can also connect local terminals to a module using a modem-eliminator (crossover) cable. The maximum baud rate for asynchronous lines associated with terminal devices using paced input is 19,200 (9,600 for asynchronous lines associated with nonkeyboard devices using continuous input, such as scanners).

### **K111 Null-Modem MultiCommunications I/O Adapter**

This null-modem I/O adapter supports four asynchronous lines and uses an RS-232-C interface. It typically connects local terminals (or printers) to a module using a direct-connect cable. The maximum baud rate for asynchronous lines associated with terminal devices using paced input is 19,200 (9,600 for asynchronous lines associated with nonkeyboard devices using continuous input, such as scanners).

**K110 Twinaxial Printer Interface I/O Adapter**

This I/O adapter provides one asynchronous line for an L322 line printer. A twinaxial cable must connect to the serial-port connector on the adapter. The connection, which corresponds to subchannel 0, typically requires a 50-foot twinaxial printer-connect cable. For distances greater than 50 feet, the connection requires both a 1-foot cable and a twinaxial extension cable. The maximum baud rate for this adapter is 9,600.

**K118 Asynchronous I/O Adapter**

This I/O adapter supports 16 asynchronous lines that can be configured to produce any combination of full-modem and/or direct-connect lines. The adapter can also meet certain system requirements by providing the RSN line and monitor-terminal connections, as well as the system's calendar/clock. The maximum baud rate for asynchronous lines associated with terminal devices using paced input is 19,200 (9,600 for lines associated with nonkeyboard devices using continuous input, such as scanners).

As shown in [Figure 1-5](#), the K118 adapter has two 100-pin connectors that connect it to an enclosure box containing two 100-pin connectors and 16 (RS-232-C) connectors. The top 100-pin connector on the adapter is labeled B and uses a 12- or 25-foot cable to connect to the 100-pin connector labeled B on the back of the enclosure box; the bottom 100-pin connector on the adapter is labeled A and uses a 12- or 25-foot cable to connect to the 100-pin connector labeled A on the back of the enclosure box. Each 100-pin connector on the box manages a set of eight DB-25 connectors, and each set of connectors manages eight device subchannels. (Note that the adapter is available in various configurations, one of which provides the enclosure box described above. For more information about the various adapter configurations, contact your Stratus representative.)

Subchannels 0 through 7 correspond to the first eight connectors, which are managed by connector B. Subchannels 8 through 15 correspond to the second eight connectors, which are managed by connector A. (The front of the enclosure box labels the two sets of connectors with the corresponding subchannel numbers. In addition, the first set is labeled B to show that it is associated with connector B; the second set is labeled A to show that it is associated with connector A.)

Since each subchannel can use either a direct-connect or full-modem configuration, the adapter has specific configuration requirements. First, each subchannel must have a `devices.tin` file entry that specifies the appropriate subchannel configuration. In addition, the system administrator (or privileged user) must issue the command `configure_async_lines` to enable the adapter to initially recognize the configuration of each subchannel. (See the *Product Configuration Bulletin: Asynchronous Devices* (R289) for a description of the subchannel entries in `devices.tin`; see the manual *VOS System Administration: Configuring a System* (R287) for a description of the `configure_async_lines` command.)

## RS-232-C Configurations

All K-series adapters providing asynchronous connections support the EIA standard RS-232-C interface. RS-232-C is implemented through a 25-pin DB-25 male connector on each of the adapters. Although RS-232-C is always used by physical asynchronous devices, it is commonly used by synchronous devices as well (for example, devices running bisynchronous protocols). RS-232-C supports line speeds of up to 19,200.

The Stratus adapters providing RS-232-C asynchronous connections are usually classified as either full-modem or null-modem (direct-connect) adapters. This classification indicates that the RS-232-C channels or subchannels associated with the adapters behave in two different ways, depending on the type of connection they handle.

The following list identifies the full-modem and null-modem adapters that provide RS-232-C asynchronous connections.

Full-Modem Adapters	Null-Modem Adapters
K101	K111
K118 (configured in software)	K118 (configured in software)
K103 (bottom connector is for RSN)	K103 (top connector is for monitor terminal)

The K118 adapter is both a full-modem and a null-modem adapter; each of its 16 subchannels can be configured independently in the software. In addition, the K103 service adapter has one full-modem subchannel and one null-modem subchannel; however, the top connector is always configured in the hardware as the null-modem subchannel (for the monitor terminal on a K-series module), and the bottom connector is always configured in the hardware as the full-modem subchannel (for the RSN line on the system).

You can use these adapters in one of three configurations.

- **Strictly full modem.** The adapter or the adapter subchannel handling the connection is full modem, the cable is full modem, and the device to be connected to the module (for example, a modem) is full modem.
- **Full modem to null modem.** The adapter or the adapter subchannel handling the connection is full modem, the cable is a modem-eliminator (crossover) cable, and the device to be connected to the module (for example, a terminal or printer) is null modem.
- **Strictly null modem.** The adapter or the adapter subchannel handling the connection is null modem, the cable is null modem (direct connect), and the device

to be connected to the module is null modem. Null-modem adapters do **not** support full-modem devices, even with cable modifications.

When selecting a cable, you should follow these guidelines.

- You should use prepared or bulk cables from Stratus instead of cables from other vendors. Do **not** use cables that swap the transmitted data (TD) pinout to the received data (RD) pinout and the DSR pinout to the DTR pinout because the Stratus **adapter** swaps these pinouts. If you use cables that swap these pinouts, the cables may not work with the Stratus hardware.
- The cable must meet or exceed Stratus specifications, as defined in the *Site Planning Guide* (R003). A substandard cable is one of the most common sources of line noise, and your application program may have difficulty recovering data that is corrupted by line noise.

For a list of the available Stratus cables, see the manual *VOS System Administration: Configuring a System* (R287) or contact your Stratus representative. RS-232-C cables have from 4 to 15 wires and 25-pin DB-25 female connectors.

A null- or full-modem configuration determines which pins in the DB-25 connector interface are active.

[Table 1-3](#) identifies the active pins on a full-modem asynchronous channel or subchannel. [Table 1-4](#) identifies the active pins on a null-modem asynchronous channel or subchannel.

**Table 1-3. Active Pins on a Full-Modem Asynchronous Channel/Subchannel**

Pin	Name	Description
1	FG	Frame ground
2	TD	Transmitted data
3	RD	Received data
4	RTS	Request to send
5	CTS	Clear to send
6	DSR	Data set ready
7	SG	Signal ground
8	DCD	Data carrier detect
20	DTR	Data terminal ready

**Table 1-4. Active Pins on a Null-Modem Asynchronous Channel/Subchannel**

Pin	Name	Description
1	FG	Frame ground
2	TD	Transmitted data (from the remote device)
3	RD	Received data (to the device)
6	DSR	Data set ready (to the device)
7	SG	Signal ground
20	DTR	Data terminal ready (from the remote device)

The pin assignments indicate that a null-modem channel or subchannel loops back the asserted request to send (RTS) signal to assert the clear to send (CTS) signal and the data carrier detect (DCD) signal. These signals are not actually present at the interface connector, but are used within the adapter channel or subchannel to simulate (for the software) full-modem operation in a null-modem configuration. [Figure 1-6](#) illustrates how these signals operate.

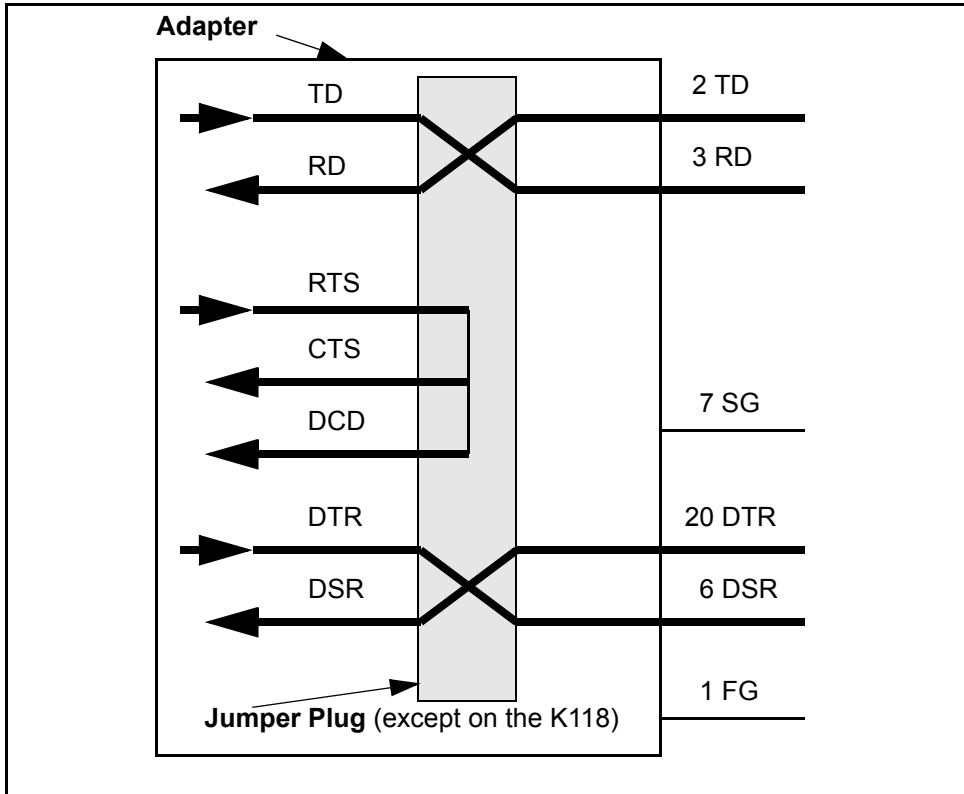


Figure 1-6. Null-Modem Adapter Signals

## Asynchronous Channel or Subchannel Characteristics

In addition to null-modem or full-modem configuration, an asynchronous channel or subchannel has several characteristics that determine how it functions. The following characteristics define an asynchronous channel or subchannel that is using the window terminal driver.

- Full-duplex transmission
- Baud-rate configuration
- Login/slave configuration
- ASCII or Baudot communication
- Parity configuration
- Bits-per-character configuration
- Stop-bits configuration



- Force-listen configuration
- Privileged-terminal configuration

Most of these characteristics must be defined in the device configuration file `devices.tin`. Some of these characteristics (the baud rate, parity, bits per character, and stop bits) can also be defined with the subroutine `s$control`, which is described in [Chapter 8](#).

The *Product Configuration Bulletin: Asynchronous Devices* (R289) provides step-by-step instructions on how to create entries for window terminal devices using the asynchronous access layer as well as for other types of asynchronous devices (for example, old asynchronous driver devices). It also provides information on the procedures for implementing new or modified device entries. Since some of the device configuration procedures have changed, you should consult the *Product Configuration Bulletin: Asynchronous Devices* (R289) before creating or editing `devices.tin` entries.

### Full-Duplex Transmission

All Stratus asynchronous channels or subchannels must use full-duplex transmission. With full-duplex transmission, signals are transmitted and received simultaneously over the same line. (With half-duplex transmission, which is not supported for Stratus asynchronous channels or subchannels, signals are either transmitted **or** received over the line at any given time; that is, signals are not transmitted and received simultaneously.) Since full-duplex transmission is the default, it does not need to be specified in `devices.tin` entries for any type of terminal.

### Baud-Rate Configuration

All Stratus asynchronous channels or subchannels support baud rates from 50 to 19,200 and can provide auto-baud detection. Terminals commonly operate at 9,600 bits per second (bps), but the high-speed adapters (the K101, K111, and K118) allow terminals to operate at up to 19,200 bps. With auto-baud detection, the operating system determines the correct baud rate for a dial-up line.

### Login/Slave Configuration

Stratus asynchronous channels or subchannels function differently for login terminal connections as opposed to slave terminal (or printer) connections. For login terminal connections, the operating system creates a prelogin process for the channel or subchannel, enabling the prelogin banner to appear when the operating system detects the DCD and DSR signals on the channel or subchannel. For slave connections, transmission cannot occur over the channel or subchannel until a process attaches and opens a port to the slave device.

## ASCII or Baudot Communication

For all asynchronous devices, Stratus channels or subchannels support both the ASCII and Baudot character sets. ASCII is a standard 8-bit code in which seven or eight bits represent the value of a character, and an additional bit is used for parity. Baudot is a binary-based code that uses 5-bit characters with 1.5 stop bits. It requires Baudot parity and is typically used with a baud rate of 50 or 75.

## Parity Configuration

Parity verifies the accuracy of information passed through the asynchronous channel or subchannel. It involves the addition of a bit to a data byte. For example, ASCII uses seven or eight bits to represent a character and uses an additional bit for parity.

The setting of the parity bit depends on the type of parity used and the number of 1 bits detected in the sequence of character bits. Stratus asynchronous channels or subchannels support even parity, odd parity, mark parity, space parity, Baudot parity, and no parity. Even parity requires an even number of 1 bits; odd parity requires an odd number of 1 bits. Therefore, if even parity is used and an odd number of 1 bits are counted in the character bits, the parity bit is set to 1; otherwise, it is set to 0. With mark or space parity, the parity bit is ignored on input and set to mark parity (1) or space parity (0), respectively, on output. Baudot parity should be used for Baudot communication. Terminals connected to a Stratus module typically use 7-bit odd parity.

The window terminal driver supports odd, even, mark, and space parity with 7-bit characters (and break handling) on K-series hardware. The driver also supports 8-bit no parity (and break handling) on K-series hardware. The window terminal driver is unique in that it also supports 8-bit odd and even parity and break handling on K-series hardware.

If you specify odd or even parity, the window terminal driver uses the value you specify for bits per character and checks the type of hardware. Specifying Baudot parity always implies 5-bit characters, and specifying mark and space parity always implies 7-bit characters. If you specify no parity, the driver always uses 8-bit characters, regardless of the value you specify for bits per character; however, the availability of break handling depends on the type of hardware.

### NOTE

The old asynchronous driver does not allow you to use eight bits per character with odd or even parity. In addition, the 8-bit no parity used by the old asynchronous driver does not provide break handling. If you need to have 8-bit no parity with break handling, you must use the window terminal driver with K-series hardware. The old

asynchronous driver provides break handling for 7-bit characters only.

### **Bits-Per-Character Configuration**

Stratus asynchronous devices can be configured to transmit either seven bits per character or eight bits per character. The bits-per-character setting should correspond to the parity setting, as described in the preceding section. Window terminal devices can use seven bits per character with even, odd, mark, space, or no parity (where 7-bit none is the same as 8-bit none); devices connected to K-series hardware can also use eight bits per character with odd, even, or no parity. (Break handling for 8-bit characters is available on K-Series hardware.)

### **Stop-Bits Configuration**

Stratus asynchronous channels or subchannels handle the one or two stop bits that are transmitted after a character to create a pause in data transmission. The use of stop bits allows the receiver to act on the character before another is sent. ASCII commonly uses one stop bit; Baudot uses 1.5 stop bits, but you must specify two stop bits.

### **Force-Listen Configuration**

A force-listen configuration forces the operating system to listen to the channel or subchannel, even if the DCD and/or DSR signals are absent (the line is hung up). In general, Stratus does not recommend this configuration; it is typically used for direct connections that do not support the DTR and DSR signals.

### **Privileged-Terminal Configuration**

A privileged-terminal configuration enables a privileged process to log in on the terminal. This configuration is valid only for login terminals and the RSN line.



---

## Chapter 2

# Simple Sequential I/O

In general, when your application program reads or writes records, it is using sequential I/O. To give application programs flexibility in dealing with the primary window/subwindow environment, the window terminal driver provides two types of sequential I/O: simple and formatted. Both types of sequential I/O provide a device-independent interface that frees programs from addressing the needs of a particular type of terminal. In addition, both types of sequential I/O ensure that data always stays within the boundaries of the current subwindow.

With both types of sequential I/O, application programs communicate with the window terminal driver using the `s$seq_read` and `s$seq_write` (or `s$seq_write_partial`) subroutines, which are described in [Chapter 7](#). (Application programs can also use the `s$read`, `s$write`, and `s$write_partial` subroutines, which in turn call the `s$seq_read` and `s$seq_write` subroutines. For a description of the `s$read`, `s$write`, and `s$write_partial` subroutines, see the VOS Subroutines manuals.)

Simple sequential I/O, sometimes referred to as line-mode I/O, enables application programs to read entire records from a terminal and write entire or partial records to a terminal. Application programs using simple sequential I/O depend on the window terminal driver to manage and maintain the display of the subwindow on the terminal screen. On the other hand, application programs using formatted sequential I/O (which is part of the application-managed I/O approach) control the display of the subwindow themselves. If you want the window terminal driver to manage the contents of the subwindow for your application program, use simple sequential I/O; if you want your application program to manage the contents of the subwindow, use formatted sequential I/O. The remainder of this chapter focuses on simple sequential I/O. For a description of formatted sequential I/O, see [Chapter 4](#).

## Characteristics of Simple Sequential I/O

Simple sequential I/O is commonly used by application programs and is also used at operating-system command level. Simple sequential I/O provides the following features.

- **One simple sequential I/O subwindow per primary window.** Each primary window can have only one simple sequential I/O subwindow. When the port is opened, the first subwindow created is **always** a simple sequential I/O subwindow

that has a subwindow ID of 0. (This is the original subwindow, as described in [Chapter 1](#).) Initially, the subwindow is the size of its primary window, but your application program can resize it with the `s$control` opcode `TERM_SET_SUBWIN_BOUNDS (2067)`. (For a description of this `s$control` opcode, see the section “Opcodes Affecting the Primary Window and Subwindow” in [Chapter 8](#).)

- **Character echoing.** All printable characters typed at the terminal are automatically echoed on the input line in the subwindow. Your application program can disable character echoing by using the `s$control` opcode `TERM_DISABLE_ECHO (2043)`. (For a description of this `s$control` opcode, see the section “Opcodes Affecting the Primary Window and Subwindow” in [Chapter 8](#).)
- **Line editing.** The terminal user can edit the input line before entering it by using a set of defined generic input requests. (For a list of the requests available with simple sequential I/O, see the section “[Generic Input Request Handling in a Simple Sequential Subwindow](#)” later in this chapter.) The terminal user enters the input line as a complete record using the keys associated with the `RETURN`, `ENTER`, `FORM`, and `HELP` generic input requests. (For COBOL users, the `EGI`, `ESI`, and `EMI` input requests can also enter the input line.) The input line entered by the terminal user is referred to hereafter as an *input record*.
- **Terminal independence for input.** The window terminal driver uses the information in a TTP to map terminal-specific input sequences (function keys and internal character-code characters) to a defined set of generic input requests. When the window terminal driver maps an input sequence to a generic input request, it executes the request within the current input record. This processing allows you to write an application program without concern for the input sequences used by a particular type of terminal.
- **Terminal independence for output.** Your application program can also maintain terminal independence for output by using generic output requests. When your application program sends a defined generic output request, the window terminal driver processes the request for the subwindow in a terminal-independent manner (that is, it interprets the request regardless of how the particular terminal implements the request). The window terminal driver then takes the result of this processing and updates the display of the subwindow based on the terminal-specific information in the TTP.
- **Line wrapping.** Any record exceeding one line wraps to the next line automatically. The portion of the record that wraps can be preceded by a single character or character string (by default, the plus sign). This character or character string is called a *continue message*, and can be set by either your application program (using the `s$control` opcode `TERM_SET_CONTINUE_CHARS (2053)`) or the terminal user (using the `set_terminal_parameters` command). (For a description of the `s$control` opcode `TERM_SET_CONTINUE_CHARS (2053)`, see the section “Opcodes Affecting the Primary Window and Subwindow” in [Chapter 8](#).)

**Chapter 8.** See the *VOS Commands Reference Manual* (R098) for a description of the `set_terminal_parameters` command.) Each input record can consist of up to 300 bytes, not including the continue message.

- **Visible typeahead.** The terminal user can type input records at any time, regardless of what your application program is doing. All input that has not been read is called *typeahead* and is always visible (unless, of course, echoing is disabled). All completed typeahead records are stored in an area of the subwindow called the *typeahead area*. Note that the typeahead area does not include the current input record, since it has not been completed. (The current input record is displayed in the *input line area*.) All typeahead in the typeahead area appears in half intensity (if the terminal supports half intensity). The current input record always appears in full intensity.
- **Pause processing.** The window terminal driver determines how many output lines can scroll up and out of the subwindow before the display of output must pause. When a pause occurs, the window terminal driver displays a pause message (by default, `--PAUSE--`). The pause message can be changed by either your application program (using the `s$control` opcode `TERM_SET_PAUSE_CHARS (2069)`) or the terminal user (using the `set_terminal_parameters` command). (For a description of the `s$control` opcode `TERM_SET_PAUSE_CHARS (2069)`, see the section “Opcodes Affecting the Primary Window and Subwindow” in [Chapter 8](#). See the *VOS Commands Reference Manual* (R098) for a description of the `set_terminal_parameters` command.)
- **Automatic refreshing of the subwindow.** Since primary windows and subwindows can overlap, there are times when portions of the simple sequential subwindow may be overlapped by primary windows or other (non-simple sequential) subwindows. The window terminal driver maintains the contents of the simple sequential subwindow so that it can redisplay (refresh) the subwindow’s contents as the overlapping primary windows or subwindows are moved or deleted.

## Subwindow Organization

When your application program uses simple sequential I/O, the window terminal driver divides the subwindow into three areas.

- The **output area** contains output from your application program.
- The **typeahead area** contains unread input records (typeahead) entered by the terminal user.
- The **input line area** contains the current (incomplete) input record.

The output area always starts at the top of the subwindow. When there is typeahead, the typeahead area appears below the output area. When there is input, the input line area appears below any typeahead at the bottom of the subwindow. When there is no

typeahead, there is no typeahead area; when there is no input, there is no input line area. (Note that from the terminal user's perspective, the input line area and the typeahead area are considered one general area, called the *command area*.)

In addition to these subwindow areas, there is a *status area* to which your application program can write a status message. The status area, sometimes referred to as the 25th line, is always associated with the primary window.

When visible at operating-system command level, the message in the status area displays information such as the date, time, percentage of wired memory used by the device, user name, the active command (or the user's current directory, if there is no active command), and the type of mode used (for example, insert or overlay). It can also contain error messages or messages concerning the status of the system. Your application program can append an error-code message to the status message by using the `s$control` opcode `TERM_STATUS_MSG_CHANGE` (2096). (For a description of this `s$control` opcode, see the section "Opcodes Affecting the Primary Window and Subwindow" in [Chapter 8](#).)

[Figure 2-1](#) illustrates how the areas are organized by providing an example of operating-system command level. In the figure, there is a partial display of information from a `list` command in the output area, several pending commands in the typeahead area and input line area, and a display of status information in the status area.

The input line area shown in [Figure 2-1](#) does not begin with a prompt character string. Your application program can set a prompt string that appears during a read by using the `s$control` opcode `TERM_SET_PROMPT_CHARS` (2073). (For a description of this `s$control` opcode, see the section "Opcodes Affecting the Primary Window and Subwindow" in [Chapter 8](#).) The terminal user can also set a prompt string by using the `set_terminal_parameters` command. (See the *VOS Commands Reference Manual* (R098) for a description of the `set_terminal_parameters` command.)

#### NOTE

Simple sequential I/O behaves differently for terminal types that do not have the most basic capabilities (for example, the ability to position the cursor anywhere on the screen). Such terminal types are referred to as *glass TTYs*. An example of a glass TTY is the `ascii` terminal type. Typically, glass TTYs are used only for the initial connection on dial-up lines. For glass TTYs, the simple sequential subwindow is not divided into the output, typeahead, and input line areas. Input is visible only when a read is active, and all output goes to the bottom line of the subwindow and scrolls up. In addition, a glass TTY



cannot support multiple primary windows or multiple subwindows.

```

w      3 schedules.new
w      1 contents.cm
w      1 create_alias_links.cm
w      1 delete_alias_links.cm
w      1 delete_carefully.cm
w      1 display_both_access_lists.cm
w      1 display_reminders.cm
w      1 dk2.cm
w      1 do_multiple.cm
w      1 edit_abbreviations.cm
w      1 edit_nonwords.cm
w      1 edit_phone_list.cm
w      1 edit_reminders.cm
w      1 find_future_day.cm
w      1 get_rid_of_accesses.cm
w      1 get_titles.cm
w      1 get_titles.ufi
w      1 give_full_access.cm
--PAUSE--
display_file schedules.new
edit_abbreviations
change_current_dir current_projects
pll test_prog
bind test_prog
06-08 09:57 | 18% | Toby Beese | list | Insert

```

**Figure 2-1. Subwindow Organization for Simple Sequential I/O**

As shown in [Figure 2-1](#), the subwindow handles a maximum of 24 lines, which represents the height of the subwindow. The output area uses 19 lines (including the pause line), the typeahead area uses 4 lines, and the input line area uses the remaining line. Since the status area is not part of the subwindow, it does not use a line in the subwindow.

In [Figure 2-1](#), the input line area displays the current input record (`bind test_prog`) in full intensity. The terminal user can edit this input record, erase it (using the key mapped to the `CANCEL` request), or enter it. When the terminal user enters the current input record shown in [Figure 2-1](#), the record moves up into the bottom of the typeahead area.

When the input record enters the typeahead area, it joins the typeahead records completed by the terminal user and changes from regular intensity to half intensity. To accommodate the newly completed input record, the typeahead area and the output

area scroll up the appropriate number of lines (in this case, they scroll up one line). The top record in the typeahead area (`display_file schedules.new`) is always the oldest unread input record and will therefore be the first record read. The bottom record is the newest unread input record and will therefore be the last record read from the typeahead area. As the terminal user completes more input records, the typeahead area grows, thereby changing the position of the completed input records and the newest unread input record in the area.

Your application program can limit the number of typeahead records allowed in the subwindow at a time. This limit is called the *typeahead limit*, and it includes all of the completed (but unread) input records in the typeahead area. For example, if the typeahead limit is set to 10 (the default), the subwindow can handle up to 10 completed input records in the typeahead area. The typeahead limit does not include the current input record in the input line area. Note that the typeahead limit is always specified in records, which may take up more than one line. To specify a typeahead limit, use the `s$control` opcode `TERM_SET_TYPEAHEAD_LINES (2077)`. (For a description of this `s$control` opcode, see the section “Opcodes Affecting the Primary Window and Subwindow” in [Chapter 8](#).)

When the number of typeahead records reaches the typeahead limit, the window terminal driver rings the terminal's bell. Once this limit is reached, the terminal user cannot enter the current input record until your application program successfully performs a read. When a read occurs, the oldest typeahead record is read and moves into the bottom of the output area, thereby reducing the number of typeahead records and allowing the terminal user to enter the current input record. When the oldest typeahead record is read and moves into the bottom of the output area, it changes from half intensity to regular intensity (if the terminal previously displayed it at half intensity).

The output area shrinks as the amount of typeahead grows, and grows as the amount of typeahead shrinks. For example, if the typeahead area in [Figure 2-1](#) grows to contain five lines, the output area shrinks to 18 output lines by scrolling the top line up and out of the subwindow. However, if the pause in [Figure 2-1](#) is released and a read occurs, the output area grows by taking the record `display_file schedules.new` and the typeahead area shrinks to three records.

The minimum size of an output area is always two lines. This limit ensures that there is always room for one output line and a pause line (when a pause occurs). When the output area shrinks to two lines, the terminal user cannot enter another input record (and the current input record cannot wrap to the next line) until your application program successfully performs a read. The maximum size of the output area is always the total number of lines in the subwindow (that is, when the subwindow does not contain any typeahead, the current input record, or a pending call to `s$seq_read`).

## Pause Processing

With simple sequential I/O, the window terminal driver determines when the display of output lines must pause. When a pause occurs, the window terminal driver cannot complete your application program's write request until the terminal user releases the pause.

The terminal user can release a pause in one of several ways.

- Press the `↓` key, which scrolls the top output line (not record) out of the subwindow and displays the next output line at the bottom of the output area.
- Press the key mapped to the `NEXT_SCREEN` request (typically, `Shift-↓`), which allows the lines currently displayed in the output area to scroll up and out of the subwindow as new lines scroll into the output area. The display of output continues until the first new line that appeared below the pause line reaches the top of the subwindow, another pause is needed, or there is no more output.
- Press the key mapped to the `NO_PAUSE` request, which disables the pause mechanism and thereby allows the remaining output lines to scroll into the output area without pausing. Once the pause mechanism is disabled, it must be reenabled by your application program, the command processor, or the terminal user. Your application program reenables the pause mechanism by performing an output reset with the `s$control` opcode `TERM_RESET_OUTPUT` (2089), the command processor also performs an output reset, and the terminal user presses the designated `NO_PAUSE` key again. (The designated `NO_PAUSE` key is a toggle key.) Note that if the window terminal driver is processing a significant amount of output when the terminal user reenables the pause mechanism, there may be a slight delay before the driver recognizes the change.
- Press the key mapped to the `ABORT_OUTPUT` request (typically, the key performing the `DISCARD` function), which releases the pause and discards all subsequent output until either your application program or the command processor resets the output or performs an `s$seq_read`. Any pending output (output that has been accepted by the window terminal driver but not yet displayed) is discarded and the error code `e$abort_output` (1279) is returned to your application program. Your application program can perform an output reset by using the `s$control` opcode `TERM_RESET_OUTPUT` (2089). (For a description of this `s$control` opcode, see the section "Opcodes Affecting the Primary Window and Subwindow" in [Chapter 8](#).)
- If the `RETURN_DOESNT_UNPAUSE` screen option is disabled (the default) and there is no current input record, press the `Return` key, which performs the same function as the `NEXT_SCREEN` request. (Your application program can set the `RETURN_DOESNT_UNPAUSE` screen option with the `s$control` opcode `TERM_SET_SCREEN_PREF` (2027), or the terminal user can set this option with the `set_terminal_parameters` command. (For a description of the `s$control` opcode `TERM_SET_SCREEN_PREF` (2027), see the section

“Opcodes Affecting the Terminal” in [Chapter 8](#). See the *VOS Commands Reference Manual* (R098) for a description of the `set_terminal_parameters` command.)

- If the `CANCEL_DOESNT_ABORT` screen option is disabled (the default) and there is no current input record, press the key mapped to the `CANCEL` request, which performs the same function as the `ABORT_OUTPUT` request. (Your application program can set the `CANCEL_DOESNT_ABORT` screen option with the `s$control` opcode `TERM_SET_SCREEN_PREF` (2027), or the terminal user can set this option with the `set_terminal_parameters` command. (For a description of the `s$control` opcode `TERM_SET_SCREEN_PREF` (2027), see the section “Opcodes Affecting the Terminal” in [Chapter 8](#). See the *VOS Commands Reference Manual* (R098) for a description of the `set_terminal_parameters` command.)

#### NOTE

---

In general, terminal users should avoid pressing XON and XOFF flow-control keys (such as `Ctrl-S` and `Ctrl-Q`) to stop and restart the display of output. Pressing these keys affects the entire communications line and all subwindows (not just the current subwindow) and produces unpredictable results. For example, if the terminal user presses `Ctrl-S` and then breaks to window manager mode, the break reverses the effect of `Ctrl-S` and the display of output resumes in all subwindows. Terminal users should use the keys described above to control pausing and should generally allow the window terminal driver to determine when a pause is necessary.

There are three times at which the window terminal driver determines when the next pause should occur:

- when the terminal user releases the pause by pressing the `↓` key or a key that performs the `NEXT_SCREEN` request.
- when a read occurs.
- when your application program issues the `s$control` opcode `TERM_RESET_COMMAND_MODES` (2088). (For a description of this `s$control` opcode, see the section “Opcodes Affecting the Primary Window and Subwindow” in [Chapter 8](#).)

When the terminal user presses a key that performs the `NEXT_SCREEN` request or when a read occurs, the window terminal driver enables all of the lines currently displayed in the output area to scroll up and out of the subwindow before the next pause. However, the terminal user or your application program can restrict the number of output lines that can scroll out of the subwindow by setting a *pause-lines limit* that is

less than the size of the current output area. If the pause-lines limit is greater than the size of the output area, the window terminal driver uses the current size of the output area as the limit. Note that if the pause-lines limit is set to 0, output never pauses.

For example, if the pause-lines limit is set to 10 and the output area currently exceeds 10 output lines, a pause occurs after every 10 output lines (not including the pause line). However, if the output area contains 8 lines and the pause-lines limit is set to 10, a pause occurs after 8 lines. By default, the pause-lines limit is the maximum size of the output area (24 in [Figure 2-1](#)).

Your application program specifies the pause-lines limit with the `s$control` opcode `TERM_SET_PAUSE_LINES (2071)`; the terminal user specifies this limit with the `set_terminal_parameters` command. (For a description of the `s$control` opcode `TERM_SET_PAUSE_LINES (2071)`, see the section “Opcodes Affecting the Primary Window and Subwindow” in [Chapter 8](#). See the *VOS Commands Reference Manual* (R098) for a description of the `set_terminal_parameters` command.)

## Retrieving Input from a Simple Sequential Subwindow

To retrieve input from a simple sequential subwindow within the current primary window, your application program must call `s$seq_read`. (See [Chapter 7](#) for a description of how to call `s$seq_read`.) If your application program calls `s$seq_read` when a complete input record is available, `s$seq_read` returns the record to the application program.

In addition to returning the input record, `s$seq_read` may return information about which key was used to complete the record. This key is referred to as a *record terminator*. The `RETURN` and `ENTER` keys are two keys that function as record terminators. (For a complete list of the record terminators, see the description of `s$seq_read` in [Chapter 7](#).)

Note that `s$seq_read` does not return the actual record terminator, but it does return information in the form of an error code for record terminators other than the `RETURN` and `ENTER` keys. In addition, `s$seq_read` does not return codes for any function keys or generic input requests that are issued by the terminal user. Codes for generic input requests are not returned because the window terminal driver automatically interprets the requests and executes them on the input line as soon as they are typed (an `s$seq_read` operation does not have to be active). The generic input requests available with simple sequential I/O are described in the next section.

If the entire input record does not fit in the input buffer supplied by your application program, `s$seq_read` returns the contents of the buffer and the error code `e$long_record (1026)`. This applies to both wait mode and no-wait mode. The `s$seq_read` call returns the error code `e$long_record (1026)` to indicate that it discarded the remaining input (that is, the portion of the record that did not fit in the input buffer).

If only part of a multibyte character can fit at the end of the input buffer, `s$seq_read` replaces the partial character with 1A hexadecimal (`SUB`). Since the remainder of the character cannot fit in the buffer, it is discarded.

## Generic Input Request Handling in a Simple Sequential Subwindow

When your application program uses simple sequential I/O, a set of generic input requests are available to the terminal user (see [Table 2-1](#)). These generic input requests allow the window terminal driver to perform basic input and editing functions for the terminal user without affecting your application program. For example, the `NEXT_SCREEN` request allows the terminal user to release a pause, and the `CANCEL` request allows the terminal user to delete the current input record. (See the *Window Terminal User's Guide* (R256) for more information about these and other requests that the terminal user can issue.) Note that most of the requests available with simple sequential I/O are also available with formatted sequential I/O; the exceptions are the `DOWN` and `NEXT_SCREEN` requests, which are available with simple sequential I/O only.

[Table 2-1](#) lists (in alphabetical order) the generic input requests available with simple sequential I/O and represents the requests using the style of the include files `video_request_defs.incl.pl1` and `video_request_defs.incl.c`. For example, the `TAB` request is represented as `TAB_REQ` in the include files. [Table 2-1](#) also identifies the decimal codes associated with these requests. Although the window terminal driver uses the decimal codes to identify the requests, it does not return these codes to your application program in sequential I/O. Additional generic input requests are available with the other types of terminal I/O. (For more information about the generic input requests, see [Chapter 5](#).)

Generic input requests are defined in the input section of the TTP using a request specifier, as shown in [Table 2-1](#). The input section associates each defined generic input request with a terminal-specific input sequence. A terminal-specific input sequence consists of function-key specifiers and/or byte specifiers that define the sequence of keystrokes or bytes required to enter a request for a particular type of terminal.

When a terminal user types input, the window terminal driver uses the information provided in the TTP to map the input to a generic input request. The window terminal driver performs this mapping before any characters are echoed on the input line, and performs the request on the current input line. (For more information about TTPs, see *VOS Communications Software: Defining a Terminal Type* (R096).)

**Table 2-1. Generic Input Requests Available with Simple Sequential I/O** (Page 1 of 2)

Generic Input Request	Code	TTP Request Specifier
ABORT_OUTPUT_REQ	212	abort-output
BACK_TAB_REQ	33	back-tab
BLANKS_LEFT_REQ	60	blanks,left
BLANKS_RIGHT_REQ	61	blanks,right
BREAK_CHAR_REQ	5	break
CANCEL_REQ	13	cancel
CHANGE_CASE_DOWN_REQ	87	change-case,down
CHANGE_CASE_UP_REQ	86	change-case,up
CLEAR_STATUS_REQ	30	clear-status
DELETE_BLANKS_REQ	88	delete,blanks
DELETE_CHAR_REQ	2	del
DELETE_LEFT_REQ	90	delete,left
DELETE_RIGHT_REQ	91	delete,right
DELETE_WORD_REQ	23	delete,word
DISCARD_REQ	49	discard
DOWN_REQ	7	down
ECHO_BEL_REQ	6	bel
EGI_REQ	189	egi
EMI_REQ	188	emi
END_OF_FILE_REQ	194	end-of-file
ENTER_REQ	1	enter
ERASE_CHAR_REQ	3	back-space
ERASE_FIELD_REQ	12	erase-field
ESI_REQ	187	esi
FORM_REQ	15	display-form
GOTO_BEGINNING_REQ	16	goto,beginning

**Table 2-1. Generic Input Requests Available with Simple Sequential I/O** (Page 2 of 2)

Generic Input Request	Code	TTP Request Specifier
GOTO_END_REQ	17	goto,end
HELP_REQ	57	help
INSERT_DEFAULT_REQ	47	insert-default
INSERT_SAVED_REQ	14	insert-saved
INTERRUPT_REQ	190	interrupt
LEFT_REQ	18	left
NEXT_SCREEN_REQ	58	next-screen
NO_PAUSE_REQ	28	no-pause
REDISPLAY_REQ	24	redisplay
REPEAT_LAST_REQ	42	repeat-last
RETURN_REQ	4	return
RIGHT_REQ	19	right
TAB_REQ	0	tab
TAB_STOP_LEFT_REQ	62	tab-stop,left
TAB_STOP_RIGHT_REQ	63	tab-stop,right
TOGGLE_OVERLAY_MODE_REQ	115	en/disable-overlay-mode
TWIDDLE_REQ	201	twiddle
UPDATE_STATUS_REQ	29	update-status
WORD_CHANGE_CASE_DOWN_REQ	133	word,change-case,down
WORD_CHANGE_CASE_LEFT_REQ	131	word,change-case,left
WORD_CHANGE_CASE_UP_REQ	132	word,change-case,up
WORD_LEFT_REQ	21	word,left
WORD_RIGHT_REQ	22	word,right



## Sending Output to a Simple Sequential Subwindow

To send output to a simple sequential subwindow, your application program must call either `s$seq_write` or `s$seq_write_partial` and specify a buffer containing data. (See [Chapter 7](#) for a description of how to use these subroutines.)

When your application program sends output to a simple sequential subwindow, the window terminal driver processes characters from the internal character coding system, graphic characters, control characters, new-line characters, tabs, and generic output requests. Output is mapped to the configuration, character-translation, output, and attributes sections of the TTP for the device. (See [Chapter 5](#) for more information about these sections of the TTP and the handling of different types of output.) The window terminal driver also performs automatic line wrapping and pause processing, described earlier in this chapter. At all times, the window terminal driver manages the display of output in the subwindow and can refresh the output when the subwindow is rearranged or resized.

To enable the window terminal driver to perform basic output functions, your application program can place sequences representing generic output requests in the output buffer. (The generic output requests available with simple sequential I/O are listed in the next section.)

If your application program is using no-wait mode, and `s$seq_write` or `s$seq_write_partial` cannot send the entire record (for example, if the display of output is paused), the call returns the error code `e$caller_must_wait` (1277) and updates the value of the `buffer_length` argument to indicate the number of characters processed thus far. In order to complete the write with another call to `s$seq_write` or `s$seq_write_partial`, your application program must advance the buffer pointer and update the value of the `buffer_length` argument before sending the remaining characters. If your application program is using wait mode and the call cannot complete the write, the window terminal driver (and therefore your application program) must wait until the call sends the complete record.

## Issuing Generic Output Requests in the Simple Sequential Subwindow

With simple sequential I/O, your application program can use a set of generic output requests to perform output functions (such as inserting a new-line sequence) without concern for the type of terminal receiving the output. [Table 2-2](#) lists (in alphabetical order) the generic output requests available with simple sequential I/O and represents the requests using the style of the include files `output_sequence_codes.incl.pl1` and `output_sequence_codes.incl.c`. For example, the `CURSOR_ON` request is represented as `CURSOR_ON_SEQ` in the include files. [Table 2-2](#) also identifies the decimal codes associated with these requests.

To specify a generic output request, your application program must place a multibyte generic output sequence in an output buffer. The first byte of the generic output sequence is always the generic sequence introducer 27 decimal (1B hexadecimal). The generic sequence introducer is followed by the code representing the generic output request, which is followed by any arguments that the request requires. (For more information about specifying generic output requests in output sequences, see [Chapter 5](#).)

Once the window terminal driver has processed all of the generic output sequences and data placed in the output buffer, it updates the display of the subwindow based on the information provided in the output and attributes sections of the TTP. These sections of the TTP define the terminal's capabilities. Each capability in the TTP is associated with a sequence that must be sent to the terminal in order for the terminal to perform the particular function. (For more information about defining capabilities in the TTP, see *VOS Communications Software: Defining a Terminal Type* (R096).)

**Table 2-2. Generic Output Requests Available with Simple Sequential I/O** (Page 1 of 2)

Generic Output Request	Code
BEEP_SEQ	69
BLANK_OFF_SEQ	130
BLANK_ON_SEQ	129
BLINK_OFF_SEQ	128
BLINK_ON_SEQ	127
BOLDFACE_OFF_SEQ	101
BOLDFACE_ON_SEQ	100
CURSOR_OFF_SEQ	11
CURSOR_ON_SEQ	10
HALF_INTENSITY_OFF_SEQ	26
HALF_INTENSITY_ON_SEQ	25
HI_INTENSITY_OFF_SEQ	126
HI_INTENSITY_ON_SEQ	125
INVERSE_VIDEO_OFF_SEQ	24
INVERSE_VIDEO_ON_SEQ	23
NEW_LINE_SEQ	61

**Table 2-2. Generic Output Requests Available with Simple Sequential I/O** (Page 2 of 2)

Generic Output Request	Code
RESET_TERMINAL_SEQ	22
SCREEN_OFF_SEQ	94
SCREEN_ON_SEQ	93
SET_CURSOR_FORMAT_SEQ	38
SET_MODE_ATTRIBUTES_SEQ	133
STANDOUT_OFF_SEQ	132
STANDOUT_ON_SEQ	131
UNDERSCORE_OFF_SEQ	99
UNDERSCORE_ON_SEQ	98

## Issuing Generic Output Requests That Switch the Subwindow to Formatted I/O Mode

The generic output requests listed in [Table 2-3](#) are valid for a simple sequential subwindow but are only a subset of the generic output requests available to your window terminal application program. Additional requests are available for a formatted sequential subwindow. If your application program attempts to use a generic output request that is available only for formatted sequential I/O (see [Table 2-3](#)), the window terminal driver switches your application program's simple sequential subwindow to formatted I/O mode. In formatted I/O mode, all sequential I/O subsequently sent to the subwindow becomes formatted sequential I/O. (For information about the arguments required by these generic output requests, see [Chapter 5](#).)

**Table 2-3. Generic Output Requests That Switch the Subwindow to Formatted I/O Mode** (Page 1 of 2)

Generic Output Request	Code
CLEAR_SCROLLING_REGION_SEQ	2
CLEAR_TO_EOL_SEQ	3
CLEAR_TO_EOR_SEQ	4
DELETE_CHARS_SEQ	15
DELETE_LINES_SEQ	17
DISPLAY_BLOCK_SEQ	33

**Table 2-3. Generic Output Requests That Switch the Subwindow to Formatted I/O Mode** (Page 2 of 2)

Generic Output Request	Code
DOWN_SEQ	6
ENTER_GRAPHICS_MODE_SEQ	27
INSERT_CHARS_SEQ	14
INSERT_LINES_SEQ	16
LEAVE_GRAPHICS_MODE_SEQ	28
LEFT_SEQ	7
POSITION_CURSOR_SEQ	1
RIGHT_SEQ	8
SCROLL_DOWN_SEQ	31
SCROLL_UP_SEQ	32
SET_SCROLLING_REGION_SEQ	95
UP_SEQ	5

When the subwindow switches to formatted I/O mode, the display of the subwindow changes. The way the display changes depends on how the terminal's port is attached. If the port is attached with the `hold_open` and `hold_attached` switches turned **off** (see the description of `s$attach_port` in [Chapter 6](#)), the window terminal driver clears the entire subwindow, discards all typeahead, and moves the cursor to the top left corner of the subwindow.

If the port is attached with either the `hold_open` or `hold_attached` switch turned on, the window terminal driver does **not** clear the entire subwindow, but does discard all typeahead. In addition, the position of the cursor does not change; that is, the cursor remains where it was prior to the switch.

Switching the simple sequential subwindow to formatted I/O mode by using the generic output requests listed in [Table 2-3](#) is the same as using the `s$control` opcode `TERM_ENABLE_FMT_IO_MODE (2046)`. If you want to switch the simple sequential subwindow to formatted I/O mode temporarily, you should design your application program to use the opcode `TERM_ENABLE_FMT_IO_MODE (2046)` instead of the generic output requests listed in [Table 2-3](#). To disable formatted I/O mode and resume simple sequential I/O in the subwindow, your application program should use the `s$control` opcode `TERM_DISABLE_FMT_IO_MODE (2047)`. (For a description of the opcodes `TERM_ENABLE_FMT_IO_MODE (2046)` and

`TERM_DISABLE_FMT_IO_MODE (2047)`, see the section “Opcodes Affecting the Primary Window and Subwindow” in [Chapter 8](#).)

If you want your application program to use formatted I/O mode **only**, make sure that it creates a new formatted sequential subwindow instead of switching the original (simple sequential) subwindow to formatted I/O mode. To explicitly create a formatted sequential subwindow, your application program must use the `s$control` opcode `TERM_CREATE_SUBWIN (2038)`. (See [Chapter 4](#) for a description of formatted sequential I/O; see the section “Opcodes Affecting the Primary Window and Subwindow” in [Chapter 8](#) for a description of the opcode `TERM_CREATE_SUBWIN (2038)`.)



---

## Chapter 3

# Unprocessed Raw I/O

When your application program reads and writes unprocessed (uninterpreted) data, it is using unprocessed raw I/O. Unprocessed raw I/O is useful when you want your application program to control the asynchronous communications line as a communications device, where the device is not necessarily a terminal.

To use unprocessed raw I/O, your application program must retrieve raw input using the `s$read_raw` subroutine and send raw output using the `s$write_raw` subroutine. This chapter describes how the `s$read_raw` and `s$write_raw` subroutines function with unprocessed raw I/O. (For a complete description of these subroutines, see [Chapter 7](#).)

### Characteristics of Unprocessed Raw I/O

Unprocessed raw I/O has the following characteristics.

- **No input or output mapping.** If your application program uses unprocessed raw I/O, it must perform all its own processing; the window terminal driver does not perform any input translation or screen formatting on behalf of your application program. Characters are simply transmitted to and from the device without being translated.
- **No character echoing.** Input is not echoed to the screen.
- **No window environment to maintain.** Unprocessed raw I/O does not take advantage of the window environment at all. In fact, raw output invalidates the contents of the subwindow display and will interfere with the processing of subsequent sequential I/O. Raw output is best combined with unprocessed raw input.
- **Terminal-specific information.** If you write an unprocessed raw I/O application program for a terminal, the application program must either contain terminal-specific information (so that it can process input sequences and send the correct output sequences), or it must use terminal-type (TTP) subroutines to retrieve the terminal-specific information. Embedding terminal-specific information in your application program usually prevents the application program from working with a terminal other than the one it was written to support. Using the TTP subroutines enables your application program to be terminal independent and to gain access to the generic input requests, generic output requests, and specific

terminal capabilities. (See [Appendix C](#) for a list and a brief description of the TTP subroutines.)

- **Control of data flow with interrupts.** Two of the unprocessed raw input modes (raw table mode and raw record mode) enable your application program to specify different types of interrupt characters in order to reduce the input processing overhead caused by frequent system interrupts. Note that the interrupt-character options available with the window terminal driver generally apply to certain protocols (such as RS-232-C), but can be used by other protocols to group data.
- **Full 8-bit support.** You can design your application program to take advantage of an 8-bit data path by specifying a 256-byte interrupt table. The interrupt table identifies the interrupt characters you specify in a buffer used by your application program. For devices using the RS-232-C interface (and the asynchronous access layer), K-series hardware supports 8-bit character transmission by providing break handling (the processing of metacharacters in the input stream to identify common asynchronous errors).
- **Automatic frame detection.** Automatic frame detection, which is supported by both K-Series hardware through the RS-232-C interface, handles protocols that use standard framing techniques. Automatic frame detection is suitable for block-oriented protocols that use a unique character in the input stream to terminate each block or packet.

In general, when you design your application program to handle unprocessed raw I/O, you should ensure that the window terminal driver cannot intercept a `BREAK_CHAR` generic input request that activates window manager mode. If your application program enables the processing of `BREAK_CHAR` generic input requests with the `s$control` opcode `TERM_SET_BREAK_ACTION` (2051), and the screen preference named `break` to window manager mode is enabled (the default), a `BREAK_CHAR` generic input request or a line break will activate window manager mode. To disable the screen preference named `break` to window manager mode and thereby prevent your application program from intercepting a `BREAK_CHAR` generic input request, use the `s$control` opcode `TERM_SET_SCREEN_PREF` (2027). (For a description of the `s$control` opcode `TERM_SET_SCREEN_PREF` (2027), see the section “Opcodes Affecting the Terminal” in [Chapter 8](#). For a description of the `s$control` opcode `TERM_SET_BREAK_ACTION` (2051), see the section “Opcodes Affecting the Primary Window and Subwindow” in [Chapter 8](#).)



## Retrieving Unprocessed Raw Input

Unprocessed raw input is raw data. It is not interpreted in any way by the window terminal driver before it is passed to your application program.

To retrieve unprocessed raw input, your application program must perform the following steps.

1. Specify an unprocessed raw input mode with the `s$control` opcode `TERM_SET_INPUT_MODE (2057)`.
2. Call the `s$read_raw` subroutine.

If your application program does not specify an unprocessed raw input mode with `TERM_SET_INPUT_MODE (2057)`, the window terminal driver uses generic input mode. Generic input mode is a **processed** raw input mode and is part of the application-managed I/O approach (see [Chapter 4](#)).

Your application program must also use `TERM_SET_INPUT_MODE (2057)` to change to a different input mode. Typically, an application program that changes input modes saves the current input mode using `TERM_GET_INPUT_MODE (2056)`, changes to a new input mode using `TERM_SET_INPUT_MODE (2057)`, performs its processing, and then restores the saved input mode by using `TERM_SET_INPUT_MODE (2057)` again. (For more information about these opcodes, see the section “Opcodes Affecting the Primary Window and Subwindow” in [Chapter 8](#).)

Regardless of the input mode you choose, `s$read_raw` always returns characters whenever they are available; that is, the input buffer supplied by your application program may or may not be full. Your application program specifies the maximum number of characters allowed in the buffer by using the `s$read_raw` argument `buffer_length`. (See [Chapter 7](#) for a description of this argument.)

The unprocessed raw input modes are described in the following section. Note that for each of the input modes, you must design your application program to use either wait mode (the default) or no-wait mode. These modes affect how `s$read_raw` functions when there are no characters available. (See [Chapter 8](#) for a description of wait mode and no-wait mode.)

## Selecting an Unprocessed Raw Input Mode

The unprocessed raw input modes use interrupts to control the flow of data in various ways. With RS-232-C communications through the asynchronous access layer, *interrupts* are control signals that cause the operating system to temporarily stop its current processing. In general, interrupts are generated by the RS-232-C communications hardware when it receives an *interrupt character*. Upon receipt of an interrupt character, data becomes available for retrieval with `s$read_raw`. Note that other access layers may handle interrupts differently; for example, the OS TELNET

access layer handles interrupts as event notifications to the user process, not as interrupts to the operating system.

There are three unprocessed raw input modes.

- In **normal raw mode**, every incoming character causes an interrupt. This mode is suitable for RS-232-C communications through the asynchronous access layer or for communications through an access layer such as OS TELNET.
- In **raw table mode**, only characters specified as interrupt characters in an interrupt table cause an interrupt. This mode is generally suitable for RS-232-C communications through the asynchronous access layer only.
- In **raw record mode**, all characters specified as interrupt characters in an interrupt table are also record terminators. This mode is suitable for RS-232-C communications through the asynchronous access layer or for communications through an access layer such as OS TELNET.

As described above, normal raw mode causes the RS-232-C communications hardware to generate an interrupt upon receipt of each incoming character (that is, each character is an interrupt character). With the raw table and raw record modes, however, the RS-232-C communications hardware generates an interrupt based on the information provided in an *interrupt table*, which is a 256-byte buffer in which each buffer position (0 to 255) represents the rank assigned to a specific character in the internal character coding system. (For information about the interrupt table, see the section “[Creating an Interrupt Table](#)” later in this chapter. See [Appendix B](#) for a description of the internal character coding system.)

The unprocessed raw input modes can support transmissions of either seven bits per character or eight bits per character. Using eight bits per character is suitable for the transfer of binary images and for taking advantage of the 256-byte interrupt-table options. Using seven bits per character is common for standard RS-232-C communications. If you are using the asynchronous access layer to establish RS-232-C communications, and you want your application program to handle eight bits per character, set the bits-per-character value to 8 and the parity value to no parity (or odd or even parity with K-Series hardware) using the `s$control` opcode `TERM_SET_OPERATING_VALUES (2014)`. (This opcode is described in [Chapter 8](#) in the section “Opcodes Affecting the RS-232-C Communications Medium.”) Note that using 8-bit no parity on K-Series hardware means that the window terminal driver will interpret the asynchronous metacharacters in the input stream. (For more information about the handling of metacharacters, see the error-code section of the `s$read_raw` description in [Chapter 7](#).)

For all of the unprocessed raw input modes, the RS-232-C communications hardware sets the minimum time between interrupts to 30 milliseconds (the default). Your application program can change the minimum time between interrupts by using the `s$control` opcode `TERM_SET_FORWARDING_TIMER (2095)`. (For a description of

this opcode, see the section “Opcodes Affecting the RS-232-C Communications Medium” in [Chapter 8](#).)

### Using Normal Raw Mode

In normal raw mode, every incoming character is an interrupt character that causes a processor interrupt. To use normal raw mode, your application program must specify `RAW_INPUT_MODE` (represented by the value 3) with the `s$control` opcode `TERM_SET_INPUT_MODE (2057)`. (See [Chapter 8](#) for a description of this opcode.)

Normal raw mode is suitable for an application program that is using either the asynchronous access layer to establish RS-232-C communications or an access layer such as OS TELNET, which establishes TELNET protocol communications over an Ethernet-based OS TCP/IP network.

If your application program uses normal raw mode with wait mode, a call to `s$read_raw` does not return to your application program until at least one character is available. When characters are available, `s$read_raw` returns the data and the code 0. When no characters are available, `s$read_raw` and your application program must wait.

In no-wait mode, `s$read_raw` returns the error code `e$caller_must_wait(1277)` **only** when no characters are available. If there are characters available, `s$read_raw` returns the data and the code 0.

### Using Raw Table Mode

In raw table mode, your application program can use its own interrupt table to control how interrupts are generated, without imposing a record structure on the input. (The section “[Creating an Interrupt Table](#)” later in this chapter describes how your application program can create and activate its own interrupt table.) Raw table mode is suitable for an application program that uses the asynchronous access layer to establish RS-232-C communications.

To use raw table mode, your application program must specify `RAW_TABLE_MODE` (represented by the value 4) with the `s$control` opcode `TERM_SET_INPUT_MODE (2057)`. (See [Chapter 8](#) for a description of this opcode.)

When your application program uses raw table mode, a call to `s$read_raw` in either wait mode or no-wait mode returns whatever input characters are available. The input buffer contains normal input characters, as well as any interrupt characters and trailer bytes that are available at the time of the read (the interrupt characters and the trailer bytes generate the interrupt).

Certain types of interrupt characters require one or two trailer bytes, while others do not require any trailer bytes. In general, the interrupt characters requiring one or two trailer bytes (`INTERRUPT_PLUS1` or `INTERRUPT_PLUS2`, respectively) are suitable for supporting automatic frame detection in raw record mode, but are not suitable for raw

table mode. If your application program uses raw table mode and activates an interrupt table that specifies one or more characters as `INTERRUPT_PLUS1` or `INTERRUPT_PLUS2` characters, the window terminal driver uses the specified interrupt type with the higher number of trailer bytes to control the processing of **all** interrupts. For example, if the interrupt table specifies a character as an `INTERRUPT_PLUS2` character, the window terminal driver will treat **all** interrupt characters as `INTERRUPT_PLUS2` characters, regardless of whether the table specifies these characters as normal interrupt characters or `INTERRUPT_PLUS1` characters. Each `INTERRUPT_PLUS2` character generates three interrupts, one for the interrupt character itself and one for each of the next two input characters. (For more information about the types of interrupt characters, see the section “[Creating an Interrupt Table](#)” later in this chapter.)

If the input buffer fills up (that is, the buffer contains the maximum number of characters specified by the `buffer_length` argument), `s$read_raw` returns the input characters that fit in the input buffer and the code 0. Any input characters that do not fit in the input buffer are retained by the window terminal driver. Your application program must then make additional calls to receive the rest of the input, which may include a number of interrupt characters and trailer bytes.

If no characters are available in wait mode, `s$read_raw` and your application program wait; if no characters are available in no-wait mode, `s$read_raw` returns the error code `e$caller_must_wait` (1277).

## Using Raw Record Mode

Your application program can use raw record mode to impose a record structure on the incoming data. Raw record mode allows your application program to use the interrupt table it created, where each interrupt character specified (including any trailer bytes it may require) is a record terminator. With raw record mode, a record consists of input characters up to and including the interrupt character and any trailer bytes it may require. (Certain types of interrupt characters require one or two trailer bytes, while others do not require any trailer bytes. See the discussion of interrupt characters in the next section, “[Creating an Interrupt Table](#).”) When the access layer receives the appropriate interrupt character and its required trailer bytes, the record is terminated. The character following the interrupt character and its trailer bytes then begins a new record.

Raw record mode is suitable for an application program that is using either the asynchronous access layer to establish RS-232-C communications or an access layer such as OS TELNET, which establishes TELNET protocol communications over an Ethernet-based OS TCP/IP network. For an application program using the asynchronous access layer, raw record mode also supports automatic frame detection, which accommodates protocols that use standard framing techniques. Therefore, this mode is suitable for block-oriented protocols in which each block or packet is terminated by a unique character that does not appear anywhere else in the data.

To use raw record mode, your application program must specify `RAW_RECORD_MODE` (represented by the value 5) with the `s$control` opcode `TERM_SET_INPUT_MODE` (2057). (See [Chapter 8](#) for a description of this opcode.) When your application program uses raw record mode, each call to `s$read_raw` returns either a single record or a partial record.

When input characters are available, a call to `s$read_raw` in either wait mode or no-wait mode returns a complete record of data and the code 0 as soon as the interrupt character and the appropriate number of trailer bytes arrive. If the input buffer fills up but does not contain the interrupt character and the appropriate number of trailer bytes (that is, the length of the record exceeds the maximum buffer length specified with the `buffer_length` argument), `s$read_raw` waits for the interrupt and then returns a partial record containing the characters that fit in the input buffer.

With raw record mode, `s$read_raw` also returns the error code `e$long_record` (1026) to indicate that the window terminal driver discarded any characters that arrived after the buffer was full and before the appropriate interrupt character. (The interrupt character and its trailer bytes are also discarded once they generate the interrupt.)

If no characters are available in wait mode, `s$read_raw` and the application program wait; if no characters are available in no-wait mode, `s$read_raw` returns the error code `e$caller_must_wait` (1277).

## Creating an Interrupt Table

If your application program uses raw table mode or raw record mode, it must create and activate its own interrupt table. In an interrupt table, your application program specifies which characters should cause interrupts and which should not. This reduces the number of interrupts generated and increases the speed at which incoming characters can be processed. If the communications line is configured to send and receive 8 bits per character, your application program's interrupt table can handle characters in the range 0 to 255, which means that it can support a full 8-bit data path between your application program and the device.

To create an interrupt table, your application program must create a buffer of 256 bytes, where each byte (buffer position) represents the rank of a particular character in the internal character coding system. For example, buffer position 0 corresponds to the ASCII NUL character, which has the decimal rank 0.

To each buffer position your application program assigns one of four interrupt values, where each value specifies a certain type of interrupt character. A description of the four types of interrupt characters and their respective values follows.

- **Noninterrupt character.** This type specifies that an interrupt should **not** be generated upon receipt of a particular character. To ensure that a character does not generate an interrupt, specify `INTERRUPT_NEVER` or the interrupt value 0 for

the buffer position representing the rank of the character. For example, assigning the interrupt value 0 to the buffer position 65 makes the ASCII character A a noninterrupt character.

- **Normal interrupt character.** This type specifies that an interrupt should be generated upon receipt of a particular character. To make a character an interrupt character, specify `INTERRUPT_NOW` or the interrupt value 1 for the buffer position representing the rank of the character. For example, assigning the interrupt value 1 to the buffer position 106 makes the j character a normal interrupt character.
- **Interrupt character plus one character.** This type specifies that an interrupt should be generated upon receipt of any character **following** the interrupt character. To use this type of interrupt character, specify `INTERRUPT_PLUS1` or the interrupt value 2 for the buffer position representing the rank of the character. For example, assigning the interrupt value 2 to the buffer position 91 causes the [ character to generate an interrupt as soon as one trailer byte arrives after the [ character. (The *trailer byte* is the character following the interrupt character. The trailer byte also generates an interrupt.)
- **Interrupt character plus two characters.** This type specifies that an interrupt should be generated upon receipt of any two characters **following** the interrupt character. To use this type of interrupt character, specify `INTERRUPT_PLUS2` or the interrupt value 3 for the buffer position representing the rank of the character. For example, assigning the interrupt value 3 to the buffer position 93 causes the ] character to generate an interrupt as soon as two trailer bytes arrive after the ] character. (The trailer bytes are the two characters following the interrupt character. Each trailer byte also generates an interrupt.)

#### NOTE

The last two interrupt types (`INTERRUPT_PLUS1` and `INTERRUPT_PLUS2`) are suitable for supporting either RS-232-C automatic frame detection in raw record mode or access layers such as OS TELNET in raw record mode, but generally are not useful for any access layer in raw table mode. In raw table mode, an interrupt table that specifies one or more characters as `INTERRUPT_PLUS1` or `INTERRUPT_PLUS2` characters will cause the window terminal driver to use the interrupt type with the most trailer bytes to control the processing of all interrupts. For example, if the interrupt table specifies normal interrupt characters, `INTERRUPT_PLUS1` characters, and `INTERRUPT_PLUS2` characters, all interrupt characters will function as `INTERRUPT_PLUS2` characters.

Once your application program creates an interrupt table, it must activate the table by using the `s$control` opcode `TERM_SET_INTERRUPT_TABLE (2061)`. However, if your application program will use more than one interrupt table, it should save the

current interrupt table before specifying a new table, and restore the saved table when it no longer needs the new table.

To save an interrupt table, your application program must use the `s$control` opcode `TERM_GET_INTERRUPT_TABLE (2060)`. (For a description of the opcode pair `TERM_SET_INTERRUPT_TABLE (2061)` and `TERM_GET_INTERRUPT_TABLE (2060)`, see the section “Opcodes Affecting the Primary Window and Subwindow” in [Chapter 8](#).)

The window terminal driver does not automatically restore the original (default) interrupt table associated with a nonlogin (slave) communications line; instead, the interrupt table that was last activated remains in effect. Therefore, it is important for your application program to restore the original (and proven) interrupt table for this type of communications line.

In addition, using the opcode `TERM_LISTEN (2023)` or `TERM_RESET_COMMAND_MODES (2088)` deletes the current interrupt table and causes the window terminal driver to use the original (default) interrupt table. (In the default interrupt table, each character is a normal interrupt character.) Therefore, your application program must re-create the desired interrupt table by using the opcode `TERM_SET_INTERRUPT_TABLE (2061)` after each call to `TERM_LISTEN (2023)` or `TERM_RESET_COMMAND_MODES (2088)`. (For a description of the opcode `TERM_LISTEN (2023)`, see the section “Opcodes Affecting the RS-232-C Communications Medium” in [Chapter 8](#). For a description of the opcode `TERM_RESET_COMMAND_MODES (2088)`, see the section “Opcodes Affecting the Primary Window and Subwindow” in [Chapter 8](#).)

## Sending Unprocessed Raw Output

To send unprocessed raw output to the terminal or device, your application program must use the `s$write_raw` subroutine. When your application program calls `s$write_raw`, the window terminal driver sends the characters without translating them. The window terminal driver does not perform pause processing or continuation-line processing, nor does it consult a terminal-type definition (`.ttp`) file. The terminal displays the output exactly as it is sent by your application program.

Because the `s$write_raw` subroutine cannot function in a subwindow environment, it is suitable only for unprocessed raw I/O application programs, not application programs designed to use the application-managed I/O approach. An application program designed to use the application-managed I/O approach must enable formatted I/O mode and use the `s$seq_write` (or `s$seq_write_partial`) subroutine. (See [Chapter 7](#) for a description of the `s$write_raw`, `s$seq_write`, and `s$seq_write_partial` subroutines. See [Chapter 4](#) for a description of application-managed I/O.)

When your application program calls `s$write_raw` and specifies a buffer containing data, `s$write_raw` tries to transfer the data to system output buffers and write the data to the device. Partial writes can occur in the following situations:

- if the time limit set with `s$set_io_time_limit` expires before enough system output buffers are available
- if your application program is using no-wait mode, and system output buffers are available only for a portion of the data supplied by your application program.

For more information about the `s$write_raw` subroutine, see [Chapter 7](#). For more information about `s$set_io_time_limit`, see [Chapter 8](#).



---

## Chapter 4

# Application-Managed I/O

Application-managed I/O enables your application program to control the display of a subwindow. It offers your application program more control over the processing of sequential output, sequential input, and raw input than the other window terminal programming approaches. The application-managed I/O approach is suitable for an interactive application program that manages the contents and organization of the subwindow display.

### Characteristics of Application-Managed I/O

Application-managed I/O has the following characteristics.

- **Formatted I/O mode subwindow.** Your application program must use formatted I/O mode to control the processing of sequential input or output in the subwindow currently handling all I/O. In this mode, all output is formatted sequential output, and your application program must use `s$seq_write` or `s$seq_write_partial` to send the output to the terminal.
- **No handling of raw output.** Your application program should not use `s$write_raw` (which sends raw output) with application-managed I/O. Raw output does not allow your application program to take advantage of the primary window/subwindow environment and will invalidate the contents of the subwindow display on subsequent calls to `s$seq_read`.
- **Large selection of generic output requests.** Approximately forty generic output requests are available for sending formatted sequential output. These requests provide your application program with more ways to control the subwindow display in a terminal-independent manner.
- **No automatic line wrapping or pause processing.** Line wrapping and pause processing are disabled to give your application program complete control of the terminal display.

- **Several input options.** Your application program has several options for retrieving input. It can send formatted sequential output and retrieve either formatted sequential input (using `s$seq_read`) or any type of raw input (using `s$read_raw`). The application-managed I/O approach supports three types of input.
  - **Formatted sequential input** offers limited automatic input mapping and typeahead processing (for example, no typeahead processing before a read operation). Your application program retrieves formatted sequential input with `s$seq_read` when the subwindow handling I/O is in formatted I/O mode.
  - **Processed raw input** provides various levels of input mapping through three input modes: generic input mode, function-key input mode, and translated input mode. Your application program must select one of these modes before calling `s$read_raw`.
  - **Unprocessed raw input**, as described in [Chapter 3](#), uses interrupts to control the flow of data and provides three input modes: normal raw mode, raw table mode, and raw record mode. Your application program must select one of these modes before calling `s$read_raw`.

To gain the most flexibility and control with application-managed I/O, your application program should handle **processed** raw input and formatted sequential output. The formatted sequential input/output combination provides a different set of features (for example, echoing and line-editing support during a read operation), but generally does not provide as much control over the appearance of the subwindow.

## Using a Formatted I/O Subwindow

To use the application-managed I/O approach, your application program must ensure that the current I/O subwindow is in formatted I/O mode. As described in [Chapter 1](#), the *current I/O subwindow* is the subwindow that receives all I/O. Your application program can make a specified subwindow the current I/O subwindow by using the `s$control` opcode `TERM_SET_CURRENT_SUBWIN (2091)`, which is described in the section “Opcodes Affecting the Primary Window and Subwindow” in [Chapter 8](#). Formatted I/O mode affects all sequential writes and sequential reads to the current I/O subwindow.

With formatted I/O mode, all sequential output sent to the subwindow is formatted sequential output, and all sequential input retrieved from the subwindow is formatted sequential input. Although the current I/O subwindow may be in formatted I/O mode when your application program is handling raw input, formatted I/O mode has no effect on the function of raw input.

There are two ways to use formatted I/O mode.

- Your application program can create a new formatted I/O subwindow.
- Your application program can switch the original subwindow from simple sequential I/O to formatted sequential I/O. (As described in [Chapter 1](#), the *original subwindow*

is the first subwindow created when your application program opens the port. The original subwindow has a subwindow ID of 0.)

The method you use depends on whether you want your application program to use formatted sequential I/O **only** or to use formatted sequential I/O temporarily in the subwindow.

## Creating a Formatted I/O Subwindow

If you want your application program to use formatted sequential I/O **only**, make sure that it always creates a new formatted I/O subwindow by using the `s$control` opcode `TERM_CREATE_SUBWIN (2038)`. Each subwindow that this opcode creates is automatically in formatted I/O mode and can **never** become a simple sequential I/O subwindow. Simple sequential I/O can be used only in the original subwindow, and the opcode `TERM_CREATE_SUBWIN (2038)` does not affect the original subwindow.

When your application program issues `TERM_CREATE_SUBWIN (2038)`, it must provide a structure that defines the horizontal and vertical offsets and the width and height of the new subwindow. This opcode returns the subwindow ID of the new subwindow, which automatically appears on top of the other subwindows and becomes the current I/O subwindow.

Note that `TERM_CREATE_SUBWIN (2038)` does not update the display of the primary window in order to clear the new subwindow. You must design your application program to clear the new subwindow. For more information, see the description of the opcode `TERM_CREATE_SUBWIN (2038)` in the section “Opcodes Affecting the Primary Window and Subwindow” in [Chapter 8](#).

To delete a formatted I/O subwindow, your application program can issue the opcode `TERM_DELETE_SUBWIN (2039)`. For a description of this opcode, see the section “Opcodes Affecting the Primary Window and Subwindow” in [Chapter 8](#).

## Enabling and Disabling Formatted I/O Mode in a Simple Sequential Subwindow

If you want your application program to use formatted sequential I/O **temporarily** in the original subwindow (the first subwindow created when your application program opened the port), switch the original subwindow to formatted I/O mode by using the `s$control` opcode `TERM_ENABLE_FMT_IO_MODE (2046)`. This opcode controls the processing of sequential input and output in the original subwindow until your application program disables formatted I/O mode or performs a reset.

When your application program uses `TERM_ENABLE_FMT_IO_MODE (2046)` to switch the original subwindow to formatted I/O mode, the original subwindow must be the current I/O subwindow (the subwindow currently handling all I/O). When the opcode performs the switch, the display of the subwindow changes, depending on how the port is attached. For more information about changes to the display, see the description of

the opcode pair `TERM_DISABLE_FMT_IO_MODE (2047)` and `TERM_ENABLE_FMT_IO_MODE (2046)` in the section “Opcodes Affecting the Primary Window and Subwindow” in [Chapter 8](#).

To return the original subwindow to simple sequential I/O, your application program should use `TERM_DISABLE_FMT_IO_MODE (2047)`. In disabling formatted I/O mode, this opcode moves the cursor to the bottom left corner of the subwindow. The window terminal driver attempts to retain the formatted I/O data that existed before the switch from simple sequential I/O to formatted sequential I/O in the subwindow. However, depending on how the port was attached, the window terminal driver may not always be able to retain the formatted I/O data.

To return the original subwindow to its initial state, your application program can perform a reset by using the opcode `TERM_RESET_OUTPUT (2089)` or `TERM_RESET_COMMAND_MODES (2088)`. For a description of these opcodes, see the section “Opcodes Affecting the Primary Window and Subwindow” in [Chapter 8](#).

The next section describes how to send formatted sequential output, which is the basis of application-managed I/O. Discussions of how to send formatted sequential input and raw input are presented later in this chapter.

## **Sending Formatted Sequential Output**

Formatted sequential output offers your application program more control over the appearance of data in the subwindow than does simple sequential output. For example, formatted sequential output offers your application program a larger set of generic output requests to control various aspects of the subwindow. One such request is `POSITION_CURSOR`, which allows your application program to specify the placement of the cursor in the subwindow.

When your application program specifies a generic output request, the window terminal driver processes the request in a terminal-independent manner (that is, it interprets the request without concern for how a particular terminal implements the request). The window terminal driver then takes the result of this processing and updates the display of the subwindow based on the terminal's capabilities, which are defined in the TTP for the terminal.

To give your application program additional flexibility, formatted sequential output does not perform line wrapping or pause processing. When data exceeds the maximum line length (the maximum width of the subwindow), it is truncated. If you want your application program to wrap lines or handle pause processing, you must design it accordingly.

To send formatted sequential output, your application program must perform the following steps.

1. Ensure that the current I/O subwindow is in formatted I/O mode.
2. Call `s$seq_write` or `s$seq_write_partial`.

If the current I/O subwindow is not in formatted I/O mode when your application program calls either `s$seq_write` or `s$seq_write_partial` (that is, your application program has not created a formatted I/O subwindow, enabled formatted I/O mode in the original subwindow, or issued generic output requests that cause the switch to formatted I/O mode), your application program will not be able to manage the display of the subwindow. Instead, the subroutine will execute in the original subwindow using simple sequential output. (See [Chapter 2](#) for a discussion of simple sequential I/O.)

When the current I/O subwindow is in formatted I/O mode, a call to `s$seq_write` or `s$seq_write_partial` enables the subroutine to handle formatted sequential output. If at any time the formatted sequential output in the subwindow is overlapped (by another subwindow or by another primary window), the status of the port determines how the window terminal driver handles the overlapped output.

If the port is attached with the `hold_open` and `hold_attached` switches turned **off** (the default), the window terminal driver can redisplay the output when the output is no longer overlapped. However, if either of these switches is turned **on**, the window terminal driver cannot guarantee subsequent redisplay of the output. (See the description of `s$attach_port` in [Chapter 6](#) for information about the `hold_open` and `hold_attached` switches.)

## Issuing Generic Output Requests Available with Formatted Sequential Output

With formatted sequential output, you can use a variety of generic output requests in your application program to perform output functions without concern for the type of terminal receiving the output.

Your application program specifies a generic output request by placing a multibyte generic output sequence in an output buffer. The first byte of the generic output sequence is always the generic sequence introducer 27 decimal (1B hexadecimal). The generic sequence introducer is followed by the code representing the generic output request, which is followed by any required arguments.

[Table 4-1](#) lists (in alphabetical order) the generic output requests available with formatted sequential output and represents the requests using the style of the system include files `output_sequence_codes.incl.pl1` and `output_sequence_codes.incl.c`. For example, the `POSITION_CURSOR` request is represented as `POSITION_CURSOR_SEQ` in the include files. [Table 4-1](#) also

identifies the decimal codes associated with these requests. For complete descriptions of the generic output requests and their arguments, see [Chapter 5](#).

**Table 4-1. Generic Output Requests Available with Formatted Sequential Output** (Page 1 of 2)

Generic Output Request	Code
BEEP_SEQ	69
BLANK_OFF_SEQ	130
BLANK_ON_SEQ	129
BLINK_OFF_SEQ	128
BLINK_ON_SEQ	127
BOLDFACE_OFF_SEQ	101
BOLDFACE_ON_SEQ	100
CLEAR_SCROLLING_REGION_SEQ	2
CLEAR_TO_EOL_SEQ	3
CLEAR_TO_EOR_SEQ	4
CURSOR_OFF_SEQ	11
CURSOR_ON_SEQ	10
DELETE_CHARS_SEQ	15
DELETE_LINES_SEQ	17
DISPLAY_BLOCK_SEQ	33
DOWN_SEQ	6
ENTER_GRAPHICS_MODE_SEQ	27
HALF_INTENSITY_OFF_SEQ	26
HALF_INTENSITY_ON_SEQ	25
HI_INTENSITY_OFF_SEQ	126
HI_INTENSITY_ON_SEQ	125
INSERT_CHARS_SEQ	14
INSERT_LINES_SEQ	16
INVERSE_VIDEO_OFF_SEQ	24

**Table 4-1. Generic Output Requests Available with Formatted Sequential Output** (Page 2 of 2)

Generic Output Request	Code
INVERSE_VIDEO_ON_SEQ	23
LEAVE_GRAPHICS_MODE_SEQ	28
LEFT_SEQ	7
NEW_LINE_SEQ	61
POSITION_CURSOR_SEQ	1
RESET_TERMINAL_SEQ	22
RIGHT_SEQ	8
SCREEN_OFF_SEQ	94
SCREEN_ON_SEQ	93
SCROLL_DOWN_SEQ	31
SCROLL_UP_SEQ	32
SET_CURSOR_FORMAT_SEQ	38
SET_MODE_ATTRIBUTES_SEQ	133
SET_SCROLLING_REGION_SEQ	95
STANDOUT_OFF_SEQ	132
STANDOUT_ON_SEQ	131
UNDERSCORE_OFF_SEQ	99
UNDERSCORE_ON_SEQ	98
UP_SEQ	5

## Generic Output Requests That Affect Subwindow Updates

Certain generic output sequences are handled differently by the window terminal driver in terms of subwindow updates. By default, the window terminal driver processes the entire output string in the buffer internally before updating the display of the formatted sequential subwindow.

However, if the output string contains output sequences for any of the following generic output requests, the window terminal driver does not wait to process the entire output string. Instead, before processing the rest of the string, the window terminal driver

updates the display of the subwindow based on the changes made by the generic output sequences issued **up to that point**.

```
CLEAR_SCROLLING_REGION_SEQ  
CLEAR_TO_EOR_SEQ  
SCREEN_OFF_SEQ  
SCREEN_ON_SEQ
```

Your application program can also use the `s$control` opcode `TERM_ENABLE_IMMED_PAINT` (2097) to enable the window terminal driver to update the display after it processes **each** generic output sequence. (For a description of this opcode, see the section “Opcodes Affecting the Primary Window and Subwindow” in [Chapter 8](#).)

In general, once the window terminal driver processes a generic output sequence internally, it updates the display of the subwindow based on the terminal-specific information provided in the output and attributes sections of the TTP. These sections of the TTP define the capabilities of the terminal. Each capability is associated with a sequence that must be sent to the terminal in order for the terminal to perform the particular function. For more information about capabilities and the sections of the TTP, see the manual *VOS Communications Software: Defining a Terminal Type* (R096).

#### NOTE

In general, terminals that do not have the most basic capabilities (for example, the ability to position the cursor anywhere on the screen) are called *glass TTYs*. These terminals cannot use formatted I/O mode, perform forms processing, or have multiple primary windows or subwindows.

## Retrieving Formatted Sequential Input

Although formatted sequential input does not offer the input-mapping flexibility available with processed raw input, it does offer your application program more flexibility than simple sequential input. Basically, formatted sequential input is not as structured as simple sequential input. For example, when your application program uses formatted sequential input, a call to `s$seq_read` does **not** force the input line to appear at the bottom of the subwindow. Instead, formatted sequential input allows your application program to position the input line anywhere in the subwindow.

Formatted sequential input does not process or display (echo) any characters that the terminal user types before a read operation (typeahead). These characters are retained in the internal input buffer until a read operation occurs. It does not matter what



the terminal user types (including any line-editing generic input requests) because characters are not translated or echoed to the subwindow until a read operation occurs. Your application program can disable the character echoing that occurs **during** a read operation by using the `s$control` opcode `TERM_DISABLE_ECHO` (2043). This opcode affects **all** subsequent read operations in **all** subwindows in the primary window until echoing is reenabled with the `s$control` opcode `TERM_ENABLE_ECHO` (2042). (For a description of the opcode pair `TERM_DISABLE_ECHO` (2043) and `TERM_ENABLE_ECHO` (2042), see the section “Opcodes Affecting the Primary Window and Subwindow” in [Chapter 8](#).)

To retrieve formatted sequential input, your application program must perform the following steps.

1. Ensure that the current I/O subwindow is in formatted I/O mode.
2. Call `s$seq_read`.

Note that if the current I/O subwindow is not in formatted I/O mode, `s$seq_read` will execute in the original subwindow using simple sequential input (see [Chapter 2](#)).

## Input Handling with Formatted Sequential Input

When the current I/O subwindow is in formatted I/O mode and your application program calls `s$seq_read`, the window terminal driver processes and echoes characters from the input buffer until it detects a record terminator or reaches the edge of the subwindow. Once the window terminal driver detects a record terminator, your application program can issue additional reads to retrieve any input that remains in the internal input buffer. (Note that both `s$seq_read` and `s$read_raw` can retrieve input from this internal buffer.)

If the window terminal driver reaches the edge of the subwindow before it detects a record terminator, it generates a beep to indicate that it is discarding any characters that cannot fit on the input line. Note that the terminal user can issue generic input requests to edit the input line (for example, the `ERASE_CHAR` request).

When your application program issues a read, the position of the cursor at that time determines the location of the input line. However, if your application program or the terminal user sets a prompt character string, the prompt appears at the cursor position and the input line immediately follows the prompt. Your application program can set a prompt character string using the `s$control` opcode `TERM_SET_PROMPT_CHARS` (2073). The terminal user can set a prompt character string using the `set_terminal_parameters` command. (For a description of the opcode `TERM_SET_PROMPT_CHARS` (2073), see the section “Opcodes Affecting the Primary Window and Subwindow” in [Chapter 8](#). See the *VOS Commands Reference Manual* (R098) for a description of the `set_terminal_parameters` command.)

When your application program is using wait mode, a call to `s$seq_read` remains active until the window terminal driver detects a record terminator. When your

application program is using no-wait mode, `s$seq_read` returns the error code `e$caller_must_wait` (1277) if the window terminal driver does not detect a record terminator. For more information about `s$seq_read`, see [Chapter 7](#).

## Generic Input Requests Available with Formatted Sequential Input

Most of the generic input requests available with simple sequential input are also available with formatted sequential input. These generic input requests enable the window terminal driver to perform line-editing functions for the terminal user without relying on the application program.

[Table 4-2](#) lists (in alphabetical order) the generic input requests available with formatted sequential input and represents the requests using the style of the system include files `video_request_defs.incl.pl1` and `video_request_defs.incl.c`. For example, the TAB request is represented as `TAB_REQ` in the include files. [Table 4-2](#) also lists the decimal codes and TTP request specifiers associated with the requests. Although the window terminal driver uses the decimal codes to identify the requests, it does not return these codes to your application program with formatted sequential input. Additional generic input requests are available with the other types of terminal I/O. (For more information about the generic input requests, see [Chapter 5](#).)

**Table 4-2. Generic Input Requests Available with Formatted Sequential Input** (Page 1 of 3)

Generic Input Request	Code	TTP Request Specifier
ABORT_OUTPUT_REQ	212	abort-output
BACK_TAB_REQ	33	back-tab
BLANKS_LEFT_REQ	60	blanks,left
BLANKS_RIGHT_REQ	61	blanks,right
BREAK_CHAR_REQ	5	break
CANCEL_REQ	13	cancel
CHANGE_CASE_DOWN_REQ	87	change-case,down
CHANGE_CASE_UP_REQ	86	change-case,up
CLEAR_STATUS_REQ	30	clear-status
DELETE_BLANKS_REQ	88	delete,blanks
DELETE_CHAR_REQ	2	del
DELETE_LEFT_REQ	90	delete,left

**Table 4-2. Generic Input Requests Available with Formatted Sequential Input** (Page 2 of 3)

Generic Input Request	Code	TTP Request Specifier
DELETE_RIGHT_REQ	91	delete,right
DELETE_WORD_REQ	23	delete,word
DISCARD_REQ	49	discard
ECHO_BEL_REQ	6	bel
EGI_REQ	189	egi
EMI_REQ	188	emi
END_OF_FILE_REQ	194	end-of-file
ENTER_REQ	1	enter
ERASE_CHAR_REQ	3	back-space
ERASE_FIELD_REQ	12	erase-field
ESI_REQ	187	esi
FORM_REQ	15	display-form
GOTO_BEGINNING_REQ	16	goto,beginning
GOTO_END_REQ	17	goto,end
HELP_REQ	57	help
INSERT_DEFAULT_REQ	47	insert-default
INSERT_SAVED_REQ	14	insert-saved
INTERRUPT_REQ	190	interrupt
LEFT_REQ	18	left
NO_PAUSE_REQ	28	no-pause
REDISPLAY_REQ	24	redisplay
REPEAT_LAST_REQ	42	repeat-last
RETURN_REQ	4	return
RIGHT_REQ	19	right
TAB_REQ	0	tab
TAB_STOP_LEFT_REQ	62	tab-stop,left
TAB_STOP_RIGHT_REQ	63	tab-stop,right

**Table 4-2. Generic Input Requests Available with Formatted Sequential Input** (Page 3 of 3)

Generic Input Request	Code	TTP Request Specifier
TOGGLE_OVERLAY_MODE_REQ	115	en/disable-overlay-mode
TWIDDLE_REQ	201	twiddle
UPDATE_STATUS_REQ	29	update-status
WORD_CHANGE_CASE_DOWN_REQ	133	word,change-case,down
WORD_CHANGE_CASE_LEFT_REQ	131	word,change-case,left
WORD_CHANGE_CASE_UP_REQ	132	word,change-case,up
WORD_LEFT_REQ	21	word,left
WORD_RIGHT_REQ	22	word,right

## Retrieving Raw Input from a Formatted I/O Subwindow

In addition to retrieving formatted sequential input, your application program can retrieve either unprocessed raw input or processed raw input from a formatted I/O subwindow. As described in [Chapter 3](#), unprocessed raw input is untranslated input that allows your application program to control the flow of data by using interrupt characters.

If you want your application program to retrieve unprocessed raw input in the subwindow, the application program must specify an unprocessed raw input mode with the `s$control opcode TERM_SET_INPUT_MODE (2057)` before calling `s$read_raw`. [Chapter 3](#) describes the three unprocessed raw input modes: normal raw mode, raw table mode, and raw record mode.

## Controlling Input Mapping with Processed Raw Input

Unlike unprocessed raw input, *processed raw input* allows your application program to control the amount of input mapping performed by the window terminal driver. *Input mapping* allows your application program to be terminal independent (that is, free of terminal-specific information).

With the processed raw input modes, the window terminal driver can perform various levels of input mapping by using various databases established by the terminal type (TTP) for the device. (A TTP defines all of the information that is specific to a particular type of terminal or printer, including the raw input and output sequences associated with the device, as well as its characteristics and capabilities.) The window terminal driver also performs a certain amount of input mapping for sequential input, as described in [Chapter 2](#) and in the section “Retrieving Formatted Sequential Input” earlier in this chapter.

If you want your application program to take advantage of the window terminal driver's input processing while maintaining control over the display of the subwindow, use processed raw input with formatted sequential output. Note, however, that formatted I/O mode has no effect on the processing of raw input, and your application program can use processed raw input with another type of output. The remainder of this chapter focuses on processed raw input.

There are three processed raw input modes.

- Translated input mode
- Function-key input mode
- Generic input mode

Each processed raw input mode offers your application program certain advantages and a different level of input processing. When your application program uses translated input mode, the window terminal driver performs the least amount of mapping; when your application program uses generic input mode, the window terminal driver performs the most mapping.

By default, the window terminal driver uses generic input mode. To use translated input mode or function-key input mode, or to change from another raw input mode to generic input mode, your application program must specify the appropriate input mode with the `s$control` opcode `TERM_SET_INPUT_MODE (2057)` before calling `s$read_raw`. (For a description of the opcode `TERM_SET_INPUT_MODE (2057)`, see the section “Opcodes Affecting the Primary Window and Subwindow” in [Chapter 8](#).)

#### NOTE

If your application program selects function-key input mode or generic input mode by specifying the old asynchronous driver opcode `SET_MODES (207)` or `SET_INFO (202)` instead of the opcode `TERM_SET_INPUT_MODE (2057)`, `s$read_raw` will return the processed input in the style used by the old asynchronous driver, not the style used by the window terminal driver. For a description of the style used by the old asynchronous driver for returning processed input, see the section “The Function of Processed Raw Input” in Appendix A.

[Figure 4-1](#) shows the different levels of raw input processing performed by the window terminal driver.

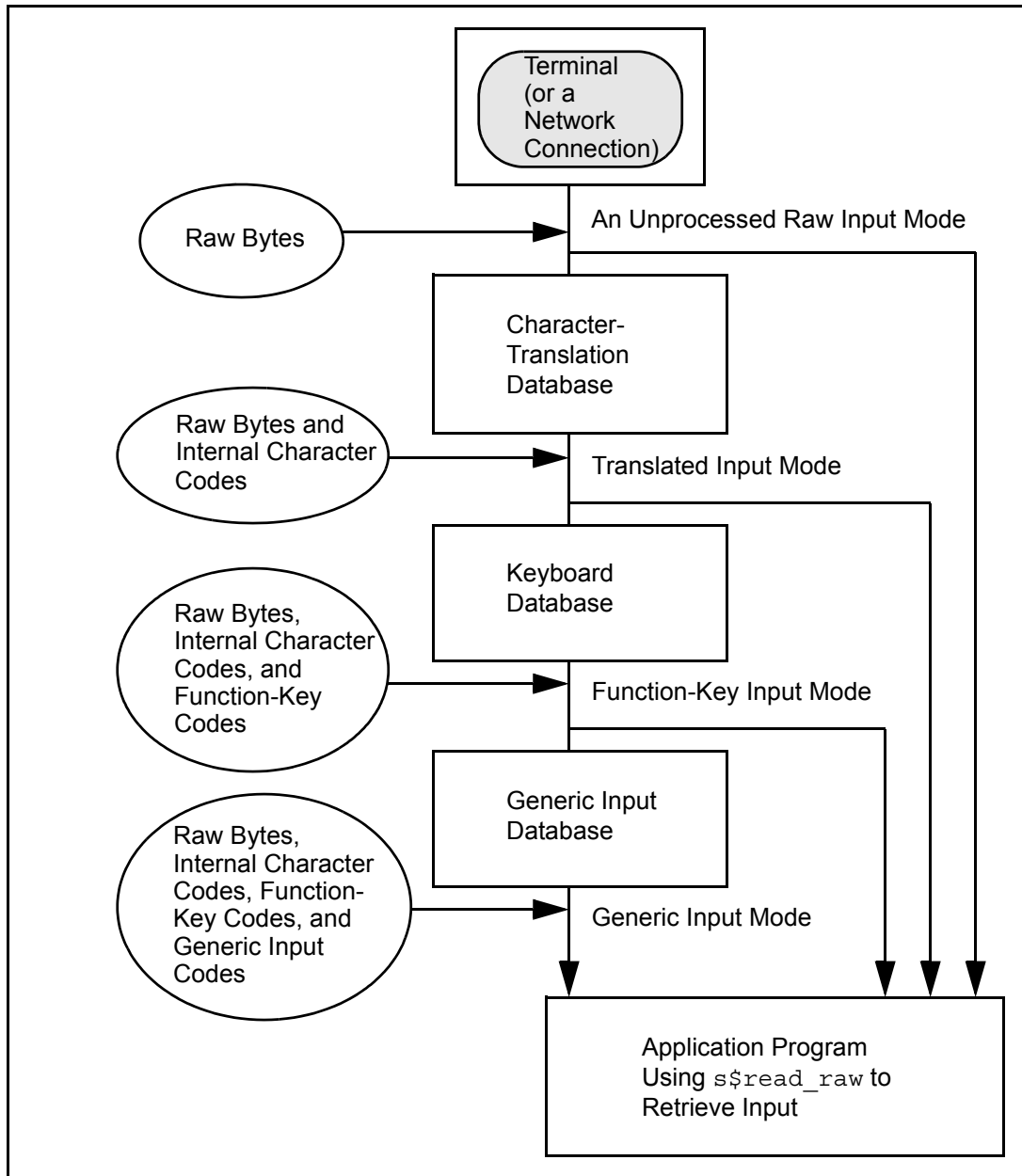


Figure 4-1. Levels of Input Processing

The following list summarizes the levels of raw input processing performed by the window terminal driver.

- If your application program is using an **unprocessed raw input mode** with `s$read_raw`, the operating system does not perform any input mapping. All characters are raw bytes, and `s$read_raw` passes them to your application program according to the unprocessed raw input mode being used. The unprocessed raw input modes use interrupts to control the flow of data in various ways. See [Chapter 3](#) for more information about the unprocessed raw input modes.
- If your application program is using **translated input mode** with `s$read_raw`, the subroutine retrieves the raw bytes and the window terminal driver compares them with the character-translation database for the TTP associated with the device. ([Appendix B](#) illustrates the internal character coding system from which character-translation databases are derived. The coding system is a matrix of hexadecimal codes that translates data using standard control sets and defined international character sets.) If the window terminal driver can map the raw bytes to the database, the matched raw bytes are replaced by internal character codes, which are then passed to your application program without further processing.
- If your application program is using **function-key input mode** with `s$read_raw`, the window terminal driver performs the mapping outlined in the preceding bulleted item, and then compares the internal character codes and raw bytes against the keyboard database derived from the TTP to determine if the codes and bytes map to a function key. (A *function key* transmits a nongraphic sequence of bytes and can perform common editing functions and other functions. The `[F1]` through `[F20]` keys on the V103 ASCII keyboard, the arrow keys, and the `[Back Space]` key are function keys.) If the window terminal driver can map the codes and bytes to function keys, the matching codes and bytes are replaced by the appropriate function-key codes (key numbers). These function-key codes are then passed to your application program, along with the internal character codes and raw bytes that did not map to function keys.
- If your application program is using **generic input mode** with `s$read_raw`, the window terminal driver performs the mapping outlined in the preceding two bulleted items, and then compares the remaining internal character codes, raw bytes, and function-key codes against the generic input database derived from the TTP to determine if they map to a generic input request. (A *generic input request* represents a common editing function and is defined as a sequence of bytes or keys. For example, the V103 function key `[F19]` can be mapped to the generic input request `FORM`, which displays a form on the terminal screen.) If the window terminal driver can perform the mapping, the matching codes and bytes are replaced by generic input request codes and passed to your application program, along with the internal character codes and bytes that did not map to a generic input request. (See [Chapter 5](#) for a list of the generic input requests.)

As described above, generic input mode is characterized by its ability to return generic input requests. In general, translated input mode and function-key input mode do not return generic input requests. One case in which translated input mode and function-key input mode do return a generic input request is when the `REDISPLAY` request is generated. This request is generated internally to indicate that the display of the subwindow has changed; your application program may want to refresh the display of the subwindow.

## Decoding the Input Returned in Binary Data Introducer (BDI) Sequences

To return function keys and generic input requests in the appropriate input mode, `s$read_raw` uses binary data introducer (BDI) sequences. Each call to `s$read_raw` can return multiple BDI sequences, where each function key and generic input request uses one BDI sequence. In generic input mode, `s$read_raw` also uses a BDI sequence to return the first raw byte of a character string that fails to map to a valid input sequence. Internal character strings (translated input) do not use BDI sequences with `s$read_raw`; instead, they are simply embedded in the input stream.

All BDI sequences use the following structure, which is the BDI header.

In PL/I:

```
declare 1 bdi_header based,
        2 introducer char (1),
        2 type      char (1),
        2 len       fixed bin (15),
        2 req_no    fixed bin (15);
```

In C:

```
struct bdi_header
{
    char      introducer;
    char      type;
    short int len;
    short int req_no;
};
```



Each BDI sequence begins with the BDI header, which consists of four parts.

- The **introducer** (147 decimal, 93 hexadecimal) is always the first part of a BDI sequence.
- The **asynchronous BDI data type** (130 decimal, 82 hexadecimal) always follows the introducer.
- The **binary data length** specifies the number of bytes used to identify the generic input request, function key, or raw byte.
- The **request number** specifies the code associated with a generic input request.

To return a generic input request, `s$read_raw` requires only the BDI header.

Therefore, a BDI sequence for a generic input request always consists of six bytes. The first two bytes are the BDI introducer and the BDI data type. The next two bytes identify the binary data length. The binary data length for a generic input request is always two bytes, since each request is identified by a 2-byte generic input request code (the request number). The last two bytes of the sequence always identify the actual request number. (See the section “[Using Generic Input Mode](#)” later in this chapter for more information about the BDI sequences for generic input requests.)

To return a function key, `s$read_raw` uses the BDI header as well as a function-key structure. When a function key is available, the request number in the BDI header identifies the `FUNCTION_KEY` request (209 decimal, 00D1 hexadecimal). This request is used by `s$read_raw` to introduce that function key, which is then identified in the following function-key structure.

#### In PL/I:

```
declare 1 FUNCTION_KEY_REQ,
        2 header like bdi_header,
        2 key_number      fixed bin (15),
        2 shift_modifier  char (1);
```

#### In C:

```
struct FUNCTION_KEY_REQ
{
    struct bdi_header header;
    short int      key_number;
    char           shift_modifier;
};
```

The entire BDI sequence for a function key consists of nine bytes. The first two bytes are the BDI introducer and the BDI type. The next two bytes identify the binary data length. The binary data length for a function key is five bytes, which consists of the `FUNCTION_KEY` request, a function-key number, and a shift modifier. The function-key number is a value in the range 0 to 511 that represents a function key. The shift modifier is a value in the range 0 to 7 that represents one of the eight shift modifiers available. See the section “[Using Function-Key Input Mode](#)” later in this chapter for more information about the shift modifiers and function-key numbers.

To return a raw byte in generic input mode, `s$read_raw` uses the BDI header as well as a raw input structure. The raw byte returned is actually the first byte of a character string that `s$read_raw` failed to map to a valid input sequence. When such a raw byte is available, the request number in the BDI header identifies the `RAW_INPUT` request (205 decimal, 00CD hexadecimal). This request is used by `s$read_raw` to introduce the individual raw byte, which is then identified in the following raw input structure.

**In PL/I:**

```
declare 1 RAW_INPUT_REQ,
        2 header like bdi_header,
        2 raw_byte    char (1);
```

**In C:**

```
struct RAW_INPUT_REQ
{
    struct bdi_header header;
    char                raw_byte;
};
```

For example, the equal sign (=) would be returned in a BDI sequence as follows:

```
'93    '82    '00'03    '00'CD    '3D
BDI type length RAW_INPUT_REQ byte
```

In this example, the binary data length is 3. The `RAW_INPUT` request takes up two bytes and the representation of the = character takes up one byte.

**NOTE** \_\_\_\_\_

Future modifications to the appropriate TTP may affect the behavior of the `RAW_INPUT` request. For example, if

the input sequences in a TTP are modified, the byte that was previously returned with the `RAW_INPUT` request may now begin a valid input sequence. In this case, the byte that was previously returned is no longer returned in the BDI sequence. In general, your application program should be flexible in handling `RAW_INPUT` BDI sequences, since they may be subject to change.

The sections that follow describe the three types of processed raw input in more detail.

## Using Translated Input Mode

In translated input mode, `s$read_raw` allows your application program to respond to the transmission of certain internal character-set characters from the terminal. Your application program can use this mode to recognize and handle characters from standard and language-specific character sets in a terminal- and language-independent manner. (See the description of National Language Support (NLS) in [Chapter 1](#).)

In translated input mode, `s$read_raw` attempts to map raw bytes from the terminal to the character-translation database for the TTP. Translated input consists of characters defined in one or more character-translation sections of the TTP. Each character-translation section of the TTP is based on the internal character coding system (shown in [Appendix B](#)), which includes ASCII characters and characters from character sets such as Latin alphabet No. 1 and kanji. The character-translation sections of a TTP generate a character-translation database for the TTP associated with the device.

To use translated input mode, your application program must specify this mode with the `s$control` opcode `TERM_SET_INPUT_MODE (2057)` before calling `s$read_raw`. (For a description of this opcode, see the section “Opcodes Affecting the Primary Window and Subwindow” in [Chapter 8](#).)

When `s$read_raw` can successfully map raw bytes to the character-translation database, it replaces the bytes in the buffer with the internal character codes that represent the translated characters. It returns these codes in the buffer, along with any characters that did not map to the database.

The following example illustrates how `s$read_raw` can return translated input in the input buffer.

```
`8E`CC`DA `61 `62 `63
```

In this example, the first three bytes represent the 木 character. The first byte is 8E hexadecimal (the kanji single-shift introducer), and the next two bytes are CCDA

hexadecimal (the hexadecimal equivalent of the character). The next three bytes represent the characters a, b, and c (61, 62, and 63 hexadecimal).

Using Function-Key Input Mode

Function-key input consists of translated input and function keys. Function keys define certain keys on the terminal keyboard according to the byte sequence that each key transmits, and also define the key legend (that is, the representation of the key on the terminal keyboard). In function-key input mode, your application program can retrieve function-key input without regard for the type of terminal being used, and can assign its own meaning to input keystrokes.

There are 4,096 possible function keys. Of these, the first 512 are the basic function keys that have a function-key name and a function-key number. The function-key numbers range from 0 to 511 decimal (00 to 01FF hexadecimal). Table 4-3 identifies the basic function keys and their respective key numbers. (Function keys are also defined in the include files `function_key_codes.incl.pll` and `function_key_codes.incl.c`.)

Table 4-3. Basic Function Keys (Page 1 of 2)

Function Key	Key Code (Number)	Description
f0-key	0	The numbered function keys that typically appear at the top of the keyboard or in the keypad on the right side of the keyboard.
f1-key	1	
f2-key	2	
.	.	
.	.	
.	.	The cursor-control keys.
f31-key	31	
up-key	32	
down-key	33	
left-key	34	
right-key	35	
goto-key	36	Typically, the [HOME] key.
enter-key	37	The [ENTER] key.
	38-43	Reserved for future use.

**Table 4-3. Basic Function Keys** (Page 2 of 2)

Function Key	Key Code (Number)	Description
misc-0-key	44	Terminal-specific keys that have no predefined meaning or keyboard location (for example, the <code>[STATUS]</code> key on the V103 terminal keyboard).
misc-1-key	45	
misc-2-key	46	
.	.	
.	.	
misc-15-key	59	Reserved for future use.
	60-242	
backspace-key	243	
tab-key	244	
linefeed-key	245	
return-key	256	Most ASCII terminals have these keys. These keys are assigned codes so that alternate or multiple sequences can be defined for them.
escape-key	247	
del-key	248	
break-key	249	
	250-254	
hex-value-key	255	Precedes the hexadecimal representation of a byte value so that the terminal can transmit it to the operating system.
space-key	256	Keys for the space character and each graphic character in the ASCII character set. The code for each key is the ASCII code for the corresponding character plus 224 (decimal).
!-key	257	
"-key	258	
.	.	
.	.	
~-key	350	Reserved for future use.
	351-511	

The basic function keys provide a wider range of function keys when they are pressed in conjunction with *shift modifiers*. The `[SHIFT]`, `[CTRL]`, and `[ALT]` (or `[FUNCT]`) keys are the typical shift modifiers. A shift modifier can be pressed with the function key alone or in

combination with other shift modifiers and the function key. There are eight shift-modifier combinations available. [Table 4-3](#) lists the function keys without any shift modifiers.

[Table 4-3](#) indicates that there are function-key numbers for most graphic key legends. These graphic key legends do not always produce graphic characters when pressed in conjunction with some of the shift modifiers (that is, `CTRL-A` is not a graphic character). In general, graphic key legends are returned to your application program **only** if they are defined in the TTP. (See [Appendix F](#) for a list of the graphic characters.)

All function keys must be defined in the keyboard section of the TTP using the key name, followed by a shift-modifier value. Each function-key entry in the TTP also contains the key legend that identifies the function key on the keyboard as well as the byte sequence transmitted when the function key is pressed.

The shift-modifier values represent a shift modifier or a combination of shift modifiers. A shift modifier of `*0` in the TTP means no shift modifier. A sample mapping for shift modifiers in the TTP follows.

*0	No Shifts
*1	Shift
*2	Control
*3	Control+Shift
*4	Alt
*5	Alt+Shift
*6	Alt+Control
*7	Alt+Control+Shift

The following example shows how three shift modifiers (`SHIFT`, `CTRL`, and `CTRL` plus `SHIFT`, represented by `*1`, `*2`, and `*3`, respectively) can be used to define a function key (`f2-key`) in the TTP. The example also indicates which byte sequences are transmitted when the key is pressed alone and with the appropriate shift modifier (for example, the sequence `ESC ! b ETX` is transmitted when the `F2` key and the `SHIFT` key are pressed simultaneously).

<code>f2-key</code>	<code>F2</code>	<code>ESC b ETX</code>
<code>*1</code>		<code>ESC ! b ETX</code>
<code>*2</code>		<code>ESC ESC b ETX</code>
<code>*3</code>		<code>ESC ESC ! b ETX</code>

When your application program is using `s$read_raw` with function-key input mode, the window terminal driver compares input character sequences with the keyboard database derived from the keyboard section of the TTP associated with the device. (See the manual *VOS Communications Software: Defining a Terminal Type* (R096) for information about the keyboard section of the TTP.) The result of the database mapping can be graphic characters or function keys.

If the TTP can map any input to a function key, `s$read_raw` returns the byte sequence associated with that function key, along with any bytes that did not map to function keys. Each function key in the input stream is returned to your application program using a 9-byte BDI sequence, which consists of the BDI header and a function-key structure. As described earlier in the section “[Decoding the Input Returned in Binary Data Introducer \(BDI\) Sequences](#),” the BDI header consists of the introducer, data type, binary data length, and request number. When `s$read_raw` retrieves a function key, the request number is 209 decimal (00D1 hexadecimal), which represents the `FUNCTION_KEY` request. The function key is then identified in the function-key structure, which consists of the function-key number and the shift-modifier value.

For example, `[F1]` (`f1-key`) uses the key number 1 (0001 hexadecimal). When this function key uses a shift modifier of 0 and is embedded in the input stream, `s$read_raw` returns the following BDI sequence in the input buffer.

```
'93  '82  '00'05          '00'D1          '00'01          '00
BDI type length FUNCTION_KEY_REQUEST f1-key shift_modifier
```

To use function-key input mode, your application program must specify this mode with the `s$control` opcode `TERM_SET_INPUT_MODE` (2057) before calling `s$read_raw`. (For a description of this opcode, see the section “Opcodes Affecting the Primary Window and Subwindow” in [Chapter 8](#).)

## Using Generic Input Mode

Generic input consists of generic input requests, function keys, and internal character codes. By using generic input mode, your application program can use the complete set of generic input requests to handle terminal input without concern for the type of terminal. The window terminal driver translates the terminal-specific input into generic input by using the generic input database derived from one or more input sections of the TTP associated with the device.

[Chapter 5](#) presents a subset of the generic input requests; the system include files `video_request_defs_incl.pll` and `video_request_defs_incl.c` define the complete set of generic input requests. Since `s$read_raw` supports the complete set of generic input requests in generic input mode, `s$read_raw` can return any request defined in `video_request_defs_incl.pll` or `video_request_defs_incl.c`, as long as the TTP defines the request. Although most of the requests defined in the include file perform editing functions, some are reserved for other functions. For example, the `FUNCTION_KEY` and `RAW_INPUT` requests do not perform editing functions. Instead, `s$read_raw` uses the `FUNCTION_KEY` request to introduce a function key and uses the `RAW_INPUT` request to introduce a raw byte that did not map to anything in the character-translation, keyboard, or generic input database.

When your application program specifies generic input mode and calls `s$read_raw`, the call returns generic input request codes (request numbers) in BDI sequences. The

generic input request codes are in the range 0 to 2047 decimal (00 to 07FF hexadecimal). The call to `s$read_raw` also returns any internal character codes, function-key codes (key numbers), and raw bytes that did not map to generic input requests in the generic input database. (The first byte of a string that did not map to an internal character, function key, or input request is returned in a `RAW_INPUT` BDI sequence, as described earlier in this chapter in the section “[Decoding the Input Returned in Binary Data Introducer \(BDI\) Sequences.](#)”)

To return each generic input request embedded in the input stream, `s$read_raw` uses a 6-byte BDI sequence that consists of the BDI header. (The BDI header is described earlier in this chapter in the section “[Decoding the Input Returned in Binary Data Introducer \(BDI\) Sequences.](#)” ) For example, the `LEFT` request is associated with decimal code 18 (0012 hexadecimal). When this request is embedded in the input stream, `s$read_raw` returns the following BDI sequence in the input buffer.

```
'93  '82  '00'02  '00'12
BDI type length LEFT_REQ
```



---

## Chapter 5

# Generic Input Requests and Generic Output Requests

This chapter describes the generic input requests and generic output requests supported by the window terminal driver. The first part of this chapter describes how the window terminal driver handles generic input requests. The second part of this chapter describes how to design your application program to send generic output requests and provides detailed descriptions of the generic output requests.

## Handling Generic Input Requests

*Generic input requests* are requests that the terminal user submits to the operating system or to an application program to perform a predefined set of common operations. Generic input requests typically perform common line-editing functions, such as entering a request, canceling a request, or requesting help.

Generic input requests allow your application program to be terminal independent (that is, your application program need not contain terminal-specific information to interpret input sequences). The window terminal driver recognizes generic input requests that perform line-editing functions when your application program is designed to retrieve one of the following types of input.

- Generic input (a type of processed raw input)
- Simple sequential input
- Formatted sequential input (formatted I/O mode under application-managed I/O)
- Forms input
- Input from a supported text editor, such as the Word Processing Editor (`edit`)
- Forms Editor input (`nls_edit_form`)

In generic input mode (described in [Chapter 4](#)), the window terminal driver recognizes all of the generic input requests. For both simple and formatted sequential I/O (described in [Chapter 2](#) and [Chapter 4](#), respectively), the window terminal driver recognizes a subset of generic input requests. Other generic input requests are available for forms I/O and the Forms Editor (both of which are described in the VOS Forms Management System manuals), as well as for other text editors.

The system include files `video_request_defs.incl.pll` and `video_request_defs.incl.c` list all of the generic input requests. These include files identify requests that are generated internally by the window terminal driver to indicate certain conditions (for example, the request `INVALID_SEQUENCE` (197) signals an invalid input sequence) and certain types of data. In addition, some requests are specific to window manager mode (for example, the requests `LOGIN_PROCESS` (198), `WINDOW_ACTIVATED` (202), and `LEAVE_WINDOW_MANAGER` (211)).

[Table 5-1](#) alphabetically lists the generic input requests that perform line-editing functions for both types of sequential I/O and forms I/O (but not window manager mode). It also identifies the decimal codes associated with these requests and their TTP request specifiers.

Note that [Table 5-1](#) represents the generic input requests using the style of the system include files. For example, the `FORM` request is represented as `FORM_REQ` in the include files. Note also that in the device's terminal type definition (`.ttp`) file, which is the source for the compiled TTP, some of the generic input requests do not have specifiers with the same name (for example, the `FORM` request has the name `display-form` in the `.ttp` file).

**Table 5-1. Generic Input Requests** (Page 1 of 4)

Generic Input Request	Code	Simple and Formatted Seq. I/O	Forms I/O	TTP Request Specifier
<code>ABORT_OUTPUT_REQ</code>	212	Yes	No	<code>abort-output</code>
<code>BACK_TAB_REQ</code>	33	Yes	Yes	<code>back-tab</code>
<code>BLANKS_LEFT_REQ</code>	60	Yes	Yes	<code>blanks, left</code>
<code>BLANKS_RIGHT_REQ</code>	61	Yes	Yes	<code>blanks, right</code>
<code>BREAK_CHAR_REQ</code>	5	Yes	Yes	<code>break</code>
<code>CANCEL_REQ</code>	13	Yes	Yes	<code>cancel</code>
<code>CHANGE_CASE_DOWN_REQ</code>	87	Yes	Yes	<code>change-case, down</code>
<code>CHANGE_CASE_UP_REQ</code>	86	Yes	Yes	<code>change-case, up</code>
<code>CLEAR_STATUS_REQ</code>	30	Yes	Yes	<code>clear-status</code>
<code>CYCLE_BACK_REQ</code>	27	No	Yes	<code>cycle-back</code>
<code>CYCLE_REQ</code>	56	No	Yes	<code>cycle</code>
<code>DELETE_BLANKS_REQ</code>	88	Yes	Yes	<code>delete, blanks</code>

**Table 5-1. Generic Input Requests** (Page 2 of 4)

<b>Generic Input Request</b>	<b>Code</b>	<b>Simple and Formatted Seq. I/O</b>	<b>Forms I/O</b>	<b>TTP Request Specifier</b>
DELETE_CHAR_REQ	2	Yes	Yes	del
DELETE_LEFT_REQ	90	Yes	Yes	delete, left
DELETE_RIGHT_REQ	91	Yes	Yes	delete, right
DELETE_WORD_REQ	23	Yes	Yes	delete, word
DISCARD_REQ	49	Yes	No	discard
DOWN_REQ	7	Yes (simple); No (fmtd.)	Yes	down
ECHO_BEL_REQ	6	Yes	Yes	bel
EGI_REQ	189	Yes	No	egi
EMI_REQ	188	Yes	No	emi
END_OF_FILE_REQ	194	Yes	No	end-of-file
ENTER_REQ	1	Yes	Yes	enter
ERASE_CHAR_REQ	3	Yes	Yes	back-space
ERASE_FIELD_REQ	12	Yes	Yes	erase-field
ESI_REQ	187	Yes	No	esi
EXIT_FORM_REQ	20	No	Yes	cancel-form
F0_REQ	154	No	Yes	function-key-0
F1_REQ to F32_REQ	155 to 186	No	Yes	function-key-1 to function-key-32
FORM_REQ	15	Yes	No	display-form
GOTO_BEGINNING_REQ	16	Yes	Yes	goto, beginning
GOTO_DOWN_REQ	99	No	Yes	goto, down
GOTO_END_REQ	17	Yes	Yes	goto, end
GOTO_UP_REQ	98	No	Yes	goto, up
HELP_REQ	57	Yes	Yes	help
INSERT_DEFAULT_REQ	47	Yes	Yes	insert-default

**Table 5-1. Generic Input Requests** (Page 3 of 4)

Generic Input Request	Code	Simple and Formatted Seq. I/O	Forms I/O	TTP Request Specifier
INSERT_SAVED_REQ	14	Yes	Yes	insert-saved
INTERRUPT_REQ	190	Yes	Yes	interrupt
LEFT_REQ	18	Yes	Yes	left
NEXT_SCREEN_REQ	58	Yes (simple); No (fmted.)	No	next-screen
NO_PAUSE_REQ	28	Yes	No	no-pause
REDISPLAY_REQ	24	Yes	Yes	redisplay
REPEAT_LAST_REQ	42	Yes	Yes	repeat-last
RETURN_REQ	4	Yes	Yes	return
RIGHT_REQ	19	Yes	Yes	right
SCROLL_DOWN_REQ	73	No	Yes	scroll,down
SCROLL_LEFT_REQ	70	No	Yes	scroll,left
SCROLL_RIGHT_REQ	71	No	Yes	scroll,right
SCROLL_UP_REQ	72	No	Yes	scroll,up
TAB_REQ	0	Yes	Yes	tab
TAB_STOP_LEFT_REQ	62	Yes	Yes	tab-stop,left
TAB_STOP_RIGHT_REQ	63	Yes	Yes	tab-stop,right
TOGGLE_OVERLAY_MODE_REQ	115	Yes	Yes	en/disable-overlay-mode
TWIDDLE_REQ	201	Yes	Yes	twiddle
UP_REQ	8	No	Yes	up
UPDATE_STATUS_REQ	29	Yes	Yes	update-status
WORD_CHANGE_CASE_DOWN_REQ	133	Yes	Yes	word, change-case,down
WORD_CHANGE_CASE_LEFT_REQ	131	Yes	Yes	word, change-case,left
WORD_CHANGE_CASE_UP_REQ	132	Yes	Yes	word, change-case,up

**Table 5-1. Generic Input Requests** (Page 4 of 4)

Generic Input Request	Code	Simple and Formatted Seq. I/O	Forms I/O	TTP Request Specifier
WORD_LEFT_REQ	21	Yes	Yes	word, left
WORD_RIGHT_REQ	22	Yes	Yes	word, right

When your application program is using `s$read_raw` with generic input mode or `s$seq_read` with sequential I/O, the window terminal driver maps the request to the generic input database. The generic input database is an input mapping scheme associated with the TTP used by the device. As described in the manual *VOS Communications Software: Defining a Terminal Type* (R096), a TTP defines terminal-specific characteristics and sequences. Generic input requests are defined in the input section of the TTP. )

The input section of the TTP maps the keys on the keyboard (function keys or other key sequences) to the generic input requests. The TTP can have more than one input section to support different types of editing environments. The TTP always has a **default** input section to ensure that it can support certain basic requests. Note that the default input section does not typically contain the complete set of generic input requests, even though they can all be used by your application program in generic input mode. To determine the name of the current input section for the TTP, your application program uses the `s$control` opcode `TERM_GET_INPUT_SECTION (2058)`; to use an input section in addition to the default input section, your application program uses the `s$control` opcode `TERM_SET_INPUT_SECTION (2059)`. (For a description of these opcodes, see the section “Opcodes Affecting the Primary Window and Subwindow” in [Chapter 8](#).

If the window terminal driver can map the input sequence to a generic input request, a call to `s$read_raw` in generic input mode returns a 6-byte binary data introducer (BDI) sequence to your application program. (The BDI sequences for generic input requests are described in [Chapter 4](#).) Every BDI sequence for a generic input request contains a 2-byte code that identifies the request. For example, if the window terminal driver receives the `HELP` request, the BDI sequence contains the code 57 decimal (0039 hexadecimal). [Table 5-1](#) lists the request codes in decimal.

Note that the `FUNCTION_KEY` request (209 decimal, 00D1 hexadecimal) and the `RAW_INPUT` request (205 decimal, 00CD hexadecimal) are not listed in [Table 5-1](#) because they do not perform editing functions. The `FUNCTION_KEY` request introduces a function key in the BDI sequence and the `RAW_INPUT` request introduces a raw byte in the BDI sequence. (See [Chapter 4](#) for more information about BDI sequences.)

If your application program is using simple sequential I/O when the window terminal driver detects a generic input request, the window terminal driver executes the request

within the current input line. An `s$seq_read` call need not be active. If your application program is using formatted sequential input, an `s$seq_read` call must be active in order for the request to be executed. With either type of sequential input (or with forms I/O), the window terminal driver does **not** return the generic input request codes to your application program.

## Handling Generic Output Requests and Other Translated Output

When your application program sends simple sequential output or formatted sequential output, the output is translated by the window terminal driver in a standard (generic) way. To send either type of sequential output, your application program must call a write subroutine such as `s$seq_write` or `s$seq_write_partial`. ([Chapter 7](#) describes these subroutines.)

Unlike raw output, both types of sequential output pass through the TTP database for the terminal (or printer). To translate the output, the window terminal driver relies on the information in the configuration, character-translation, output, and attributes sections (databases) of the TTP for the terminal. These sections, which are described in the manual *VOS Communications Software: Defining a Terminal Type* (R096), contain the following terminal-specific information.

- The *configuration section* of a TTP specifies how a terminal is set up and how it handles output. It defines output defaults, such as the hexadecimal notation character variable (`hex-notation-char`), which introduces the hexadecimal equivalent of a character for which there is no graphic representation or corresponding capability.
- The *character-translation section* of a TTP defines the mappings between the operating-system internal character sets and the character sets used by the terminal.
- The *output section* of a TTP defines the terminal's capabilities and the raw ASCII sequences that handle requests specified by your application program. The requests that your application program specifies are called *generic output requests*.
- The *attributes section* of a TTP identifies the video-display attributes supported by the terminal and defines how the attributes are implemented.

In general, the translation of sequential output involves the processing of generic output requests; however, it also involves the processing of the following characters:

- National Language Support (NLS) characters (for example, kanji characters)
- Stratus-specific control characters from the internal character coding system
- standard ASCII control characters (for example, BS, HT, and BEL)

- line-graphics characters (by enabling graphics mode with a generic output request)
- new-line characters
- tabs.

#### NOTE

With `$seq_write`, the window terminal driver inserts a new-line character to **wrap** the line (in simple sequential I/O) or to **truncate** the line (in formatted sequential I/O) based on the line length defined for the terminal. If the configuration section of the TTP defines the line length as 80 (the default), the line wraps after the 80th character. To determine the current line length, your application program uses the window terminal opcode `TERM_GET_SCREEN_WIDTH (2028)` or the global opcode `GET_LINE_LENGTH (1)`. Your application program can specify a different line length by verifying and then selecting a different configuration section of the TTP with the opcodes `TERM_GET_TERMINAL_SETUP (2030)` and `TERM_SET_TERMINAL_SETUP (2031)`, respectively. (For more information about these opcodes, see the section “Opcodes Affecting the Terminal” in [Chapter 8](#).)

## Handling Generic Output Requests

Generic output requests allow your application program to perform output functions without concern for the type of terminal displaying the output. For example, your application program can specify a generic output request that clears the scrolling region without having to know the ASCII sequence that causes the particular terminal type to clear the scrolling region.

Your application program can specify a small subset of the generic output requests when using simple sequential I/O and can specify a larger subset when using formatted sequential I/O. Note that if your application program is using simple sequential I/O and specifies a generic output request available only with formatted sequential I/O, the window terminal driver switches the subwindow to formatted I/O mode before executing the request. (See [Chapter 2](#) for more information about this switch to formatted I/O mode.)

[Table 5-2](#) lists the generic output requests commonly used for terminals, along with their respective 1-byte decimal codes. [Table 5-2](#) categorizes the requests according to function and identifies the requests offered by simple sequential I/O and the requests offered by formatted sequential I/O.

**Table 5-2** represents the generic output requests as they appear in the system include files `output_sequence_codes.incl.pll` and `output_sequence_codes.incl.c`. For example, the `POSITION_CURSOR` request is represented as `POSITION_CURSOR_SEQ` in the include files. Note that the style used to define capabilities in the TTP is lowercase with hyphens instead of uppercase with underscores. For example, the capability associated with the `POSITION_CURSOR` request is defined as `position-cursor` in the TTP.

When your application program specifies a generic output request, the window terminal driver processes the request internally, then updates the display of the subwindow based on the information in the TTP for the device. The TTP contains output and attributes sections that define the output capabilities of the terminal. In the output section, each capability is associated with a sequence that enables the terminal to perform the desired function. Therefore, generic output requests must be associated with capabilities in the device's TTP before your application program can use them and the window terminal driver can recognize them.

Your application program can use the TTP subroutine `s$ttp_get_output_cap` to retrieve information about the capabilities defined in the TTP. ([Appendix C](#) briefly describes this subroutine.) If your application program writes an output request to a terminal whose TTP is not able to handle the request, the request is simply ignored and no error code is returned. (For more information about placing terminal-specific information in the TTP, see the manual *VOS Communications Software: Defining a Terminal Type* (R096).)

#### NOTE

Your application program can change the way in which the window terminal driver processes the output string and updates the display of the subwindow. By default, the window terminal driver processes the entire output string internally before updating the display of the subwindow.

Specifying the `s$control` opcode

`TERM_ENABLE_IMMED_PAINT` (2097) causes the window terminal driver to update the display after it processes **each** generic output sequence. (For a description of this opcode, see the section "Opcodes Affecting the Primary Window and Subwindow" in [Chapter 8](#).) If your application program is sending output to a formatted sequential I/O subwindow, a change in the processing of the output string occurs automatically when certain generic output sequences appear in the output string. In this case, the window terminal driver updates the subwindow display with the visible changes **up to that point** before it continues to process the rest of the string. The generic output sequences causing this change



contain one of the following requests:

`CLEAR_SCROLLING_REGION`, `CLEAR_TO_EOR`,  
`SCREEN_ON`, or `SCREEN_OFF` .

## Compatibility Issues Concerning Generic Output Requests

If you are migrating an application program from the old asynchronous driver to the window terminal driver, you should be aware that the two drivers handle some of the generic output requests differently. Since there are differences in how the two drivers handle attributes (the window terminal driver treats all attributes as mode attributes, while the old asynchronous driver handles both mode attributes and area attributes), and since certain window terminal opcodes perform some of the operations previously performed by generic output requests, you should avoid using any generic output requests that are not listed in [Table 5-2](#).

### NOTE

Your application program can continue to use the two generic output requests that have been renamed for the window terminal driver. Although the `CLEAR_SCREEN` (2) and `CLEAR_TO_EOS` (4) requests have been renamed `CLEAR_SCROLLING_REGION` and `CLEAR_TO_EOR`, respectively, each request still performs the same function and uses the same code number.

If you have an application program that works with the old asynchronous driver, see Table A-3 in [Appendix A](#). Table A-3 explains how to replace generic output requests used by the old asynchronous driver. For example, your application program should avoid using output requests such as `SET_ATTRIBUTES` (18) and `SET_ATTRIBUTE_CHAR` (72), since they cannot set markers for area attributes with the window terminal driver. In addition, output requests that previously set screen preferences (such as `KEY_CLICK_ON` (36) and `SET_SMOOTH_SCROLL` (39)) should be replaced by the opcode `TERM_SET_SCREEN_PREF` (2027). (For a description of this opcode, see the section “Opcodes Affecting the Primary Window and Subwindow” in [Chapter 8](#).)

**Table 5-2. Generic Output Requests** (Page 1 of 4)

Generic Output Request	Code	Arguments	Simple Seq. I/O	Formatted Seq. I/O
<b><i>Cursor Operations</i></b>				
<code>CURSOR_OFF_SEQ</code>	11		Yes	Yes
<code>CURSOR_ON_SEQ</code>	10		Yes	Yes

**Table 5-2. Generic Output Requests** (Page 2 of 4)

Generic Output Request	Code	Arguments	Simple Seq. I/O	Formatted Seq. I/O
DOWN_SEQ	6		No	Yes
LEFT_SEQ	7		No	Yes
NEW_LINE_SEQ	61		Yes	Yes
POSITION_CURSOR_SEQ	1	<i>line, column</i>	No	Yes
RIGHT_SEQ	8		No	Yes
SCROLL_DOWN_SEQ	31		No	Yes
SCROLL_UP_SEQ	32		No	Yes
SET_CURSOR_FORMAT_SEQ	38	<i>cursor_format</i> (0 to 4)	Yes	Yes
UP_SEQ	5		No	Yes
<b>Clearing Operations</b>				
CLEAR_SCROLLING_REGION_SEQ	2		No	Yes
CLEAR_TO_EOL_SEQ	3		No	Yes
CLEAR_TO_EOR_SEQ	4		No	Yes
<b>Line Operations</b>				
DELETE_CHARS_SEQ	15	<i>line, column, number_of_chars</i>	No	Yes
DELETE_LINES_SEQ	17	<i>line, number_of_lines</i>	No	Yes
INSERT_CHARS_SEQ	14	<i>line, column, number_of_chars</i>	No	Yes
INSERT_LINES_SEQ	16	<i>line, number_of_lines</i>	No	Yes
SET_SCROLLING_REGION_SEQ	95	<i>top_line, bottom_line</i>	No	Yes

Table 5-2. Generic Output Requests (Page 3 of 4)

Generic Output Request	Code	Arguments	Simple Seq. I/O	Formatted Seq. I/O
<b>Visual Attributes</b>				
BLANK_OFF_SEQ	130		Yes	Yes
BLANK_ON_SEQ	129		Yes	Yes
BLINK_OFF_SEQ	128		Yes	Yes
BLINK_ON_SEQ	127		Yes	Yes
BOLDFACE_OFF_SEQ	101		Yes	Yes
BOLDFACE_ON_SEQ	100		Yes	Yes
HALF_INTENSITY_OFF_SEQ	26		Yes	Yes
HALF_INTENSITY_ON_SEQ	25		Yes	Yes
HI_INTENSITY_OFF_SEQ	126		Yes	Yes
HI_INTENSITY_ON_SEQ	125		Yes	Yes
INVERSE_VIDEO_OFF_SEQ	24		Yes	Yes
INVERSE_VIDEO_ON_SEQ	23		Yes	Yes
SET_MODE_ATTRIBUTES_SEQ	133	2-byte attribute	Yes	Yes
STANDOUT_OFF_SEQ	132		Yes	Yes
STANDOUT_ON_SEQ	131		Yes	Yes
UNDERSCORE_OFF_SEQ	99		Yes	Yes
UNDERSCORE_ON_SEQ	98		Yes	Yes
<b>Modes</b>				
ENTER_GRAPHICS_MODE_SEQ	27		No	Yes
LEAVE_GRAPHICS_MODE_SEQ	28		No	Yes
<b>Miscellaneous</b>				
BEEP_SEQ	69		Yes	Yes
DISPLAY_BLOCK_SEQ	33		No	Yes
RESET_TERMINAL_SEQ	22		Yes	Yes

Table 5-2. Generic Output Requests (Page 4 of 4)

Generic Output Request	Code	Arguments	Simple Seq. I/O	Formatted Seq. I/O
SCREEN_OFF_SEQ	94		Yes	Yes
SCREEN_ON_SEQ	93		Yes	Yes

To specify a generic output request, your application program must place a multibyte generic output sequence in an output buffer and call `s$seq_write` or `s$seq_write_partial`. (Note that when your application program uses `$seq_write`, a new-line sequence is appended to the output string automatically.)

Your application program must build a generic output sequence as follows:

- Begin the sequence with the generic sequence introducer (GSI). The introducer is 27 decimal (1B hexadecimal).
- Specify (as the second byte) the 1-byte code associated with the request (see [Table 5-2](#)).
- Specify any arguments required by the request.

For example, an output sequence for the `POSITION_CURSOR` request consists of the following four bytes.

- 27 decimal (1B hexadecimal)
- 1 decimal (01 hexadecimal), the code associated with `POSITION_CURSOR`
- *line* (0-based origin)
- *column* (0-based origin)

As indicated, all *line* and *column* values begin at 0; that is, they have a 0-based origin. The character position at the top left corner of the scrolling region is line number 0 and column number 0. Therefore, to position the cursor at the ninth character position on the top line, your application program should specify the following hexadecimal sequence.

``1B `01 `00`08`

NOTE \_\_\_\_\_

Your application program **must** write all of the bytes in a generic output sequence to the device using a single subroutine call. For example, your application program cannot call `s$seq_write_partial` to write the generic sequence introducer (1B hexadecimal), and then call the

subroutine again to write the byte containing the code for the generic output request.

The command `display_line` uses translated output and can test the action of a generic output request on a terminal. For example, both of the following commands clear the terminal display.

```
display_line `1B `02
display_line (byte 27) (byte 2)
```

Executing a generic output request directly (using the keyboard or `s$write_raw`) often has a different effect than using the corresponding generic output request, because the window terminal driver does not process the request. For example, the window terminal driver stores the current cursor position internally. When your application program executes the `POSITION_CURSOR` generic output request, the window terminal driver updates the internally stored current cursor position using the coordinates specified in the request; subsequent output appears at the new cursor position. If the terminal user or an application program repositions the cursor directly using the keyboard or `s$write_raw`, on the next I/O call, the window terminal driver repaints (refreshes) the screen to ensure that the actual screen states match the states stored internally (for example, the state of the cursor position).

## Output Requests for Performing Cursor Operations

All generic output requests that perform cursor operations observe the subwindow size defined with the `s$control` opcode `TERM_SET_SUBWIN_BOUNDS` (2067) and the scrolling region defined with the `SET_SCROLLING_REGION` request. (For a description of the opcode `TERM_SET_SUBWIN_BOUNDS` (2067), see the section “Opcodes Affecting the Primary Window and Subwindow” in Chapter 8; for a description of the `SET_SCROLLING_REGION` request, see the section “[Output Requests for Performing Line Operations](#)” later in this chapter.)

If your application program does not specify a scrolling region, the default scrolling region includes all of the lines in the subwindow. All generic output requests operate within the scrolling region.

### **CURSOR\_OFF\_SEQ (11)**

For either simple or formatted sequential I/O, this request makes the cursor invisible. Your application program can also set this cursor format by specifying the value 0 with the `SET_CURSOR_FORMAT` request.

The sequence for this request consists of the following two bytes.

- 27 decimal (1B hexadecimal)
- 11 decimal (0B hexadecimal), the code for the request

#### **CURSOR\_ON\_SEQ (10)**

For either simple or formatted sequential I/O, this request makes the cursor visible. The sequence for this request consists of the following two bytes.

- 27 decimal (1B hexadecimal)
- 10 decimal (0A hexadecimal), the code for the request

#### **DOWN\_SEQ (6)**

If your application program is using formatted sequential I/O, this request moves the cursor down one line in the scrolling region. However, the cursor remains on the current line if it is already on the bottom line of the scrolling region.

If your application program specifies this request with simple sequential I/O, the window terminal driver switches the subwindow to formatted I/O mode before it executes the request.

The sequence for this request consists of the following two bytes.

- 27 decimal (1B hexadecimal)
- 6 decimal (06 hexadecimal), the code for the request

#### **LEFT\_SEQ (7)**

If your application program is using formatted sequential I/O, this request moves the cursor one column to the left. However, the cursor cannot move one column to the left if it is positioned at column 0 of the subwindow.

If your application program specifies this request with simple sequential I/O, the window terminal driver switches the subwindow to formatted I/O mode before it executes the request.

The sequence for this request consists of the following two bytes.

- 27 decimal (1B hexadecimal)
- 7 decimal (07 hexadecimal), the code for the request

#### **NEW\_LINE\_SEQ (61)**

For either simple or formatted sequential I/O, this request moves the cursor to the beginning of the next line. If the cursor is already at the bottom of a formatted I/O subwindow (or scrolling region), this request scrolls the data up and inserts a blank line at the bottom of the subwindow (or scrolling region). For simple sequential I/O, the request adds a blank line at the bottom of the output area.

This request is similar to the `SCROLL_UP` request, except that it can be used with simple sequential I/O.

The sequence for this request consists of the following two bytes.

- 27 decimal (1B hexadecimal)
- 61 decimal (3D hexadecimal), the code for the request

#### **POSITION\_CURSOR\_SEQ (1)**

*line, column*

If your application program is using formatted sequential I/O, this request moves the cursor to the line and column specified, relative to the origin of the subwindow's scrolling region. Note that line 0 is always the first line in the scrolling region; you must specify the line relative to this first line. Column 0 is always the first column in the subwindow.

For example, suppose the scrolling region begins at line 5 and ends at line 10 of the subwindow, and your application program uses this request to position the cursor at line 0, column 0. The request positions the cursor at line 0, column 0 of the scrolling region, which is actually line 5, column 0 of the **subwindow**.

If your application program specifies the value 255 for either a line or a column, the cursor's line or column position does not change. If your application program specifies a line or a column outside the scrolling region or subwindow, the cursor is positioned at the nearest scrolling region or subwindow boundary.

If your application program specifies this request with simple sequential I/O, the window terminal driver switches the subwindow to formatted I/O mode before it executes the request.

The sequence for this request consists of the following four bytes.

- 27 decimal (1B hexadecimal)
- 1 decimal (01 hexadecimal), the code for the request
- *line* (0-based origin)
- *column* (0-based origin)

#### **RIGHT\_SEQ (8)**

If your application program is using formatted sequential I/O, this request moves the cursor one column to the right. However, the cursor cannot move one column to the right if it is positioned at the last (rightmost) column of the subwindow.

If your application program specifies this request with simple sequential I/O, the window terminal driver switches the subwindow to formatted I/O mode before it executes the request.

The sequence for this request consists of the following two bytes.

- 27 decimal (1B hexadecimal)
- 8 decimal (08 hexadecimal), the code for the request

#### **SCROLL\_DOWN\_SEQ (31)**

If your application program is using formatted sequential I/O, this request moves the cursor to the first column of the preceding line, unless the cursor is at the top of the scrolling region. When the cursor is at the top of the scrolling region, the request moves the cursor to the first column of the current line and scrolls the data within the region down one line. This inserts a blank line at the top of the scrolling region and causes the bottom line to scroll out of the region. In this case, the cursor is positioned at the top left corner of the region when the request finishes executing.

If your application program specifies this request with simple sequential I/O, the window terminal driver switches the subwindow to formatted I/O mode before it executes the request.

The sequence for this request consists of the following two bytes.

- 27 decimal (1B hexadecimal)
- 31 decimal (1F hexadecimal), the code for the request

#### **SCROLL\_UP\_SEQ (32)**

If your application program is using formatted sequential I/O, this request moves the cursor to the first column of the following line, unless the cursor is at the bottom of the scrolling region. When the cursor is at the bottom of the scrolling region, the request moves the cursor to the first column of the current line and scrolls the data within the region up one line. This inserts a blank line at the bottom of the scrolling region and causes the top line to scroll out of the region. In this case, the cursor is positioned at the bottom left corner of the region when the request finishes executing.

If your application program specifies this request with simple sequential I/O, the window terminal driver switches the subwindow to formatted I/O mode before it executes the request.

The sequence for this request consists of the following two bytes.

- 27 decimal (1B hexadecimal)
- 32 decimal (20 hexadecimal), the code for the request



**SET\_CURSOR\_FORMAT\_SEQ (38)***cursor\_format* (0 to 4)

For either simple or formatted sequential I/O, this request changes the display of the cursor to the format specified. There are five cursor formats available, each of which is associated with a code.

Format	Code
Cursor off	0
Blinking block	1
Steady block	2
Blinking underline	3
Steady underline	4

All but the cursor-off format make the cursor visible (if it is not already visible).

The sequence for this request consists of the following three bytes.

- 27 decimal (1B hexadecimal)
- 38 decimal (26 hexadecimal), the code for the request
- *cursor\_format* (0 to 4)

The cursor format that you specify must be supported by the device and must be defined as a display-control capability in the TTP. Your application program can also set the cursor format using the `s$control` opcode `TERM_SET_CURSOR_FORMAT (2055)`. (For a description of this opcode, see the section “Opcodes Affecting the Primary Window and Subwindow” in Chapter 8.)

**NOTE**

If you are migrating application programs from the old asynchronous driver to the window terminal driver, you should specify the cursor format using either the `SET_CURSOR_FORMAT` request or the opcode `TERM_SET_CURSOR_FORMAT (2055)` instead of using specific cursor-format requests (for example, `SET_CURSOR_INVISIBLE (117)`, which is recognized by the old asynchronous driver). [Appendix A](#) discusses

the compatibility issues for migrating application programs to the window terminal driver.

#### **UP\_SEQ (5)**

If your application program is using formatted sequential I/O, this request moves the cursor up one line in the scrolling region. However, the cursor remains on the current line if it is already positioned at the top line of the scrolling region.

If your application program specifies this request with simple sequential I/O, the window terminal driver switches the subwindow to formatted I/O mode before it executes the request.

The sequence for this request consists of the following two bytes.

- 27 decimal (1B hexadecimal)
- 5 decimal (05 hexadecimal), the code for the request

## **Output Requests for Performing Clearing Operations**

These requests clear specific areas of the defined scrolling region. If your application program specifies any of these requests with simple sequential I/O, the window terminal driver switches the subwindow to formatted I/O mode before it executes the request.

#### **CLEAR\_SCROLLING\_REGION\_SEQ (2) (alias CLEAR\_SCREEN\_SEQ (2))**

If your application program is using formatted sequential I/O, this request clears all characters from the scrolling region. It then moves the cursor to the top left corner of the scrolling region. This request also causes the window terminal driver to process output up to and including the request in the output string, and to update the subwindow display with the visible changes before it continues to process the rest of the string.

The sequence for this request consists of the following two bytes.

- 27 decimal (1B hexadecimal)
- 2 decimal (02 hexadecimal), the code for the request

#### **CLEAR\_TO\_EOL\_SEQ (3)**

If your application program is using formatted sequential I/O, this request clears all characters from the current cursor position to the end of the line. This includes the character marked by the cursor.

The sequence for this request consists of the following two bytes.

- 27 decimal (1B hexadecimal)
- 3 decimal (03 hexadecimal), the code for the request

**CLEAR\_TO\_EOR\_SEQ (4)** (alias **CLEAR\_TO\_EOS\_SEQ (4)**)

If your application program is using formatted sequential I/O, this request clears all characters from the current cursor position to the end of the scrolling region. This includes the character marked by the cursor. This request also causes the window terminal driver to process output up to and including the request in the output string, and to update the subwindow display with the visible changes before it continues to process the rest of the string.

The sequence for this request consists of the following two bytes.

- 27 decimal (1B hexadecimal)
- 4 decimal (04 hexadecimal), the code for the request

## Output Requests for Performing Line Operations

These requests edit lines in the subwindow's scrolling region when your application program is using formatted sequential I/O. Note that for each of these requests, the line and column specified must be relative to the top line and leftmost column of the subwindow's scrolling region. (The subwindow always begins at line 0, and the scrolling region begins at the appropriate line of the subwindow.)

If your application program specifies the value 255 for either a line or a column, the cursor's line or column position does not change when the request executes. If your application program specifies a line or a column outside the scrolling region or subwindow, the appropriate request moves the cursor to the nearest scrolling region or subwindow boundary and then performs the line operation.

If your application program specifies any of these requests with simple sequential I/O, the window terminal driver switches the subwindow to formatted I/O mode before it executes the request.

**DELETE\_CHARS\_SEQ (15)**

*line, column, number\_of\_chars*

If your application program is using formatted sequential I/O, this request deletes the specified number of characters at the line and column specified. All characters to the right of the deleted characters move the appropriate number of character positions to the left. The request then positions the cursor at the line and column specified.

The sequence for this request consists of the following five bytes.

- 27 decimal (1B hexadecimal)

- 15 decimal (0F hexadecimal), the code for the request
- *line* (0-based origin; 255 for current line position)
- *column* (0-based origin; 255 for current column position)
- *number\_of\_chars* (the number of characters to be deleted)

**DELETE\_LINES\_SEQ (17)**

*line, number\_of\_lines*

If your application program is using formatted sequential I/O, this request moves the cursor to the beginning of the line specified and deletes the appropriate number of lines. All lines below the last line deleted scroll up the region. When the request finishes executing, the cursor is positioned at column 0 of the line specified.

The sequence for this request consists of the following four bytes.

- 27 decimal (1B hexadecimal)
- 17 decimal (11 hexadecimal), the code for the request
- *line* (0-based origin; 255 for current line position)
- *number\_of\_lines* (the number of lines to be inserted)

**INSERT\_CHARS\_SEQ (14)**

*line, column, number\_of\_chars*

If your application program is using formatted sequential I/O, this request inserts the specified number of blank characters (also called *fill characters*) at the line and column specified. When the blank characters are inserted, the characters located at the line and column specified move the appropriate number of character positions to the right. However, any characters that move beyond the right boundary of the subwindow are discarded. When the request finishes executing, the cursor is positioned at the line and column specified.

The sequence for this request consists of the following five bytes.

- 27 decimal (1B hexadecimal)
- 14 decimal (0E hexadecimal), the code for the request
- *line* (0-based origin; 255 for current line position)
- *column* (0-based origin; 255 for current column position)
- *number\_of\_chars* (the number of characters to be inserted)

**INSERT\_LINES\_SEQ (16)***line, number\_of\_lines*

If your application program is using formatted sequential I/O, this request moves the cursor to the beginning of the specified line and inserts the appropriate number of blank lines. The line specified and all lines below it scroll down the region. However, any lines that go beyond the bottom of the scrolling region are discarded. When the request finishes executing, the cursor is positioned at column 0 of the line specified.

The sequence for this request consists of the following four bytes.

- 27 decimal (1B hexadecimal)
- 16 decimal (10 hexadecimal), the code for the request
- *line* (0-based origin; 255 for current line position)
- *number\_of\_lines* (the number of lines to be inserted)

**SET\_SCROLLING\_REGION\_SEQ (95)***top\_line, bottom\_line*

If your application program is using formatted sequential I/O, this request sets the scrolling region within the subwindow. Your application program must specify the top and bottom lines relative to the top line of the subwindow. (The subwindow always begins at line 0, and a scrolling region begins at the appropriate line in the subwindow.) For example, in a 24-line subwindow, your application program can set a scrolling region that begins at line 5 and ends at line 15 of the subwindow.

If your application program specifies this request with simple sequential I/O, the window terminal driver switches the subwindow to formatted I/O mode before it executes the request.

The sequence for this request consists of the following four bytes.

- 27 decimal (1B hexadecimal )
- 95 decimal (5F hexadecimal), the code for the request
- *top\_line* (0-based origin)
- *bottom\_line* (0-based origin)

The lines in the scrolling region scroll when a `SCROLL_UP` or `SCROLL_DOWN` request executes, or when the cursor is positioned at the bottom line of the scrolling region and a `NEW_LINE` request sequence is written to the subwindow. Lines outside the scrolling region remain frozen (or locked) until they are included in a new scrolling region.

To scroll the scrolling region, your application program must move the cursor to the top or bottom line of the subwindow before executing the `SCROLL_UP` or `SCROLL_DOWN` request, respectively. Unless the cursor is at the top or bottom of the subwindow, these requests are equivalent to the `UP` and `DOWN` requests. Note, however, that the `UP` and `DOWN` requests do not move the cursor beyond the top or bottom of the subwindow (that is, they do not cause the subwindow to scroll).

## Output Requests for Setting Visual Attributes

These requests allow your application program to set various display attributes. The window terminal driver supports all attributes as mode attributes. *Mode attributes* enable or disable a given attribute (or group of attributes) for all **subsequent** write operations. Setting a mode attribute does not affect characters already displayed on the screen, but affects the display of all characters subsequently sent by your application program.

An attribute can take effect **only** if the terminal supports it; therefore, the attribute must be defined in the attributes section of the TTP for the terminal. (See the manual *VOS Communications Software: Defining a Terminal Type* (R096) for more information about defining attributes in the TTP.)

Specifying an attribute with one of the attribute requests does not affect the setting of any other attribute. To reset **all** of the current display attributes, your application program must use the request `SET_MODE_ATTRIBUTES (133)`.

### NOTE

The old asynchronous driver supports mode attributes **and** area attributes. *Area attributes* mark a specific area of the screen for a given attribute. Setting an area attribute affects all characters in the marked area, either existing characters or characters sent after the attribute is set. Since there are differences in how the old asynchronous driver and the window terminal driver handle attributes (the window terminal driver treats all attributes as mode attributes), you should avoid using the output requests that set area attributes for the old asynchronous driver. These output requests include `SET_ATTRIBUTES (18)` and `SET_ATTRIBUTE_CHAR (72)`.

`BLANK_OFF_SEQ (130)`

`BLANK_ON_SEQ (129)`

For either simple or formatted sequential I/O, these requests toggle the blanked attribute. The blanked attribute is typically used to suppress the display of characters written to the subwindow.

The sequence for either of these requests consists of the following two bytes.

- 27 decimal (1B hexadecimal)
- 129 or 130 decimal (81 or 82 hexadecimal), the code for the request

**BLINK\_OFF\_SEQ** (128)

**BLINK\_ON\_SEQ** (127)

For either simple or formatted sequential I/O, these requests toggle the blinking attribute. The blinking attribute is typically used to make characters blink on and off.

The sequence for either of these requests consists of the following two bytes.

- 27 decimal (1B hexadecimal)
- 127 or 128 decimal (7F or 80 hexadecimal), the code for the request

**BOLDFACE\_OFF\_SEQ** (101)

**BOLDFACE\_ON\_SEQ** (100)

For either simple or formatted sequential I/O, these requests toggle the boldface attribute. The sequence for either of these requests consists of the following two bytes.

- 27 decimal (1B hexadecimal)
- 100 or 101 decimal (64 or 65 hexadecimal), the code for the request

**HALF\_INTENSITY\_OFF\_SEQ** (26)

**HALF\_INTENSITY\_ON\_SEQ** (25)

For either simple or formatted sequential I/O, these requests toggle the half-intensity (low-intensity) attribute. The sequence for either of these requests consists of the following two bytes.

- 27 decimal (1B hexadecimal)
- 25 or 26 decimal (19 or 1A hexadecimal), the code for the request

**HI\_INTENSITY\_OFF\_SEQ** (126)

**HI\_INTENSITY\_ON\_SEQ** (125)

For either simple or formatted sequential I/O, these requests toggle the high-intensity attribute. The sequence for either of these requests consists of the following two bytes.

- 27 decimal (1B hexadecimal)
- 125 or 126 decimal (7D or 7E hexadecimal), the code for the request

INVERSE\_VIDEO\_OFF\_SEQ (24)  
INVERSE\_VIDEO\_ON\_SEQ (23)

For either simple or formatted sequential I/O, these requests toggle the inverse-video (reverse-video) attribute. The sequence for either of these requests consists of the following two bytes.

- 27 decimal (1B hexadecimal)
- 23 or 24 decimal (17 or 18 hexadecimal), the code for the request

SET\_MODE\_ATTRIBUTES\_SEQ (133)

2-byte *attribute*

For either simple or formatted sequential I/O, this request resets the terminal's display attributes and enables the attribute or group of attributes specified. (Your application program should use the individual attribute requests if it does not want to affect other active attributes.) This request does not move the cursor to another position. Your application program can use this request **only** if the terminal's TTP includes an attributes section.

When your application program specifies an attribute or group of attributes with this request, it must provide a 2-byte value that specifies the code representing the attribute or a code representing the sum of a group of attributes.

The sequence for this request consists of the following four bytes.

- 27 decimal (1B hexadecimal)
- 133 decimal (85 hexadecimal), the code for the request
- 2-byte *attribute*

Table 5-3 lists the display attributes and their associated decimal codes. Your application program can specify attributes in the range 0 to 1023 decimal (00 to 03FF hexadecimal).

#### NOTE \_\_\_\_\_

The strikeout and ribbon-color attributes are currently ignored by simple sequential I/O. These attributes are typically printer attributes that are defined in TTPs for printers.



To specify the blinking and inverse-video attributes (whose decimal codes are 2 and 4, respectively), your application program would specify the following sequence (in hexadecimal).

```
'1B      '85      '00'06
GSI request_number attributes
```

**Table 5-3. Display Attributes for the Window Terminal Driver**

Attribute	Code
Normal (attributes off)	0
Blanked	1
Blinking	2
Inverse video	4
Underlined	8
Low intensity	16
High intensity	32
Standout	64
Boldface	128
Strikeout	256
Ribbon-color	512

**STANDOUT\_OFF\_SEQ (132)**

**STANDOUT\_ON\_SEQ (131)**

For either simple or formatted sequential I/O, these requests toggle the standout attribute. This attribute is typically used to highlight a particular area.

The sequence for either of these requests consists of the following two bytes.

- 27 decimal (1B hexadecimal)
- 131 or 132 decimal (83 or 84 hexadecimal), the code for the request

`UNDERSCORE_OFF_SEQ (99)`

`UNDERSCORE_ON_SEQ (98)`

For either simple or formatted sequential I/O, these requests toggle the underline attribute. The sequence for either of these requests consists of the following two bytes.

- 27 decimal (1B hexadecimal)
- 98 or 99 decimal (62 or 63 hexadecimal), the code for the request

## Output Requests for Setting Modes

These requests allow your application program to enable or disable a particular terminal mode. They must have corresponding capabilities defined in the output section of the TTP.

`ENTER_GRAPHICS_MODE_SEQ (27)`

If your application program is using formatted sequential I/O, this request enables graphics mode in the subwindow, which allows the subwindow to display line-graphics characters. If your application program specifies this request with simple sequential I/O, the window terminal driver switches the subwindow to formatted I/O mode before it executes the request.

Graphics mode should not be confused with a local terminal's graphics mode, in which the terminal translates the output stream to graphics characters without involving the window terminal driver. In graphics mode, the window terminal driver allows the output stream to contain only generic output requests and bytes in the range 0 to 15 decimal (00 to 0F hexadecimal). If the output stream contains anything out of the range 0 to 15 decimal, the window terminal driver recognizes only the lower four bits of the character and interprets these bits according to the 0-to-15 scheme.

The window terminal driver maps the output characters 0 to 15 decimal to the 16 graphics-character sequences specified in the `line-graphics` entry in the output section of the TTP. For Stratus terminals, byte values 0 to 15 are mapped to the output sequences for the graphics characters shown in [Table 5-4](#).

The sequence for this request consists of the following two bytes.

- 27 decimal (1B hexadecimal)
- 27 decimal (1B hexadecimal), the code for the request

The following PL/I call to `s$seq_write_partial` enables graphics mode in the subwindow, writes a vertical-line character, and then disables graphics mode in the subwindow.

```
call s$seq_write_partial (port_id, buffer_length, byte(27) ||
```










```
byte(27) || byte(VERTICAL_LINE_CHAR) || byte(27) || byte(28),
error_code);
```

#### LEAVE\_GRAPHICS\_MODE\_SEQ (28)

If your application program is using formatted sequential I/O, this request disables graphics mode in the subwindow. The sequence for this request consists of the following two bytes.

- 27 decimal (1B hexadecimal)
- 28 decimal (1C hexadecimal), the code for the request

**Table 5-4. Line Graphics Characters** (Page 1 of 2)

Character Name	Symbol	Number	Description
SPACE_CHAR		0	Blank
LOWER_LEFT_ROUND_CHAR		1	Round lower left corner
UPPER_LEFT_ROUND_CHAR		2	Round upper left corner
UPPER_RIGHT_ROUND_CHAR		3	Round upper right corner
LOWER_RIGHT_ROUND_CHAR		4	Round lower right corner
LOWER_LEFT_SQUARE_CHAR		5	Square lower left corner
UPPER_LEFT_SQUARE_CHAR		6	Square upper left corner
UPPER_RIGHT_SQUARE_CHAR		7	Square upper right corner
LOWER_RIGHT_SQUARE_CHAR		8	Square lower right corner
INTERSECTION_JOIN_CHAR		9	Intersection

**Table 5-4. Line Graphics Characters** (Page 2 of 2)

Character Name	Symbol	Number	Description
VERTICAL_LINE_CHAR		10	Vertical line
HORIZONTAL_LINE_CHAR	—	11	Horizontal line
RIGHT_LINE_JOIN_CHAR	┤	12	Left tree
LEFT_LINE_JOIN_CHAR	├	13	Right tree
TOP_LINE_JOIN_CHAR	┐	14	Down tree
BOTTOM_LINE_JOIN_CHAR	└	15	Up tree

## Miscellaneous Generic Output Requests

These requests perform various functions for your application program. They must have corresponding capabilities defined in the output section of the TTP.

### **BEEP\_SEQ (69)**

For either simple or formatted sequential I/O, this request sounds the terminal's bell. The sequence for this request consists of the following two bytes.

- 27 decimal (1B hexadecimal)
- 69 decimal (45 hexadecimal), the code for the request

### **DISPLAY\_BLOCK\_SEQ (33)**

If your application program is using formatted sequential I/O, this request displays a solid block at the current cursor position.

If your application program specifies this request with simple sequential I/O, the window terminal driver switches the subwindow to formatted I/O mode before executing the request.

The sequence for this request consists of the following two bytes.

- 27 decimal (1B hexadecimal)
- 33 decimal (21 hexadecimal), the code for the request

#### **RESET\_TERMINAL\_SEQ (22)**

For either simple or formatted sequential I/O, this request sends a reset sequence (as defined by the `reset-terminal` capability in the output section of the TTP) and refreshes the display of the entire screen. A reset occurs automatically when the terminal is powered up. If the `reset-terminal` capability is defined in the TTP, this request causes the following events to occur.

- The bytes that implement the `reset-terminal` capability are sent to the terminal.
- The internal status information stored by the window terminal driver for terminal output is reset.
- The cursor is moved to the position stored internally by the window terminal driver for the line, column, and page.

The sequence for this request consists of the following two bytes.

- 27 decimal (1B hexadecimal)
- 22 decimal (16 hexadecimal), the code for the request

#### **SCREEN\_OFF\_SEQ (94)**

For either simple or formatted sequential I/O, this request turns off the display of the entire screen, including the status area.

If your application program is using formatted sequential I/O, this request also causes the window terminal driver to process output up to and including the request in the output string, and to update the subwindow display with the visible changes before it continues to process the rest of the string.

The sequence for this request consists of the following two bytes.

- 27 decimal (1B hexadecimal)
- 94 decimal (5E hexadecimal), the code for the request

#### **SCREEN\_ON\_SEQ (93)**

For either simple or formatted sequential I/O, this request turns on the display of the entire screen. If your application program is using formatted sequential I/O, this request also causes the window terminal driver to process output up to and including the

request in the output string, and to update the subwindow display with the visible changes before it continues to process the rest of the string.

Your application program should issue this request **after** the display of the screen has been turned off. The sequence for this request consists of the following two bytes.

- 27 decimal (1B hexadecimal)
- 93 decimal (5D hexadecimal), the code for the request

## Handling National Language Support (NLS) Characters

To send NLS characters from a specific character set represented in the internal character coding system, your application program typically invokes the supplementary character set and specifies the internal codes for the characters. The character code may be a series of bytes, depending on the character set, and is typically expressed in hexadecimal. See [Appendix B](#) for a depiction of the codes in the internal character coding system; see [Appendix F](#) for a list of the characters in the default internal character sets.

Your application program does not need to invoke the standard ASCII character set, which includes the left-hand part of Latin alphabet No. 1. In addition, if the TTP for the device defines the right-hand part of Latin alphabet No.1 (called `latin-1`), your application program can simply send the internal character code for the character; it need not invoke that character set by preceding the character with the `latin-1` single-shift code (128 decimal, 80 hexadecimal).

For example, if the TTP defines `latin-1`, then your application program simply specifies an internal character code such as 246 decimal (F6 hexadecimal) in the buffer. This code causes the terminal to print the character `ö` from `latin-1`.

In order for the terminal to print a character from a double-byte character set such as kanji, your application program must specify the single-shift code that invokes the character set, followed by the bytes that represent the appropriate kanji character. For example, in order for the terminal to print the kanji character 木, your application program must specify the following:

```
~8E ~CC ~DA
```

In this example, the first byte is 8E hexadecimal (142 decimal), which invokes the kanji character set. The next two bytes are the hexadecimal equivalent of the actual kanji character. If the character set is defined in the TTP, the character will be displayed on the terminal screen.

If the character set is not defined in the TTP, the handling of the character is determined by the undisplayable notation character mode specified by your application program. (See the next section for more information about this mode.) The default setting for this

mode causes the code for the character to appear in *escaped notation* (the hexadecimal equivalent of the character, preceded by the hexadecimal notation character variable `hex-notation-char`, which is defined in the configuration section of the TTP).

For more information about using NLS characters, see the *National Language Support User's Guide* (R212).

## Defining the Handling of Standard ASCII Control Characters

The standard ASCII control characters are bytes less than 32 decimal (20 hexadecimal). ([Appendix F](#) lists these control characters.) The following control characters are supported by the window terminal driver and perform standard operations.

- `BEL` (7 decimal) performs the `BEEP` request.
- `BS` (8 decimal) performs a backspace (a `LEFT` request) in formatted sequential I/O mode and causes the switch to formatted I/O mode in simple sequential I/O.
- `HT` (9 decimal) performs a horizontal tab (inserts spaces up to the next tab).
- `LF` (10 decimal) performs a line feed (a `NEW_LINE` request).
- `CR` (13 decimal) performs a carriage return in formatted sequential I/O mode and causes the switch to formatted I/O mode in simple sequential I/O.

The TTP for the device should include capabilities that define these control characters, except for `HT`, which does not require a TTP definition. If your application program embeds these control characters in the output string, and they are defined with capabilities in the TTP, `s$seq_write` or `s$seq_write_partial` performs the appropriate function. However, if these control characters (except for `HT`) are **not** defined with capabilities in the TTP, they are ignored (that is, discarded).

If your application program embeds any other control characters in the output string, those characters are handled according to the *undisplayable notation character mode* specified by your application program. Selecting an undisplayable notation character mode allows your application program to determine the handling of characters for which there is no graphic representation (that is, control characters and non-ASCII characters).

### NOTE

Unlike the old asynchronous driver, the window terminal driver does not support the control characters `VT` (11 decimal), which performs a vertical tab, and `FF` (12 decimal), which performs a form feed. With the window terminal driver, these control characters are always

handled according to the undisplayable notation character mode specified by your application program. In general, the old asynchronous driver handles control characters differently and offers the edited-output option to suppress (discard) the control characters. See [Appendix A](#) for more information about the differences in handling control characters.

Your application program can use the `s$control` opcodes `TERM_GET_UNDISP_MODE (2064)` and `TERM_SET_UNDISP_MODE (2065)` to verify and then select an undisplayable notation character mode. The default undisplayable notation character mode is `UNDISP_ESCAPED` (associated with the value 0), which replaces the control character with escaped notation (the character's hexadecimal equivalent, preceded by the hexadecimal notation character variable `hex-notation-char`, which is defined in the configuration section of the TTP; the default `hex-notation-char` is the grave accent (```)).

Your application program can also select `UNDISP_REPLACED` (associated with the value 2) or `UNDISP_SUPPRESSED` (associated with the value 3) as the undisplayable notation character mode. With `UNDISP_REPLACED`, the window terminal driver represents the character as a printable character that is defined in the TTP. With `UNDISP_SUPPRESSED`, the window terminal driver suppresses the display of a character that is typically represented in escaped notation (that is, the character is discarded).

For more information about selecting an undisplayable notation character mode, see the description of the opcode pair `TERM_GET_UNDISP_MODE (2064)` and `TERM_SET_UNDISP_MODE (2065)` in the section “Opcodes Affecting the Primary Window and Subwindow” in [Chapter 8](#).



---

## Chapter 6

# Attaching and Opening a Port

This chapter describes three VOS subroutines that enable your application program to attach and open a port.

- `s$attach_port`
- `s$open`
- `s$seq_open`

The descriptions of these subroutines focus on terminal I/O. For complete descriptions of the subroutines and how they would apply to other types of I/O, see the VOS Subroutines manuals.

Each subroutine description in this chapter explains how to make the call from an application program written in either PL/I or C.

## `s$attach_port`

Use the `s$attach_port` subroutine to associate a port with a terminal device or a virtual device. You must attach and then open a port to the device (using `s$open`) before attempting to communicate with the device.

When you attach the port, the value of the `duration_switches` argument affects how the window terminal driver functions. The value of `duration_switches` is the binary coding of two logic variables (switches): `hold_attached` and `hold_open`. These switches affect the window terminal driver as follows:

- If these switches are turned **off**, the window terminal driver can redisplay any formatted sequential output that has been overlapped (for example, by another subwindow or primary window). (In general, the window terminal driver waits until your application program performs a sequential I/O operation before it refreshes the contents of all formatted sequential I/O subwindows. Your application program can also use the `s$control` global opcode `RUNOUT (2)` to refresh the subwindows, as described in [Chapter 8](#).) If the switches are turned off when the original (simple sequential) subwindow switches to formatted I/O mode, the window terminal driver clears the entire subwindow, discards all typeahead, and moves the cursor to the top left corner of the subwindow.
- If either of these switches is turned **on**, the window terminal driver cannot guarantee subsequent redisplay of any formatted sequential output that has been overlapped. (See [Chapter 4](#) for a description of formatted sequential output.) If either of the switches is turned on when the original subwindow switches to formatted I/O mode, the window terminal driver discards all typeahead, but does not clear the entire subwindow or reposition the cursor. The opcode `TERM_ENABLE_FMT_IO_MODE (2046)` switches the original subwindow to formatted I/O mode. (For a description of this opcode, see the section “Opcodes Affecting the Primary Window and Subwindow” in [Chapter 8](#).)

### NOTE \_\_\_\_\_

When the operating-system command processor starts a login process, both of these switches are turned on.

For more information about the setting of the `hold_attached` and `hold_open` switches, see the description of the `duration_switches` argument in the section “Arguments,” which appears later in this subroutine description.

## PL/I Usage

```

declare port_name          char (32) varying;
declare path_name          char (256) varying;
declare duration_switches  fixed bin (15);
declare port_id            fixed bin (15);
declare error_code         fixed bin (15);

declare s$attach_port entry(
                                char (32) varying,
                                char (256) varying,
                                fixed bin (15),
                                fixed bin (15),
                                fixed bin (15));

                                call s$attach_port(
                                    port_name,
                                    path_name,
                                    duration_switches,
                                    port_id,
                                    error_code);

```

## C Usage

```

char_varying (32)          port_name;
char_varying (256)         path_name;
short int                  duration_switches;
short int                  port_id;
short int                  error_code;

void s$attach_port();

                                s$attach_port(
                                    &port_name,
                                    &path_name,
                                    &duration_switches,
                                    &port_id,
                                    &error_code);

```

## Arguments

The `s$attach_port` subroutine takes five arguments, which are listed and described in the following table.

Argument	Description
<code>port_name</code> (input)	The name of the port to be attached.
<code>path_name</code> (input)	The name of the terminal device or virtual device to which the port will be attached. Your process can attach more than one port to a terminal. In addition, a privileged process can attach a port to a terminal to which another user has already attached a port.
<code>duration_switches</code> (input)	A value from 0 to 3 that is the binary coding of two logic variables (switches): <code>hold_attached</code> and <code>hold_open</code> . The switches are coded in the “1” and “2” bits of the value. The value 0 (the default) ensures that the <code>hold_attached</code> and <code>hold_open</code> switches are turned <b>off</b> . The value 1 turns on the <code>hold_attached</code> switch and tells the operating system to leave the port attached the next time <code>s\$stop_program</code> executes. The value 2 turns on the <code>hold_open</code> switch and tells the operating system to leave the port attached and open the next time <code>s\$stop_program</code> executes. The value 3 turns on both switches. For more information about the logic variables, see the VOS Subroutines manuals.
<code>port_id</code> (output)	The identifier of the port. The value of <code>port_id</code> will be used as input for subsequent read, write, or control subroutines.
<code>error_code</code> (output)	A returned error code.

## Error Codes

The `s$attach_port` subroutine may return the error codes listed in the following table.

Error Code	Description
e\$device_already_assigned (1155)	Another process is already attached to the device; either your process is not privileged or the device cannot be shared.
e\$device_not_found (1220)	The specified device is not known to the system. To list the known devices, issue the <code>list_devices</code> command. (This command is documented in the <i>VOS Commands Reference Manual</i> (R098).)
e\$no_ports_available (1006)	The system does not have any more ports available.
e\$port_already_attached (1008)	The specified port is already attached.

## s\$open

Use the `s$open` subroutine to initialize the terminal device or virtual device attached to the specified port. Any call to `s$open` **must** be preceded by a call to `s$attach_port` so that the port is associated with the device.

The `s$open` subroutine causes the window terminal driver to create a primary window. As soon as there is I/O, the primary window appears on the screen of the terminal associated with the specified port. Usually, each subsequent call to `s$open` for the same terminal creates another primary window. The exception is when you use the `s$control` opcode `TERM_OPEN_EXISTING_WINDOW_OPCODE` (2107) to open an existing primary window. See the description of this opcode in [Chapter 8](#) for further information.

Most `s$control` operations are valid **only** if the terminal's RS-232-C channel/subchannel is open or the virtual device's port is open. The `s$control` operations that your application program can issue before a call to `s$open` either set characteristics of the communications line or control certain aspects of the primary window. (The description of `s$control` in [Chapter 8](#) identifies the opcodes that can and cannot be issued before the port is open.)

For example, the `s$control` opcode `TERM_SET_OPEN_ACTION` (2012) allows your application program to define the action performed by `s$open` when it opens the terminal's RS-232-C channel/subchannel or the virtual device's port. (For a description of the opcode `TERM_SET_OPEN_ACTION` (2012), see the section "Opcodes Affecting All Communications Media" in [Chapter 8](#).)

If your application program is using no-wait mode (see [Chapter 8](#)) and `s$open` returns the error code `e$caller_must_wait` (1277), your application program must either make another call to `s$open` or abort the operation to cancel the pending no-wait call. To abort the operation, your application program must use the `s$control` global opcode `ABORT` (3). (For a description of the opcode `ABORT` (3), see the section "Global Control Opcodes" in [Chapter 8](#).)

## PL/I Usage

```
declare port_id           fixed bin (15);
declare file_organization fixed bin (15); /* not for terminals */
declare maximum_length    fixed bin (15);
declare io_type           fixed bin (15);
declare locking_mode      fixed bin (15); /* not for terminals */
declare access_mode       fixed bin (15);
declare index_name        char (32) varying; /* not for terminals */
declare error_code        fixed bin (15);

declare s$open entry(
                                fixed bin (15),
                                fixed bin (15),
                                fixed bin (15),
                                fixed bin (15),
                                fixed bin (15),
                                fixed bin (15),
                                char (32) varying,
                                fixed bin (15));

                                call s$open(
                                port_id,
                                file_organization,
                                maximum_length,
                                io_type,
                                locking_mode,
                                access_mode,
                                index_name,
                                error_code);
```

C Usage

```
short int      port_id;
short int      file_organization; /* not for terminals */
short int      maximum_length;
short int      io_type;
short int      locking_mode;      /* not for terminals */
short int      access_mode;
char_varying(32) index_name;      /* not for terminals */
short int      error_code);

void s$open();

s$open(
    &port_id,
    &file_organization,
    &maximum_length,
    &io_type,
    &locking_mode,
    &access_mode,
    &index_name,
    &error_code);
```

Arguments

The s\$open subroutine takes eight arguments, but only five are relevant to terminal I/O. The other three arguments are for file I/O only and are not used by the window terminal driver. These s\$open arguments should be set to 0 (or the null string).

The s\$open arguments are listed and described in the following table.

(Page 1 of 2)

Argument	Description
port_id (input)	The identifier of the port attached to the device.
file_organization (input)	Not relevant to terminal or printer I/O.
maximum_length (input)	The maximum number of characters that can be placed in the internal input buffer. The value specified determines the size of the internal input buffers allocated to store input from the terminal. The limit is 2,048. Specifying a value of 0 allocates the default length of 300 bytes. The s\$control opcode TERM_SET_MAX_BUFFER_SIZE (2010) performs the same function as maximum_length.



(Page 2 of 2)

Argument	Description
<code>io_type</code> (input)	The type of I/O access. You can specify 1 (input), 2 (output), 3 (append), or 4 (update). Stratus recommends that you specify the value 4 (update).
<code>locking_mode</code> (input)	Not relevant to terminal or printer I/O.
<code>access_mode</code> (input)	The type of access. You can specify 1 (sequential), 2 (random), or 3 (indexed). Stratus recommends that you specify the value 1 (sequential).
<code>index_name</code> (input)	Not relevant to terminal or printer I/O.
<code>error_code</code> (output)	A returned error code.

## Error Codes

The `s$open` subroutine may return the error codes listed in the following table.

(Page 1 of 2)

Error Code	Description
<code>e\$caller_must_wait</code> (1277)	The I/O operation could not be completed. Your application program is in no-wait mode and must wait for the connection to be established. To recover from this error, you must either reissue the call to <code>s\$open</code> or call <code>s\$control</code> with the opcode <code>ABORT</code> (3).
<code>e\$invalid_access_mode</code> (1071)	The <code>access_mode</code> argument does not have a valid value. You must correct your application program by specifying a value of 1, 2, or 3 for <code>access_mode</code> .
<code>e\$invalid_buffer_size</code> (1215)	The value of the <code>maximum_length</code> argument is less than 0 or greater than 2048. You must correct your application program by specifying a value in the range 0 to 2048.
<code>e\$invalid_hardware_type</code> (2998)	The port is not attached to an asynchronous line or I/O adapter.
<code>e\$invalid_io_type</code> (1070)	The <code>io_type</code> argument does not have a valid value. You must correct your application program by specifying a value of 1, 2, 3, or 4 for <code>io_type</code> .

(Page 2 of 2)

Error Code	Description
e\$line_hangup (1365)	For an RS-232-C channel/subchannel that is configured for dial-up, this error code indicates a loss of the DSR signal and/or the DCD signal. For most access layers, this error code indicates that the connection state is no longer <code>TERM_CONNECTION_ESTABLISHED</code> and that the connection is being terminated (that is, the connection state is most likely <code>TERM_CONNECTION_BREAKING</code> ). To determine the connection state, use the opcode <code>TERM_GET_CONNECTION_STATE</code> (2002). For information on how to return an RS-232-C channel/subchannel to service, see the description of the opcodes <code>TERM_HANGUP</code> (2001) and <code>TERM_LISTEN</code> (2023) in Chapter 8.
e\$no_alloc_wired_heap (3077)	There is insufficient space in the wired heap for the operating system to allocate the number of buffers that are needed in order to accommodate the <code>maximum_length</code> argument.
e\$out_of_service (2535)	The RS-232-C channel/subchannel is out of service. This error may indicate that the asynchronous line or I/O adapter was removed, or that the error threshold for the channel/subchannel was exceeded, causing the operating system to remove the channel/subchannel from service. Your system administrator can return the channel/subchannel to service.

## s\$seq\_open

Use the `s$seq_open` subroutine to attach **and** open a port to the device identified by `path_name`.

When the `s$seq_open` subroutine attaches the port to the device, it uses the default value 0 for the duration switches. The value 0 ensures that the `hold_attached` and `hold_open` switches are turned **off**.

Unlike `s$open`, `s$seq_open` has no `maximum_length` argument and cannot be used to control the allocation of input buffers. For `s$seq_open`, the window terminal driver allocates a maximum length of 300 bytes.

PL/I Usage

```
declare path_name          char (256) varying;
declare io_type            fixed bin (15);
declare port_id            fixed bin (15);
declare error_code         fixed bin (15);

declare s$seq_open entry(   char (256) varying,
                           fixed bin (15),
                           fixed bin (15),
                           fixed bin (15));

      call s$seq_open(      path_name,
                           io_type,
                           port_id,
                           error_code);
```

C Usage

```
char_varying (256)      path_name;
short int              io_type;
short int              port_id;
short int              error_code;

void s$seq_open();

      s$seq_open(         &path_name,
                           &io_type,
                           &port_id,
                           &error_code);
```

Arguments

The s\$seq\_open subroutine takes four arguments, which are listed and described in the following table.

Argument	Description
path_name (input)	The name of the terminal device or virtual device to which the port will be attached and then opened.
io_type (input)	The type of I/O access. You can specify 1 (input), 2 (output), 3 (append), or 4 (update). Stratus recommends that you specify the value 4 (update).
port_id (output)	The identifier of the port attached to the device.
error_code (output)	A returned error code.

## Error Codes

The `s$seq_open` subroutine may return the error codes listed in the following table.

(Page 1 of 2)

Error Code	Description
<code>e\$caller_must_wait</code> (1277)	The I/O operation could not be completed. Your application program is in no-wait mode and must wait for the connection to be established. To recover from this error, you must either issue a call to <code>s\$open</code> or call <code>s\$control</code> with the opcode <code>ABORT</code> (3).
<code>e\$device_already_assigned</code> (1155)	Another process is already attached to the device; either your process is not privileged or the device cannot be shared.
<code>e\$device_not_found</code> (1220)	The specified device is not known to the system. To list the known devices, issue the <code>list_devices</code> command. (This command is documented in the <i>VOS Commands Reference Manual</i> (R098).)
<code>e\$invalid_hardware_type</code> (2998)	The port is not attached to an asynchronous line or I/O adapter.
<code>e\$invalid_io_type</code> (1070)	The <code>io_type</code> argument does not have a valid value. You must correct your application program by specifying a value of 1, 2, 3, or 4 for <code>io_type</code> .
<code>e\$line_hangup</code> (1365)	For an RS-232-C channel/subchannel that is configured for dial-up, this error code indicates a loss of the DSR signal and/or the DCD signal. For most access layers, this error code indicates that the connection state is no longer <code>TERM_CONNECTION_ESTABLISHED</code> and that the connection is being terminated (that is, the connection state is most likely <code>TERM_CONNECTION_BREAKING</code> ). To determine the connection state, use the opcode <code>TERM_GET_CONNECTION_STATE</code> (2002). For information on how to return an RS-232-C channel/subchannel to service, see the description of the opcodes <code>TERM_HANGUP</code> (2001) and <code>TERM_LISTEN</code> (2023) in Chapter 8.
<code>e\$no_alloc_wired_heap</code> (3077)	There is insufficient space in the wired heap for the operating system to allocate the number of buffers that are needed in order to accommodate the <code>maximum_length</code> argument.

(Page 2 of 2)

Error Code	Description
e\$no_ports_available (1006)	The system does not have any more ports available.
e\$out_of_service (2535)	The RS-232-C channel/subchannel is out of service. This error may indicate that the asynchronous line or I/O adapter was removed, or that the error threshold for the channel/subchannel was exceeded, causing the operating system to remove the channel/subchannel from service. Your system administrator can return the channel/subchannel to service.

---

## Chapter 7

# Performing Input and Output Operations

This chapter describes five VOS subroutines that enable your application program to read input from a window terminal device and write output to a window terminal device.

- `s$read_raw`
- `s$seq_read`
- `s$write_raw`
- `s$seq_write`
- `s$seq_write_partial`

The `s$read_raw` and `s$seq_read` subroutines allow your application program to retrieve input from a terminal. The `s$read_raw` subroutine retrieves **raw** input and the `s$seq_read` subroutine retrieves **sequential** input. Although your application program can also retrieve sequential input using the `s$read` and `s$read_code` subroutines (which are variations of `s$seq_read`), these subroutines do not have a `record_length` argument and require you to use the `record_buffer` argument in your application program to determine the number of characters returned. Additionally, `s$read` cannot return error codes. Note that you cannot use `s$seq_read`, `s$read`, or `s$read_code` in your application program to retrieve **raw** input. To retrieve raw input, your application program must call `s$read_raw`.

This chapter also describes how your application program can send output using the subroutines `s$write_raw`, `s$seq_write`, and `s$seq_write_partial`. Your application program can also send output using the following alternative write subroutines: `s$write`, `s$write_code`, `s$write_partial`, `s$write_partial_code`, `s$write_wrap`, `s$write_wrap_indent`, and `s$write_wrap_partial`. These subroutines are similar to `s$seq_write` in that they allow the application program to write to subwindows and send **generic** output (that is, the output buffer can contain internal-character-set characters and generic output requests, and the window terminal driver can interpret the output requests). With these subroutines, the window terminal driver can interpret the generic output requests and display control characters based on the setting of the undisplayable notation character mode (see [Chapter 8](#)). However, these subroutines do **not** have a `buffer_length` argument (unlike `s$seq_write` and `s$seq_write_partial`)

and cannot report the number of characters actually written if the port is in no-wait mode or if a time limit expires.

These subroutines do not handle **raw** output. To handle raw output, your application program must call `s$write_raw`. For more information about the alternative read and write subroutines, see the VOS Subroutines manuals.

The five subroutines described in this chapter focus on terminal I/O. Each subroutine description explains how to make the call using PL/I or C. For complete descriptions of these subroutines and how they would apply to other types of I/O, refer to the VOS Subroutines manuals.



## s\$read\_raw

Use this subroutine to read raw input from the primary window attached to the specified port.

If the specified port is set for POSIX mode, `s$read_raw` implements POSIX read semantics. When the port is in POSIX mode and the ICANON `tcsetattr` bit is true, `s$read_raw` does the equivalent of an `s$seq_read` operation. When the port is in POSIX mode and the ICANON bit is false, `s$read_raw` follows POSIX raw input semantics. See the POSIX.1 Standard ISO/IEC 9945-1:1996(E) ANSI/IEEE Std. 1003.1, 1996 Edition, for information about POSIX read semantics. For general information about running POSIX-compliant programs on VOS, see the *Software Release Bulletin: VOS Release 14.3.0* (R914) and the online doc file `>system>doc>posix>posix.doc`.

The `s$read_raw` subroutine can retrieve two types of raw input: *unprocessed raw input* and *processed raw input*. Unprocessed raw input is raw data; it is not translated in any way before being passed to your application program. Processed raw input, on the other hand, is interpreted before being passed to your application program.

To collect unprocessed raw input, your application program must specify an unprocessed raw input mode with the opcode `TERM_SET_INPUT_MODE (2057)` before calling `s$read_raw`. (For a description of the opcode `TERM_SET_INPUT_MODE (2057)`, see the section “Opcodes Affecting the Primary Window and Subwindow” in [Chapter 8](#).)

There are three unprocessed raw input modes.

- Normal raw mode
- Raw table mode
- Raw record mode

These input modes use interrupts to control the flow of data in various ways. For a description of each of these modes, see [Chapter 3](#).

There are three processed raw input modes.

- Translated input mode
- Function-key input mode
- Generic input mode

These modes offer different levels of input-character mapping. To use translated input mode or function-key input mode, your application program must specify the mode with the opcode `TERM_SET_INPUT_MODE (2057)` before calling `s$read_raw`. Since generic input mode is the default input mode, your application program need only specify generic input mode to change from another raw input mode to generic input mode.

When your application program uses generic input mode or function-key input mode, `s$read_raw` uses binary data introducer (BDI) sequences to return generic input requests and function keys. After a single call to `s$read_raw`, the input buffer established by your application program may contain multiple BDI sequences. (For a description of the processed raw input modes and BDI sequences, see [Chapter 4](#).)

Although the level of processing varies among the `s$read_raw` input modes, they all share a common characteristic: they do **not** provide character echoing. Application programs using `s$read_raw` must perform their own character echoing.

[Table 7-1](#) and [Table 7-2](#) summarize the behavior of `s$read_raw` with the various input modes. [Table 7-1](#) describes the function of `s$read_raw` when your application program is using wait mode with the input modes. [Table 7-2](#) describes the function of `s$read_raw` when your application program is using no-wait mode with the input modes.

Although the normal raw, translated, function-key, and generic input modes are grouped together in [Table 7-1](#) and [Table 7-2](#), the data returned by each of these input modes is different and reflects the level of input mapping. For example, data in normal raw mode is not mapped at all, but data in function-key input mode is mapped to the internal character coding system and the keyboard database, and therefore contains representative codes that are passed to the application program. (For a description of translated input mode, function-key input mode, and generic input mode, see [Chapter 4](#). For a description of normal raw mode, raw table mode, and raw record mode, see [Chapter 3](#).)

In general, `s$read_raw` returns characters **as soon as they are available**, regardless of whether the application program is using wait mode or no-wait mode. The buffer does not have to be full in order for characters to be returned.

Characters are considered “available” for retrieval with `s$read_raw` upon the arrival of the appropriate interrupt character. With the normal raw, translated, function-key, and generic input modes, each incoming character is an interrupt character. With raw

table mode and raw record mode, only characters specified as interrupt characters in the interrupt table generate an interrupt. In raw record mode, each interrupt character is also a record terminator. (Note that while raw record mode is suitable for RS-232-C communications through the asynchronous access layer or for communications through an access layer such as OS TELNET, raw table mode is generally suitable for RS-232-C communications only.) There are four types of interrupt characters that can be specified in the interrupt table. See [Chapter 3](#) for more information about these interrupt characters.

If you are using `s$read_raw` with no-wait mode, your application program must be prepared to handle no-wait mode actions. With no-wait mode (see [Table 7-2](#)), the error code `e$caller_must_wait` (1277) is returned immediately when there is no data available. Your application program must then decide how to proceed. With wait mode, however, the window terminal driver waits until data is available before returning to your application program. (For information on how to use either wait mode or no-wait mode, see [Chapter 8](#).)

#### NOTE \_\_\_\_\_

In [Table 7-1](#) and [Table 7-2](#), *N* is the length of the buffer specified by the `buffer_length` argument, and *X* is the number of characters exceeding `buffer_length` (that is, the number of characters that did not fit in the buffer).

**Table 7-1. Wait Mode and `s$read_raw` (Page 1 of 2)**

	Normal Raw, Translated, Function-Key, or Generic Input Mode	Raw Table Mode	Raw Record Mode
<b>Zero Characters Available</b>	Wait.	Wait.	Wait.
<b>Partial Buffer</b> (1 to <i>N</i> -1 characters available, with interrupt character for raw table mode and raw record mode)	Data and a return code of 0.	Data and a return code of 0.	A record of data up to and including the interrupt character, plus a return code of 0.

Table 7-1. Wait Mode and s\$read\_raw (Page 2 of 2)

	Normal Raw, Translated, Function-Key, or Generic Input Mode	Raw Table Mode	Raw Record Mode
<b>Full Buffer</b>  ( <i>N</i> characters available, with interrupt character for raw table mode and raw record mode)	<i>N</i> characters and a return code of 0.	<i>N</i> characters and a return code of 0.	A record of data up to and including the interrupt character, plus a return code of 0.
<b>Buffer Maximum Exceeded</b>  ( <i>N</i> + <i>X</i> characters available, with interrupt character)	Not applicable to these modes. Call returns <i>N</i> characters and a return code of 0.	<i>N</i> characters and a return code of 0. Additional calls can retrieve <i>X</i> characters.	Partial record of <i>N</i> characters and error code e\$long_record (1026). Interrupt character generates interrupt, but is discarded along with <i>X</i> characters.

Table 7-2. No-Wait Mode and s\$read\_raw

	Normal Raw, Translated, Function-Key, or Generic Input Mode	Raw Table Mode	Raw Record Mode
<b>Zero Characters Available</b>	Error code e\$caller_must_wait (1277).	Error code e\$caller_must_wait (1277).	Error code e\$caller_must_wait (1277).
<b>Partial Buffer</b> (1 to $N-1$ characters available, with interrupt character for raw table mode and raw record mode)	Data and a return code of 0.	Data and a return code of 0.	A record of data up to and including the interrupt character, plus a return code of 0.
<b>Full Buffer</b> ( $N$ characters available, with interrupt character for raw table mode and raw record mode)	$N$ characters and a return code of 0.	$N$ characters and a return code of 0.	A record of data up to and including the interrupt character, plus a return code of 0.
<b>Buffer Maximum Exceeded</b> ( $N + X$ characters available, with interrupt character)	Not applicable to these modes. Call returns $N$ characters and a return code of 0.	$N$ characters and a return code of 0. Additional calls can retrieve $X$ characters.	Partial record of $N$ characters and error code e\$long_record (1026). Interrupt character generates interrupt, but is discarded along with $X$ characters.

## PL/I Usage

```
declare port_id                fixed bin (15);
declare buffer_length          fixed bin (15);
declare record_length          fixed bin (15);
declare record_buffer          char (N);
declare error_code             fixed bin (15);

declare s$read_raw entry(      fixed bin (15),
                              fixed bin (15),
                              fixed bin (15),
                              char (N),
                              fixed bin (15));

                                call s$read_raw(
                                port_id,
                                buffer_length,
                                record_length,
                                record_buffer,
                                error_code);
```

## C Usage

```
short int                      port_id;
short int                      buffer_length;
short int                      record_length;
char                           record_buffer [N];
short int                      error_code;

void s$read_raw();

s$read_raw(                    &port_id,
                              &buffer_length,
                              &record_length,
                              record_buffer,
                              &error_code);
```

## Arguments

The `s$read_raw` subroutine takes five arguments, which are listed and described in the following table.

Argument	Description
<code>port_id</code> (input)	The identifier of the port attached to the device.
<code>buffer_length</code> (input)	The length of the record in <code>record_buffer</code> , where <code>buffer_length</code> cannot exceed the length of <code>record_buffer</code> .
<code>record_length</code> (output)	The length of the record read.
<code>record_buffer</code> (output)	The number of characters actually returned.
<code>error_code</code> (output)	A returned error code.

## Error Codes

Of the error codes that `s$read_raw` can return, six are common RS-232-C asynchronous errors that indicate that data has been corrupted or lost:

`e$block_overrun` (2917), `e$frame_error` (2309), `e$line_hangup` (1365), `e$out_of_service` (2535), `e$overrun_error` (2919), and `e$parity_error` (2916). Note that the error code `e$line_hangup` (1365) is also valid for access layers such as OS TELNET.

The RS-232-C communications hardware detects frame, parity, and overrun errors when the input data from the device arrives at the adapter. The hardware identifies frame, parity, and overrun errors by inserting metacharacters in the input stream.

[Table 7-3](#) lists these asynchronous input-stream metacharacters.

**Table 7-3. Input-Stream Metacharacters**

<b>Metacharacter (in hexadecimal)</b>	<b>Function</b>
<code>^80</code>	Indicates a line break or a <code>BREAK_CHAR</code> request
<code>^81</code>	Indicates a frame error
<code>^82</code>	Indicates a parity error
<code>^83</code>	Indicates an overrun error
<code>^84</code>	Used internally for echo negotiation
<code>^85</code>	Used internally for echo negotiation
<code>^86</code>	Used internally as a quoted metacharacter

If your application program is using a raw input mode with 7-bit character transmission on K-series (RS-232-C) hardware, or with 8-bit character transmission on K-series hardware (with parity set to odd, even, or none), the operating system converts metacharacters in the range 81 through 83 hexadecimal to error codes. The subroutine `s$read_raw` then returns the appropriate error code to your application program (`e$frame_error` (2309), `e$parity_error` (2916), or `e$overrun_error` (2919)).

If your application program is using the `RETURN_ERROR` break option, `s$read_raw` can also return the error code `e$break_signalled` (1086). Your application program selects a break action using the opcode `TERM_SET_BREAK_ACTION` (2051), as described in the section “Opcodes Affecting the Primary Window and Subwindow” in [Chapter 8](#). The error code `e$break_signalled` (1086) is associated with the metacharacter 80 hexadecimal and indicates a line break or a `BREAK_CHAR` request.

If `s$read_raw` returns the error code `e$frame_error` (2309), `e$parity_error` (2916), or `e$overrun_error` (2919) in any raw input mode except raw record mode, this indicates that the data in an internal input buffer is corrupted and you should consider discarding the input returned in that buffer. Your application program receives the error code when `s$read_raw` processes an internal input buffer that contains the corrupted data. If your application program is retrieving raw input, it receives the error code on successive calls to `s$read_raw` until `s$read_raw` finishes reading the characters from the corrupted internal input buffer. Since your application program may receive the error code on successive calls to `s$read_raw`, it may not be obvious which call returned the corrupted data. This problem occurs in raw record mode when the user-supplied buffer is too small, and `s$read_raw` returns the error code `e$long_record` (1026). Typically, `s$read_raw` returns the entire contents of an internal input buffer in raw record mode. Application programs that use checksum



characters can retain the data and try to determine the exact location of the corrupted data in the internal input buffer.

Another common asynchronous error code is `e$line_hangup` (1365), which indicates that an RS-232-C dial-up channel/subchannel has lost the data set ready (DSR) signal and/or the data carrier detect (DCD) signal, thereby terminating the input stream. The application program can handle this error situation **only** if the RS-232-C channel/subchannel belongs to a slave dial-up device. If this is the case, the application program should issue the opcode `TERM_HANGUP` (2001) to hang up the slave line and then issue the opcode `TERM_LISTEN` (2023) to listen to the slave line. When an RS-232-C dial-up line for a login terminal hangs up, the error code is written to the system error log and the connection (and process) is terminated. Since all unread input is discarded with `e$line_hangup` (1365), you should design an interactive application program to handle a line hang-up. Note that for most access layers, the error code `e$line_hangup` (1365) generally indicates that the connection (and process) is being terminated.

The `s$read_raw` subroutine may return the error codes listed in [Table 7-4](#).

**Table 7-4. `s$read_raw` Error Codes** (Page 1 of 3)

Error Code	Description
<code>e\$block_overnun</code> (2917)	With RS-232-C communications, this error code indicates that data has been lost because the internal input buffer is too small. To avoid this error, you can increase the size of the internal input buffer with the <code>s\$control</code> opcode <code>TERM_SET_MAX_BUFFER_SIZE</code> (2010). You can also decrease the baud rate or change the reading method (for example, by using a different interrupt table). This error may also occur if your application program cannot process the incoming data fast enough (for example, if your application program handles several terminals and performs a great deal of input processing). If this is the case, design your application program to use input flow control. (See the description of the <code>s\$control</code> opcode <code>TERM_SET_INPUT_FLOW_INFO</code> (2008) in Chapter 8.)
<code>e\$break_signalled</code> (1086)	A line break occurred or the terminal user issued a <code>BREAK_CHAR</code> generic input request; however, this did not signal a break condition in the current process because your application program enabled the <code>RETURN_ERROR</code> break option. Your application program can specify a break action with the <code>s\$control</code> opcode <code>TERM_SET_BREAK_OPTION</code> (2051). (See Chapter 8 for a description of this opcode.)

**Table 7-4. s\$read\_raw Error Codes** (Page 2 of 3)

Error Code	Description
e\$buffer_too_small (1133)	Your application program specified an incorrect value (for example, a negative number) for the <code>buffer_length</code> argument. You must correct the error in your application program by specifying a valid buffer length. This error can also occur if your application program specifies the same value for the <code>buffer_length</code> and <code>record_length</code> arguments.
e\$caller_must_wait (1277)	The no-wait mode I/O operation could not be completed. Reissue the call to <code>s\$read_raw</code> .
e\$frame_error (2309)	The RS-232-C communications hardware has detected corrupted data. You should check the RS-232-C line and the hardware.
e\$line_hangup (1365)	If the RS-232-C channel/subchannel is configured for dial-up, this error code indicates a loss of the DSR signal and/or the DCD signal. For most access layers, this error code indicates that the connection state is no longer <code>TERM_CONNECTION_ESTABLISHED</code> and that the connection is being terminated (that is, the connection state is most likely <code>TERM_CONNECTION_BREAKING</code> ). All unread input has been discarded. To determine the connection state, use the opcode <code>TERM_GET_CONNECTION_STATE</code> (2002). For information on how to return an RS-232-C dial-up channel/subchannel to service, see the description of the <code>s\$control</code> opcodes <code>TERM_HANGUP</code> (2001) and <code>TERM_LISTEN</code> (2023) in Chapter 8.
e\$long_record (1026)	Your application program is using raw record mode and the input supplied exceeds the size of the internal input buffer. (The buffer does not contain the interrupt character or any of its trailer bytes.) In this case, <code>s\$read_raw</code> waits for the interrupt character and returns a partial record of <code>buffer_length</code> characters. (The <code>record_length</code> argument is set to the length of the partial record.) The window terminal driver discards any characters that arrived after the buffer was filled and before the interrupt character arrived.
e\$out_of_service (2535)	The RS-232-C channel/subchannel is out of service. This error code may indicate that the line or I/O adapter was removed or that the error threshold for the channel/subchannel was exceeded, causing the operating system to remove the channel/subchannel from service. Your system administrator can return the channel/subchannel to service.

**Table 7-4. s\$read\_raw Error Codes** (Page 3 of 3)

Error Code	Description
e\$overrun_error (2919)	The RS-232-C communications hardware cannot handle the volume of data. Data may be overwritten or lost. You should lower the baud rate for the RS-232-C channel/subchannel associated with the device, or increase the size of the internal input buffer. This error may occur if your application program cannot process the incoming data fast enough. If this is the case, design your application program to use input flow control. (See the description of the s\$control opcode TERM_SET_INPUT_FLOW_INFO (2008) in Chapter 8.)
e\$parity_error (2916)	The RS-232-C communications hardware has detected corrupted data. The parity bit of the incoming byte did not match the parity specified for the device's configuration. You should also check the hardware and the RS-232-C line.
e\$port_not_attached (1021)	The specified port is not attached. (It may never have been attached or it may have been detached.) This error code may also indicate a problem on the network.
e\$timeout (1081)	An I/O time limit expired before enough input characters were available.

## **s\$seq\_read**

Use this subroutine to read records from the current I/O subwindow of the primary window attached to the specified port. To call `s$seq_read`, your application program must use one of two types of sequential input.

- Simple sequential input (part of the simple sequential I/O approach)
- Formatted sequential input (part of the application-managed I/O approach)

With both types of sequential input, the window terminal driver observes subwindow boundaries to ensure that data always stays within the subwindow. (See [Chapter 1](#) for a description of subwindows.)

With simple sequential I/O (line-mode I/O), the window terminal driver manages the entire display of the subwindow (the original subwindow). Simple sequential I/O is used at command level. It provides character echoing, input line-editing features, visible typeahead, line wrapping, pause processing, and automatic refreshing of the subwindow. Simple sequential I/O uses generic input mode, so your application program need not specify generic input mode with `s$control`. Note that simple sequential I/O uses a variation of generic input mode that does not cause `s$seq_read` to return function-key codes or generic input request codes. The window terminal driver automatically interprets the requests and executes them on the input line. For more information about simple sequential I/O, see [Chapter 2](#).

With formatted sequential input, the application program (not the window terminal driver) manages the display of the subwindow. Input is echoed on the input line **only** if an `s$seq_read` operation is active. This means that typeahead is not processed or visible with formatted sequential input. In addition, input line-editing requests are processed only if an `s$seq_read` operation is active.

To use formatted sequential input, your application program must ensure that the current I/O subwindow is in formatted I/O mode. This mode affects all sequential reads and writes to the current I/O subwindow. (See [Chapter 4](#) for a description of formatted I/O mode.)

For both types of sequential I/O, `s$seq_read` can return input to your application program when it receives a complete record. If `s$seq_read` detects a record terminator and then finds that the entire input record does not fit in the input buffer, it

returns the partial record that does fit in the buffer and the error code `e$long_record` (1026). This error code indicates that the remaining part of the input record (that is, the buffer overflow) has been discarded. This situation applies to both wait mode and no-wait mode. In addition, if only part of a multibyte character can fit at the end of the buffer, `s$seq_read` replaces the partial character with 1A hexadecimal (SUB); the remainder of the character is discarded.

If your application program is using no-wait mode with `s$seq_read`, it receives the error code `e$caller_must_wait` (1277) if no record is available. The application program must then decide how to proceed.

If your application program is using wait mode, the window terminal driver waits until a record is available before returning (unless a time-out occurs; see [Chapter 8](#)). Since the operating system (not your application program) controls the waiting period in wait mode, your application program must wait until the call returns.

The `s$seq_read` subroutine recognizes the following requests as valid record terminators.

- RETURN
- ENTER
- FORM
- HELP
- BREAK\_CHAR (depending on the break action; see [Chapter 8](#))
- END\_OF\_FILE
- ESI, EGI, and EMI (for COBOL users)

Note that these record terminators are not returned in the record buffer. However, if the record is terminated with FORM, HELP, END\_OF\_FILE, ESI, EGI, or EMI, and no other error occurs, `s$seq_read` returns the appropriate error code: `e$form_requested` (1225), `e$help_requested` (4085), `e$end_of_file` (1025), `e$esi` (2856), `e$egi` (2858), or `e$emi` (2857). If the record is terminated with either RETURN or ENTER, `s$seq_read` does not return an error code.

## PL/I Usage

```
declare port_id           fixed bin (15);
declare buffer_length     fixed bin (15);
declare record_length     fixed bin (15);
declare record_buffer     char (N);
declare error_code        fixed bin (15);

declare s$seq_read entry(      fixed bin (15),
                              fixed bin (15),
                              fixed bin (15),
                              char (N),
                              fixed bin (15));

      call s$seq_read(        port_id,
                              buffer_length,
                              record_length,
                              record_buffer,
                              error_code);
```

## C Usage

```
short int                port_id;
short int                buffer_length;
short int                record_length;
char                    record_buffer [N];
short int                error_code;

void s$seq_read();

      s$seq_read(            &port_id,
                              &buffer_length,
                              &record_length,
                              record_buffer,
                              &error_code);
```

## Arguments

The `s$seq_read` subroutine takes five arguments, which are listed and described in the following table.

Argument	Description
<code>port_id</code> (input)	The identifier of the port attached to the device.
<code>buffer_length</code> (input)	The length of the record in <code>record_buffer</code> , where <code>buffer_length</code> cannot exceed the length of <code>record_buffer</code> . Your application program should supply a buffer of at least 300 bytes in order to prevent the error code <code>e\$long_record</code> (1026).
<code>record_length</code> (output)	The length of the record read.
<code>record_buffer</code> (output)	The record of data actually returned.
<code>error_code</code> (output)	A returned error code.

## Error Codes

With `s$seq_read`, RS-232-C communications hardware detects frame, parity, and overrun errors when the input data from the device arrives at the adapter. The hardware inserts the appropriate metacharacters in the input stream when these errors occur. (For a list of the metacharacters used by the hardware, see [Table 7-4](#), which appears in the `s$read_raw` “Error Codes” section earlier in this chapter.) With 7-bit character transmission on K-series (RS-232-C) hardware, or 8-bit transmission on K-series hardware (with parity set to odd, even, or none), the operating system converts metacharacters in the range 81 through 83 hexadecimal to asynchronous error codes. The subroutine `s$seq_read` then returns the appropriate error code to your application program (`e$frame_error` (2309), `e$parity_error` (2916), or `e$overrun_error` (2919)). Each of these error codes indicates that the data in the input record may be corrupted or lost. The operating system returns the contents of the input record but discards all typeahead (unread input) or pending output.

If your application program is using the `RETURN_ERROR` break option, `s$seq_read` can also return the error code `e$break_signalled` (1086). Your application program selects a break action using the opcode `TERM_SET_BREAK_ACTION` (2051), as described in the section “Opcodes Affecting the Primary Window and Subwindow” in [Chapter 8](#). The error code `e$break_signalled` (1086) is associated with the metacharacter 80 hexadecimal and indicates a line break or a `BREAK_CHAR` request.

Another common asynchronous error code is `e$line_hangup` (1365), which indicates that an RS-232-C dial-up channel/subchannel has lost the data set ready (DSR) signal and/or the data carrier detect (DCD) signal, and that unread input has

been discarded. The application program can handle this error situation **only** if the channel/subchannel belongs to a slave dial-up device. If this is the case, the application program should issue the opcode `TERM_HANGUP` (2001) to hang up the slave line and then issue the opcode `TERM_LISTEN` (2023) to listen to the slave line.

When an RS-232-C dial-up line for a login terminal hangs up, the error code is written to the system error log and the connection (and process) is terminated. To avoid the loss of unread input, you should design an interactive application program to handle a line hang-up.

Note that for most access layers, the error code `e$line_hangup` (1365) generally indicates that the connection (and process) is being terminated.

The `s$seq_read` subroutine may return the following error codes.

(Page 1 of 4)

Error Code	Description
<code>e\$block_overnrun</code> (2917)	With RS-232-C communications, this error code indicates that data has been lost because the internal input buffer is too small. To avoid this error, you can increase the size of the internal input buffer with the <code>s\$control</code> opcode <code>TERM_SET_MAX_BUFFER_SIZE</code> (2010). You can also decrease the baud rate or change the reading method (for example, by using a different interrupt table). This error may also occur if your application program cannot process the incoming data fast enough (for example, if your application program handles several terminals and performs a great deal of input processing). If this is the case, design your application program to use input flow control. (See the description of the <code>s\$control</code> opcode <code>TERM_SET_INPUT_FLOW_INFO</code> (2008) in Chapter 8.)
<code>e\$break_signalled</code> (1086)	A line break occurred or the terminal user issued a <code>BREAK_CHAR</code> generic input request; however, this did not signal a break condition in the current process because your application program enabled the <code>RETURN_ERROR</code> break option. Your application program can specify a break action with the <code>s\$control</code> opcode <code>TERM_SET_BREAK_OPTION</code> (2051). (See Chapter 8 for a description of this opcode.)



(Page 2 of 4)

Error Code	Description
e\$buffer_too_small (1133)	Your application program specified an incorrect value (for example, a negative number) for the <code>buffer_length</code> argument. You must correct the error in your application program by specifying a valid buffer length. This error can also occur if your application program specifies the same value for the <code>buffer_length</code> and <code>record_length</code> arguments.
e\$caller_must_wait (1277)	The no-wait mode I/O operation could not be completed. (A complete input record was unavailable.) Reissue the call to <code>s\$seq_read</code> .
e\$egi (2858), e\$emi (2857), e\$esi (2856)	The input line was terminated with the EGI, EMI, or ESI COBOL request.
e\$end_of_file (1025)	The input line was terminated with the <code>END_OF_FILE</code> request.
e\$feature_unavailable (1687)	The requested feature is currently unavailable, possibly due to the status of the current I/O subwindow.
e\$form_requested (1225)	The input line was terminated with the <code>FORM</code> request.
e\$frame_error (2309)	The RS-232-C communications hardware has detected corrupted data. You should check the RS-232-C line and the hardware.
e\$help_requested (4085)	The input line was terminated with the <code>HELP</code> request.
e\$invalid_io_operation (1040)	The read cannot be performed because at least one input character must be echoed to the current I/O subwindow.

(Page 3 of 4)

Error Code	Description
e\$line_hangup (1365)	If the RS-232-C channel/subchannel is configured for dial-up, this error code indicates a loss of the DSR signal and/or the DCD signal. For most access layers, this error code indicates that the connection state is no longer <code>TERM_CONNECTION_ESTABLISHED</code> and that the connection is being terminated (that is, the connection state is most likely <code>TERM_CONNECTION_BREAKING</code> ). All unread input has been discarded. To determine the connection state, use the opcode <code>TERM_GET_CONNECTION_STATE</code> (2002). For information on how to return an RS-232-C dial-up channel/subchannel to service, see the description of the <code>s\$control</code> opcodes <code>TERM_HANGUP</code> (2001) and <code>TERM_LISTEN</code> (2023) in Chapter 8.
e\$long_record (1026)	The input supplied exceeds the size of the internal input buffer. In this case, <code>s\$seq_read</code> returns <code>buffer_length</code> characters ( <code>record_length</code> is set to the length of the record). If only part of a multibyte character can fit at the end of the buffer, the partial character is replaced with 1A hexadecimal (SUB); the remainder of the character is discarded. For simple sequential I/O, your application program should supply a buffer of at least 300 bytes in order to avoid this error situation.
e\$out_of_service (2535)	The RS-232-C channel/subchannel is out of service. This error code may indicate that the line or I/O adapter was removed, or that the error threshold for the channel/subchannel was exceeded, causing the operating system to remove the channel/subchannel from service. Your system administrator can return the channel/subchannel to service.
e\$overrun_error (2919)	The RS-232-C communications hardware cannot handle the volume of data. Data may be overwritten or lost. You should lower the baud rate for the channel/subchannel associated with the device, or increase the size of the internal input buffer. This error may occur if your application program cannot process the incoming data fast enough. If this is the case, design your application program to use input flow control. (See the description of the <code>s\$control</code> opcode <code>TERM_SET_INPUT_FLOW_INFO</code> (2008) in Chapter 8.)

(Page 4 of 4)

Error Code	Description
e\$parity_error (2916)	The RS-232-C communications hardware has detected corrupted data. The parity bit of the incoming byte did not match the parity specified for the device's configuration. You should also check the hardware and the RS-232-C line.
e\$port_not_attached (1021)	The specified port is not attached. (It may never have been attached or it may have been detached.) This error code may also indicate a problem on the network.
e\$redisplay_form (3363)	The visible part of a subwindow has not been refreshed since the last time the subwindow handled an I/O operation. The call returns this error code only if the port is attached with the <code>hold_open</code> switch and/or the <code>hold_attached</code> switch turned on. (See the description of <code>s\$attach_port</code> in Chapter 6 for more information about the effect of these switches on the port.) If your application program receives this error code, it must refresh the subwindow.
e\$timeout (1081)	An I/O time limit expired before enough input characters were available.

## s\$write\_raw

Use `s$write_raw` to send raw data to the primary window attached to the specified port. All data sent to the primary window with `s$write_raw` is unprocessed. Characters are not translated, there is no pause processing or line wrapping, and generic output requests are not interpreted in any way.

If the specified port is set for POSIX mode, `s$write_raw` implements POSIX write semantics. When the port is in POSIX mode and the `OPOST tcsetattr` bit is true, `s$write_raw` does the equivalent of an `s$seq_write_partial` operation. When the port is in POSIX mode and the `OPOST` bit is false, `s$write_raw` follows POSIX raw output semantics. See the POSIX documentation for information about POSIX output semantics. See the POSIX.1 Standard ISO/IEC 9945-1:1996(E) ANSI/IEEE Std. 1003.1, 1996 Edition, for information about POSIX write semantics. For general information about running POSIX-compliant programs on VOS, see the *Software Release Bulletin: VOS Release 14.3.0* (R914) and the online doc file `>system>doc>posix>posix.doc`.

It is important to note that `s$write_raw` does **not** observe subwindow boundaries, which makes it unsuitable for a terminal's primary window/subwindow environment. Executing `s$write_raw` in the primary window invalidates the state of the terminal display, which is maintained by the window terminal driver. Therefore, your application program should use `s$write_raw` **only** for unprocessed raw I/O (see [Chapter 3](#)).

If your application program is using no-wait mode, and the data does not fit in the available internal output buffers, `s$write_raw` sends as much data as possible and returns the error code `e$caller_must_wait` (1277). The value of the `buffer_length` argument is then updated to reflect the number of characters actually sent. To send the remaining data, your application program must repeatedly call `s$write_raw`. (For more information, see the description of no-wait mode in [Chapter 3](#) and [Chapter 8](#).)

If your application program is using wait mode, `s$write_raw` does not return until `buffer_length` bytes have been transferred to the available internal output buffers. Your application program can define the amount of time that `s$write_raw` waits before returning by calling `s$set_io_time_limit` (see [Chapter 8](#)). If the time limit expires, `s$write_raw` updates `buffer_length` with the number of bytes actually transferred and returns the error code `e$timeout` (1081).

**PL/I Usage**

```
declare port_id                fixed bin (15);
declare buffer_length          fixed bin (15);
declare record_buffer          char (N);
declare error_code             fixed bin (15);

declare s$write_raw entry(      fixed bin (15),
                                fixed bin (15),
                                char (N),
                                fixed bin (15));

                                call s$write_raw(  port_id,
                                                    buffer_length,
                                                    record_buffer,
                                                    error_code);
```

**C Usage**

```
short int    port_id;
short int    buffer_length;
char         record_buffer [N];
short int    error_code;

void s$write_raw();

s$write_raw(    &port_id,
                &buffer_length,
                record_buffer,
                &error_code);
```

**Arguments**

The `s$write_raw` subroutine takes four arguments, which are listed and described in the following table.

*(Page 1 of 2)*

Argument	Description
port_id (input)	The identifier of the port attached to the device.
buffer_length (input/output)	On input, the number of bytes in record_buffer, where buffer_length cannot exceed the length of record_buffer. On output, buffer_length is the number of bytes in record_buffer successfully written to the device.
record_buffer (input)	The bytes to be written to the device.

(Page 2 of 2)

Argument	Description
error_code (output)	A returned error code.

## Error Codes

The `s$write_raw` subroutine may return the following error codes.

Error Code	Description
e\$buffer_too_small (1133)	Your application program specified an incorrect value (for example, a negative number) for the <code>buffer_length</code> argument. You must correct the error in your application program by specifying a valid buffer length.
e\$caller_must_wait (1277)	The no-wait mode I/O operation could not be completed. Reissue the call to <code>s\$write_raw</code> .
e\$line_hangup (1365)	If the RS-232-C channel/subchannel is configured for dial-up, this error code indicates a loss of the DSR signal and/or the DCD signal. For most access layers, this error code indicates that the connection state is no longer <code>TERM_CONNECTION_ESTABLISHED</code> and that the connection is being terminated (that is, the connection state is most likely <code>TERM_CONNECTION_BREAKING</code> ). All unread input has been discarded. To determine the connection state, use the opcode <code>TERM_GET_CONNECTION_STATE</code> (2002). For information on how to return an RS-232-C dial-up channel/subchannel to service, see the description of the <code>s\$control</code> opcodes <code>TERM_HANGUP</code> (2001) and <code>TERM_LISTEN</code> (2023) in Chapter 8.
e\$out_of_service (2535)	The RS-232-C channel/subchannel is out of service. This error code may indicate that the line or I/O adapter was removed, or that the error threshold for the channel/subchannel was exceeded, causing the operating system to remove the channel/subchannel from service. Your system administrator can return the channel/subchannel to service.
e\$timeout (1081)	An I/O time limit expired before the operation could be completed.

## s\$seq\_write

Use the `s$seq_write` subroutine to send a record to the current I/O subwindow of the primary window attached to the specified port. Your application program should call `s$seq_write` if it uses one of two types of sequential output.

- Simple sequential output (part of the simple sequential I/O approach)
- Formatted sequential output (part of the application-managed I/O approach)

With both types of sequential output, `s$seq_write` observes subwindow boundaries. It handles the processing of characters defined in the internal character coding system, graphic characters, control characters, new-line characters, generic output requests, and tabs. Note that `s$seq_write` performs a new-line operation after it processes the data.

The behavior of `s$seq_write` depends on whether your application program is using simple sequential output (part of the simple sequential I/O approach) or formatted sequential output (part of the application-managed I/O approach). If your application program is using simple sequential output, the window terminal driver manages the display of output in the original subwindow. (See [Chapter 1](#) for a description of the original subwindow.) Simple sequential output allows `s$seq_write` to perform automatic line wrapping and pause processing. In addition, simple sequential output ensures that the window terminal driver always preserves the contents of the subwindow and refreshes the display when the subwindow is rearranged or resized. (See [Chapter 2](#) for more information about simple sequential I/O.)

If your application program is using formatted sequential output to manage the display of the subwindow, the window terminal driver does not restrict your application program by performing any line wrapping or pause processing. When data exceeds the maximum line length, it is truncated. To enable your application program to control the display of the subwindow, formatted sequential output offers an extended set of generic output requests (see [Chapter 4](#)). As long as the port is operating in cached I/O mode, the window terminal driver can ensure the integrity of the data within the subwindow (On the next `s$seq_read` or `s$seq_write`, it can automatically redisplay any data that has been overlapped by another subwindow or primary window).

To send formatted sequential output, your application program must ensure that the current I/O subwindow is in formatted I/O mode. To enable formatted I/O mode in the

current I/O subwindow, your application program must either create a new formatted sequential I/O subwindow or switch the original subwindow from simple sequential I/O to formatted sequential I/O. (See [Chapter 4](#) for more information about formatted I/O mode.) Note that if your application program uses simple sequential I/O and embeds certain generic output requests in the output buffer, *s\$seq\_write* automatically switches the original subwindow to formatted I/O mode before it processes the requests. (These requests are listed at the end of [Chapter 2](#) and are described in [Chapter 5](#).)

In formatted I/O mode, certain generic output requests cause the window terminal driver to process the output string differently in terms of subwindow updates. These requests are listed in the section “Generic Output Requests That Affect Subwindow Updates” in Chapter 4.

If control characters are embedded in the output string (that is, bytes less than 20 hexadecimal), *s\$seq\_write* functions in the following manner.

- If the TTP for the device supports the control-character operation, *s\$seq\_write* performs the appropriate action.
- If the TTP for the device does not support the control-character operation, *s\$seq\_write* handles control characters according to the undisplayable notation character mode specified by your application program. Your application program can select an undisplayable notation character mode that either replaces control characters with hexadecimal notation, replaces control characters with printable characters defined in the TTP, or discards control characters. Your application program selects the undisplayable notation character mode using the *s\$control* opcode `TERM_SET_UNDISP_MODE (2065)`. (See [Chapter 8](#).)

The control characters `VT` (vertical tab) and `FF` (form feed) are not supported and are always handled according to the undisplayable notation character mode.

The following list identifies the control characters that are typically supported in the TTP and specifies the action performed by *s\$seq\_write* when processing simple sequential I/O and formatted sequential I/O. Note that these control characters (except for `HT`) should be supported in the TTP; if they are not, they are ignored (discarded).



Control Character	Action with Simple Sequential I/O	Action with Formatted Sequential I/O
BEL (bell)	Performs a BEEP request	Performs a BEEP request
BS (backspace)	Causes the switch to formatted I/O mode	Performs a backspace (LEFT request)
HT (horizontal tab)	Inserts spaces up to the next tab	Inserts spaces up to the next tab
LF (line feed)	Performs a NEW_LINE request	Performs a NEW_LINE request
CR (carriage return)	Causes the switch to formatted I/O mode	Performs a carriage return

How the `s$seq_write` subroutine returns to your application program depends on whether your application program is using wait mode or no-wait mode.

Typically, if your application program is using no-wait mode and `s$seq_write` cannot complete the write (for example, your application program is using simple sequential output and the display of output is paused), `s$seq_write` returns the error code `e$caller_must_wait` (1277) and updates the `buffer_length` argument to contain the number of characters actually processed. In this case, your application program can advance the buffer pointer to point to the first unsent byte and then reissue the call to `s$seq_write` to complete the operation. Note that if either your application program or the operating system issues the `s$control` opcode `TERM_DISCARD_OUTPUT` (2041) before or during an `s$seq_write` operation, the call returns the error code `e$abort_output` (1279).

If your application program is using wait mode, `s$seq_write` does not return until `buffer_length` characters have been processed. Your application program can define the amount of time that `s$seq_write` waits before returning by calling the `s$set_io_time_limit` subroutine (see [Chapter 8](#)).

## PL/I Usage

```
declare port_id                fixed bin (15);
declare buffer_length          fixed bin (15);
declare record_buffer          char (N);
declare error_code             fixed bin (15);

declare s$seq_write entry(      fixed bin (15),
                                fixed bin (15),
                                char (N),
                                fixed bin (15));

                                call s$seq_write(
                                port_id,
                                buffer_length,
                                record_buffer,
                                error_code);
```

## C Usage

```
short int                      port_id;
short int                      buffer_length;
char                           record_buffer [N];
short int                      error_code;

void s$seq_write();

s$seq_write(                    &port_id,
                                &buffer_length,
                                record_buffer,
                                &error_code);
```

## Arguments

The `s$seq_write` subroutine takes four arguments, which are listed and described in the following table.

Argument	Description
<code>port_id</code> (input)	The identifier of the port attached to the device.
<code>buffer_length</code> (input/output)	On input, the length of the record in <code>record_buffer</code> , where <code>buffer_length</code> cannot exceed the length of <code>record_buffer</code> . On output, <code>buffer_length</code> is the number of characters in <code>record_buffer</code> processed when the call returned. The buffer length cannot exceed 32,767.
<code>record_buffer</code> (input)	The record to be written to the device.
<code>error_code</code> (output)	A returned error code.

## Error Codes

The `s$seq_write` subroutine may return the following error codes.

(Page 1 of 2)

Error Code	Description
<code>e\$abort_output</code> (1279)	The terminal user pressed the key mapped to the <code>ABORT_OUTPUT</code> request when simple sequential I/O was enabled. After receiving this error code, your application program should issue the <code>s\$control</code> opcode <code>TERM_RESET_OUTPUT</code> (2089). Otherwise, the operating system continues to discard pending output, and subsequent calls to <code>s\$seq_write</code> return this error code (see Chapter 8).
<code>e\$buffer_too_small</code> (1133)	Your application program specified an incorrect value (for example, a negative number) for the <code>buffer_length</code> argument. You must correct the error in your application program by specifying a valid buffer length.
<code>e\$caller_must_wait</code> (1277)	The no-wait mode I/O operation could not be completed. Reissue the call to <code>s\$seq_write</code> .
<code>e\$feature_unavailable</code> (1687)	The requested feature is currently unavailable, possibly due to the status of the current I/O subwindow.

(Page 2 of 2)

Error Code	Description
e\$line_hangup (1365)	If the RS-232-C channel/subchannel is configured for dial-up, this error code indicates a loss of the DSR signal and/or the DCD signal. For most access layers, this error code indicates that the connection state is no longer TERM_CONNECTION_ESTABLISHED and that the connection is being terminated (that is, the connection state is most likely TERM_CONNECTION_BREAKING). All unread input has been discarded. To determine the connection state, use the opcode TERM_GET_CONNECTION_STATE (2002). For information on how to return an RS-232-C dial-up channel/subchannel to service, see the description of the s\$control opcodes TERM_HANGUP (2001) and TERM_LISTEN (2023) in Chapter 8.
e\$out_of_service (2535)	The RS-232-C channel/subchannel is out of service. This error code may indicate that the line or I/O adapter was removed, or that the error threshold for the channel/subchannel was exceeded, causing the operating system to remove the channel/subchannel from service. Your system administrator can return the channel/subchannel to service.
e\$redisplay_form (3363)	The visible part of a subwindow has not been refreshed since the last time the subwindow handled an I/O operation. The call returns this error code only if the port is attached with the hold_open switch and/or the hold_attached switch turned on. (See the description of s\$attach_port in Chapter 6 for more information about the effect of these switches on the port.) If your application program receives this error code, it must refresh the subwindow.
e\$timeout (1081)	An I/O time limit expired before the operation could be completed.

## **s\$seq\_write\_partial**

Use this subroutine to write a partial record to the current I/O subwindow of the primary window attached to the specified port.

In general, `s$seq_write_partial` performs the same processing as `s$seq_write`. However, `s$seq_write_partial` does not perform a new-line operation after it processes the output data. Once the output data in `record_buffer` is sent to the device, the next output character appears at the next column position and in the same row as the last output character; the next input character appears at this position as well.

PL/I Usage

```
declare port_id                fixed bin (15);
declare buffer_length          fixed bin (15);
declare record_buffer          char (N);
declare error_code             fixed bin (15);

declare s$seq_write_partial entry( fixed bin (15),
                                   fixed bin (15),
                                   char (N),
                                   fixed bin (15));

      call s$seq_write_partial( port_id,
                                buffer_length,
                                record_buffer,
                                error_code);
```

C Usage

```
short int      port_id;
short int      buffer_length;
char           record_buffer [N];
short int      error_code;

void s$seq_write_partial();

      s$seq_write_partial(      &port_id,
                                &buffer_length,
                                record_buffer,
                                &error_code);
```

Arguments

The `s$seq_write_partial` subroutine takes four arguments, which are listed and described in the following table.

(Page 1 of 2)

Argument	Description
port_id (input)	The identifier of the port attached to the device.
buffer_length (input/output)	On input, the length of the record in <code>record_buffer</code> , where <code>buffer_length</code> cannot exceed the length of <code>record_buffer</code> . On output, <code>buffer_length</code> is the number of characters in <code>record_buffer</code> processed when the call returned. The buffer length cannot exceed 32,767.

(Page 2 of 2)

Argument	Description
record_buffer (input)	The record to be written to the device.
error_code (output)	A returned error code.

## Error Codes

The `s$seq_write_partial` subroutine may return the following error codes.

(Page 1 of 2)

Error Code	Description
e\$abort_output (1279)	The terminal user pressed the key mapped to the <code>ABORT_OUTPUT</code> request when simple sequential I/O was enabled. After receiving this error code, your application program should issue the <code>s\$control</code> opcode <code>TERM_RESET_OUTPUT</code> (2089). Otherwise, the operating system continues to discard pending output, and subsequent calls to <code>s\$seq_write_partial</code> return this error code (see Chapter 8).
e\$buffer_too_small (1133)	Your application program specified an incorrect value (for example, a negative number) for the <code>buffer_length</code> argument. You must correct the error in your application program by specifying a valid buffer length.
e\$caller_must_wait (1277)	The no-wait mode I/O operation could not be completed. Reissue the call to <code>s\$seq_write_partial</code> .
e\$feature_unavailable (1687)	The requested feature is currently unavailable, possibly due to the status of the current I/O subwindow.

(Page 2 of 2)

Error Code	Description
e\$line_hangup (1365)	If the RS-232-C channel/subchannel is configured for dial-up, this error code indicates a loss of the DSR signal and/or the DCD signal. For most access layers, this error code indicates that the connection state is no longer TERM_CONNECTION_ESTABLISHED and that the connection is being terminated (that is, the connection state is most likely TERM_CONNECTION_BREAKING). All unread input has been discarded. To determine the connection state, use the opcode TERM_GET_CONNECTION_STATE (2002). For information on how to return an RS-232-C dial-up channel/subchannel to service, see the description of the s\$control opcodes TERM_HANGUP (2001) and TERM_LISTEN (2023) in Chapter 8.
e\$out_of_service (2535)	The RS-232-C channel/subchannel is out of service. This error code may indicate that the line or I/O adapter was removed, or that the error threshold for the channel/subchannel was exceeded, causing the operating system to remove the channel/subchannel from service. Your system administrator can return the channel/subchannel to service.
e\$redisplay_form (3363)	The visible part of a subwindow has not been refreshed since the last time the subwindow handled an I/O operation. The call returns this error code only if the port is attached with the hold_open switch and/or the hold_attached switch turned on. (See the description of s\$attach_port in Chapter 6 for more information about the effect of these switches on the port.) If your application program receives this error code, it must refresh the subwindow.
e\$timeout (1081)	An I/O time limit expired before the operation could be completed.



---

## Chapter 8

# Performing Control Operations

This chapter describes six VOS subroutines that allow your application program to perform various control operations, such as setting 10.0 as an I/O time limit, switching the port to no-wait mode, returning the port to wait mode, using device events, and specifying control parameters for the window terminal connection.

- `s$set_io_time_limit`
- `s$set_wait_mode`
- `s$set_no_wait_mode`
- `s$read_device_event`
- `s$control`
- `s$control_device`

Note that the subroutine descriptions focus on terminal I/O. For complete descriptions of the `s$set_io_time_limit`, `s$set_wait_mode`, `s$set_no_wait_mode`, and `s$control` subroutines, and how they apply to other types of I/O, refer to the VOS Subroutines manuals. The `s$read_device_event` and `s$control_device` subroutines are documented in this manual only.

Each subroutine description in this chapter explains how to make the call from an application program written in either PL/I or C.

## `s$set_io_time_limit`

Use this subroutine to control how long the operating system waits for an I/O subroutine call to complete before returning control to your application program. Using `s$set_io_time_limit` is an alternative to using no-wait mode.

If an I/O operation (for example, an `s$read_raw` or `s$seq_write` subroutine call) does not complete before the time limit expires, the appropriate subroutine returns the error code `e$timeout (1081)`. The kernel uses no-wait mode to implement this, but it does an `ABORT` when it times out; using `s$wait_event` may not work.

If the time limit expires during an `s$read_raw` operation (that is, the `s$read_raw` subroutine returns the error code `e$timeout (1081)`), your application program must be prepared to receive partial input, just as in no-wait mode. If the time limit expires during an `s$write_raw` operation (that is, the `s$write_raw` subroutine returns the error code `e$timeout (1081)`), your application program must be prepared to send partial output, just as in no-wait mode.

Setting the time limit to 0 has the same effect as switching the port to no-wait mode. Setting the time limit to -1 allows the operating system to wait as long as necessary and is equivalent to normal wait-mode I/O.

A time limit set with `s$set_io_time_limit` is nullified by any subsequent call to `s$set_io_time_limit`, `s$set_no_wait_mode`, or `s$set_wait_mode`.

## PL/I Usage

```

declare port_id                fixed bin (15);
declare time_limit             fixed bin (31);
declare error_code             fixed bin (15);

declare s$$set_io_time_limit entry( fixed bin (15),
                                   fixed bin (31),
                                   fixed bin (15));

      call s$$set_io_time_limit( port_id,
                                time_limit,
                                error_code);

```

## C Usage

```

short int    port_id;
long int     time_limit;
short int    error_code;

void s$$set_io_time_limit();

      s$$set_io_time_limit(    &port_id,
                              &time_limit,
                              &error_code);

```

## Arguments

The `s$$set_io_time_limit` subroutine takes three arguments, which are listed and described in the following table.

Argument	Description
<code>port_id</code> (input)	The identifier of the port attached to the device.
<code>time_limit</code> (input)	The I/O time limit, in units of 1/1024 of a second.
<code>error_code</code> (output)	A returned error code.

**Error Codes**

The `s$set_io_time_limit` subroutine may return the following error codes.

Error Code	Description
<code>e\$invalid_io_operation</code> (1040)	Your application program attempted to perform an I/O operation that it is not authorized or correctly initialized to perform.
<code>e\$port_not_attached</code> (1021)	The specified port is not attached. (It may never have been attached or it may have been detached.) This error code may also indicate a problem on the network.

## s\$\$set\_wait\_mode

Use this subroutine to enable the operating system to control waiting for your application program. In wait mode, your application program **cannot** expect read or write calls to return immediately. The operating system waits when necessary and decides when the call can return. By default, all ports operate in wait mode initially. Therefore, your application program need only call `s$$set_wait_mode` to disable an earlier call to `s$$set_no_wait_mode` or `s$$set_io_time_limit`.

When wait mode is used with any of the raw input modes, a call to `s$$read_raw` returns as soon as data is available (that is, the input buffer need not be full). Note that in raw record mode, data is available for retrieval with `s$$read_raw` when a record is terminated. (In raw record mode, a record is terminated when the window terminal driver detects the appropriate interrupt character.) In raw table mode, data is available when the window terminal driver detects the appropriate interrupt character. In generic input mode, translated input mode, function-key input mode, and normal raw input mode, data is available when the window terminal driver receives at least one character. (See the description of `s$$read_raw` in Chapter 7 for more information.)

When wait mode is used with either simple sequential I/O or formatted I/O mode, a call to `s$$seq_read` returns as soon as a complete record is available. (See the description of `s$$seq_read` in Chapter 7 for more information.)

When wait mode is used with `s$$seq_write`, `s$$seq_write_partial`, or `s$$write_raw`, the write call does not return until all output characters have been transferred to system output buffers. If there are not enough output buffers currently available, the write call waits as long as necessary for the required number of buffers. (Note that `s$$seq_write` and `s$$seq_write_partial` will also wait if the display of output is paused.) You can use `s$$set_io_time_limit` to define the amount of time that an I/O subroutine waits before returning control to the application program.

PL/I Usage

```
declare port_id                fixed bin (15);
declare error_code             fixed bin (15);

declare s$set_wait_mode entry(  fixed bin (15),
                                fixed bin (15));

                                call s$set_wait_mode(  port_id,
                                                        error_code);
```

C Usage

```
short int                      port_id;
short int                      error_code;

void s$set_wait_mode();

                                s$set_wait_mode(      &port_id,
                                                        &error_code);
```

Arguments

The s\$set\_wait\_mode subroutine takes two arguments, which are listed and described in the following table.

Argument	Description
port_id (input)	The identifier of the port attached to the device.
error_code (output)	A returned error code.

Error Codes

The s\$set\_wait\_mode subroutine may return the following error codes.

Error Code	Description
e\$invalid_io_operation (1040)	Your application program attempted to perform an I/O operation that it is not authorized or correctly initialized to perform.
e\$port_not_attached (1021)	The specified port is not attached. (It may never have been attached or it may have been detached.) This error code may also indicate a problem on the network.

## s\$set\_no\_wait\_mode

Use this subroutine to switch the port to no-wait mode. No-wait mode enables your application program to explicitly control waiting. With no-wait mode, the operating system does not wait, nor does it force your application program to wait for the operation to complete. No-wait mode allows the appropriate subroutine call to return to your application program immediately, whether the call processes data or not. This enables your application program to decide when it should wait (for example, it may decide to wait on an event using `s$wait_event`).

When no-wait mode is used with any of the raw input modes, a call to `s$read_raw` returns data as soon as it is available. Just as with wait mode, the input buffer need not be full. Note that in raw record mode, data is available for retrieval with `s$read_raw` when a record is terminated. (In raw record mode, a record is terminated when the window terminal driver detects the appropriate interrupt character.) In raw table mode, data is available when the window terminal driver detects the appropriate interrupt character. In generic input mode, translated input mode, function-key input mode, and normal raw input mode, data is available when the window terminal driver receives at least one character. If there is no data available, `s$read_raw` returns the error code `e$caller_must_wait (1277)`.

When no-wait mode is used with either simple sequential I/O or formatted I/O mode, a call to `s$seq_read` returns a complete record as soon as the record is terminated. If there is no record available, `s$seq_read` returns the error code `e$caller_must_wait (1277)`.

When no-wait mode is used with `s$seq_write`, `s$seq_write_partial`, or `s$write_raw`, and the output buffers cannot accommodate all of the output, the write call sends as many bytes as possible and returns the number of bytes actually sent in the `buffer_length` argument. The write call also returns the error code `e$caller_must_wait (1277)`. To send the remaining output, your application program must reissue the call. The write call also returns `e$caller_must_wait (1277)` when the display of output is paused.

Each port associated with the terminal or connection is also associated with a system event. Whenever the window terminal driver makes input available to your application program or frees an output buffer, it notifies the per port event. A call to `s$set_no_wait_mode` returns the ID of the event in the `event_id` argument.

If you want your application program to explicitly wait for input characters or output buffers, design it to call `s$read_event` and `s$wait_event` using the event ID. (Your application program can also call `s$read_device_event`, which requires a port ID instead of an event ID.)

As an alternative to no-wait mode, you can design your application program to use `s$set_io_time_limit` or to handle multitasking (that is, design the application program to perform multiple tasks).

As a general rule, if you design your application program to use no-wait mode, you should also design it to handle any partial data transfers and then loop until all input or output is transferred. (This applies to read operations performed with `s$read_raw` and write operations performed with `s$seq_write`, `s$seq_write_partial`, or `s$write_raw`.) After switching a port to no-wait mode, your application program should perform the following steps to ensure that all data transfers are completed.

1. Read the per port event to initialize your copy of the event count. Your application program can use `s$read_event` (which requires an event ID) or `s$read_device_event` (which requires a port ID).
2. Read input or write output.
3. Test for an error code. If the error code is 0, process data and return to Step 2. If the error code is `e$caller_must_wait(1277)`, proceed to Step 4. For any other error code, exit your application program with the error.
4. Assume that a partial input record was returned or that partial output bytes were transferred. If a partial input record was received, perform the following two steps.
  - a. Increment the number of bytes actually returned by the value of the `record_length` output argument.
  - b. Advance the starting point within the buffer (the `record_buffer` argument) to the position immediately following the last byte of the bytes just received. Also, decrease the value of the `buffer_length` argument by the number of bytes just received to reflect the number of bytes available in the buffer.

If partial output bytes were transferred, advance the starting point within the buffer so that the `record_buffer` argument reflects the correct position in the output string (that is, use the value returned in `buffer_length`, which indicates the number of bytes actually sent to the device, to determine the position at which output should resume). In addition, reset the value of the `buffer_length` argument to reflect the number of remaining bytes to be sent to the device.

5. Use `s$wait_event` to wait for more input bytes or output buffers.
6. Repeat Steps 2 through 5.

As described earlier, the subroutine `s$set_no_wait_mode` returns the event ID of the per port event. Your application program uses this event ID in calls to



`s$read_event` and `s$wait_event`. Since the per port event always exists, it is not necessary to attach the event before reading and waiting on the event.

If your application program uses no-wait mode to read input, it should not use the `s$read` subroutine. The `s$read` subroutine does not return error-code information to the application program; it simply signals the system error condition. (See the VOS Subroutines manuals for information about system error conditions.)

If your application program uses no-wait mode to write output, it should only use `s$write_raw`, `s$seq_write`, or `s$seq_write_partial`. Of the nine operating system subroutines that write output, only these three have the `buffer_length` parameter. The other subroutines do not return the number of characters actually transferred and therefore should not be used in no-wait mode.

PL/I Usage

```
declare port_id                fixed bin (15);
declare event_id              fixed bin (31);
declare error_code            fixed bin (15);

declare s$set_no_wait_mode entry(  fixed bin (15),
                                   fixed bin (31),
                                   fixed bin (15));

      call s$set_no_wait_mode(  port_id,
                                event_id,
                                error_code);
```

C Usage

```
short int    port_id;
long int     event_id;
short int    error_code;

void s$set_no_wait_mode();

      s$set_no_wait_mode(    &port_id,
                              &event_id,
                              &error_code);
```

Arguments

The s\$set\_no\_wait\_mode subroutine takes three arguments, which are listed and described in the following table.

Argument	Description
port_id (input)	The identifier of the port attached to the device.
event_id (output)	The identifier of the event. The operating system notifies the event when the I/O operation on the port has finished executing. Your application program can wait for the event to be notified.
error_code (output)	A returned error code.

## Error Codes

The `s$set_no_wait_mode` subroutine may return the error codes listed in the following table.

Error Code	Description
<code>e\$invalid_io_operation</code> (1040)	Your application program attempted to perform an I/O operation that it is not authorized or properly initialized to perform.
<code>e\$port_not_attached</code> (1021)	The specified port is not attached. (It may never have been attached or it may have been detached.) This error code may also indicate a problem on the network.

## **s\$read\_device\_event**

Use this subroutine to retrieve the event ID and event count associated with a port. If you want your application program to wait on this event, you must design it to use `s$read_device_event` to determine the current event count and then supply this event count to the `s$wait_event` subroutine.

The event count indicates the number of times that the event was notified before the call. (The event is notified each time the window terminal driver completes an I/O operation while something is waiting for completion.) When your application program supplies an event count to `s$wait_event` that is based on the event count returned by `s$read_device_event`, `s$wait_event` returns to your application program when the new event count exceeds the supplied event count. (For more information on `s$wait_event`, see the VOS Subroutines manuals.)

If a port is in no-wait mode and a subroutine that starts an I/O operation returns to the application program before the operation is completed, the subroutine returns the error code `e$caller_must_wait (1277)`. Your application program can then decide whether or not to wait on the event.

As an alternative to `s$read_device_event`, your application program can use the `s$read_event` subroutine. (See the VOS Subroutines manuals for more information on the `s$read_event` subroutine.)

## PL/I Usage

```

declare port_id                fixed bin (15);
declare event_id               fixed bin (31);
declare event_count            fixed bin (31);
declare error_code             fixed bin (15);

declare s$read_device_event entry( fixed bin (15),
                                   fixed bin (31),
                                   fixed bin (31),
                                   fixed bin (15));

      call s$read_device_event(      port_id,
                                   event_id,
                                   event_count,
                                   error_code);

```

## C Usage

```

short int      port_id;
long int       event_id;
long int       event_count;
short int      error_code;

void s$read_device_event();

      s$read_device_event(      &port_id,
                                   &event_id,
                                   &event_count,
                                   &error_code);

```

## Arguments

The `s$read_device_event` subroutine takes four arguments, which are listed and described in the following table.

(Page 1 of 2)

Argument	Description
<code>port_id</code> (input)	The identifier of the port attached to the device.
<code>event_id</code> (output)	The identifier of the event. The operating system notifies the event when the I/O operation on the port has finished executing. Your application program can wait for the event to be notified.
<code>event_count</code> (output)	The current event count.

(Page 2 of 2)

Argument	Description
error_code (output)	A returned error code.

**Error Codes**

The `s$read_device_event` subroutine may return the following error code.

Error Code	Description
e\$port_not_attached (1021)	The specified port is not attached. (It may never have been attached or it may have been detached.) This error code may also indicate a problem on the network.

## s\$control

Use this subroutine to set the control parameters of the terminal attached to the specified port (or the network connection) and to return information about the current settings of the control parameters.

The window terminal driver supports the following types of `s$control` operations (opcodes).

- **Global control opcodes.** These opcodes affect both terminal and nonterminal devices (including virtual devices), and perform standard device operations.
- **Opcodes affecting the RS-232-C communications medium.** These opcodes apply only to the asynchronous access layer (`async_al`). If you are using another access layer (for example, `telnet_al`), these opcodes are not relevant. You should refer to the appropriate system administrator's manual for information about the opcodes that apply to a particular access layer.
- **Opcodes affecting all communications media.** These opcodes generally apply to all access layers.
- **Opcodes affecting the terminal.** These opcodes allow your application program to perform such operations as setting terminal-screen preferences and selecting a terminal type.
- **Opcodes affecting the primary window and subwindow.** These opcodes allow your application program to control certain aspects of the primary window as well as the creation, deletion, and arrangement of subwindows.

**Note:** The window terminal driver generally supports the complete set of opcodes associated with the old asynchronous driver; however, there are differences in the handling of certain functions. See Appendix A for more information about the handling of standard asynchronous opcodes. For complete descriptions of the standard asynchronous opcodes, see the manual *VOS Communications Software: Asynchronous Communications* (R025).

Table 8-1, which appears later in this subroutine description, lists all relevant opcodes that are generally available to your application program. A description of each of the opcodes immediately follows [Table 8-1](#).

As shown in Table 8-1, the names of the window terminal opcodes begin with the prefix `TERM_`; these opcodes perform either get or set operations. (The names of the standard asynchronous opcodes do not have a specific prefix.) A *get operation* allows your application program to retrieve information and verify current settings. A *set operation* allows your application program to change a current setting or a group of settings.

If you design your application program to perform old asynchronous driver operations, make sure that it does not perform a window terminal set operation between a pair of standard asynchronous get and set operations. Mixing the two types of opcodes may destroy correct settings and put other (incorrect) settings into effect. The same rule applies when your application program performs window terminal operations; make sure that the program does not perform a standard asynchronous set operation (or any other set operation) between a pair of window terminal get and set operations. In general, you should design your application program to perform a pair of get and set operations together, and to check the current settings using a get operation before and after performing a set operation.

The window terminal opcodes that are available to application programs are listed in the system include files `window_control_opcodes.incl.pl1` and `window_control_opcodes.incl.c`. (The standard asynchronous opcodes are listed in the system include files `standard_term_opcodes.incl.pl1` and `standard_term_opcodes.incl.c`.) You can define the appropriate include file in your application program instead of designing the program to include a definition for each relevant opcode.

For most of the `s$control` operations, the terminal port or virtual-device port must be open before your application program can make the call. [Table 8-1](#) identifies the subset of `s$control` operations that your application program can perform when the port is attached but not yet open.

All of the `s$control` opcodes require the `control_info` argument. The `control_info` argument may be a structure or a simple variable, depending on the opcode specified in the `opcode` argument. Each opcode description identifies the structures and variables required by `control_info` in both PL/I and C. The system include files `window_term_info.incl.pl1` and `window_term_info.incl.c` contain all of the constants and structure definitions that your application program needs to perform `s$control` operations.

For all opcodes requiring a **structure** for the `control_info` argument, the first element of the structure is always the `version` field. This field must contain the appropriate structure version number.

Most of the structures supported by the window terminal driver require version number 1. The names of these structures contain the suffix `V1`. For structures that require version number 2, the structure name contains the suffix `V2`. For example, the



structure `SUBWINDOW_INFO_V2` requires the value 2 in its `version` field. (The structure `SUBWINDOW_INFO_V2` is associated with the opcodes `TERM_CREATE_SUBWIN` (2038), `TERM_GET_SUBWIN_BOUNDS` (2066), and `TERM_SET_SUBWIN_BOUNDS` (2067).) Note that the opcode `TERM_STATUS_MSG_CHANGE` (2096) is associated with two entirely different structures and therefore requires version number 1 for the first structure and version number 2 for the second structure.

For opcodes that do not have a `control_info` argument (indicated by `control_info: 0` in the opcode descriptions), `control_info` should be defined in PL/I as a 2-byte variable (a fixed bin (15) with the value 0) or defined in C as a pointer to a 2-byte variable (a short int with the value 0).

Note that in the opcode descriptions, where `control_info` is a `char_varying` string, the C equivalent is defined in the include file `window_term_info.incl.c` as follows:

```
#define STRING(N)      char_varying(N)
```

To accommodate the Stratus XA/R reduced instruction-set computer (RISC) hardware environment, all PL/I and C structures associated with the window terminal opcodes contain either `longmap` or `shortmap` directives. Note that where `control_info` is a `longmap` or `shortmap` structure, the C equivalent of the directive for the structure is defined in the include file `window_term_info.incl.c` as follows:

```
#define SHORTMAP      $shortmap
#define LONGMAP       $longmap
```

Most of the structures associated with the window terminal opcodes are `longmap` structures. The only `shortmap` structures are associated with the `s$control` opcodes `TERM_GET_PWIN_DEFAULTS` (2034), `TERM_SET_PWIN_DEFAULTS` (2035), `TERM_GET_PWIN_PARAMS` (2082), and `TERM_SET_PWIN_PARAMS` (2083). If you specify these opcodes in your application program, their structures must contain `shortmap` directives. The opcodes `TERM_GET_PWIN_DEFAULTS` (2034) and `TERM_SET_PWIN_DEFAULTS` (2035) are described later in this chapter in the section “Opcodes Affecting the Terminal.” The opcodes `TERM_GET_PWIN_PARAMS` (2082) and `TERM_SET_PWIN_PARAMS` (2083) are described later in this chapter in the section “Opcodes Affecting the Primary Window and Subwindow.”

When declaring the structure or variable for `control_info`, remember that you must allocate it on an even-byte boundary. Refer to the appropriate language manual for storage-allocation rules.

PL/I Usage

```
declare port_id           fixed bin (15);
declare opcode            fixed bin (15);
declare control_info      fixed bin (15);
declare error_code        fixed bin (15);

declare s$control entry(   fixed bin (15),
                           fixed bin (15),
                           fixed bin (15),
                           fixed bin (15));

call s$control(            port_id,
                           opcode,
                           control_info,
                           error_code);
```

C Usage

```
short int port_id;
short int opcode;
short int control_info;
short int error_code;

void s$control();

    s$control(    &port_id,
                  &opcode,
                  &control_info,
                  &error_code);
```

Arguments

The s\$control subroutine takes four arguments, which are listed and described in the following table.

(Page 1 of 2)

Argument	Description
port_id (input)	The identifier of the port attached to the device.
opcode (input)	The operation performed on the device. Opcodes 1 through 4 are global opcodes that apply to asynchronous I/O. Opcodes 2001 through 2200 apply to devices using the window terminal driver. (Opcodes 200 through 299 apply to devices using the old asynchronous driver.) See <a href="#">Table 8-1</a> for a list of the opcodes.

(Page 2 of 2)

Argument	Description
control_info (input/output)	The structure or variable associated with each opcode. (See the individual opcode descriptions in this chapter.)
error_code (output)	A returned error code. See “Error Codes” later in this subroutine description for a list of the error codes returned by s\$control.

Table 8-1 lists the window terminal opcodes, their numbers, and whether they are valid before the port is open.

**Table 8-1. Window Terminal Opcodes** (Page 1 of 5)

Control Opcode	Number	Valid before Port Is Open
<b>Global Control Opcodes</b>		
GET_LINE_LENGTH	1	No
RUNOUT	2	No
ABORT	3	No
SET_SPOOLER	4	Yes
<b>Opcodes Affecting the RS-232-C Communications Medium</b>		
TERM_GET_BAUD_RATE	2003	Yes
TERM_SET_BAUD_RATE	2004	Yes
TERM_GET_BITS_PER_CHAR	2005	Yes
TERM_SET_BITS_PER_CHAR	2006	Yes
TERM_GET_INPUT_FLOW_INFO	2007	Yes
TERM_SET_INPUT_FLOW_INFO	2008	Yes
TERM_GET_OPERATING_VALUES	2013	Yes
TERM_SET_OPERATING_VALUES	2014	Yes
TERM_GET_OUTPUT_FLOW_INFO	2015	Yes
TERM_SET_OUTPUT_FLOW_INFO	2016	Yes
TERM_GET_PARITY	2017	Yes

**Table 8-1. Window Terminal Opcodes** (Page 2 of 5)

<b>Control Opcode</b>	<b>Number</b>	<b>Valid before Port Is Open</b>
TERM_SET_PARITY	2018	Yes
TERM_GET_STOP_BITS	2019	Yes
TERM_SET_STOP_BITS	2020	Yes
TERM_LISTEN	2023	No
TERM_SET_FORWARDING_TIMER	2095	No
<b><i>Opcodes Affecting All Communications Media</i></b>		
TERM_HANGUP	2001	Yes
TERM_GET_CONNECTION_STATE	2002	Yes
TERM_GET_MAX_BUFFER_SIZE	2009	No
TERM_SET_MAX_BUFFER_SIZE	2010	No
TERM_GET_OPEN_ACTION	2011	Yes
TERM_SET_OPEN_ACTION	2012	Yes
TERM_HOLD_CONNECTION	2021	No
<b><i>Opcodes Affecting the Terminal</i></b>		
TERM_GET_SCREEN_HEIGHT	2024	No
TERM_GET_SCREEN_PREF	2026	No
TERM_SET_SCREEN_PREF	2027	No
TERM_GET_SCREEN_WIDTH	2028	No
TERM_GET_TERMINAL_SETUP	2030	No
TERM_SET_TERMINAL_SETUP	2031	No
TERM_GET_TERMINAL_TYPE	2032	No
TERM_SET_TERMINAL_TYPE	2033	No
TERM_GET_PWIN_DEFAULTS	2034	Yes
TERM_SET_PWIN_DEFAULTS	2035	Yes
TERM_SWITCH_TO_WIN_MGR	2036	No
TERM_SEND_BREAK	2106	No

**Table 8-1. Window Terminal Opcodes** (Page 3 of 5)

<b>Control Opcode</b>	<b>Number</b>	<b>Valid before Port Is Open</b>
<b><i>Opcodes Affecting the Primary Window and Subwindow</i></b>		
TERM_SET_SUBWIN_TO_TOP	2025	No
TERM_SET_SUBWIN_TO_BOTTOM	2029	No
TERM_CREATE_SUBWIN	2038	No
TERM_DELETE_SUBWIN	2039	No
TERM_DISCARD_INPUT	2040	No
TERM_DISCARD_OUTPUT	2041	No
TERM_ENABLE_ECHO	2042	No
TERM_DISABLE_ECHO	2043	No
TERM_ENABLE_KEY	2044	No
TERM_DISABLE_KEY	2045	No
TERM_ENABLE_FMT_IO_MODE	2046	No
TERM_DISABLE_FMT_IO_MODE	2047	No
TERM_GET_BREAK_ACTION	2050	No
TERM_SET_BREAK_ACTION	2051	No
TERM_GET_CONTINUE_CHARS	2052	No
TERM_SET_CONTINUE_CHARS	2053	No
TERM_GET_CURSOR_FORMAT	2054	No
TERM_SET_CURSOR_FORMAT	2055	No
TERM_GET_INPUT_MODE	2056	No
TERM_SET_INPUT_MODE	2057	No
TERM_GET_INPUT_SECTION	2058	No
TERM_SET_INPUT_SECTION	2059	No
TERM_GET_INTERRUPT_TABLE	2060	No
TERM_SET_INTERRUPT_TABLE	2061	No
TERM_GET_KEY_BIT_MASK	2062	No

**Table 8-1. Window Terminal Opcodes** (Page 4 of 5)

<b>Control Opcode</b>	<b>Number</b>	<b>Valid before Port Is Open</b>
TERM_SET_KEY_BIT_MASK	2063	No
TERM_GET_UNDISP_MODE	2064	No
TERM_SET_UNDISP_MODE	2065	No
TERM_GET_SUBWIN_BOUNDS	2066	No
TERM_SET_SUBWIN_BOUNDS	2067	No
TERM_GET_PAUSE_CHARS	2068	No
TERM_SET_PAUSE_CHARS	2069	No
TERM_GET_PAUSE_LINES	2070	No
TERM_SET_PAUSE_LINES	2071	No
TERM_GET_PROMPT_CHARS	2072	No
TERM_SET_PROMPT_CHARS	2073	No
TERM_GET_TABS	2074	No
TERM_SET_TABS	2075	No
TERM_GET_TYPEAHEAD_LINES	2076	No
TERM_SET_TYPEAHEAD_LINES	2077	No
TERM_GET_PWIN_BOUNDS	2078	Yes
TERM_GET_PWIN_BOUND_LIMS	2080	Yes
TERM_GET_PWIN_PARAMS	2082	No
TERM_SET_PWIN_PARAMS	2083	No
TERM_GET_PWIN_TITLE	2084	Yes
TERM_SET_PWIN_TITLE	2085	Yes
TERM_MOVE_PWIN_TO_BOTTOM	2086	No
TERM_MOVE_PWIN_TO_TOP	2087	No
TERM_RESET_COMMAND_MODES	2088	No
TERM_RESET_OUTPUT	2089	No
TERM_SAVE_COMMAND_INPUT	2090	No

**Table 8-1. Window Terminal Opcodes** (Page 5 of 5)

<b>Control Opcode</b>	<b>Number</b>	<b>Valid before Port Is Open</b>
TERM_SET_CURRENT_SUBWIN	2091	No
TERM_SET_PWIN_TO_DEFAULTS	2092	No
TERM_STATUS_MSG_CHANGE	2096	Yes
TERM_ENABLE_IMMED_PAINT	2097	No
TERM_DISABLE_IMMED_PAINT	2098	No
TERM_WRITE_SYSTEM_MESSAGE	2101	No
TERM_KNOCK_DOWN_FORM	2102	No
TERM_WRITE_SYMSG_NOBEEP	2103	No
TERM_KNOCK_DOWN_FORM_OK	2104	No
TERM_GET_KNOCKDOWN_ID	2105	No
TERM_OPEN_EXISTING_WINDOW_OPCODE	2107	Yes
TERM_ENABLE_CACHED_IO_OPCODE	2108	No
TERM_DISABLE_CACHED_IO_OPCODE	2109	No
TERM_POSIX_OPEN_OPCODE	2110	No
TERM_POSIX_CLOSE_OPCODE	2111	No
TERM_GET_SUBWIN_BORDER_OPCODE	2112	No
TERM_SET_SUBWIN_BORDER_OPCODE	2113	No
TERM_GET_SESSION_ID_OPCODE	2114	No
TERM_POSIX_GETATTR_OPCODE	2115	No
TERM_POSIX_SETATTR_OPCODE	2116	No

## Global Control Opcodes

This section describes the global control opcodes that apply to window terminal connections. The opcode descriptions are presented in alphabetical order for easy reference.

### **ABORT (3)**

```
control_info: 0
```

This global opcode aborts the current I/O operation. If your application program is using no-wait mode, this opcode terminates any incomplete I/O operation so that another operation can be initiated.

Your application program must issue this opcode to terminate incomplete operations associated with the following subroutines.

```
s$close  
s$control  
s$open
```

Although it is beneficial, your application program is not required to issue this opcode to terminate incomplete operations associated with the following subroutines.

```
s$read_form  
s$read_raw  
s$seq_read  
s$seq_write  
s$seq_write_partial  
s$write_raw
```

### **GET\_LINE\_LENGTH (1)**

```
control_info: fixed bin (15) or short int
```

This global opcode returns the length of a line on the terminal screen. The line length is initially derived from information in the TTP when the terminal port is opened. Specifically, the line length is initially derived from the `width` variable defined in the configuration section of the terminal's TTP. The configuration section can define more than one `width` variable if it includes multiple configuration *setups*, where each setup defines a group of variable settings such as screen width, screen height (size), and display pages to handle a specific type of terminal-screen configuration.

If the configuration section of the TTP includes more than one configuration setup, you can change the line length in your application program by specifying a different configuration setup with the opcode `TERM_SET_TERMINAL_SETUP (2031)`. You can also verify the current line length with the opcode `TERM_GET_SCREEN_WIDTH`.



(2028). (These two opcodes are described later in this chapter in the section “Opcodes Affecting the Terminal.”)

#### NOTE

Unlike the line-length value returned by the old asynchronous driver, the line-length value returned by the window terminal driver matches the `width` variable defined in the TTP. The line-length value returned by the old asynchronous driver is one less than the `width` variable defined in the TTP.

#### RUNOUT (2)

```
control_info: 0
```

If you are using the asynchronous access layer (`async_al`), this global opcode ensures that the window terminal driver has forwarded all pending output to the RS-232-C communications hardware for transmission to the terminal. To ensure that the communications hardware has enough time to send the output to the terminal, you should design your application program to wait a specified period of time before closing or hanging up the line (terminating the connection). In general, you should design your application program to issue this opcode before attempting to reconfigure the RS-232-C connection (for example, before changing the baud rate).

#### NOTE

This opcode works with the asynchronous access layer, but may not work with other access layers, such as the OS TELNET access layer `telnet_al`.

The subroutine `s$sleep` allows you to set a time interval during which your application program suspends all processing. See the VOS Subroutines manuals for more information about the `s$sleep` subroutine.

In wait mode, `s$control` does not return until all pending output is sent to the communications hardware for transmission to the terminal. (Your application program can call `s$set_io_time_limit` to specify the period of time that `s$control` will wait.) In no-wait mode, `s$control` returns the error code `e$caller_must_wait` (1277) until all pending output has been sent to the communications hardware for transmission to the terminal.

If the port is attached with the `hold_attached` and `hold_open` switches turned **off**, your application program can use the opcode `RUNOUT (2)` to refresh the contents of all formatted sequential I/O subwindows. This is useful if your application program is not designed to perform a sequential I/O operation immediately after it performs either

a subwindow operation (for example, creating, moving, or deleting a subwindow) or a primary-window operation (for example, moving a primary window to the top of the primary-window stack). If your application program does not use `RUNOUT` after performing one of these operations, the subwindows are not refreshed until your application program performs a sequential I/O operation.

#### **SET\_SPOOLER (4)**

```
control_info: 0
```

This global opcode allows your application program to specify that a connection is to be used by the VOS spooler. It enables the operating system to return information about the spooler devices (printers). Your application program can issue this opcode before the port is open.

For information about the VOS spooler, see the manual *VOS System Administration: Administering the Spooler Facility* (R286).

## **Opcodes Affecting the RS-232-C Communications Medium**

This section describes the opcodes that your application program uses to control various characteristics of the RS-232-C connection to the asynchronous device. The asynchronous access layer (`async_al`) of the window terminal driver provides the interface to the RS-232-C communications medium, as described in Chapter 1, “Overview of Window Terminal Communications.”

If you are using another access layer, such as the OS TELNET access layer (`telnet_al`), these `s$control` opcodes are generally not relevant. Access layers often have a unique set of requirements and may have different ways of handling these `s$control` opcodes.

For example, the OS TELNET access layer does not support opcodes affecting asynchronous operating values (baud rate, bits per character, parity, and stop bits), input and output flow control, and the forwarding timer. If the OS TELNET access layer does not support a particular get operation (such as `TERM_GET_BAUD_RATE(2003)`), `s$control` returns an error code of 0 with the appropriate default values (such as 9,600 for the default baud rate). If the OS TELNET access layer does not support a particular set operation, `s$control` simply does not perform the operation. For information about how the OS TELNET access layer provides the interface to the Ethernet communications medium and functions as part of the OS TCP/IP network, refer to the OS TCP/IP manual set, paying particular attention to the *VOS OS TCP/IP Administrator's Guide* (R223).

The opcodes that generally apply to all access layers are described in the next section, “Opcodes Affecting All Communications Media.”

This section presents the opcode descriptions in alphabetical order; the get and set opcode pairs appear together.

As indicated in [Table 8-1](#), your application program can issue many of the opcodes in this group when the port is attached but not yet open. The following opcodes are the exceptions; your application program cannot issue them before the port is open.

```
TERM_LISTEN (2023)
TERM_SET_FORWARDING_TIMER (2095)
```

```
TERM_GET_BAUD_RATE (2003)
TERM_SET_BAUD_RATE (2004)
```

```
control_info: fixed bin (15) or short int
```

The opcode `TERM_GET_BAUD_RATE (2003)` returns the current baud rate; the opcode `TERM_SET_BAUD_RATE (2004)` allows your application program to change the baud rate.

In addition, the opcode `TERM_GET_OPERATING_VALUES (2013)` returns the current baud rate; the opcode `TERM_SET_OPERATING_VALUES (2014)` allows your application program to change the baud rate. These opcodes are useful for retrieving and/or changing more than one operating value (for example, the bits-per-character, parity, and stop-bits values). These opcodes are described later in this section.

The following list identifies the valid baud rates and their corresponding values.

(Page 1 of 2)

Baud Rate	Value
BAUD_50	1
BAUD_75	2
BAUD_110	3
BAUD_134	4
BAUD_150	5
BAUD_300	6
BAUD_600	7
BAUD_1200	8
BAUD_1800	9
BAUD_2400	10

(Page 2 of 2)

BAUD_3600	11
BAUD_4800	12
BAUD_7200	13
BAUD_9600	14
BAUD_19200	15
AUTO_BAUD	32

**TERM\_GET\_BITS\_PER\_CHAR (2005)****TERM\_SET\_BITS\_PER\_CHAR (2006)**`control_info: fixed bin (15) or short int`

The opcode `TERM_GET_BITS_PER_CHAR (2005)` returns the number of bits per character; the opcode `TERM_SET_BITS_PER_CHAR (2006)` allows your application program to change the number of bits per character.

In addition, the opcode `TERM_GET_OPERATING_VALUES (2013)` returns the current bits-per-character value; the opcode `TERM_SET_OPERATING_VALUES (2014)` allows your application program to change the bits-per-character value. These opcodes are useful for retrieving and/or changing more than one operating value (for example, the baud-rate, parity, and stop-bits values). These opcodes are described later in this section.

There can be either 7 bits per character or 8 bits per character, represented by the following values.

Bits per Character	Value
SEVEN_BITS_PER_CHARACTER	7
EIGHT_BITS_PER_CHARACTER	8

The handling of 8-bit data transmission depends on the type of hardware used to make the connection to the device. (The parity options for 8-bit data transmission also depend on the type of hardware used to make the connection.) For more information about how the hardware handles 8-bit data transmission and parity options, see the description of the opcode `TERM_GET_PARITY (2017)` later in this section and see the section “Parity Configuration” in Chapter 1.

**TERM\_GET\_INPUT\_FLOW\_INFO (2007)**

**TERM\_SET\_INPUT\_FLOW\_INFO (2008)**

`control_info: FLOW_CONTROL_INFO_V1 structure`

The opcode `TERM_GET_INPUT_FLOW_INFO (2007)` returns input flow-control information; the opcode `TERM_SET_INPUT_FLOW_INFO (2008)` allows your application program to set the type of flow control (none, XON/XOFF, or DSL) and the input flow-control characters. Note that input flow control is available when the device is using the window terminal driver and is connected to the module using a K-series I/O adapter. (See Chapter 1 for a description of K-series hardware.)

Input flow control is the process of temporarily stopping and restarting the flow of input from the device to the module. Input flow control is used when a device such as a personal computer is sending the module more data than it can handle (for example, during a file transfer from the personal computer's asynchronous communications port). When this happens, the I/O adapter sends an XOFF signal to the device to tell it to temporarily stop sending input. When the input buffers are free, the adapter sends the XON signal to the device to tell it to resume the flow of input.

The following characteristics apply to input flow control.

- With XON/XOFF flow control, the special characters XOFF and XON temporarily stop and resume the flow of input. Receipt of the XOFF character (usually ASCII DC3, which is often `[Ctrl]-[S]`) stops the flow of input. The flow of input resumes upon receipt of the XON character (usually ASCII DC1, which is often `[Ctrl]-[Q]`). XON/XOFF is the most common form of flow control and is the default.
- With data set lead (DSL) flow control, the window terminal driver automatically reconfigures the channel or subchannel associated with the device for a force-listen configuration, where the loss of the data set ready (DSR) signal (the device drops the DTR signal) does not indicate a line hang-up. Instead, the window terminal driver interprets the loss of the DSR signal as XOFF and the return of the DSR signal as XON. Note that DSL flow control can only be used with devices that are directly connected to the module.

You can design your application program to use input flow control in conjunction with output flow control. This combination of input and output flow control is called *bidirectional flow control*. If you use bidirectional flow control, you must keep in mind the following considerations.

- When input flow control takes effect (the device receives the XOFF character sent by the module to stop the flow of input to the module), the device must stop sending input immediately. An input buffer size of approximately 30 characters is available, which gives the device enough time to recognize the XOFF character and stop sending data. A device can only resume sending data when it receives the XON character. If the device continues to send data after the module has sent the XOFF signal, data will be lost because the input buffer size has been exceeded. The

window terminal driver returns a buffer-overflow error to the application program on the module. This applies to both input flow control and bidirectional flow control.

- When output flow control takes effect (the device sends an XOFF character to the module to stop the flow of output from the module to the device), the device must continue to process incoming data. This is required to enable the module to send XON/XOFF characters to the device even though the flow of output has stopped. The module sends the XON/XOFF characters for input flow control regardless of the status of output flow control.
- When necessary, the device must send XON/XOFF characters to the module in order to control the flow of output from the module, even after the module sends the XOFF character to enable input flow control and stop the flow of input to the module. Since the XON/XOFF characters sent to the module do not use buffer space, they do not cause a buffer-overflow error and the module continues to recognize them.
- The flow-control characters must be the same for both input flow control and output flow control. This means that the flow-control characters specified with the opcode `TERM_SET_INPUT_FLOW_INFO (2008)` must be the same as those specified with the opcode `TERM_SET_OUTPUT_FLOW_INFO (2016)`. For more information about output flow control, see the description of the opcode `TERM_SET_OUTPUT_FLOW_INFO (2016)` later in this section.
- You cannot mix DSL input flow control with XON/XOFF output flow control, or XON/XOFF input flow control with DSL output flow control.

To retrieve or specify input flow-control information, use the following structure in your application program.

The asynchronous and Console Controller access layers for the window terminal driver recognize a new `-rts_cts_flow_control` option. This option is specified in the device table "parameters" string. The `-rts_cts_flow_control` option overrides the XON/XOFF or DSL flow control mechanisms specified with the `s$control` calls. The RTS/CTS flow control is for use with high-speed modems that require flow control to do reliable binary transfers. This option requires that an RTS/CTS cable be used to connect the modem.

## PL/I Usage

```
declare 1 FLOW_CONTROL_INFO_V1 based longmap,
        2 version                fixed bin (15), /* 1 */
        2 flow_type              fixed bin (15),
        2 flow_off_char          char (1),
        2 flow_on_char           char (1);
```

## C Usage

```
typedef struct LONGMAP flow_control_info_v1
{
    short int          version; /* 1 */
    short int          flow_type;
    char              flow_off_char;
    char              flow_on_char;
} FLOW_CONTROL_INFO_V1;
```

The `flow_type` field can contain one of the following values.

Flow-Control Type	Value
NO_FLOW_CONTROL	0
XON_XOFF	1
DSL	2

**TERM\_GET\_OPERATING\_VALUES (2013)**

**TERM\_SET\_OPERATING\_VALUES (2014)**

`control_info: OPERATING_VALUES_V1 structure`

The opcode `TERM_GET_OPERATING_VALUES (2013)` returns four operating values of the device: baud rate, bits per character, parity, and number of stop bits. The opcode `TERM_SET_OPERATING_VALUES (2014)` allows your application program to change these operating values. For a list of the possible values, see the descriptions of the individual opcodes (for example, the opcodes `TERM_SET_BAUD_RATE (2004)` and `TERM_SET_PARITY (2018)` in this section).

### NOTE

These opcodes affect only a copy of the device entry in the device configuration table (`devices.table`). The

operating values of the device remain in effect until the next bootstrap.

To retrieve or change the operating values of the device, use the following structure in your application program.

## PL/I Usage

```
declare 1 OPERATING_VALUES_V1 based longmap,
        2 version          fixed bin (15), /* 1 */
        2 baud             fixed bin (15),
        2 bits_per_char    fixed bin (15),
        2 parity           fixed bin (15),
        2 stop_bits        fixed bin (15);
```

## C Usage

```
typedef struct LONGMAP operating_values_v1
{
    short int    version; /* 1 */
    short int    baud;
    short int    bits_per_char;
    short int    parity;
    short int    stop_bits;
} OPERATING_VALUES_V1;
```

**TERM\_GET\_OUTPUT\_FLOW\_INFO (2015)**

**TERM\_SET\_OUTPUT\_FLOW\_INFO (2016)**

control\_info: FLOW\_CONTROL\_INFO\_V1 structure

The opcode **TERM\_GET\_OUTPUT\_FLOW\_INFO (2015)** returns output flow-control information; the opcode **TERM\_SET\_OUTPUT\_FLOW\_INFO (2016)** allows your application program to set a type of flow control (none, XON/XOFF, or DSL) and the output flow-control characters.

Output flow control is the process of temporarily stopping and restarting the flow of output sent from your application program on the module to the device. Output flow control is supported on K-Series I/O adapters; it is also supported by both the window terminal driver and the old asynchronous driver.

Output flow control is used when the module is sending more data than the attached device (for example, a V103 terminal) can handle. When this happens, the device needs to control the flow of output from the module.



**NOTE**

If you are using output flow control with input flow control (known as **bidirectional** flow control), you must keep in mind the considerations presented in the description of the opcode `TERM_SET_INPUT_FLOW_INFO` (2008), which appears earlier in this section.

The following characteristics apply to output flow control.

- With XON/XOFF flow control, the special characters XOFF and XON temporarily stop and resume the flow of output. Receipt of the XOFF character (usually ASCII DC3, which is often `Ctrl-S`) stops the flow of output. The flow of output resumes upon receipt of the XON character (usually ASCII DC1, which is often `Ctrl-Q`). XON/XOFF is the most common form of flow control and is the default. When the device needs to control the flow of output from the module, it sends the XOFF signal to the module to tell it to stop sending data until it receives the XON signal. This is what happens when the terminal user presses `Ctrl-S` (XOFF) and then presses `Ctrl-Q` (XON).
- With data set lead (DSL) flow control, the window terminal driver automatically reconfigures the channel or subchannel associated with the device for a force-listen configuration, where the loss of the data set ready (DSR) signal (the device drops the DTR signal) does not indicate a line hang-up. Instead, the window terminal driver interprets the loss of the DSR signal as XOFF and the return of the DSR signal as XON. Note that DSL flow control can only be used with devices that are directly connected to the module.

To retrieve or specify output flow-control information, use the following structure in your application program.

The asynchronous and Console Controller access layers for the window terminal driver recognize a new `-rts_cts_flow_control` option. This option is specified in the device table “parameters” string. The `-rts_cts_flow_control` option overrides the XON/XOFF or DSL flow control mechanisms specified with the `s$control` calls. The RTS/CTS flow control is for use with high-speed modems that require flow control to do reliable binary transfers. This option requires that an RTS/CTS cable be used to connect the modem.

PL/I Usage

```
declare 1 FLOW_CONTROL_INFO_V1 based longmap,
        2 version                fixed bin (15), /* 1 */
        2 flow_type              fixed bin (15),
        2 flow_off_char          char (1),
        2 flow_on_char           char (1);
```

C Usage

```
typedef struct LONGMAP flow_control_info_v1
{
    short int          version; /* 1 */
    short int          flow_type;
    char               flow_off_char;
    char               flow_on_char;
} FLOW_CONTROL_INFO_V1;
```

The flow\_type field can contain the following values.

Flow-Control Type	Value
NO_FLOW_CONTROL	0
XON_XOFF	1
DSL	2

**TERM\_GET\_PARITY (2017)**  
**TERM\_SET\_PARITY (2018)**

control\_info: fixed bin (15) or short int

The opcode TERM\_GET\_PARITY (2017) returns the current parity value; the opcode TERM\_SET\_PARITY (2018) allows your application program to change the parity value.

In addition, the opcode TERM\_GET\_OPERATING\_VALUES (2013) returns the current parity value; the opcode TERM\_SET\_OPERATING\_VALUES (2014) allows your application program to change the parity value. These opcodes are useful for retrieving and/or changing more than one operating value (for example, the baud-rate, bits-per-character, and stop-bits values). These opcodes are described earlier in this section.

The following list identifies the six types of parity available and their corresponding values.

Parity	Value
EVEN_PARITY	0
ODD_PARITY	1
NO_PARITY	2
MARK_PARITY	3
SPACE_PARITY	4
BAUDOT	5

#### NOTE \_\_\_\_\_

You can configure the line to use odd, even, or no parity with 8 bits per character only if the terminal is connected to K-series hardware. (For a description of parity, see the section “Parity Configuration” in Chapter 1.)

**TERM\_GET\_STOP\_BITS (2019)**

**TERM\_SET\_STOP\_BITS (2020)**

`control_info: fixed bin (15) or short int`

The opcode `TERM_GET_STOP_BITS (2019)` returns the number of stop bits; the opcode `TERM_SET_STOP_BITS (2020)` allows your application program to change the number of stop bits. (For a description of stop bits, see the section “Stop-Bits Configuration” in Chapter 1.)

In addition, the opcode `TERM_GET_OPERATING_VALUES (2013)` returns the current stop-bits value; the opcode `TERM_SET_OPERATING_VALUES (2014)` allows your application program to change the stop-bits value. These opcodes are useful for retrieving and/or changing more than one operating value (for example, the baud-rate, bits-per-character, and parity values). These opcodes are described earlier in this section.

There can be either one stop bit or two stop bits, represented by the following values.

Stop Bits	Value
ONE_STOP_BIT	1
TWO_STOP_BITS	2

### **TERM\_LISTEN (2023)**

```
control_info: 0
```

This opcode causes the operating system to listen to the RS-232-C channel or subchannel associated with the device and then enable or reenable the connection to the device. This opcode can reenable a slave RS-232-C channel or subchannel that has been disconnected. The operating system handles login RS-232-C channels and subchannels differently.

Once a slave RS-232-C channel or subchannel has been disconnected, I/O cannot resume over the channel or subchannel until the port is closed and then reopened, or until your application program issues the opcode `TERM_LISTEN (2023)` for the channel or subchannel.

Before issuing the opcode `TERM_LISTEN (2023)`, consider the following points.

- Issuing this opcode on a dial-up line discards all pending input and output. Therefore, do not issue this opcode multiple times between line hang-ups or between a call to `s$open` and a hang-up on an open dial-up line.
- Issuing this opcode destroys the K-series interrupt table previously set with the opcode `TERM_SET_INTERRUPT_TABLE (2061)`. Each time your application program issues the `TERM_LISTEN` opcode or performs a reset, it must reissue `TERM_SET_INTERRUPT_TABLE (2061)` to re-create the desired interrupt table.

To reestablish a connection after a line hang-up, follow these steps.

1. Call `s$control` with the opcode `TERM_HANGUP (2001)`.
2. Read the channel event.
3. Call `s$control` with the opcode `TERM_LISTEN (2023)`.
4. Perform the following additional steps in a loop until the connection is established.
  - a. Issue the opcode `TERM_GET_CONNECTION_STATE (2002)` to determine the status of the connection.
  - b. If the connection state is not `TERM_CONNECTION_ESTABLISHED`, perform Step c.

- c. Call `s$wait_event` to wait for the connection state to change.

For descriptions of the `s$control` opcodes `TERM_HANGUP` (2001) and `TERM_GET_CONNECTION_STATE` (2002), see the next section, “Opcodes Affecting All Communications Media.”

#### **TERM\_SET\_FORWARDING\_TIMER (2095)**

`control_info`: fixed bin (15) or short int

This opcode sets the minimum time between interrupts (in milliseconds). By default, the minimum time between interrupts is 30 milliseconds. The RS-232-C communications hardware uses this timer for all of the raw input modes.

## **Opcodes Affecting All Communications Media**

This section describes opcodes that generally apply to all access layers. It presents the opcode descriptions in alphabetical order; the get and set opcode pairs appear together.

As indicated in [Table 8-1](#), your application program can issue some of the opcodes in this group when the port is attached but not yet open. The following opcodes are the exceptions; your application program cannot issue them before the port is open.

`TERM_GET_MAX_BUFFER_SIZE` (2009)  
`TERM_SET_MAX_BUFFER_SIZE` (2010)  
`TERM_HOLD_CONNECTION` (2021)

#### **TERM\_GET\_CONNECTION\_STATE (2002)**

`control_info`: fixed bin (15) or short int

This opcode returns the status of the connection to the communications medium. For a list of the steps involved in establishing and terminating/reestablishing RS-232-C dial-up connections, see the section “Modem Connections” in Chapter 1.

[Table 8-2](#) lists the seven connection states and their values.

**Table 8-2. Connection States** (Page 1 of 2)

Connection State	Value	Description
<code>TERM_NO_CONNECTION</code>	1	No connection exists.
<code>TERM_CONNECTION_ALLOWED</code>	2	The module will accept a connection.
<code>TERM_CONNECTION_REQUESTED</code>	3	The module is attempting to establish a connection.
<code>TERM_CONNECTION_PENDING</code>	4	(Reserved for future use)

Table 8-2. Connection States (Page 2 of 2)

Connection State	Value	Description
TERM_CONNECTION_ESTABLISHED	5	In general, a connection exists. For RS-232-C connections, the DSR and DCD signals are asserted, which usually indicates that the connection exists. However, this connection state may also indicate that the host modem has not dropped the DSR signal from a previous RS-232-C dial-up connection.
TERM_CONNECTION_BREAKING	6	The window terminal driver is handling a request to terminate the connection.
TERM_FAILED_CONNECTION	7	The module was unable to initiate a connection. This connection state applies only to access layers that can initiate connections (such as OS TELNET).

**TERM\_GET\_MAX\_BUFFER\_SIZE (2009)**  
**TERM\_SET\_MAX\_BUFFER\_SIZE (2010)**

control\_info: fixed bin (15) or short int

The opcode `TERM_GET_MAX_BUFFER_SIZE (2009)` returns the maximum input buffer size; the opcode `TERM_SET_MAX_BUFFER_SIZE (2010)` allows your application program to change the maximum input buffer size. The value supplied with this opcode determines the size of the internal input buffers allocated to hold input from the terminal (or network connection). This opcode discards any pending input that has not yet been read by your application program.

The value of the `control_info` argument must be a 2-byte integer specifying the maximum buffer size. The value must be in the range 0 to 2048. The value 0 tells the window terminal driver to use the internal default of 300 bytes.

The maximum input buffer size specified corresponds to the `maximum_length` parameter of `s$open`. (See the description of `s$open` in Chapter 6 for information on the `maximum_length` field.)

**TERM\_GET\_OPEN\_ACTION (2011)**  
**TERM\_SET\_OPEN\_ACTION (2012)**

control\_info: fixed bin (15) or short int

The opcode `TERM_GET_OPEN_ACTION (2011)` returns information about the action (the connection action) that is performed when the terminal port or virtual-device port

is opened; the opcode `TERM_SET_OPEN_ACTION` (2012) allows your application program to change the connection action that is performed when the terminal port or virtual-device port is opened.

There are three connection actions, represented by the following values.

Connection Action	Value	Description
<code>ALLOW_CONN_ACTION</code>	0	Allow the connection (the default).
<code>ALLOW_AND_WAIT_ACTION</code>	1	Allow the connection and wait for a response.
<code>NO_ALLOW_ACTION</code>	2	Do not allow the connection.

### **TERM\_HANGUP (2001)**

```
control_info: 0
```

This opcode terminates the connection to the device. For network connections, this means that a new connection must be established. For RS-232-C connections, this means that the module will drop the data terminal ready (DTR) signal. When the module drops the DTR signal, the window terminal driver discards any pending input and output.

This opcode is useful if there is an unrecoverable error. For information on reenabling an RS-232-C line after it has been disconnected, see the description of the opcode `TERM_LISTEN` (2023) earlier in this chapter in the section “Opcodes Affecting the RS-232-C Communications Medium.”

### **TERM\_HOLD\_CONNECTION (2021)**

```
control_info: 0
```

This opcode prevents the connection to the terminal or virtual device from being terminated when the last port opened for the terminal or virtual device is closed. For RS-232-C connections, the channel or subchannel associated with the line remains dialed up, and the line can transmit data as soon as that channel or subchannel is reopened. For example, the command `logout -hold` causes the `HOLD_CONNECTION` operation to be executed for an RS-232-C channel or subchannel.

When the command `logout -hold` is executed from a login terminal over an RS-232-C dial-up line, the operating system performs the following steps.

1. It terminates the login process.

2. It determines that the RS-232-C line is already connected, which means that the data set ready (DSR) signal is asserted.
3. It creates a new prelogin process, which opens the RS-232-C channel or subchannel.
4. It displays the login banner on the terminal screen.

## Opcodes Affecting the Terminal

This section describes the opcodes that allow your application program to control various aspects of the terminal's operation, such as the default parameter settings, screen height (size), screen width (line length), screen preferences, and break transmission.

As indicated in Table 8-1, your application program cannot issue most of the opcodes in this group before the port is open. The opcodes `TERM_GET_PWIN_DEFAULTS` (2034) and `TERM_SET_PWIN_DEFAULTS` (2035) are the exceptions; your application program can issue them before the port is open.

**`TERM_GET_PWIN_DEFAULTS` (2034)**

**`TERM_SET_PWIN_DEFAULTS` (2035)**

`control_info`: `PRIMARY_WINDOW_PARAMS_V1` structure

The opcode `TERM_GET_PWIN_DEFAULTS` (2034) returns the default parameters of the primary window; the opcode `TERM_SET_PWIN_DEFAULTS` (2035) allows your application program to change the default parameters of the primary window. Each newly created primary window inherits these parameters (for example, the prompt message, cursor format, and tabs).

To retrieve or change the primary-window parameters, use the structure `PRIMARY_WINDOW_PARAMS_V1` in your application program. When using this structure, note that the `pwin_param_flags` field is currently ignored and is reserved for future use.

### NOTE \_\_\_\_\_

The `PRIMARY_WINDOW_PARAMS_V1` structure contains a shortmap directive; therefore, you should use the include file `window_term_info.incl.pl1` or `window_term_info.incl.c` to define the structure in your application program. If you do not use the include file and instead insert the actual structure in your application program, make sure that the structure contains the shortmap directive and that all declarations correspond



with those defined in the include file for the current VOS release.

For more information on setting specific primary-window parameters, see the descriptions of the appropriate opcodes (for example, the opcode `TERM_SET_CONTINUE_CHARS (2053)`) later in this chapter.

## PL/I Usage

```
declare 1 PRIMARY_WINDOW_PARAMS_V1 based shortmap,
        2 version                      fixed bin (15), /* 1 */
        2 pwin_param_flags             fixed bin (31),
        2 pause_lines                  fixed bin (15),
        2 cursor_format                 fixed bin (15),
        2 prompt_chars                  char (32) var,
        2 continue_chars                char (32) var,
        2 pause_chars                   char (32) var,
        2 undisplayable_notation_mode  fixed bin (15),
        2 max_typeahead                 fixed bin (15),
        2 num_tab_stops                 fixed bin (15),
        2 tabs (25)                    fixed bin (15);
```

## C Usage

```
typedef struct SHORTMAP primary_window_params_v1
{
    short int      version; /* 1 */
    long int       pwin_param_flags;
    short int      pause_lines;
    short int      cursor_format;
    STRING(32)     prompt_chars;
    STRING(32)     continue_chars;
    STRING(32)     pause_chars;
    short int      undisplayable_notation_mode;
    short int      max_typeahead;
    short int      num_tab_stops;
    short int      tabs [25];
} PRIMARY_WINDOW_PARAMS_V1;
```

### **TERM\_GET\_SCREEN\_HEIGHT (2024)**

`control_info: fixed bin (15) or short int`

This opcode returns the maximum number of lines that can appear on the terminal screen at a time (the screen height, also referred to as the screen size). The number of lines is always within the range specified in the configuration section of the terminal's TTP.

The screen height is initially derived from the `height` variable defined in the configuration section of the terminal's TTP when the terminal port is opened. The configuration section can define more than one `height` variable if it includes multiple configuration setups, where each setup defines a group of variable settings such as screen width, screen height, and display pages to handle a specific type of terminal-screen configuration.

If the configuration section of the TTP includes more than one configuration setup, you can change the screen height in your application program by specifying a different configuration setup with the opcode `TERM_SET_TERMINAL_SETUP (2031)`. This opcode and the opcode `TERM_GET_TERMINAL_SETUP (2030)` are described later in this section.

#### NOTE \_\_\_\_\_

Unlike the screen-height value returned by the old asynchronous driver, the screen-height value returned by the window terminal driver matches the `height` variable defined in the TTP. The screen-height value returned by the old asynchronous driver is one less than the `height` variable defined in the TTP.

Note that the screen-height value does not include the line commonly used for status messages.

**TERM\_GET\_SCREEN\_PREF (2026)**

**TERM\_SET\_SCREEN\_PREF (2027)**

`control_info: SCREEN_PREF_V1` structure

The opcode `TERM_GET_SCREEN_PREF (2026)` returns the current screen characteristics; the opcode `TERM_SET_SCREEN_PREF (2027)` allows your application program to change the screen characteristics.

The following screen options are available.

- Audible key click or no audible key click (the default).
- Smooth scrolling or jump scrolling (the default).
- Black-on-white or white-on-black (the default).
- Initial overlay mode or insert mode (the default).
- `CANCEL` request does not cancel a pause or cancels a pause (the default).
- At command level, `RETURN` request does not release a pause or releases a pause (the default).

- In a display form, RETURN request does not tab horizontally or tabs horizontally (the default).
- No break to window manager mode or break to window manager mode (the default).

The default settings for the screen options involving the CANCEL request and the RETURN request allow your application program to provide the various pausing characteristics of both the old asynchronous driver and the window terminal driver. (See the section “Pause Processing” in Chapter 2 for a description of the window terminal driver’s pausing characteristics; see the section “The Function of Simple Sequential I/O” in Appendix A for a description of the old asynchronous driver’s pausing characteristics.)

With the default settings, the CANCEL request performs the same function as the ABORT\_OUTPUT request when there is no current input record at command level, the RETURN request performs the same function as the NEXT\_SCREEN request when there is no current input record at command level, and the RETURN request tabs horizontally to the next field in a display form. If you do not want the CANCEL and RETURN requests to function in this manner, your application program can set the appropriate screen options (CANCEL\_DOESNT\_ABORT, RETURN\_DOESNT\_UNPAUSE, and RETURN\_DOESNT\_TAB) with the opcode TERM\_SET\_SCREEN\_PREF (2027). (The terminal user can also set these screen options with the set\_terminal\_parameters command.) When these screen options are set, both the CANCEL and RETURN requests display a command-level message that explains to the terminal user how to properly release the pause. In a display form, the RETURN request moves the cursor to the first field on the next line instead of performing a tab. Regardless of how your application program controls these settings, the CANCEL request always cancels the current input record (if one exists) and the RETURN request always enters the current input record (if one exists).

Window manager mode allows the terminal user and the application program to manipulate the primary windows. (See the *Window Terminal User’s Guide* (R256) for information about this mode.) When handling unprocessed raw I/O, your application program should make sure that the window terminal driver cannot intercept a break request that activates window manager mode. If the break action specified with the s\$control opcode TERM\_SET\_BREAK\_ACTION (2051) enables the processing of BREAK\_CHAR generic input requests, your application program should use the opcode TERM\_SET\_SCREEN\_PREF (2027) to disable the break to window manager mode.

To retrieve or change the screen characteristics, use the following structure in your application program.

PL/I Usage

```
declare 1 SCREEN_PREF_V1 based longmap,
        2 version          fixed bin (15), /* 1 */
        2 terminal_state    fixed bin (15);
```

C Usage

```
typedef struct LONGMAP screen_pref_v1
{
    short int          version; /* 1 */
    short int          terminal_state;
} SCREEN_PREF_V1;
```

The `terminal_state` field can contain one of the following values or the sum of the following values. For example, if your application program uses `BLACK_ON_WHITE` and `INITIAL_OVERLAY_ON`, the field should contain the value 12.

Terminal State	Value
KEY_CLICK	1
SMOOTH_SCROLL	2
BLACK_ON_WHITE	4
INITIAL_OVERLAY_ON	8
BREAK_TO_WMGR	16
CANCEL_DOESNT_ABORT	32
RETURN_DOESNT_UNPAUSE	64
RETURN_DOESNT_TAB	128

**TERM\_GET\_SCREEN\_WIDTH (2028)**

control\_info: fixed bin (15)Or short int

This opcode returns the width of the screen. The screen width (line length) must be within the range specified in the configuration section of the terminal's TTP. You can also use the global control opcode `GET_LINE_LENGTH (1)` to determine the current screen width.

The screen width is initially derived from the `width` variable defined in the configuration section of the terminal's TTP when the terminal port is opened. The

configuration section of the TTP can define more than one `width` variable if it includes multiple configuration setups, where each setup defines a group of variable settings such as screen width, screen height (size), and display pages to handle a specific type of terminal-screen configuration.

If the configuration section of the TTP includes more than one configuration setup, you can change the screen width in your application program by specifying a different configuration setup with the opcode `TERM_SET_TERMINAL_SETUP (2031)`. This opcode is described next in this section.

#### NOTE

Unlike the line-length value returned by the old asynchronous driver, the line-length value returned by the window terminal driver matches the `width` variable defined in the TTP. The line-length value returned by the old asynchronous driver is one less than the `width` variable defined in the TTP.

**TERM\_GET\_TERMINAL\_SETUP (2030)**

**TERM\_SET\_TERMINAL\_SETUP (2031)**

`control_info: char (32) varying or STRING (32)`

The opcode `TERM_GET_TERMINAL_SETUP (2030)` returns the name of the current configuration setup; the opcode `TERM_SET_TERMINAL_SETUP (2031)` allows your application program to change the current configuration setup.

A *configuration setup* defines a specific set of terminal variables in the configuration section of the TTP. The configuration section can include multiple configuration setups, where each setup defines a group of variable settings such as screen height (size), screen width, and display pages to handle a specific type of terminal-screen configuration.

The configuration setup name that you specify can have a maximum length of 32. The default character set for the configuration setup is Latin alphabet No. 1, and the default shift mode is single shift.

**TERM\_GET\_TERMINAL\_TYPE (2032)**

**TERM\_SET\_TERMINAL\_TYPE (2033)**

`control_info: char (32) varying or STRING (32)`

The opcode `TERM_GET_TERMINAL_TYPE (2032)` returns the name of the current terminal type; the opcode `TERM_SET_TERMINAL_TYPE (2033)` allows your application program to change the terminal type. The terminal-type name specified by

your application program can have a maximum length of 32. The terminal type must be installed on the system.

**TERM\_SEND\_BREAK (2106)**

`control_info: fixed bin(31) or long int`

This opcode allows your application program to generate a break condition and transmit a break over the line to the terminal. It accommodates devices that must receive a break in order to initiate certain operations. Note that this opcode applies to the asynchronous access layer (`async_al`) but may not apply to other access layers, such as OS TELNET (`telnet_al`).

The `control_info` argument must specify the duration of the break in 100ths of a second (centiseconds). Specifying the value 0 for this argument generates a break of .25 seconds (25 centiseconds).

**TERM\_SWITCH\_TO\_WIN\_MGR (2036)**

`control_info: 0`

This opcode turns on window manager mode. See the *Window Terminal User's Guide* (R256) for a description of window manager mode.

## Opcodes Affecting the Primary Window and Subwindow

This section describes the opcodes that allow your application program to retrieve information about a primary window or subwindow and to control various aspects of a primary window or subwindow. For example, these opcodes can control primary-window parameters such as cursor format and tab stops, and can create, delete, and arrange subwindows.

As indicated in [Table 8-1](#), your application program cannot issue most of the opcodes in this group before the port is open. The following opcodes are the exceptions; your application program can issue them before the port is open.

```
TERM_GET_PWIN_BOUNDS (2078)
TERM_GET_PWIN_BOUND_LIMS (2080)
TERM_GET_PWIN_TITLE (2084)
TERM_SET_PWIN_TITLE (2085)
TERM_STATUS_MSG_CHANGE (2096)
```

This section presents the opcode descriptions in alphabetical order; the get and set opcode pairs appear together.

## Duration-Switch Settings and the Subwindow Opcodes

If the port is attached with either the `hold_attached` or `hold_open` duration switch turned on, opcodes that affect the arrangement of a subwindow (for example, opcodes that can cause a subwindow to be overlapped) can affect the information returned by the TTP subroutine `s$ttp_get_supported_output_seqs`. (This TTP subroutine tells your application program which generic output requests are supported in the output database of the TTP used by the current subwindow.)

For example, if the current subwindow becomes overlapped, output requests such as `DELETE_LINES` can no longer be performed. Therefore, if your application program calls the TTP subroutine and then changes the arrangement of any subwindow by issuing one of the following `s$control` opcodes, it must call the TTP subroutine again to determine how the change affects the status of the output requests. Your application program should call the TTP subroutine **each** time it changes the arrangement of any subwindow using one of these opcodes.

```
TERM_CREATE_SUBWIN (2038)
TERM_DELETE_SUBWIN (2039)
TERM_SET_CURRENT_SUBWIN (2091)
TERM_SET_SUBWIN_BOUNDS (2067)
TERM_SET_SUBWIN_TO_BOTTOM (2029)
TERM_SET_SUBWIN_TO_TOP (2025)
```

### **TERM\_CREATE\_SUBWIN (2038)**

`control_info`: SUBWINDOW\_INFO\_V2 structure

This opcode creates a formatted sequential subwindow. The new subwindow moves on top of the other subwindows and becomes the current I/O subwindow. (The current I/O subwindow is the subwindow that handles all I/O.)

When your application program creates a subwindow using this opcode, the new subwindow is always in formatted I/O mode and can never be modified to use simple sequential I/O. Simple sequential I/O can be used only in the original subwindow, and this opcode has no effect on the original subwindow.

Your application program must issue this opcode in order to use formatted I/O mode **exclusively** within a subwindow. To use formatted I/O mode temporarily in the original subwindow, your application program must issue the `s$control` opcode

```
TERM_ENABLE_FMT_IO_MODE (2046).
```

Formatted I/O mode affects all sequential reads and writes to the current subwindow, which makes all I/O **formatted** sequential I/O. (See Chapter 4 for a description of formatted sequential I/O, which is part of the application program-managed I/O approach.)

The opcode `TERM_CREATE_SUBWIN` (2038) returns the subwindow ID of the new subwindow. Since the ID of the original subwindow is always 0, the new subwindow ID is always a value other than 0.

#### NOTE

---

This opcode does **not** update the display of the primary window with the new, cleared subwindow area. If the port is attached with the `hold_attached` and `hold_open` switches turned off (the default), your application program can clear the new subwindow area by performing any sequential I/O operation or by issuing the `s$control` opcode `RUNOUT` (2). (The opcode `RUNOUT` (2) is described earlier in this chapter in the section “Global Control Opcodes.”) When your application program updates the display in this manner, the window terminal driver positions the cursor at line 0, column 0 and initializes the scrolling region to include all of the lines in the subwindow. However, if the port is attached with either the `hold_attached` or `hold_open` switch turned on, you must design your application program to clear the initial subwindow area by calling `s$seq_write_partial` with the `CLEAR_SCROLLING_REGION` output request. In addition, your application program must maintain the contents of the subwindow and refresh the display of the subwindow when necessary. (See Chapter 7 for a description of `s$seq_write_partial`; see Chapter 5 for a description of the `CLEAR_SCROLLING_REGION` output request.)

To specify the subwindow values, use the following structure in your application program. When using this structure, note that the version number must be 2, and the horizontal and vertical offsets (represented by `upper_left_corner_h` and `upper_left_corner_v`, respectively) are zero based. Note also that the `reserved` field is currently ignored.



## PL/I Usage

```
declare 1 SUBWINDOW_INFO_V2 based longmap,
        2 version                fixed bin (15), /* 2 */
        2 subwin_id              fixed bin (15), /* output */
        2 upper_left_corner_h    fixed bin (15),
        2 upper_left_corner_v    fixed bin (15),
        2 width                  fixed bin (15),
        2 height                  fixed bin (15),
        2 reserved                fixed bin (15);
```

## C Usage

```
typedef struct longmap subwindow_info_v2
{
    short int    version; /* 2 */
    short int    subwin_id; /* output */
    short int    upper_left_corner_h;
    short int    upper_left_corner_v;
    short int    width;
    short int    height;
    short int    reserved;
} SUBWINDOW_INFO_V2;
```

### TERM\_DELETE\_SUBWIN (2039)

control\_info: fixed bin (15) or short int

This opcode deletes the specified subwindow. Your application program must specify the subwindow ID to identify the subwindow to delete.

The original subwindow in a primary window cannot be deleted. If your application program tries to delete the original subwindow, s\$control returns the error code e\$form\_cant\_delete\_orig\_subwin (3956).

If your application program deletes the current subwindow, the subwindow below it becomes the current subwindow and the window terminal driver determines the data that will be displayed.

### NOTE

This opcode does **not** refresh the display of the subwindow. (For more information about making changes to the display of the subwindow, see the description of the opcode TERM\_CREATE\_SUBWIN (2038).)

```
TERM_DISABLE_CACHED_IO_OPCODE (2109)
TERM_ENABLE_CACHED_IO_OPCODE (2108)
```

```
control_info: 0
```

These two opcodes explicitly disable and enable cached formatted I/O for the current subwindow. When cached formatted I/O is disabled, the subwindow remains in formatted I/O mode.

To turn off cached formatted I/O, your application program should use the opcode `TERM_DISABLE_CACHED_IO_OPCODE (2109)`. This opcode disables all cached formatted I/O until it is explicitly reenabled. To reenable cached formatted I/O, your application program should use the opcode `TERM_ENABLE_CACHED_IO_OPCODE (2108)`.

By default, cached formatted I/O is enabled if the `hold_open` and `hold_attached` options were off when the port was attached.

```
TERM_DISABLE_ECHO (2043)
TERM_ENABLE_ECHO (2042)
```

```
control_info: 0
```

With either simple or formatted sequential I/O, these opcodes control the automatic echoing of input characters to the subwindow display. By default, character echoing is enabled automatically for simple sequential I/O and is enabled for formatted sequential I/O during a read operation.

To turn off automatic character echoing, your application program should use the opcode `TERM_DISABLE_ECHO (2043)`. This opcode disables echoing for all subsequent read operations in **all** subwindows in the primary window until echoing is reenabled. To reenable automatic character echoing, your application program uses the opcode `TERM_ENABLE_ECHO (2042)`.

Disabling automatic character echoing is useful for application programs that require the terminal user to enter a password.

```
TERM_DISABLE_FMT_IO_MODE (2047)
TERM_ENABLE_FMT_IO_MODE (2046)
```

```
control_info: 0
```

These opcodes control the processing of sequential I/O in the original subwindow. The original subwindow always has a subwindow ID of 0.

When the port is opened, the original subwindow uses simple sequential I/O. The opcode `TERM_ENABLE_FMT_IO_MODE (2046)` allows your application program to switch the original subwindow to formatted I/O mode. In formatted I/O mode, all

sequential I/O in the subwindow is **formatted** sequential I/O until your application program issues the opcode `TERM_DISABLE_FMT_IO_MODE` (2047) or performs a reset. Your application program can perform a reset by using the opcode `TERM_RESET_OUTPUT` (2089) or `TERM_RESET_COMMAND_MODES` (2088).

Your application program should issue the opcode `TERM_ENABLE_FMT_IO_MODE` (2046) to use formatted I/O mode **temporarily** in the original subwindow. To use formatted I/O mode **only**, your application program should create a new subwindow using the opcode `TERM_CREATE_SUBWIN` (2038). Note that when your application program switches the original subwindow to formatted I/O mode, the original subwindow must be the **current** subwindow (that is, the subwindow that handles all I/O).

When the original subwindow switches to formatted I/O mode, the display of the subwindow changes depending on how the terminal's port is attached. If the port is attached with the `hold_open` and `hold_attached` switches turned off (see the description of `s$attach_port` in Chapter 6), the window terminal driver clears the entire subwindow, discards all typeahead, and moves the cursor to the top left corner of the subwindow. If the port is attached with either of these switches turned on, the window terminal driver does **not** clear the entire subwindow, but discards all typeahead. In addition, the cursor remains at the same position.

To return the subwindow to simple sequential I/O mode, your application program should issue the opcode `TERM_DISABLE_FMT_IO_MODE` (2047). When the subwindow returns to simple sequential I/O mode, the cursor moves to the bottom left corner of the subwindow and the window terminal driver tries to preserve the formatted I/O data in the subwindow.

If, at any time, the formatted I/O data in the subwindow becomes overlapped (by another subwindow or primary window), the status of the port determines how the window terminal driver handles the overlapped data. If the port is attached with the `hold_open` and `hold_attached` switches turned **off**, the window terminal driver will redisplay the data when it is no longer overlapped. However, if either of these switches is turned **on**, the window terminal driver cannot guarantee subsequent redisplay of the data. In addition, if your application program returns to simple sequential I/O mode and then changes the size of the subwindow, the formatted I/O data disappears completely.

Note that the opcode `TERM_DISABLE_FMT_IO_MODE` (2047) has no effect on a subwindow created with the opcode `TERM_CREATE_SUBWIN` (2038).

For more information about simple sequential I/O, see Chapter 2; for more information about formatted sequential I/O, see Chapter 4.

**TERM\_DISABLE\_IMMED\_PAINT (2098)****TERM\_ENABLE\_IMMED\_PAINT (2097)**`control_info: 0`

For formatted sequential I/O subwindows, these opcodes control the way in which the window terminal driver updates the subwindow display when handling generic output sequences. (Generic output sequences are multibyte sequences that contain generic output requests.) These opcodes have no effect on simple sequential I/O subwindows.

By default, the window terminal driver processes the entire output string internally before updating the subwindow display. The opcode **TERM\_ENABLE\_IMMED\_PAINT (2097)** causes the window terminal driver to update the display after it processes **each** generic output sequence. (See Chapter 5 for more information about specifying a generic output request in a generic output sequence.)

To disable this option, your application program must issue the opcode **TERM\_DISABLE\_IMMED\_PAINT (2098)**. Your application program can also issue the opcode **TERM\_RESET\_OUTPUT (2089)** or **TERM\_RESET\_COMMAND\_MODES (2088)** to return the window terminal driver to its default state.

**TERM\_DISABLE\_KEY (2045)****TERM\_ENABLE\_KEY (2044)**`control_info: fixed bin (15) or short int`

These opcodes control the processing of a single keyboard request (generic input request). For example, your application program can use this opcode to enable or disable a request such as `TAB`. Your application program uses the `control_info` argument to specify the generic input request to be enabled or disabled. (See Chapter 5 for a list of the relevant generic input requests and their respective codes.)

**TERM\_DISCARD\_INPUT (2040)**`control_info: 0`

This opcode deletes all characters that the terminal user has typed but your application program has not yet read. These advance input characters are called *typeahead*. This opcode allows your application program to delete typeahead in response to a specific error or change in the environment. For example, the operating system issues this opcode when there is an RS-232-C line-break signal.

**TERM\_DISCARD\_OUTPUT (2041)**`control_info: 0`

This opcode empties the output buffers and deletes any pending output that has not yet been sent to the terminal by the communications hardware on the module. This

opcode is useful when you need to free the output buffers as part of a general cleanup after an unrecoverable error.

#### NOTE

You should use this opcode **only** if your application program is performing **unprocessed raw I/O** (for example, your application program is using the terminal solely as a simple communications device with only one primary window). See Chapter 3 for a discussion of unprocessed raw I/O.

**TERM\_GET\_BREAK\_ACTION (2050)**

**TERM\_SET\_BREAK\_ACTION (2051)**

control\_info: fixed bin (15) or short int

The opcode **TERM\_GET\_BREAK\_ACTION (2050)** returns information on how the window terminal driver processes **BREAK\_CHAR** input requests and RS-232-C line breaks; the opcode **TERM\_SET\_BREAK\_ACTION (2051)** allows your application program to select a break action that determines how the window terminal driver processes **BREAK\_CHAR** input requests and line breaks.

The following list identifies the available break actions and their corresponding values.

Break Action	Value	Description
<b>SIGNAL_AND_DISCARD</b>	0	Notifies the application program of the break condition and discards any pending input or output
<b>SIGNAL_ONLY</b>	1	Notifies the application program of the break condition
<b>RETURN_NUL_CHAR</b>	2	Returns the NUL character
<b>RETURN_ERROR</b>	3	Returns an error to the application program
<b>IGNORE</b>	4	Ignores the break condition

If you select the **RETURN\_ERROR** break action, a **BREAK\_CHAR** input request or a line break does not immediately generate the break condition in the current process. Instead, sequential and raw read operations return the error code **e\$break\_signalled (1086)** to your application program. This is intercepted by the kernel and causes break to get signaled when the terminal port is accessed for reading. In other words, the error code is never returned to the application program. This break

action is useful in a multitasking application to make sure the break gets signaled in the task that is reading from the terminal.

#### NOTE

The old asynchronous driver does not offer as many break actions as the window terminal driver. The old asynchronous driver uses two mode bits (`break_char` and `break_enabled`) to control the processing of line breaks and `BREAK_CHAR` input requests (that is, these mode bits determine whether line breaks and `BREAK_CHAR` input requests generate the break condition in the current process). For more information about the break actions offered by the old asynchronous driver, see the manual *VOS Communications Software: Asynchronous Communications* (R025).

**TERM\_GET\_CONTINUE\_CHARS (2052)**

**TERM\_SET\_CONTINUE\_CHARS (2053)**

`control_info: char (32) varying or STRING (32)`

The opcode `TERM_GET_CONTINUE_CHARS (2052)` returns the current continue message; the opcode `TERM_SET_CONTINUE_CHARS (2053)` allows your application program to change the continue message.

The *continue message* is the character string that is always displayed at the beginning of a wrapped line. The default continue message is the plus sign (+). The string can contain up to 32 characters.

**TERM\_GET\_CURSOR\_FORMAT (2054)**

**TERM\_SET\_CURSOR\_FORMAT (2055)**

`control_info: fixed bin (15) or short int`

The opcode `TERM_GET_CURSOR_FORMAT (2054)` returns the current cursor format; the opcode `TERM_SET_CURSOR_FORMAT (2055)` allows your application program to change the cursor format.

Your application program can specify the following cursor formats and values.

Cursor Format	Value	Description
CURSOR_OFF	0	The cursor is invisible.
BLINKING_BLOCK	1	The cursor is a blinking block.
STEADY_BLOCK	2	The cursor is a steady block.
BLINKING_UNDERLINE	3	The cursor is a blinking underline.
STEADY_UNDERLINE	4	The cursor is a steady underline.

**TERM\_GET\_INPUT\_MODE (2056)**

**TERM\_SET\_INPUT\_MODE (2057)**

`control_info: fixed bin (15) or short int`

The opcode `TERM_GET_INPUT_MODE (2056)` returns the current input mode; the opcode `TERM_SET_INPUT_MODE (2057)` allows your application program to set or change the input mode.

[Table 8-3](#) identifies the various input modes and their corresponding values.

**Table 8-3. Input Modes and Values** (Page 1 of 2)

Input Mode	Value	Description
GENERIC_INPUT_MODE	0	The window terminal driver maps input to its respective generic input code. If your application program uses processed raw input, the window terminal driver returns this code in a binary data introducer (BDI) sequence. (See Chapter 4 for a description of BDI sequences.)
FUNCTION_KEY_INPUT_MODE	1	The window terminal driver maps input to its respective function-key code and returns the code in a BDI sequence.
TRANSLATED_INPUT_MODE	2	The window terminal driver maps input to its respective internal-character code.
RAW_INPUT_MODE	3	The window terminal driver passes input to your application program unprocessed.
RAW_TABLE_MODE	4	The window terminal driver uses an interrupt table that specifies which characters generate interrupts and how.

Table 8-3. Input Modes and Values (Page 2 of 2)

Input Mode	Value	Description
RAW_RECORD_MODE	5	The window terminal driver uses an interrupt table in which each interrupt character also functions as a record terminator.

NOTE \_\_\_\_\_

If your application program selects function-key input mode or generic input mode by specifying the standard asynchronous opcode `SET_MODES (207)` or `SET_INFO (202)` instead of the window terminal opcode `TERM_SET_INPUT_MODE (2057)`, `s$read_raw` will return the processed input in the style used by the old asynchronous driver, not the style used by the window terminal driver. For a description of the style used by the old asynchronous driver for returning processed input, see the section “The Function of Processed Raw Input” in Appendix A.

If your application program uses unprocessed raw I/O (see Chapter 3), it should specify a raw input mode before performing a read operation. If it does not specify a raw input mode, the window terminal driver uses generic input mode. Generic input mode is an acceptable input mode for application programs using processed raw input (see Chapter 4) or simple sequential I/O (see Chapter 2). If your application program is using simple sequential I/O, it need not specify generic input mode before performing a read operation. Application programs using processed raw input can also use function-key input mode or translated input mode (see Chapter 4).

`TERM_GET_INPUT_SECTION (2058)`  
`TERM_SET_INPUT_SECTION (2059)`

`control_info: char (32) varying` or `STRING (32)`

The opcode `TERM_GET_INPUT_SECTION (2058)` returns the name of the TTP’s current input section; the opcode `TERM_SET_INPUT_SECTION (2059)` allows your application program to use another input section of the TTP.

The input section of the TTP maps the keys on the terminal keyboard (function keys and other key sequences) to generic input requests, which perform common editing functions. The TTP can have multiple input sections to support different types of editing environments.



To ensure that certain basic requests are always recognized, the TTP has a default input section. When your application program specifies an input section using the opcode `TERM_SET_INPUT_SECTION(2059)`, it gains access to the specified input section **in addition to** the default input section. If there is a conflict between the definitions in the specified input section and the definitions in the default input section, the window terminal driver overrides the definitions in the default input section and uses the definitions in the specified input section.

Your application program must specify the name of the appropriate input section using the opcode `TERM_SET_INPUT_SECTION(2059)`. The name can be a string of up to 32 characters. The default input section does not have a unique name; therefore, your application program must specify a null string (length 0) to select the default input section.

If your application program specifies an input section that does not exist, the `s$control` call returns the error code `e$name_not_found (3084)`. For more information about the input section of a TTP, see the manual *VOS Communications Software: Defining a Terminal Type* (R096).

#### NOTE

The default character set for the input-section name is Latin alphabet No. 1, and the default shift mode is single shift.

**TERM\_GET\_INTERRUPT\_TABLE (2060)**

**TERM\_SET\_INTERRUPT\_TABLE (2061)**

`control_info: INTERRUPT_TABLE_V1 structure`

The opcode `TERM_GET_INTERRUPT_TABLE (2060)` returns the interrupt table that the access layer of the window terminal driver is currently using; the opcode `TERM_SET_INTERRUPT_TABLE (2061)` allows your application program to use another interrupt table. The interrupt table generally applies to the RS-232-C asynchronous access layer; it may not apply to other access layers.

The values in the interrupt table determine how the access layer processes input. Your application program should specify a new interrupt table only if it is using an unprocessed raw input mode (see Chapter 3).

To retrieve the current interrupt table or specify another interrupt table, use the following structure in your application program. When using this structure, note that the `flags` field is currently ignored and is reserved for future use.

PL/I Usage

```
declare 1 INTERRUPT_TABLE_V1 based longmap,
        2 version          fixed bin (15), /* 1 */
        2 flags            fixed bin (15),
        2 characters (0:255) char (1);
```

C Usage

```
typedef struct LONGMAP interrupt_table_v1
{
    short int      version; /* 1 */
    short int      flags;
    char           characters [256];
} INTERRUPT_TABLE_V1;
```

To use another interrupt table, perform the following steps.

- 1. If you plan to use more than one interrupt table, save the current interrupt table in a buffer using the opcode `TERM_GET_INTERRUPT_TABLE (2060)`.
- 2. Use the `INTERRUPT_TABLE_V1` structure and create a buffer of 256 bytes, where each byte represents the rank of a character in the internal character coding system. (For example, buffer position 0 corresponds to the ASCII NUL character, which has the decimal rank 0.) Then, assign an available interrupt value to each buffer position. (The interrupt values are listed in the table that follows.)
- 3. Issue the opcode `TERM_SET_INTERRUPT_TABLE (2061)` to pass the completed structure to the window terminal driver. You must ensure that the `version` field contains the value 1, the `flags` field contains the value 0, and the `characters` field contains the interrupt-character information.
- 4. When you are finished with this interrupt table, restore the interrupt table that you saved in Step 1 by issuing the opcode `TERM_SET_INTERRUPT_TABLE (2061)` and specifying the appropriate buffer name.

(Page 1 of 2)

Type of Interrupt	Value	Description
INTERRUPT_NEVER	0	The character is a noninterrupt character and does not generate an interrupt.
INTERRUPT_NOW	1	The character is a normal interrupt character and generates an interrupt.

(Page 2 of 2)

Type of Interrupt	Value	Description
INTERRUPT_PLUS1	2	The character is an interrupt-plus-one character and generates an interrupt upon receipt of any character following the interrupt character.
INTERRUPT_PLUS2	3	The character is an interrupt-plus-two character and generates an interrupt upon receipt of any two characters following the interrupt character.

If your application program issues the opcode `TERM_LISTEN (2023)` or `TERM_RESET_COMMAND_MODES (2088)`, the interrupt table will be destroyed. If this happens, you must re-create the interrupt table by issuing the opcode `TERM_SET_INTERRUPT_TABLE (2061)` after each call to `TERM_LISTEN (2023)` or `TERM_RESET_COMMAND_MODES (2088)`.

See Chapter 3 for more information about creating an interrupt table.

**`TERM_GET_KEY_BIT_MASK (2062)`**

**`TERM_SET_KEY_BIT_MASK (2063)`**

`control_info: KEY_BIT_MASK_V1` structure

The opcode `TERM_GET_KEY_BIT_MASK (2062)` returns information about the bit mask for generic input requests; the opcode `TERM_SET_KEY_BIT_MASK (2063)` allows your application program to change the bit mask.

The *bit mask* is an array of 0 to 255 bits, where each bit corresponds to a generic input request. For example, bit 0 corresponds to the `TAB` request, which has a decimal code of 0. (The codes associated with the generic input requests are presented in Chapter 5.)

A request is enabled if its corresponding bit is set, and is disabled if its corresponding bit is cleared. Your application program can enable or disable requests by setting or clearing the appropriate bits in the mask.

To retrieve or change bit-mask information, use the following structure in your application program.

## PL/I Usage

```
declare 1 KEY_BIT_MASK_V1 based longmap,  
        2 version          fixed bin (15), /* 1 */  
        2 keys_bit_mask (16)  fixed bin (15);
```

## C Usage

```
typedef struct LONGMAP key_bit_mask_v1  
{  
    short int          version; /* 1 */  
    short int          keys_bit_mask [16];  
} KEY_BIT_MASK_V1;
```

### **TERM\_GET\_KNOCKDOWN\_ID (2105)**

control\_info: fixed bin (15) or short int

This opcode returns the ID of the primary window attached to the port (that is, the port specified in the `port_id` argument of `s$control`). The primary window attached to the port may or may not be the top primary window. Your application program must specify the primary-window ID when issuing the opcode `TERM_KNOCK_DOWN_FORM` (2102) or `TERM_KNOCK_DOWN_FORM_OK` (2104) to abort (knock down) a displayed form. These opcodes are described later in this section.

### **TERM\_GET\_PAUSE\_CHARS (2068)**

### **TERM\_SET\_PAUSE\_CHARS (2069)**

control\_info: char (32) varying or STRING (32)

The opcode `TERM_GET_PAUSE_CHARS` (2068) returns the pause message displayed when a pause occurs; the opcode `TERM_SET_PAUSE_CHARS` (2069) allows your application program to change the pause message.

By default, the pause message is `--PAUSE--`. The string representing the message can contain up to 32 characters.

Pause characters are used only by simple sequential I/O (see Chapter 2).

### **TERM\_GET\_PAUSE\_LINES (2070)**

### **TERM\_SET\_PAUSE\_LINES (2071)**

control\_info: fixed bin (15) or short int

The opcode `TERM_GET_PAUSE_LINES` (2070) returns the number of lines that can scroll up and out of the output area in the simple sequential I/O subwindow before a pause occurs (the pause-lines limit). The opcode `TERM_SET_PAUSE_LINES`

(2071) allows your application program to change the pause-lines limit. By default, the pause-lines limit is the maximum size of the output area (see Chapter 2), which is always the height of the subwindow.

When specifying a pause-lines limit, the limit is observed only if it is less than the size of the current output area. If the limit is greater than the size of the current output area (for example, if the limit is 18 lines and the output area is 15 lines), the limit will not be observed until the output area shrinks to contain fewer lines than the limit. The window terminal driver will not pause more lines than the number of lines in the current output area. If the limit is 0 lines, a pause never occurs.

These opcodes apply only to simple sequential I/O. The display of raw output and formatted sequential output never pauses.

**TERM\_GET\_PROMPT\_CHARS (2072)**

**TERM\_SET\_PROMPT\_CHARS (2073)**

`control_info: char (32) varying or STRING (32)`

The opcode **TERM\_GET\_PROMPT\_CHARS (2072)** returns the character string (prompt) displayed by the operating system at the beginning of the input line. The string can be null or can contain up to 32 characters.

The opcode **TERM\_SET\_PROMPT\_CHARS (2073)** allows your application program to set a prompt. (By default, there is no prompt.)

**TERM\_GET\_PWIN\_BOUND\_LIMS (2080)**

`control_info: PRIMARY_WINDOW_LIMITS_V1 structure`

This opcode returns the current size limits for the primary window attached to the port.

To retrieve the primary-window size limits, use the following structure in your application program.

## PL/I Usage

```
declare 1 PRIMARY_WINDOW_LIMITS_V1 based longmap,
        2 version                fixed bin (15), /* 1 */
        2 min_width              fixed bin (15),
        2 max_width              fixed bin (15),
        2 min_height             fixed bin (15),
        2 max_height             fixed bin (15);
```

## C Usage

```
typedef struct LONGMAP primary_window_limits_v1
{
    short int    version; /* 1 */
    short int    min_width;
    short int    max_width;
    short int    min_height;
    short int    max_height;
} PRIMARY_WINDOW_LIMITS_V1;
```

### **TERM\_GET\_PWIN\_BOUNDS (2078)**

control\_info: PRIMARY\_WINDOW\_BOUNDS\_V1 structure

This opcode returns the size and location of the primary window attached to the port.

To retrieve the current primary-window values, use the PRIMARY\_WINDOW\_BOUNDS\_V1 structure in your application program. Note that the horizontal and vertical offsets (represented by upper\_left\_corner\_h and upper\_left\_corner\_v, respectively) are zero based.

## PL/I Usage

```
declare 1 PRIMARY_WINDOW_BOUNDS_V1 based longmap,
        2 version                fixed bin (15), /* 1 */
        2 upper_left_corner_h    fixed bin (15),
        2 upper_left_corner_v    fixed bin (15),
        2 width                  fixed bin (15),
        2 height                 fixed bin (15);
```

## C Usage

```
typedef struct LONGMAP primary_window_bounds_v1
{
    short int    version; /* 1 */
    short int    upper_left_corner_h;
    short int    upper_left_corner_v;
    short int    width;
    short int    height;
} PRIMARY_WINDOW_BOUNDS_V1;
```

**TERM\_GET\_PWIN\_PARAMS (2082)**

**TERM\_SET\_PWIN\_PARAMS (2083)**

control\_info: PRIMARY\_WINDOW\_PARAMS\_V1 structure

The opcode **TERM\_GET\_PWIN\_PARAMS (2082)** returns the current primary-window parameters; the opcode **TERM\_SET\_PWIN\_PARAMS (2083)** allows your application program to change the primary-window parameters. Your application program can also use other opcodes presented in this section (for example, **TERM\_SET\_CURSOR\_FORMAT (2055)**) to return or change individual primary-window parameters.

To retrieve or change the primary-window parameters, use the **PRIMARY\_WINDOW\_PARAMS\_V1** structure in your application program. Note that the **pwin\_param\_flags** field is reserved for future use.

### NOTE

The **PRIMARY\_WINDOW\_PARAMS\_V1** structure contains a shortmap directive; therefore, you should use the include file **window\_term\_info.incl.pl1** or **window\_term\_info.incl.c** to define the structure in your application program. If you do not use the include file and instead insert the actual structure in your application program, make sure that the structure contains the shortmap directive and that all declarations correspond

with those defined in the include file for the current VOS release.

## PL/I Usage

```
declare 1 PRIMARY_WINDOW_PARAMS_V1 based shortmap,
        2 version                      fixed bin (15), /* 1 */
        2 pwin_param_flags             fixed bin (31),
        2 pause_lines                  fixed bin (15),
        2 cursor_format                fixed bin (15),
        2 prompt_chars                 char (32) var,
        2 continue_chars               char (32) var,
        2 pause_chars                  char (32) var,
        2 undisplayable_notation_mode  fixed bin (15),
        2 max_typeahead                fixed bin (15),
        2 num_tab_stops                 fixed bin (15),
        2 tabs (25)                    fixed bin (15);
```

## C Usage

```
typedef struct SHORTMAP primary_window_params_v1
{
    short int      version; /* 1 */
    long int       pwin_param_flags;
    short int      pause_lines;
    short int      cursor_format;
    STRING(32)     prompt_chars;
    STRING(32)     continue_chars;
    STRING(32)     pause_chars;
    short int      undisplayable_notation_mode;
    short int      max_typeahead;
    short int      num_tab_stops;
    short int      tabs [25];
} PRIMARY_WINDOW_PARAMS_V1;
```

To determine the valid values for a specific field, see the description of the appropriate opcode in this chapter. For example, the description of the opcode `TERM_SET_CURSOR_FORMAT (2055)` identifies the valid values for the `cursor_format` field.

**TERM\_GET\_PWIN\_TITLE (2084)**

**TERM\_SET\_PWIN\_TITLE (2085)**

`control_info: char (256) varying or STRING (256)`



The opcode `TERM_GET_PWIN_TITLE` (2084) returns the title (name) of the current primary window; the opcode `TERM_SET_PWIN_TITLE` (2085) allows your application program to change the title. You can design your application program to display the title of the current primary window in the status area of the terminal screen. (The title can be the name of a command that is currently executing.) Your application program can specify a title of up to 256 characters.

#### **TERM\_GET\_SESSION\_ID\_OPCODE (2114)**

`control_info`: fixed bin (31) or long int

The opcode `TERM_GET_SESSION_ID_OPCODE` (2114) returns the `control_info` variable as the session ID associated with the primary window for the port attached to window terminal device. It returns the `control_info` value as 0 if it fails to get the session ID of the primary window.

#### **TERM\_GET\_SUBWIN\_BORDER\_OPCODE (2112)**

#### **TERM\_SET\_SUBWIN\_BORDER\_OPCODE (2113)**

`control_info`: `SUBWINDOW_BORDER_V1` structure

These opcodes allow your application program to get and set the border attributes of a specified sequential I/O subwindow. The `TERM_SET_SUBWIN_BORDER_OPCODE` (2113) does not cause the border output to be displayed, but the border becomes visible when another operation forces a screen update.

The opcode `TERM_GET_SUBWIN_BORDER_OPCODE` (2112) gets the border attributes with the `version` field is set to the value 1 and the `subwin_id` field is set to the ID of the sequential I/O subwindow. The subroutine `s$control` returns with the `border_type` field containing one of the following values:

- `NO_SUBWIN_BORDER` (1). Default and current behavior
- `EMPTY_SUBWIN_BORDER` (2). Single space border with no line drawing
- `SQUARE_CORNERRED_BORDER` (3). Single line border with square corners
- `ROUND_CORNERRED_BORDER` (4). Single line border with round corners

The `border_visual` variable indicates the visual attributes to be applied when the subwindow border is painted. The interpretation of these attributes is terminal dependent. The `flags` field must be zero. FMS uses two bits of the `flags` field.

## PL/I Usage

```
declare 1 SUBWINDOW_BORDER_V1 based longmap,
        2 version                fixed bin (15), /* 1 */
        2 subwin_id              fixed bin (15), /* input */
        2 flags                  fixed bin (15),
        2 border_type            fixed bin (15),
        2 border_visual          fixed bin (31);
```

## C Usage

```
typedef struct LONGMAP subwindow_border_v1
{
    short int    version;        /* 1 */
    short int    subwin_id;      /* input */
    short int    flags;
    short int    border_type;
    long int     border_visual;
} SUBWINDOW_BORDER_V1;
```

**TERM\_GET\_SUBWIN\_BOUNDS (2066)**

**TERM\_SET\_SUBWIN\_BOUNDS (2067)**

control\_info: SUBWINDOW\_INFO\_V2 structure

The opcode **TERM\_GET\_SUBWIN\_BOUNDS (2066)** returns the size and location of the subwindow specified in the `subwin_id` field; the opcode

**TERM\_SET\_SUBWIN\_BOUNDS (2067)** allows your application program to change the size and location of the specified subwindow. The size and location should be specified in terms of character positions within the primary window. If your application program specifies the value 32767 for both the height and width, the subwindow will be the same size as the primary window.

To retrieve or change the subwindow values, use the `SUBWINDOW_INFO_V2` structure in your application program. When using this structure, note that the version number must be 2 and the horizontal and vertical offsets (represented by `upper_left_corner_h` and `upper_left_corner_v`, respectively) are zero based. Note also that the `reserved` field is reserved for future use.

## PL/I Usage

```

declare 1 SUBWINDOW_INFO_V2 based longmap,
        2 version                fixed bin (15), /* 2 */
        2 subwin_id              fixed bin (15), /* input */
        2 upper_left_corner_h    fixed bin (15),
        2 upper_left_corner_v    fixed bin (15),
        2 width                  fixed bin (15),
        2 height                  fixed bin (15),
        2 reserved                fixed bin (15);

```

## C Usage

```

typedef struct LONGMAP subwindow_info_v2
{
    short int    version; /* 2 */
    short int    subwin_id; /* input */
    short int    upper_left_corner_h;
    short int    upper_left_corner_v;
    short int    width;
    short int    height;
    short int    reserved;
} SUBWINDOW_INFO_V2;

```

**TERM\_GET\_TABS (2074)**

**TERM\_SET\_TABS (2075)**

control\_info: TAB\_INFO\_V1 structure

The opcode **TERM\_GET\_TABS (2074)** returns the current tab settings; the opcode **TERM\_SET\_TABS (2075)** allows your application program to change the tab settings. The tab settings are an array of column positions that define tab stops. The left-most column is column 0. There can be up to 25 tab stops. By default, tab stops are set every five characters, beginning at character position six and ending at character position 126.

To retrieve or change the tab settings, use the **TAB\_INFO\_V1** structure in your application program.

## PL/I Usage

```
declare 1 TAB_INFO_V1 based longmap,
        2 version          fixed bin (15), /* 1 */
        2 num_tab_stops     fixed bin (15),
        2 tabs (25)         fixed bin (15);
```

## C Usage

```
typedef struct LONGMAP tab_info_v1
{
    short int      version; /* 1 */
    short int      num_tab_stops;
    short int      tabs [25];
} TAB_INFO_V1;
```

**TERM\_GET\_TYPEAHEAD\_LINES (2076)**

**TERM\_SET\_TYPEAHEAD\_LINES (2077)**

control\_info: fixed bin (15) or short int

The opcode **TERM\_GET\_TYPEAHEAD\_LINES (2076)** returns the number of completed unread input records allowed in the subwindow at a time (the typeahead limit). The opcode **TERM\_SET\_TYPEAHEAD\_LINES (2077)** allows your application program to change the typeahead limit; the default limit is 10 records. Note that the limit refers to a specified number of records, not lines. (A record may take up more than one line.) In addition, the limit does not include the current input record.

Use these opcodes for simple sequential I/O. For more information about typeahead, see Chapter 2.

**TERM\_GET\_UNDISP\_MODE (2064)**

**TERM\_SET\_UNDISP\_MODE (2065)**

control\_info: fixed bin (15) or short int

The opcode **TERM\_GET\_UNDISP\_MODE (2064)** returns the current undisplayable notation character mode. This mode allows your application program to determine the handling of characters for which there is no graphic representation (control characters and non-ASCII characters). The opcode **TERM\_SET\_UNDISP\_MODE (2065)** allows your application program to select an undisplayable notation character mode. The default undisplayable notation character mode is **UNDISP\_ESCAPED**.

The following list identifies the available modes and their corresponding values.

Undisplayable Notation Character Mode	Value	Description
UNDISP_ESCAPED	0	The character is represented in escaped notation (the character's hexadecimal equivalent preceded by the hexadecimal notation character variable ( <code>hex-notation-char</code> ), as defined in the TTP).
UNDISP_REPLACED	2	The character is represented by a printable character that is defined in the TTP.
UNDISP_SUPPRESSED	3	The character is suppressed (discarded).

The undisplayable notation character mode associated with the value 1 is reserved for future use.

#### NOTE

The old asynchronous driver handles control characters differently than the window terminal driver; it also uses the edited-output option to suppress the display of control characters. For more information about how the old asynchronous driver and the window terminal driver handle control characters, see Chapter 5 and Appendix A.

#### **TERM\_KNOCK\_DOWN\_FORM (2102)**

`control_info: fixed bin (15) or short int`

This opcode enables your application program to abort the display of the form that is currently on the terminal screen. Your application program should issue the opcode `TERM_GET_KNOCKDOWN_ID (2105)` before issuing this opcode so that it can specify the form's primary-window ID.

The opcode `TERM_KNOCK_DOWN_FORM (2102)` enables your application program to abort a form from a process or task other than the one displaying the form. When your application program aborts the display of a form using this opcode, any data that has been entered in the form is returned to the application program but is not validated. An application program that displays a form by using an `accept` statement receives a `keyused` option value of -6 if it aborts the form using this opcode. For more

information about the handling of forms, see the VOS Forms Management System manuals.

**TERM\_KNOCK\_DOWN\_FORM\_OK (2104)**

`control_info`: 0, or fixed bin (15) or short int

This opcode enables your application program to abort the display of the form that is currently on the terminal screen; it also returns the error code `e$invalid_form_id` (3794) if a form is not currently displayed on the terminal screen. Other than the return of the error code, this opcode performs the same function as the opcode `TERM_KNOCK_DOWN_FORM (2102)`.

Your application program must issue the opcode `TERM_GET_KNOCKDOWN_ID` (2105) before issuing this opcode so that it can specify the form's primary-window ID.

Like `TERM_KNOCK_DOWN_FORM (2102)`, this opcode enables your application program to abort a form from a process or task other than the one displaying the form. When your application program aborts the display of a form using this opcode, any data that has been entered in the form is returned to the application program but is not validated. An application program that displays a form by using an `accept` statement receives a `keyused` option value of -6 if it aborts the form using this opcode. For more information about the handling of forms, see the VOS Forms Management System manuals.

**TERM\_MOVE\_PWIN\_TO\_BOTTOM (2086)**

`control_info`: 0

This opcode makes the primary window attached to the port the bottom window. Since primary windows overlap, moving a primary window to the bottom means that it will be completely overlapped by another primary window.

**TERM\_MOVE\_PWIN\_TO\_TOP (2087)**

`control_info`: 0

This opcode makes the primary window attached to the port the top window. When a primary window becomes the top window, it receives all keyboard input.

**NOTE** 

---

This opcode does **not** refresh the display of the primary window. If the port is attached with the `hold_attached` and `hold_open` switches turned off (the default), your application program can refresh the display by performing any sequential I/O operation or by issuing the `s$control` opcode `RUNOUT (2)`. (The opcode `RUNOUT (2)` is

described earlier in this chapter.) When your application program refreshes the display, the window terminal driver ensures that the entire primary window is visible (that is, it completely overlaps any other primary windows). However, if the port is attached with either the `hold_attached` or `hold_open` switch turned on, your application program is responsible for maintaining the contents of all formatted I/O subwindows and must perform a subwindow redisplay when necessary.

#### **TERM\_OPEN\_EXISTING\_WINDOW\_OPCODE (2107)**

`control_info`: fixed bin (15) or short int

To use this opcode, you must attach a port by a call to `s$attach_port` just before using this opcode in the `s$control` call. The opcode then causes the subsequent call to `s$open` to open an existing primary window instead of opening a new primary window. The call to `s$open` opens the new port to the window terminal device.

To specify the primary window, your application program must set the variable `control_info` to the primary window's ID. The primary window's ID can be retrieved by using the `TERM_GET_KNOCKDOWN_ID (2105)` on another port that has the primary window open.

#### **TERM\_POSIX\_CLOSE\_OPCODE (2111)**

`control_info`: 0

The opcode `TERM_POSIX_CLOSE_OPCODE (2111)` performs all necessary functions to close POSIX compatibility mode on the primary window. Normally, this opcode is executed by the POSIX runtime. This opcode does not reenale cached formatted I/O when cached formatted I/O was disabled by `TERM_POSIX_OPEN_OPCODE (2110)`. To reenale cached formatted I/O, see the discussion for the opcode `TERM_ENABLE_CACHED_IO_OPCODE (2108)`.

#### **TERM\_POSIX\_OPEN\_OPCODE (2110)**

`control_info`: fixed bin (31) or long int

The opcode `TERM_POSIX_OPEN_OPCODE (2110)` performs all necessary functions to open the primary window in POSIX compatibility mode. Normally, this opcode is executed by the POSIX runtime. This opcode establishes the primary window as the "controlling terminal" for the current POSIX session. It also disables cached formatted I/O if it has been enabled.

The `control_info` variable is a flags word that should be initialized to 0 for normal operation and to 1 to disable establishing the primary window as a controlling terminal.

It returns with the value 1 if the terminal is a controlling terminal and 0 otherwise. If the primary window is already the controlling terminal of a POSIX session, `s$control` returns the error code `e$device_already_assigned` (1155).

**TERM\_POSIX\_GETATTR\_OPCODE (2115)**

**TERM\_POSIX\_SETATTR\_OPCODE (2116)**

`control_info`: `termios` structure

The opcode `TERM_POSIX_GETATTR_OPCODE` (2115) fills in a POSIX `termios` structure according to the state of the primary window associated with the port. The `termios` structure defines the terminal attributes for POSIX. The opcode `TERM_POSIX_SETATTR_OPCODE` (2116) applies the options specified in a `termios` structure to the primary window associated with the port attached to the window terminal device.

**TERM\_RESET\_COMMAND\_MODES (2088)**

`control_info`: 0

The command processor calls this opcode to reset the command modes (and return the port to a stable state) after the application program terminates.

This opcode performs the following functions:

- clears the input mode
- resets the interrupt table
- performs an output reset on all subwindows
- enables automatic character echoing
- if the original subwindow is in formatted I/O mode, returns the subwindow to simple sequential I/O mode.

**TERM\_RESET\_OUTPUT (2089)**

`control_info`: 0

If your application program is reading from or writing to the original subwindow, this opcode performs the following functions.

- If the original subwindow is in simple sequential I/O mode, this opcode performs an output reset. Your application program should use this opcode to terminate the effects of pressing the key mapped to the `ABORT_OUTPUT` request (typically, the key associated with the `DISCARD` function) or to terminate the effects of pressing the key mapped to the `NO_PAUSE` request. (See Chapter 2 for more information about the `ABORT_OUTPUT` request and the `NO_PAUSE` request.)



- If the original subwindow is in formatted I/O mode, this opcode returns the subwindow to simple sequential I/O mode.
- It enables automatic character echoing.

If your application program is using a formatted sequential I/O subwindow that it created with the opcode `TERM_CREATE_SUBWIN (2038)`, this opcode disables graphics mode and makes the cursor visible.

#### **TERM\_SAVE\_COMMAND\_INPUT (2090)**

`control_info: 0`

This opcode determines which input record the terminal user retrieves by pressing the key mapped to the `INSERT_DEFAULT` request. The opcode retains the next record retrieved by `s$seq_read` in the `INSERT_DEFAULT` buffer. The terminal user can retrieve the record from this buffer by pressing the key mapped to the `INSERT_DEFAULT` request. If your application program reissues this opcode, `s$seq_read` will replace the contents of the buffer when it retrieves the next record.

#### **TERM\_SET\_CURRENT\_SUBWIN (2091)**

`control_info: fixed bin (15) or short int`

This opcode makes the specified subwindow the current I/O subwindow. When a subwindow becomes the current I/O subwindow, it receives all keyboard input and all output from your application program.

Note that this opcode does not move the specified subwindow on top of the other subwindows. To do this, your application program must issue the opcode `TERM_SET_SUBWIN_TO_TOP (2025)`.

#### **TERM\_SET\_PWIN\_TO\_DEFAULTS (2092)**

`control_info: 0`

This opcode sets to the default settings the parameters of the primary window attached to the port. For more information, see the description of the opcode `TERM_SET_PWIN_DEFAULTS (2035)` earlier in this chapter.

#### **TERM\_SET\_SUBWIN\_TO\_BOTTOM (2029)**

`control_info: fixed bin (15) or short int`

This opcode moves the specified I/O subwindow below any other subwindows. To specify a subwindow, your application program must specify the subwindow's ID.

**NOTE** \_\_\_\_\_

This opcode does **not** refresh the display of the subwindow. For more information about making changes to the display, see the description of the opcode `TERM_CREATE_SUBWIN` (2038) earlier in this section.

**TERM\_SET\_SUBWIN\_TO\_TOP** (2025)

`control_info`: fixed bin (15) or short int

This opcode moves the specified I/O subwindow on top of any other subwindows, and makes that subwindow the current I/O subwindow. (The current I/O subwindow is the subwindow that handles all I/O.) To specify a subwindow, your application program must specify the subwindow's ID.

**NOTE** \_\_\_\_\_

This opcode does **not** refresh the display of the subwindow. For more information about making changes to the display, see the description of the opcode `TERM_CREATE_SUBWIN` (2038) earlier in this section.

**TERM\_STATUS\_MSG\_CHANGE** (2096)

`control_info`: `MESSAGE_CHANGE_V1` or `MESSAGE_CHANGE_V2` structure

This opcode allows your application program either to append a system error message to the standard status message, or to toggle the display of the status message. (A system error message provides a brief description of an error and identifies a specific error code.) The status message appears in the status area (which is usually the bottom line on the screen) and is always associated with the primary window.

To write a different status message to the status area, your application program must use either the opcode `TERM_WRITE_SYSTEM_MESSAGE` (2101) or the opcode `TERM_WRITE_SYMSG_NOBEEP` (2103). These opcodes are described later in this section.

When visible at operating-system command level, the standard status message displays information such as the date, time, percentage of wired memory used, user name, current command, and the type of mode used (for example, insert or overlay). It can also contain an error message or a message describing the status of the system (for example, a message from the system administrator). (See the section "Subwindow Organization" in Chapter 2 for a sample status message.)

To append an error message to the standard status message, use the `MESSAGE_CHANGE_V1` structure in your application program. This structure works as follows:

- If the old (existing) message in the `old_msg` field is 0 (that is, the status message does not contain an error message) and the new message specified by your application program in the `new_msg` field is 0, this opcode has no effect.
- If the old message is 0 and the new message identifies a valid error code, the error message is appended to the status message.
- If the old message identifies an error code and the new message specified is 0, the error message is deleted from the status message.
- If the old message identifies an error code and the new message identifies a valid error code, the existing error message is replaced by the new error message.

## PL/I Usage

```
declare 1 MESSAGE_CHANGE_V1 based longmap,
        2 version                fixed bin (15), /* 1 */
        2 old_msg                fixed bin (15),
        2 new_msg                fixed bin (15);
```

## C Usage

```
typedef struct longmap message_change_v1
{
    short int          version; /* 1 */
    short int          old_msg;
    short int          new_msg;
} MESSAGE_CHANGE_V1;
```

To display or clear the status message, use the following structure in your application program. Setting the `status_flag` field to 1 causes the window terminal driver to display the status message in the primary window (the equivalent of pressing the key mapped to the `UPDATE_STATUS` request). Setting the field to 0 causes the window terminal driver to clear the status message from the primary window (the equivalent of pressing the key or keys mapped to the `CLEAR_STATUS` request).

### NOTE \_\_\_\_\_

The `reserved` field is ignored.

## PL/I Usage

```
declare 1 MESSAGE_CHANGE_V2 based longmap,
        2 version                fixed bin (15), /* 2 */
        2 status_flag            fixed bin (15),
        2 reserved               fixed bin (15);
```

## C Usage

```
typedef struct LONGMAP message_change_v2
{
    short int    version; /* 2 */
    short int    status_flag;
    short int    reserved;
} MESSAGE_CHANGE_V2;
```

### **TERM\_WRITE\_SYMSG\_NOBEEP (2103)**

`control_info: char(N) varying` **OR** `char_varying (N)`

This opcode writes a message to the terminal screen but does not sound the terminal's bell when the message appears on the screen. If the terminal screen has a status line designated for messages (typically, the 25th line), the message appears there; otherwise, the message appears on the line below the cursor.

The value of the `control_info` argument must be less than or equal to the current line length (width) defined for the terminal. For more information about the line length defined for a terminal, see the description of the global control opcode `GET_LINE_LENGTH (1)` and the terminal opcode `TERM_GET_SCREEN_WIDTH (2028)`.

### **TERM\_WRITE\_SYSTEM\_MESSAGE (2101)**

`control_info: char(N) varying` **OR** `char_varying (N)`

This opcode writes a message to the terminal screen and sounds the terminal's bell when the message appears on the screen. If the terminal screen has a status line designated for messages (typically, the 25th line), the message appears there; otherwise, the message appears on the line below the cursor.

The value of the `control_info` argument must be less than or equal to the current line length (width) defined for the terminal. For more information about the line length defined for a terminal, see the description of the global control opcode `GET_LINE_LENGTH (1)` and the terminal opcode `TERM_GET_SCREEN_WIDTH (2028)`.

## Error Codes

The `s$control` subroutine may return the following error codes.

(Page 1 of 3)

Error Code	Description
<code>e\$caller_must_wait</code> (1277)	The data in the output buffers has not yet been written to the device. This error code is returned if the port is in no-wait mode and your application program issues the opcode <code>RUNOUT (2)</code> .
<code>e\$device_not_assigned_to_you</code> (1158)	The calling process is not the primary user of the device.
<code>e\$form_cant_delete_orig_subwin</code> (3956)	Your application program attempted to delete the original subwindow.
<code>e\$form_invalid_subwindow_id</code> (3964)	Your application program specified a subwindow that does not exist.
<code>e\$invalid_baud_rate</code> (4761)	Your application program specified an invalid baud-rate value.
<code>e\$insufficient_access</code> (1141)	The caller is not privileged and the user ID of the caller does not match the user ID of the first process to open the terminal.
<code>e\$invalid_arg</code> (1371)	The version number is wrong, the <code>Reserved</code> field is nonzero, or the <code>border_visual</code> field contains an invalid value.
<code>e\$invalid_bits_per_char</code> (4762)	Your application program specified an invalid bits-per-character value.
<code>e\$invalid_break_action</code> (4770)	Your application program specified an invalid break-action value.
<code>e\$invalid_buffer_size</code> (1215)	Your application program did not specify an internal buffer size in the range 0 to 2048.
<code>e\$invalid_comm_model</code> (3144)	The communications hardware does not support the specified operating parameter (for example, a baud-rate value that exceeds the maximum for the adapter) or feature.
<code>e\$invalid_control_operation</code> (1366)	Your application program did not specify an opcode in one of the following ranges: 1 to 4, 201 to 300, or 2001 to 2200.

(Page 2 of 3)

Error Code	Description
e\$invalid_cursor_format (4767)	Your application program specified an invalid cursor format.
e\$invalid_form_id (3794)	There is no displayed form to abort.
e\$invalid_input_mode (4760)	Your application program specified an invalid input mode.
e\$invalid_io_operation (1040)	Your application program did not open the specified port before calling s\$control.
e\$invalid_key (4769)	Your application program specified an invalid key.
e\$invalid_knockdown_id (5207)	The specified window was not found.
e\$invalid_open_action (4768)	Your application program specified an invalid open action.
e\$invalid_parity (4763)	Your application program specified an invalid parity value.
e\$invalid_stop_bits (4764)	Your application program specified an invalid stop-bits value.
e\$invalid_string_size (4776)	Your application program specified an invalid string size.
e\$invalid_subwindow_bounds (4771)	Your application program specified invalid subwindow boundaries (that is, an invalid location or invalid dimensions).
e\$invalid_tab_settings (1499)	The specified tab columns either are not in ascending order or extend beyond column 255. The window terminal driver responds by setting tab stops every five characters. By default, the first tab stop is at character position six.
e\$invalid_undisp_mode (4774)	Your application program specified an invalid undisplayable notation character mode.
e\$wrong_version (1083)	The specified version was not 1.

(Page 3 of 3)

Error Code	Description
e\$long_record (1026)	The data portion of the <code>control_info</code> structure is too small to hold the returned output.
e\$name_not_found (3084)	The specified input section or configuration setup does not exist.
e\$not_yet_implemented (1062)	The specified opcode is in the range reserved for the window terminal driver, but is not yet implemented.
e\$out_of_range (1038)	Your application program did not specify a function-key code in the range 0 to 511.
e\$unknown_terminal_type (1221)	The TTP specified by your application program or the TTP specified in the <code>terminal_type</code> field of the <code>devices.tin</code> entry is not known to the system.
e\$wrong_version (1083)	The <code>version</code> field of the <code>control_info</code> structure contains an invalid value for the specified opcode.

## **s\$control\_device**

Use this subroutine to perform device-specific operations not handled by `s$control`. The `s$control_device` subroutine passes information to the device driver using an input buffer and passes information to your application program using an output buffer. Your application program specifies the size of these buffers.

There are two `s$control_device` opcodes available to your application program: `TERM_SAVE_PWIN_PARAMS_DEVOP` (201) and `TERM_RESTORE_PWIN_PARAMS_DEVOP` (202). For more information about these opcodes, see “Opcodes for `s$control_device`” later in this subroutine description.

If the port associated with the device is in no-wait mode, and the call to `s$control_device` returns the error code `e$caller_must_wait` (1277), this indicates that no data has been transferred (that is, the call cannot perform a partial transfer or operation). In this case, your application program must wait for the device event to be notified and then reissue the call.

If `s$control_device` can complete the transfer or operation, it returns the number of bytes actually stored in the output buffer (returned in the `out_length` argument). If the value of `out_length` is greater than the length of the output buffer (supplied by your application program using the `out_buffer_length` argument), and the output buffer length is nonzero, the call returns the error code `e$long_record` (1026). If your application program supplies an output buffer length of 0, the call returns the correct data length (returned in the `out_length` argument). In this case, no data and no error code are returned. If you do not know how large a buffer to allocate, supply an output buffer length of 0 to enable your application program to retrieve the correct data length.



## PL/I Usage

```

declare port_id                fixed bin (15);
declare opcode                 fixed bin (15);
declare in_buffer_length      fixed bin (31);
declare in_buffer (N)         char (1);
declare out_buffer_length     fixed bin (31);
declare out_length            fixed bin (31);
declare out_buffer (N)        char (1);
declare error_code            fixed bin (15);

declare s$control_device entry( fixed bin (15),
                                fixed bin (15),
                                fixed bin (31),
                                (N) char (1),
                                fixed bin (31),
                                fixed bin (31),
                                (N) char (1),
                                fixed bin (15));

call s$control_device(          port_id,
                                opcode,
                                in_buffer_length,
                                in_buffer (N),
                                out_buffer_length,
                                out_length,
                                out_buffer (N),
                                error_code);

```

## C Usage

```

short int port_id;
short int opcode;
long int in_buffer_length;
char in_buffer [N];
long int out_buffer_length;
long int out_length;
char out_buffer [N];
short int error_code;

void s$control_device();

s$control_device( &port_id,
                  &opcode,
                  &in_buffer_length,
                  in_buffer,
                  &out_buffer_length,
                  &out_length,
                  out_buffer,
                  &error_code);

```

## Arguments

The `s$control_device` subroutine takes eight arguments, which are listed and described in the following table.

Argument	Description
<code>port_id</code> (input)	The identifier of a port attached to the device. The port must be attached to an asynchronous communications line.
<code>opcode</code> (input)	The operation to be performed on the device.
<code>in_buffer_length</code> (input)	The number of bytes available in <code>in_buffer</code> .
<code>in_buffer</code> (input)	A buffer containing the data to be sent to the device driver.
<code>out_buffer_length</code> (input)	The number of bytes available in <code>out_buffer</code> .
<code>out_length</code> (output)	The number of bytes returned in <code>out_buffer</code> . This value can be greater than the value of <code>out_buffer_length</code> if the error code <code>e\$long_record</code> (1026) is returned or if the value of <code>out_buffer_length</code> is 0.
<code>out_buffer</code> (output)	The buffer into which the results of the operation are stored.
<code>error_code</code> (output)	A returned error code. The error codes returned by <code>s\$control_device</code> are listed in “Error Codes” later in this subroutine description.

## Opcodes for `s$control_device`

There are two `s$control_device` opcodes available to your application program: `TERM_SAVE_PWIN_PARAMS_DEVOP` (201) and `TERM_RESTORE_PWIN_PARAMS_DEVOP` (202). These opcodes are useful when your application program must temporarily change primary-window parameters. In this case, your application program saves the current primary-window parameters, changes the appropriate primary-window parameter (for example, using the `s$control` opcode `TERM_SET_CURSOR_FORMAT` (2055)) or group of parameters (using the `s$control` opcode `TERM_SET_PWIN_PARAMS` (2083)), and then restores the previous parameters when the change is no longer necessary.

**TERM\_SAVE\_PWIN\_PARAMS\_DEVOP (201)**

`out_buffer: char(N) or STRING(N)`

This opcode saves the current primary-window parameters. For this opcode, your application program must supply an output buffer length that is large enough to hold the primary-window data. The buffer can hold from 0 to 32,767 bytes, depending on the operation to be performed.

The `out_buffer_length` argument contains the length of the output buffer supplied by your application program, and the `out_length` argument contains the actual number of bytes returned in `out_buffer`. If your application program receives the error code `e$long_record (1026)`, it can use the value of the `out_length` argument to determine an appropriate value for the `out_buffer_length` argument. Your application program can also determine how large a buffer to allocate by supplying a value of 0 for the `out_buffer_length` argument. Supplying a value of 0 causes the `out_length` argument to contain the correct data length; in addition, no error code is returned.

**TERM\_RESTORE\_PWIN\_PARAMS\_DEVOP (202)**

`in_buffer: char(N) or STRING(N)`

This opcode restores the previous primary-window parameters. For this opcode, your application program must supply an input buffer that was retrieved previously with the opcode `TERM_SAVE_PWIN_PARAMS_DEVOP (201)`. The buffer can hold from 0 to 32,767 bytes, depending on the operation to be performed.

The `in_buffer_length` argument contains the length of the input buffer supplied by your application program. The `in_buffer` argument contains data that was retrieved in a previous save operation (performed by `TERM_SAVE_PWIN_PARAMS_DEVOP (201)`).

If the buffer size specified by the `in_buffer_length` argument is too small, your application program receives the error code `e$buffer_too_small (1133)`.

## Error Codes

The s\$control\_device subroutine may return the following error codes.

Error Code	Description
e\$buffer_too_small (1133)	Your application program specified a buffer size that was too small or an incorrect value (for example, a negative number) for the in_buffer_length argument. You must correct the error in your application program.
e\$caller_must_wait (1277)	No data in the output buffers can be written to the device because the call cannot perform a partial transfer operation. This error code is returned if the port is in no-wait mode. Your application program must wait for the device event to be notified and then reissue the call to s\$control_device.
e\$device_already_assigned (1155)	The primary window is already the “controlling terminal” of a POSIX session. This is a nonfatal error and will be mapped to zero.
e\$device_not_assigned_to_you (1158)	The calling process is not the primary user of the device.
e\$invalid_comm_model (3144)	The communications hardware does not support the specified operating parameter.
e\$invalid_control_operation (1366)	Your application program did not specify an opcode in the range 100 to 300.
e\$invalid_io_operation (1040)	Your application program did not open the specified port before calling s\$control_device.
e\$long_record (1026)	The out_buffer structure is too small to hold the returned output.
e\$not_yet_implemented (1062)	The specified opcode is in the range reserved for the window terminal driver, but is not yet implemented.

---

## Chapter 9

# Closing and Detaching a Port

This chapter describes the two VOS subroutines that enable your application program to close and detach a port.

- `s$close`
- `s$detach_port`

Note that the descriptions of the subroutines focus on terminal I/O. For complete descriptions of the subroutines and how they would apply to other types of I/O, refer to the VOS Subroutines manuals. Each subroutine description in this chapter explains how to make the call from an application program written in either PL/I or C.

## **s\$close**

Use this subroutine to close the specified terminal port or virtual-device port. When your application program calls `s$close`, the primary window associated with the specified port disappears from the terminal screen. When the last primary window attached to a terminal is closed, the operating system clears the terminal's I/O buffers.

Once your application program calls `s$close`, it can call `s$detach_port` to detach the port from the terminal or virtual device. Note that the primary window remains attached to the port after `s$close` finishes executing, and is not freed until your application program calls `s$detach_port`. As long as the primary window is attached to the port, your application program can reopen the port.

If your application program is using wait mode, `s$close` does not return until the port attached to the device is closed. In no-wait mode, your application program receives the error code `e$caller_must_wait` (1277) and must reissue `s$close`. (See [Chapter 8](#) for a description of no-wait mode.)

### **NOTE**

---

Before closing the port, your application program should issue the `s$control` global opcode `RUNOUT (2)` to ensure that all pending output has been sent to the RS-232-C communications hardware for transmission to the terminal. (The opcode `RUNOUT (2)` is described in [Chapter 8](#) in the section “Global Control Opcodes.”)

## PL/I Usage

```

declare port_id                fixed bin (15);
declare error_code             fixed bin (15);

declare  s$close entry(        fixed bin (15),
                               fixed bin (15));

                               call s$close(      port_id,
                                                    error_code);

```

## C Usage

```

short int                      port_id;
short int                      error_code;

void s$close();

s$close(                        &port_id,
                               &error_code);

```

## Arguments

The `s$close` subroutine takes two arguments, which are listed and described in the following table.

Argument	Description
<code>port_id</code> (input)	The identifier of the port to be closed.
<code>error_code</code> (output)	A returned error code.

## Error Codes

The `s$close` subroutine may return the following error codes.

(Page 1 of 2)

Error Code	Description
<code>e\$caller_must_wait</code> (1277)	The I/O operation could not be completed. Your application program must reissue <code>s\$close</code> .

(Page 2 of 2)

Error Code	Description
e\$close_invalid_on_sys_port (3425)	The port is a system port and cannot be closed by your application program. Your application program can close only user ports.
e\$invalid_io_operation (1040)	The specified port is not open.
e\$port_not_attached (1021)	The specified port is not attached.



## s\$detach\_port

Use this subroutine to detach a port from the terminal or virtual device. The port must be closed before your application program can call this subroutine.

### PL/I Usage

```
declare port_id                fixed bin (15);
declare error_code             fixed bin (15);

declare s$detach_port entry(    fixed bin (15),
                                fixed bin (15));

                                call s$detach_port(  port_id,
                                                        error_code);
```

### C Usage

```
short int                      port_id;
short int                      error_code;

void s$detach_port();

                                s$detach_port(      &port_id,
                                                        &error_code);
```

### Arguments

The s\$detach\_port subroutine takes two arguments, which are listed and described in the following table.

Argument	Description
port_id (input)	The identifier of the port to be detached from the device.
error_code (output)	A returned error code.

**Error Codes**

The `s$detach_port` subroutine may return the following error codes.

Error Code	Description
<code>e\$bad_port_number</code> (1029)	The specified port ID is invalid.
<code>e\$invalid_io_operation</code> (1040)	The specified port is not closed.
<code>e\$port_not_attached</code> (1021)	The specified port is not attached.

---

# Appendix A

## Compatibility Issues

The window terminal driver supports a primary-window and subwindow environment; therefore, it differs significantly from the old asynchronous driver, which is described in the manual *VOS Communications Software: Asynchronous Communications* (R025).

If you plan to use an old asynchronous driver application program with the window terminal driver, you should be aware of various compatibility issues. Some of these issues may warrant changes to the application program. This appendix addresses the following compatibility issues.

- 8-bit support and parity options
- Flow-control options
- The function of simple sequential I/O
- The function of unprocessed raw I/O
- The function of processed raw input
- Generic input and generic output requests
- The handling of attributes
- The handling of control characters
- The handling of `s$control` opcodes
- The handling of the complete-write option
- Asynchronous device configuration

### 8-Bit Support and Parity Options

The window terminal driver and the old asynchronous driver support six types of parity: odd, even, mark, space, Baudot, and no parity. With a parity type other than no parity or Baudot, the old asynchronous driver always transmits 7 bits per character. With no parity, the old asynchronous driver always transmits 8 bits per character without break handling (the processing of metacharacters to signal errors to your application program). The old asynchronous driver does not provide break handling for 8-bit transmission on K-series hardware; it provides break handling only for 7-bit transmission on the hardware.

The window terminal driver offers your application program more parity options and break handling for 8-bit transmission. Using a parity type other than no parity or Baudot does **not** force the window terminal driver to transmit 7 bits per character. The window terminal driver handles 8-bit transmission with odd, even, or no parity (with break handling), as long as the device is connected to K-series hardware (for example, the K101, K111, or K118 I/O adapter). (See [Chapter 1](#) for a description of K-series hardware.)

With the old asynchronous driver, the application program uses the opcode `TERM_CONFIGURE` (227) to set the parity, baud-rate, and stop-bits values. With the window terminal driver, the application program sets the parity, bits-per-character, baud-rate, and stop-bits values with the opcode `TERM_SET_OPERATING_VALUES` (2014); or, the application program can set these values individually with opcodes such as `TERM_SET_PARITY` (2018) and `TERM_SET_BITS_PER_CHAR` (2006). If you design your application program to use 8-bit odd parity or 8-bit even parity, make sure that it uses the window terminal opcodes instead of the standard asynchronous opcode `TERM_CONFIGURE` (227). For a description of the window terminal opcodes, see the section “Opcodes Affecting the RS-232-C Communications Medium” in [Chapter 8](#).

Transmitting 8 bits per character with the appropriate type of parity also affects the entries for asynchronous devices in the `devices.tin` configuration file. See the table in the section “Asynchronous Device Configuration” later in this appendix for a brief summary of the differences between `devices.tin` entries for window terminal devices and standard asynchronous (nonwindow terminal) devices.

## Flow-Control Options

The window terminal driver offers more flow-control options than the old asynchronous driver. The old asynchronous driver offers only output flow control (DSL flow control or XON/XOFF flow control) for terminals; it does not support input flow control for terminals.

The window terminal driver supports DSL or XON/XOFF output flow control, DSL or XON/XOFF input flow control, and bidirectional flow control (output flow control with input flow control). To use input flow control or bidirectional flow control, the device must be connected to K-series hardware. For more information about flow control, see the section “Using Flow Control” in [Chapter 1](#). See the section “Opcodes Affecting the RS-232-C Communications Medium” in [Chapter 8](#) for descriptions of the following opcodes:

```
TERM_GET_INPUT_FLOW_INFO (2007)
TERM_SET_INPUT_FLOW_INFO (2008)
TERM_GET_OUTPUT_FLOW_INFO (2015)
TERM_SET_OUTPUT_FLOW_INFO (2016)
```

## Function of Simple Sequential I/O


The three-area subwindow organization used by the window terminal driver is different from the screen organization used by the old asynchronous driver, and therefore presents a different user interface for simple sequential (command-line) I/O. With the old asynchronous driver, typeahead is often mixed with output, and the position of the input line often changes. With the window terminal driver, typeahead has its own area, and the position of the input line does not change. See [Chapter 2](#) for a description of the three-area subwindow organization used by the window terminal driver for simple sequential I/O.


In general, the window terminal driver offers more line-editing options than the old asynchronous driver. [Table A-1](#) summarizes some of these line-editing options.

Table A-1. Differences in Line-Editing Options (Page 1 of 2)

Key	Function with Old Asynchronous Driver	Function with Window Terminal Driver
<code>RETURN</code> (at command level)	Depends on the context; scrolls paused output, enters the input record, or both.	Enters the input record when one exists. The function of the key when there is no current input record depends on the setting of the <code>RETURN_Doesnt_Unpause</code> screen option, which is controlled with the opcode <code>TERM_SET_SCREEN_PREF (2027)</code> . By default, the key performs the same function as the key mapped to the <code>NEXT_SCREEN</code> request. However, if the <code>RETURN_Doesnt_Unpause</code> screen option is set, the key does not release the pause. Instead, it causes the display of a message that identifies the proper way to release a pause and enters an empty input record.
<code>RETURN</code> (in a form)	Performs a tab up, down, left, or right.	The function of the key depends on the setting of the <code>RETURN_Doesnt_Tab</code> screen option, which is controlled with the opcode <code>TERM_SET_SCREEN_PREF (2027)</code> . By default, the key can tab left and right as well as up and down. However, if the <code>RETURN_Doesnt_Tab</code> screen option is set, the key does not tab left or right. Instead, it moves the cursor to the first field of the next line, and the user must press the <code>TAB</code> key to tab horizontally to a field.

Table A-1. Differences in Line-Editing Options (Page 2 of 2)

Key	Function with Old Asynchronous Driver	Function with Window Terminal Driver
The key mapped to the CANCEL request	Depends on the context; cancels the pause (and the output), cancels the input record, or both. When there is typeahead at a pause, the key performs both actions.	Cancels the input record when one exists. The function of the key when there is no current input record depends on the setting of the CANCEL_Doesnt_Abort screen option, which is controlled with the opcode TERM_SET_SCREEN_PREF (2027) . By default, the key performs the same function as the key mapped to the ABORT_OUTPUT request. However, if the CANCEL_Doesnt_Abort screen option is set, the key does not perform the same function as the key mapped to the ABORT_OUTPUT request. Instead, it causes the display of a message that identifies the proper way to abort a pause.
	No function.	Scrolls paused output one line at a time.

As indicated in the table, the window terminal driver provides screen options that enable your application program to control the handling of certain line-editing characteristics. The default screen options provide the pausing characteristics of the old asynchronous driver. Note that certain line-editing functions are exactly the same for both drivers. For example, the key mapped to the NEXT\_SCREEN request (often ) scrolls a screen of paused output, and the key mapped to the ABORT\_OUTPUT request releases the pause and discards all subsequent output.

## The Function of Unprocessed Raw I/O

The old asynchronous driver offers three unprocessed raw input modes: normal raw, use break table, and break table record. In normal raw or use break table mode, your application program can retrieve *bulk raw input*, which involves the setting of the `bulk_raw_input` mode bit and allows `s$read_raw` to return data whenever data is available. The window terminal driver offers equivalent raw input modes (normal raw, raw table, and raw record); however, they function differently than the raw input modes offered by the old asynchronous driver.

Table A-2 explains the differences between the unprocessed raw input modes offered by the old asynchronous driver and the window terminal driver.

Table A-2. Differences in the Unprocessed Raw Input Modes

Standard Asynchronous Raw Input Mode	Window Terminal Raw Input Mode	Description
Normal raw with bulk raw retrieval	Normal raw	Every character is an interrupt character, and <code>s\$read_raw</code> can return data whenever data is available. With the normal/bulk raw combination, <code>s\$read_raw</code> returns the error code <code>e\$short_record</code> (1299) when the buffer is partially full. This error does not apply to the normal raw mode offered by the window terminal driver.
Normal raw without bulk raw retrieval	No functional equivalent	Every character is an interrupt character, but <code>s\$read_raw</code> can return data in wait mode <b>only</b> when the buffer is full. In no-wait mode, <code>s\$read_raw</code> returns partial data and the error code <code>e\$caller_must_wait</code> (1277).
Break table with bulk raw retrieval	Raw table	Each character specified as an interrupt character in the interrupt (or break) table causes an interrupt, and <code>s\$read_raw</code> can return data when the buffer is partially full. With the break table/bulk raw combination, <code>s\$read_raw</code> also returns the error code <code>e\$short_record</code> (1299) when the buffer is partially full. This error does not apply to the raw table mode offered by the window terminal driver. Break table mode and raw table mode are suitable for RS-232-C connections.
Break table without bulk raw retrieval	No functional equivalent	Each character specified as an interrupt character in the interrupt table causes an interrupt, and <code>s\$read_raw</code> returns data in wait mode <b>only</b> when the buffer is full. In no-wait mode, <code>s\$read_raw</code> returns partial data and the error code <code>e\$caller_must_wait</code> (1277). Break table mode is suitable for RS-232-C connections.
Break table record	Raw record	Each character specified as an interrupt character in the interrupt table causes an interrupt and terminates a record. Thus, <code>s\$read_raw</code> returns a complete record when the buffer is partially full or when the buffer is full and includes the interrupt character. When the buffer is full and does not include the interrupt character, <code>s\$read_raw</code> returns a partial record and the error code <code>e\$long_record</code> (1026).

An application program using the old asynchronous driver selects an unprocessed raw input mode by setting certain bits with the `s$control` opcode `SET_MODES` (207) or `SET_INFO` (202). With the window terminal driver, the `s$control` opcode

`TERM_SET_INPUT_MODE` (2057) enables an application program to set the raw input mode.

An application program that creates its own interrupt table should do so by using the window terminal opcode `TERM_SET_INTERRUPT_TABLE` (2061) instead of the standard asynchronous opcode `SET_BREAK_TABLE` (225), which was designed to accommodate a 128-bit break (interrupt) table.

See the section “Opcodes Affecting the Primary Window and Subwindow” in [Chapter 8](#) for a description of the opcode `TERM_SET_INTERRUPT_TABLE` (2061). See [Chapter 3](#) for a description of the unprocessed raw input modes.

**NOTE** \_\_\_\_\_  
If you have an application program that uses no-wait mode, verify that it performs a read operation before waiting for input.

## The Function of Processed Raw Input

The window terminal driver and the old asynchronous driver handle processed raw input (function-key input and generic input) differently. The two drivers handle translated input (characters that map to the internal character coding system) in the same way. Note that some generic input requests apply only to the window terminal driver, and the range of function keys is 0 through 511 for the window terminal driver (rather than 0 through 255).

[Table A-3](#) summarizes how the window terminal driver and the old asynchronous driver return processed raw input.

**Table A-3. Methods of Returning Processed Raw Input** (Page 1 of 2)

Input Mode	Method of Return with Old Asynchronous Driver	Method of Return with Window Terminal Driver
Translated input mode	If the application program specifies bulk raw retrieval, the method of return is the same as that of the window terminal driver. Without bulk raw retrieval, <code>s\$read_raw</code> waits until the buffer is full before returning (in wait mode).	In this mode, <code>s\$read_raw</code> embeds in the input stream the codes representing translated internal character strings, as well as unprocessed characters that did not map to the character-translation database of a TTP.



**Table A-3. Methods of Returning Processed Raw Input** (Page 2 of 2)

Input Mode	Method of Return with Old Asynchronous Driver	Method of Return with Window Terminal Driver
Function-key input mode	When a function key is available, <code>s\$read_raw</code> returns a function-key sequence that is introduced by FE hexadecimal (254 decimal), followed by the code for the function key. In this mode, the application program can use bulk raw retrieval.	When a function key is available, <code>s\$read_raw</code> uses a 9-byte binary data introducer (BDI) sequence to identify the function key by using its key number and shift-modifier value. Chapter 4 describes BDI sequences.
Generic input mode	When a generic input request is available, <code>s\$read_raw</code> returns a generic input sequence that is introduced by 1B hexadecimal (ESC), followed by the 1-byte code for the request. In this mode, the application program can use bulk raw retrieval. This mode does not allow <code>s\$read_raw</code> to return raw bytes.	When a generic input request is available, <code>s\$read_raw</code> uses a 6-byte BDI sequence to identify a generic input request. Each generic input request uses one BDI sequence. If a string does not map to an input request, function key, or translated input string, <code>s\$read_raw</code> uses a raw-input BDI sequence to return the first byte of the string.

If you have an application program that uses `s$read_raw` to read generic input sequences or function keys, you may want to update the sequence-decoding logic so that it recognizes the BDI syntax.

An application program using the window terminal driver selects a processed raw input mode by specifying the mode with the `s$control` opcode `TERM_SET_INPUT_MODE` (2057). An application program using the old asynchronous driver selects a processed raw input mode by setting certain bits with the `s$control` opcode `SET_MODES` (207) or `SET_INFO` (202). In general, the window terminal opcode offers a better method of handling mode settings.

#### NOTE \_\_\_\_\_

If an application program using the window terminal driver selects a processed raw input mode by specifying the standard asynchronous opcode `SET_MODES` (207) or `SET_INFO` (202), `s$read_raw` will use the method of return for the old asynchronous driver, as outlined in [Table A-3](#). In this situation, `s$read_raw` does not return a received function key or generic input request using the BDI syntax of the window terminal driver; instead, function keys are introduced by FE hexadecimal (254 decimal)

and generic input requests are introduced by 1B hexadecimal (27 decimal, the `ESC` character).

## Generic Input and Generic Output Requests

In some cases, the window terminal driver and the old asynchronous driver recognize different generic input and generic output requests, and handle some requests differently. The following list summarizes the differences.

- Certain input requests apply only to the window terminal driver.
- Certain output requests do not apply to the window terminal driver.
- Certain requests typically recognized by the old asynchronous driver should be replaced by `s$control` opcodes, TTP subroutines, or other requests for the window terminal driver.
- Certain requests have different names (aliases) to reflect their role in the window environment.

### Requests That Apply Only to the Window Terminal Driver

The generic input requests with codes 198 (`LOGIN_PROCESS`) and 201 (`TWIDDLE`) through 211 (`LEAVE_WINDOW_MANAGER`) are used by the window terminal driver only. (Many of these requests are available only in window manager mode.) These generic input requests and their codes are listed in the system include files `video_request_defs.incl.pll` and `video_request_defs.incl.c`. (Request 212 (`ABORT_OUTPUT`) is recognized by both drivers.)

Three of the window terminal requests deserve special mention: `TWIDDLE` (201), `RAW_INPUT` (205), and `FUNCTION_KEY` (209). The `TWIDDLE` request, available with sequential I/O, edits the input line by transposing the two characters to the left of the cursor. Note that this request is not included in the standard TTPs; to be supported, it must be defined in the appropriate TTP.

The `RAW_INPUT` request, available when retrieving processed raw input under the application-managed I/O approach (specifically, generic input mode), is used in BDI sequences to return the first byte of a string that did not map to the appropriate database (character translation, keyboard, or generic input).

The `FUNCTION_KEY` request, also available when retrieving processed raw input under the application-managed I/O approach, is used to introduce function keys in BDI sequences. See the description of processed raw input in [Chapter 4](#) for more information about the `RAW_INPUT` and `FUNCTION_KEY` requests.

## Requests That Do Not Apply to the Window Terminal Driver

The following generic output requests do not apply to the window terminal driver (that is, they are ignored in either simple or formatted sequential I/O mode). If you have an application program that uses any of these requests, you should revise your application program.

```
ENTER_INSERT_MODE_SEQ (12)
LEAVE_INSERT_MODE_SEQ (13)
ENTER_MONITOR_MODE_SEQ (29)
LEAVE_MONITOR_MODE_SEQ (30)
SET_INSERT_CHAR_SEQ (41)
SET_SCREEN_PAGE_SIZE_SEQ (42)
```

These requests are defined in the manual *VOS Communications Software: Asynchronous Communications* (R025).

## Requests That Should Be Replaced by Opcodes, TTP Subroutines, or Other Requests

The window terminal driver replaces various generic output requests used by the old asynchronous driver with `s$control` opcodes, TTP subroutines, or other requests. Although these generic output requests are available, Stratus recommends that you use the window terminal opcodes, TTP subroutines, and other requests because they provide a more convenient and efficient user interface. All generic output requests are defined in the system include files `output_sequence_codes.incl.pll` and `output_sequence_codes.incl.c`.

[Table A-4](#) explains how to replace the generic output requests used by the old asynchronous driver. [Table A-4](#) does **not** address the requests that control attributes. Attributes are discussed in the section “The Handling of Attributes” later in this appendix.

For more information about the `s$control` opcodes listed in [Table A-4](#), see the description of the `s$control` subroutine in [Chapter 8](#). For information about the generic output requests used by the window terminal driver, see [Chapter 5](#).

**Table A-4. Replacing Generic Output Requests Used by the Old Asynchronous Driver** (Page 1 of 2)

Generic Output Request(s)	Instructions
INITIAL_STRING_SEQ (9)	Remove the request. The window terminal driver monitors the status of the screen display and sends the initial-string sequence when necessary.
RESET_25TH_LINE_SEQ (19), START_25TH_LINE_SEQ (20), END_25TH_LINE_SEQ (21)	Replace the request RESET_25TH_LINE_SEQ (19) with version 2 (V2) of the <code>s\$control</code> opcode TERM_STATUS_MSG_CHANGE (2096). Replace all START_25TH_LINE_SEQ (20) and END_25TH_LINE_SEQ (21) requests with the <code>s\$control</code> opcode TERM_WRITE_SYSTEM_MESSAGE (2101) or TERM_WRITE_SYSMSG_NOBEEP (2103). (See the section “Opcodes Affecting the Primary Window and Subwindow” in <a href="#">Chapter 8</a> for more information about these opcodes.)
SET_BLACK_ON_WHITE_SEQ (34), SET_WHITE_ON_BLACK_SEQ (35)	Replace these requests by specifying the appropriate option with the <code>s\$control</code> opcode TERM_SET_SCREEN_PREF (2027). (See the section “Opcodes Affecting the Terminal” in Chapter 8 for more information about this opcode.)
KEY_CLICK_ON_SEQ (36), KEY_CLICK_OFF_SEQ (37)	Replace these requests by specifying the appropriate option with the <code>s\$control</code> opcode TERM_SET_SCREEN_PREF (2027).
SET_SMOOTH_SCROLL_SEQ (39), SET_JUMP_SCROLL_SEQ (40)	Replace these requests by specifying the appropriate option with the <code>s\$control</code> opcode TERM_SET_SCREEN_PREF (2027).
SPECIAL_1_SEQ (51), SPECIAL_2_SEQ (52), SPECIAL_3_SEQ (53), SPECIAL_4_SEQ (54)	Define these requests as capabilities in the output section of the TTP. Use the TTP subroutine <code>s\$http_get_output_cap</code> to retrieve information about the capabilities defined in the output section of the TTP. (Appendix C briefly describes this subroutine.)
GOTO_PAGE_SEQ (68)	Replace this request with the <code>s\$control</code> opcode TERM_SET_CURRENT_SUBWIN (2091), which makes the specified subwindow the current I/O subwindow. If you use this request with formatted sequential I/O, you must specify the appropriate subwindow ID with the request's page argument.

**Table A-4. Replacing Generic Output Requests Used by the Old Asynchronous Driver** (Page 2 of 2)

Generic Output Request(s)	Instructions
GOTO_PAGE_SEQ (68) <i>(Continued)</i>	If there is no subwindow with the specified subwindow ID, this request creates a subwindow that is the same size as the primary window and makes it the top subwindow. The window terminal driver immediately updates the display when it processes the request. The section “Generic Output Requests That Affect Subwindow Updates” in Chapter 4 describes display updates. If your application program specifies this request while using simple sequential I/O in the original subwindow, the original subwindow switches to formatted I/O mode. For information about formatted I/O mode, see the description of the <code>s\$control</code> opcode <code>TERM_ENABLE_FMT_IO_MODE</code> (2046) in the section “Opcodes Affecting the Primary Window and Subwindow” in Chapter 8.
UNFREEZE_LINES_SEQ (71)	Replace this request with the request <code>SET_SCROLLING_REGION_SEQ</code> (95). If your application program uses the <code>UNFREEZE_LINES_SEQ</code> (71) request with formatted sequential I/O, the request resets the scrolling region to include all of the lines in the subwindow. If your application program specifies this request with simple sequential I/O, the request switches the original subwindow to formatted I/O mode.
ESI_DELIMITER_SEQ (90), EMI_DELIMITER_SEQ (91), EGI_DELIMITER_SEQ (92)	Define these requests as capabilities in the output section of the TTP. Use the TTP subroutine <code>s\$ttp_get_output_cap</code> to retrieve information about the capabilities defined in the output section of the TTP. (Appendix C briefly describes this subroutine.)
SET_CURSOR_INVISIBLE_SEQ (117), SET_CURSOR_BLINKING_BLOCK_SEQ (118), SET_CURSOR_STEADY_BLOCK_SEQ (119), SET_CURSOR_BLINKING_UNDERLINE_SEQ (120), SET_CURSOR_STEADY_UNDERLINE_SEQ (121)	With either simple or formatted sequential I/O, select a cursor format with the request <code>SET_CURSOR_FORMAT_SEQ</code> (38) or with the <code>s\$control</code> opcode <code>TERM_SET_CURSOR_FORMAT</code> (2055). (See the section “Opcodes Affecting the Primary Window and Subwindow” in Chapter 8 for more information about this opcode.)

## Window Terminal Requests That Have Aliases

Two window terminal generic output requests have specific names to reflect their function in the primary-window and subwindow environment. These requests have old asynchronous driver aliases.

- The window terminal request `CLEAR_TO_EOR (4)` (which clears all characters from the cursor to the end of the scrolling region) has the old asynchronous driver alias `CLEAR_TO_EOS (4)` (which clears all characters from the cursor to the end of the screen). Although both requests use the same code number (4) and therefore require no change to existing application programs, new application programs should use the window terminal request `CLEAR_TO_EOR (4)`.
- The window terminal request `CLEAR_SCROLLING_REGION (2)` has the old asynchronous driver alias `CLEAR_SCREEN (2)`. Although both requests use the same code number (2) and therefore require no change to existing application programs, new application programs should use the window terminal request `CLEAR_SCROLLING_REGION (2)`.

See [Chapter 5](#) for a description of the requests `CLEAR_TO_EOR (4)` and `CLEAR_SCROLLING_REGION (2)`.

## Handling of Attributes

Regardless of how a terminal functions, the window terminal driver treats all attributes as *mode attributes*, where the setting of an attribute affects all **subsequent** writes to the subwindow. Since the old asynchronous driver allows the setting of both mode and area attributes (*area attributes* mark a specific area of the subwindow for a given attribute), the window terminal driver may introduce compatibility issues for application programs using the `SET_ATTRIBUTES (18)` or `SET_ATTRIBUTE_CHAR (72)` request to explicitly control area attributes. You must modify such application programs to work in a mode-attribute environment. You can no longer use the `SET_ATTRIBUTES (18)` and `SET_ATTRIBUTE_CHAR (72)` requests to place area-attribute markers.

In general, for either simple or formatted sequential I/O, you should use the `SET_MODE_ATTRIBUTES (133)` request instead of the `SET_ATTRIBUTES (18)` request. Note that the `SET_MODE_ATTRIBUTES (133)` request takes a two-byte argument, whereas the `SET_ATTRIBUTES (18)` request requires only a one-byte argument.

If you want to position the cursor while setting the attributes, use the `POSITION_CURSOR (1)` request to position the cursor and use the `SET_MODE_ATTRIBUTES (133)` request to set the attributes. Do not use the `SET_ATTRIBUTE_CHAR (72)` request, which positions the cursor **and** sets the attributes.

If you continue to use the `SET_ATTRIBUTE_CHAR` (72) request, note that specifying a cursor position other than the current position (represented by a line and column value of 255) causes the request to switch the original subwindow to formatted I/O mode. (For information about formatted I/O mode, see the description of the `s$control` opcode `TERM_ENABLE_FMT_IO_MODE` (2046) in the section “Opcodes Affecting the Primary Window and Subwindow” in [Chapter 8](#). See [Chapter 5](#) for a description of the `SET_MODE_ATTRIBUTES` (133) request.)

## Handling of Control Characters

The window terminal driver and the old asynchronous driver handle certain control characters differently when the characters appear in output strings. [Table A-5](#) lists these control characters and describes the differences in the way the two drivers handle the control characters when they are defined in the TTP.

**Table A-5. Differences in the Handling of Control Characters**

Control Character	Function with Window Terminal Driver	Function with Old Asynchronous Driver
BEL	Performs a BEEP request.	Performs a BEEP request.
BS	Performs a backspace with the <code>LEFT</code> request in a formatted sequential I/O subwindow. This character causes the switch to formatted I/O mode in a simple sequential I/O subwindow.	Performs a backspace with the <code>LEFT</code> request.
CR	Performs a carriage return in a formatted sequential I/O subwindow. This character causes the switch to formatted I/O mode in a simple sequential I/O subwindow.	Performs a carriage return.
FF	Not supported. This character is handled according to the setting of the undisplayable notation character mode.	Performs a form feed.
HT	Inserts spaces up to the next tab. This character is always supported, even if it is not defined in the TTP.	Inserts spaces up to the next tab. This character is always supported, even if it is not defined in the TTP.
LF	Performs a line feed with the <code>NEW_LINE</code> request.	Performs a line feed with the <code>NEW_LINE</code> request.
VT	Not supported. This character is handled according to the setting of the undisplayable notation character mode.	Performs a vertical tab.

With the old asynchronous driver, if these control characters (except for `HT`) are not defined in the TTP, they are either replaced by hexadecimal notation or discarded (if the edited-output option is enabled). Any other control characters are also replaced by hexadecimal notation or discarded (if the edited-output option is enabled). (See the manual *VOS Communications Software: Asynchronous Communications* (R025) for information about the edited-output option.)

With the window terminal driver, control characters are handled as described in the `s$seq_write` subroutine description in [Chapter 7](#). The `BEL`, `BS`, `LF`, and `CR` control characters should always be defined in the TTP. If the TTP does not define these control characters, they are discarded.

The `VT` and `FF` control characters are no longer supported and are always handled according to the setting of the undisplayable notation character mode. Any other control characters are also handled according to the setting of the undisplayable notation character mode. For a description of this mode, see the section “Defining the Handling of Standard ASCII Control Characters” in [Chapter 5](#), the description of the `s$seq_write` subroutine in [Chapter 7](#), and the description of the `s$control` opcode `TERM_SET_UNDISP_MODE` (2065) in the section “Opcodes Affecting the Primary Window and Subwindow” in [Chapter 8](#).

## Handling of `s$control` Opcodes

In general, the window terminal driver supports the standard asynchronous opcodes for which it has `s$control` equivalents. If your application program uses a standard asynchronous opcode that does not offer all of the features of its window terminal equivalent, the window terminal driver emulates the behavior of the standard asynchronous opcode and ignores the features of the window terminal equivalent.

You should revise your application program if you want it to take advantage of features not offered by the standard asynchronous opcodes (for example, the break options and the interrupt-table options).

[Table A-6](#) lists (in alphabetical order) the standard asynchronous opcodes and the equivalent window terminal opcodes, and compares the opcode functions. See [Chapter 8](#) for a description of the window terminal opcodes. See the manual *VOS Communications Software: Asynchronous Communications* (R025) for a description of the standard asynchronous opcodes.

As of VOS Release 11.4, the old asynchronous driver opcode `SEND_BREAK` (275) and the window terminal opcode `TERM_SEND_BREAK` (2106) allow your application program to generate a break condition and transmit a break to the terminal over an RS-232-C line. (See the section “Opcodes Affecting the Terminal” in [Chapter 8](#) for a description of the opcode `TERM_SEND_BREAK` (2106).) In addition, the window terminal driver provides equivalent opcodes for the standard asynchronous opcodes that write messages to the status line and abort (knock down) a form.



**Table A-6. Comparison of Standard Asynchronous and Window Terminal Opcodes** (Page 1 of 7)

Standard Asynchronous Opcode	Equivalent Window Terminal Opcode	Description
ASYNC_HANGUP (234)	TERM_HANGUP (2001)	Both opcodes break the connection on an RS-232-C line by dropping the DSR signal.
BREAK_DISABLE (223)	TERM_SET_BREAK_ACTION (2051)	The opcode BREAK_DISABLE (223) disables the terminal's <b>BREAK</b> key. The opcode TERM_SET_BREAK_ACTION (2051) provides the IGNORE option, which tells the window terminal driver to ignore the break condition.
BREAK_ENABLE (222)	TERM_SET_BREAK_ACTION (2051)	The opcode BREAK_ENABLE (222) enables the terminal's <b>BREAK</b> key and requires another mode bit to determine the handling of the break. The opcode TERM_SET_BREAK_ACTION (2051) provides various break options and is beneficial because it controls <b>all</b> of the break options (including disabling a break).
DISCARD_INPUT (204)	TERM_DISCARD_INPUT (2040)	Both opcodes discard characters that have been typed but not read.
DISCARD_OUTPUT (205)	TERM_DISCARD_OUTPUT (2041)	Both opcodes discard characters that have been placed in a buffer but not written.
DISPLAY_OFF (221)	TERM_DISABLE_ECHO (2043)	Both opcodes disable character echoing.
DISPLAY_ON (220)	TERM_ENABLE_ECHO (2042)	Both opcodes enable character echoing.
GET_BREAK_TABLE (243)	TERM_GET_INTERRUPT_TABLE (2060)	The opcode GET_BREAK_TABLE (243) retrieves a 128-bit break (interrupt) table. The opcode TERM_GET_INTERRUPT_TABLE (2060) retrieves a 256-bit interrupt table. The opcode GET_INTERRUPT_TABLE (267) retrieves a 256-bit interrupt table for the old asynchronous driver.
GET_DEFINED_KEYS (251)	TERM_GET_KEY_BIT_MASK (2062)	Both opcodes return the defined function-key sequences.
GET_INFO (201)	Various opcodes, such as TERM_GET_PROMPT_CHARS (2072)	The window terminal driver does not use a single opcode to return all of the terminal parameters returned with the opcode GET_INFO (201). The window terminal driver uses a specific opcode to return each terminal parameter.

**Table A-6. Comparison of Standard Asynchronous and Window Terminal Opcodes** (Page 2 of 7)

Standard Asynchronous Opcode	Equivalent Window Terminal Opcode	Description
GET_INPUT_SECTION (268)	TERM_GET_INPUT_SECTION (2058)	Both opcodes return the name of the current input section of the TTP.
GET_INTERRUPT_TABLE (267)	TERM_GET_INTERRUPT_TABLE (2060)	Both opcodes retrieve the interrupt table for K-series hardware. The opcode <code>TERM_GET_INTERRUPT_TABLE (2060)</code> also works with C-series hardware, but provides fewer interrupt-character options than for K-series hardware.
GET_KEY_LEGEND (252)	No equivalent; defined in the TTP	The opcode <code>GET_KEY_LEGEND (252)</code> returns the defined function keys.
GET_KNOCKDOWN_ID (272)	TERM_GET_KNOCKDOWN_ID (2105)	Both opcodes return an ID. For the old asynchronous driver, this is the ID of the form that is currently displayed; for the window terminal driver, this is the primary-window ID.
GET_TERMINAL_SETUP (270)	TERM_GET_TERMINAL_SETUP (2030)	Both opcodes return the name of the configuration setup that the application program is using. The configuration section of the TTP can define multiple configuration setups.
HOLD_CONNECTION (244)	TERM_HOLD_CONNECTION (2021)	Both opcodes maintain the connection to the device when the last port is closed.
INTERRUPT_KEY_DISABLE (246)	TERM_DISABLE_KEY (2045)	Both opcodes disable the terminal's interrupt key.
INTERRUPT_KEY_ENABLE (245)	TERM_ENABLE_KEY (2044)	Both opcodes enable the terminal's interrupt key.
KNOCK_DOWN_FORM (240)	TERM_KNOCK_DOWN_FORM (2102)	Both opcodes abort the form that is currently displayed without returning an error code. These opcodes also require you to specify the primary-window ID returned with the opcode <code>TERM_GET_KNOCKDOWN_ID (2105)</code> .
KNOCK_DOWN_FORM_OK (264)	TERM_KNOCK_DOWN_FORM_OK (2104)	Both opcodes abort the form that is currently displayed and return the error code <code>e\$invalid_form_id (3794)</code> if the specified ID is not valid. These opcodes require you to specify the primary-window ID returned with the opcode <code>TERM_GET_KNOCKDOWN_ID (2105)</code> .

**Table A-6. Comparison of Standard Asynchronous and Window Terminal Opcodes** (Page 3 of 7)

Standard Asynchronous Opcode	Equivalent Window Terminal Opcode	Description
LISTEN (203)	TERM_LISTEN (2023)	Both opcodes cause the operating system to listen to the RS-232-C channel or subchannel for the terminal.
RESET_COMMAND_INPUT (226)	TERM_SAVE_COMMAND_INPUT (2090)	Both opcodes control the action of the key to which the INSERT_DEFAULT request is mapped.
RESET_OUTPUT (224)	TERM_RESET_OUTPUT (2089)	Both opcodes resume the normal display of all processed output.
RESET_TERMINAL (230)	TERM_RESET_COMMAND_MODES (2088)	Both opcodes reset certain characteristics of the terminal's port after the application program has finished executing.
SET_BREAK_TABLE (225)	TERM_SET_INTERRUPT_TABLE (2061)	The opcode SET_BREAK_TABLE (225) sets a 128-bit break (interrupt) table. The opcode TERM_SET_INTERRUPT_TABLE (2061) sets 256-bit interrupt tables. The opcode SET_INTERRUPT_TABLE (266) sets a 256-bit interrupt table for the old asynchronous driver.
SET_CONTINUE_CHARS (214)	TERM_SET_CONTINUE_CHARS (2053)	Both opcodes set the continue message, which is the character string that is displayed when a line wraps to the next line. The opcode SET_CONTINUE_CHARS (214) allows the application program to specify a string of up to eight characters. The opcode TERM_SET_CONTINUE_CHARS (2053) allows the application program to specify a string of up to 32 characters.
SET_ESCAPE_CHAR (217)	No equivalent; defined in the TTP	The opcode SET_ESCAPE_CHAR (217) allows the application program to set the terminal's escape character.

**Table A-6. Comparison of Standard Asynchronous and Window Terminal Opcodes** (Page 4 of 7)

Standard Asynchronous Opcode	Equivalent Window Terminal Opcode	Description
SET_INFO (202)	Various opcodes, such as TERM_SET_PROMPT_CHARS (2073) and TERM_SET_INPUT_MODE (2057)	The window terminal driver does not use a single opcode to set all of the terminal parameters set with the opcode SET_INFO (202). The window terminal driver uses a specific opcode to return each terminal parameter. Note that setting function-key input mode or generic input mode with SET_INFO (202) when using the window terminal driver causes <i>s\$read_raw</i> to return input in the style used by the old asynchronous driver.
SET_INPUT_SECTION (269)	TERM_SET_INPUT_SECTION (2059)	Both opcodes allow the application program to specify and use a different TTP input section.
SET_INTERRUPT_TABLE (266)	TERM_SET_INTERRUPT_TABLE (2061)	Both opcodes set the interrupt table for K-series hardware. The opcode TERM_SET_INTERRUPT_TABLE (2061) also works with C-series hardware, but provides fewer interrupt-character options than for K-series hardware.
SET_LANGUAGE (258)	No equivalent	The opcode SET_LANGUAGE (258) sets the channel to match the language used by the current process.
SET_LINE_LENGTH (209)	TERM_SET_TERMINAL_SETUP (2031)	The opcode SET_LINE_LENGTH (209) sets the maximum line length for the screen display. The opcode TERM_SET_TERMINAL_SETUP (2031) allows the application program to change the line length by specifying a configuration setup that is defined in the configuration section of the TTP.
SET_MAX_BUFFER_SIZE (247)	TERM_SET_MAX_BUFFER_SIZE (2010)	Both opcodes change the number of internal buffers allocated for the device's channel or subchannel.

**Table A-6. Comparison of Standard Asynchronous and Window Terminal Opcodes** (Page 5 of 7)

Standard Asynchronous Opcode	Equivalent Window Terminal Opcode	Description
SET_MODES (207)	Various opcodes, such as TERM_SET_INPUT_MODE (2057)	The opcode SET_MODES (207) uses mode bits and can set all of the terminal modes recognized by the old asynchronous driver. This opcode sets internal flags to ensure compatibility with the window terminal driver. The window terminal driver supports different modes, most of which have dedicated opcodes. The opcode TERM_SET_INPUT_MODE (2057) sets the window terminal raw input modes. Note that setting function-key input mode or generic input mode with SET_MODES (207) when using the window terminal driver causes <i>s\$read_raw</i> to return input in the style used by the old asynchronous driver.
SET_OUTPUT_FLOW (216)	TERM_SET_OUTPUT_FLOW (2016)	Both opcodes allow the application program to select the type of output flow control and specify the flow-control characters. The opcode SET_OUTPUT_FLOW (216) controls the setting of two internal mode bits and has many restrictions. The opcode TERM_SET_OUTPUT_FLOW_INFO (2016) uses a structure to store the output flow-control information. Window terminal application programs using bidirectional flow control must follow the guidelines presented in Chapter 8 for both input flow control and output flow control.
SET_PAUSE_CHARS (215)	TERM_SET_PAUSE_CHARS (2069)	Both opcodes set the terminal's pause message, which is a string of characters displayed when the display of output pauses. The opcode SET_PAUSE_CHARS (215) allows the application program to specify a string of up to 20 characters. The opcode TERM_SET_PAUSE_CHARS (2069) allows the application program to specify a string of up to 32 characters.
SET_PAUSE_LINES (211)	TERM_SET_PAUSE_LINES (2071)	Both opcodes set the number of lines between pauses.

**Table A-6. Comparison of Standard Asynchronous and Window Terminal Opcodes** (Page 6 of 7)

Standard Asynchronous Opcode	Equivalent Window Terminal Opcode	Description
SET_PRIMARY_USER (231)	TERM_MOVE_PWIN_TO_TOP (2087)	The opcode SET_PRIMARY_USER (231) makes the calling process the primary user. The opcode TERM_MOVE_PWIN_TO_TOP (2087) moves a primary window to the top of the primary-window stack so that it receives all keyboard input.
SET_PROMPT_CHARS (208)	TERM_SET_PROMPT_CHARS (2073)	Both opcodes set the prompt character string. The opcode SET_PROMPT_CHARS (208) allows the application program to specify a string of up to 8 characters. The opcode TERM_SET_PROMPT_CHARS (2073) allows the application program to specify a string of up to 32 characters.
SET_SCREEN_SIZE (210)	TERM_SET_TERMINAL_SETUP (2031)	The opcode SET_SCREEN_SIZE (210) sets the terminal's screen size (height). The opcode TERM_SET_TERMINAL_SETUP (2031) allows the application program to change the screen size by specifying a configuration setup that is defined in the configuration section of the TTP.
SET_TABS (213)	TERM_SET_TABS (2075)	Both opcodes set the terminal's tab stops. The opcode TERM_SET_TABS (2075) has different parameter definitions than the opcode SET_TABS (213).
SET_TERMINAL_SETUP (271)	TERM_SET_TERMINAL_SETUP (2031)	Both opcodes allow the application program to specify and use a configuration setup that is defined in the configuration section of the TTP. A configuration setup defines terminal characteristics such as line length (width) and screen size (height).
SET_TERMINAL_TYPE (218)	TERM_SET_TERMINAL_TYPE (2033)	Both opcodes set the terminal type (TTP) for the device.

**Table A-6. Comparison of Standard Asynchronous and Window Terminal Opcodes** (Page 7 of 7)

Standard Asynchronous Opcode	Equivalent Window Terminal Opcode	Description
TERM_CONFIGURE (227)	TERM_SET_OPERATING_VALUES (2014)	Both opcodes set the baud-rate, stop-bits, and parity values. The opcode TERM_SET_OPERATING_VALUES (2014) also allows your application program to set the bits-per-character value. The window terminal driver allows your application program to set these values collectively or individually.
TERM_GET_CONFIG (265)	TERM_GET_OPERATING_VALUES (2013)	Both opcodes return the baud-rate, stop-bits, and parity values. The opcode TERM_GET_OPERATING_VALUES (2014) also allows your application program to retrieve the bits-per-character value. The window terminal driver allows your application program to retrieve these values collectively or individually.
WRITE_SYSTEM_MESSAGE (206)	TERM_WRITE_SYSTEM_MESSAGE (2101)	Both opcodes write a message to the terminal and generate a beep.
WRITE_SYSTEM_MSG_NO_BEEP (259)	TERM_WRITE_SYMSG_NOBEEP (2103)	Both opcodes write a message to the terminal without generating a beep.

## Handling of the Complete-Write Option

The old asynchronous driver allows an application program using wait mode to prevent partial writes to the terminal by enabling the complete-write option (that is, by setting the `complete_write` mode bit of the `terminal_info` structure). When this option is enabled, the old asynchronous driver does not send any output until all characters specified by the write subroutine can be sent.

The window terminal driver does not handle the complete-write option. If an application program sets the `complete_write` mode bit, the window terminal driver simply ignores it. If you have an application program that uses the complete-write option, you should determine whether your program can execute properly without it.

## Asynchronous Device Configuration

As described in the *Product Configuration Bulletin: Asynchronous Devices* (R289), every asynchronous device must be defined for the system in the device configuration file (`devices.tin`). A `devices.tin` entry for a device using the window terminal

driver is different from the entry for a device using the old asynchronous driver.  
[Table A-7](#) summarizes these differences.

**Table A-7. Differences in Asynchronous Device Configuration**

<code>devices.tin</code> Field	Window Terminal Driver <code>devices.tin</code> Value	Old Asynchronous Driver <code>devices.tin</code> Value
<code>device_type</code>	<code>window_term</code> for a terminal or printer.	terminal or printer.
<code>parity</code>	<code>odd</code> , <code>even</code> , or <code>none</code> specifies 8-bit transmission on K-series hardware.	<code>none</code> specifies 8-bit transmission, regardless of the type of hardware.
<code>bits_per_char</code>	8 does not assume no parity. The parity value and the type of hardware determines the parity used.	8 assumes no parity.
<code>parameters</code>	<code>-access_layer async_al</code> , <code>-access_layer telnet_al</code> , <code>-access_layer tli_al</code> , or <code>-access_layer recc_al</code> .	Not applicable.



---

## Appendix B

# Internal Character Coding System

The Stratus internal character coding system is based on the International Standards Organization (ISO) document *ISO 2022, ISO 7-bit and 8-bit Coded Character Sets – Code Extension Techniques*. The coding system consists of the following elements.

- The left-hand control character set (0 to 31 decimal, 00 to 1F hexadecimal)
- The ASCII `SP` character (32 decimal, 20 hexadecimal)
- The ASCII `DEL` character (127 decimal, 7F hexadecimal)
- The left-hand graphic character set (33 to 126 decimal, 21 to 7E hexadecimal)
- The right-hand control character set (128 to 159 decimal, 80 to 9F hexadecimal)
- The right-hand supplementary graphic character set (160 to 255 decimal, A0 to FF hexadecimal)

The two graphic character sets contain codes for characters that display images on the terminal screen, while the two control character sets contain codes for characters that cause a specific action (for example, the control character `CR` does not produce an image, but causes a carriage return). The character assignments for the two control character sets and the left-hand graphic character set are fixed, while those for the right-hand graphic character set can vary.

The `SP` and `DEL` characters are not part of either control character set or graphic character set, but are part of standard ASCII. The `SP` and `DEL` characters, the left-hand control character set, and the left-hand graphic character set form the subset of the ASCII set used by VOS. The right-hand control character set is specific to the Stratus internal character coding system and consists of characters used to select supplementary internal character sets and to perform other internal tasks.

[Figure B-1](#) illustrates the fixed assignments for the two control character sets and the left-hand graphic character set. (See [Appendix C](#) for the complete names of the control characters.)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	<div>Control</div> <div>Left-hand Control and Graphic Character Sets</div>		SP	0	@	P	`	p	SS1	LSI	<div>Graphic</div> <div>Supplementary Set of Graphic Characters</div>					
1			!	1	A	Q	a	q	SS4	WPI						
2			"	2	B	R	b	r	SS5							
3			#	3	C	S	c	s	SS6	BDI						
4			\$	4	D	T	d	t	SS7							
5			%	5	E	U	e	u	SS8							
6			&	6	F	V	f	v	SS9							
7			'	7	G	W	g	w	SS10							
8	BS	(	8	H	X	h	x	SS11								
9	HT	)	9	I	Y	i	y	SS12								
A	LF	SUB	*	:	J	Z	j	z	SS13							
B	VT	ESC	+	;	K	[	k	{	SS14							
C	FF		,	<	L	\	l		SS15							
D	CR		-	=	M	]	m	}								
E			.	>	N	^	n	~	SS2							
F			/	?	O	_	o	DEL	SS3							

The control and graphic character sets are described in the sections that follow.

## Left-hand Control Character Set

This set is derived from the control set of *ISO 646-1983* and *ANSI X3.4-1977* (registration number 1). It includes the characters 7 to 13 decimal (07 to 0D hexadecimal), the substitute character `SUB`, and the `ESC` character, which functions as a generic sequence introducer (GSI).

### Control Characters 7 to 13 Decimal (07 to 0D Hexadecimal)

The control characters in the range 7 to 13 decimal (07 to 0D hexadecimal) perform special functions with the window terminal driver. See the description of the `s$seq_write` subroutine in [Chapter 7](#) for information about the function of these characters.

### `SUB` Character

The `SUB` character (26 decimal, 1A hexadecimal) replaces a character that is corrupted during transmission (for example, due to a parity error) or translation. The substitute character can produce either a backward question mark or `^1A`. (The backward question mark is the standard representation of `SUB` specified by *ISO 2047-1975*.) The `SUB` character is also returned by `s$seq_read` when only part of a multibyte character fits in the input buffer. (See the description of the `s$seq_read` subroutine in [Chapter 7](#) for more information.)

### `ESC` Character

In the coding scheme, the `ESC` character (27 decimal, 1B hexadecimal) functions as a generic sequence introducer (GSI) to mark the beginning of a window terminal generic output sequence. (With the standard asynchronous driver, the GSI marks the beginning of either a generic input or generic output sequence.) A generic output sequence always contains the GSI, followed by a code that represents a generic output request. [Chapter 5](#) lists the generic output requests.

## Left-hand Graphic Character Set

This set contains the left-hand part of the ISO 8859 8-bit graphic character sets. It also contains the ASCII graphic character set.

## Right-hand Control Character Set

This set is specific to Stratus. It invokes supplementary graphic character sets (by means of single-shift *N* characters) and identifies locking shift introducer (LSI)

sequences, word processing introducer (WPI) sequences, and binary data introducer (BDI) sequences.

**Single-Shift *N* Characters**

A single-shift *N* character (*SS N*) indicates that the next character is from a supplementary graphic character set. The following byte (or two or three, if the set is a multibyte set) must be in the range 160 to 255 decimal (A0 to FF hexadecimal).

**Locking Shift Introducer**

The locking shift introducer (LSI) is a prefix for locking-shift *N* right (*LS N R*) control sequences. Its rank is 144 decimal (90 hexadecimal). In hexadecimal, the byte following the LSI is 20 hexadecimal greater than the corresponding *SS N* character.

SS1	80 hexadecimal	LS1R	90 hexadecimal	A0 hexadecimal
SS2	8E hexadecimal	LS2R	90 hexadecimal	AE hexadecimal
SS3	8F hexadecimal	LS3R	90 hexadecimal	AF hexadecimal
SS4	81 hexadecimal	LS4R	90 hexadecimal	A1 hexadecimal
SS5	82 hexadecimal	LS5R	90 hexadecimal	A2 hexadecimal
SS6	83 hexadecimal	LS6R	90 hexadecimal	A3 hexadecimal
SS7	84 hexadecimal	LS7R	90 hexadecimal	A4 hexadecimal
SS8	85 hexadecimal	LS8R	90 hexadecimal	A5 hexadecimal
SS9	86 hexadecimal	LS9R	90 hexadecimal	A6 hexadecimal
SS10	87 hexadecimal	LS10R	90 hexadecimal	A7 hexadecimal
SS11	88 hexadecimal	LS11R	90 hexadecimal	A8 hexadecimal
SS12	89 hexadecimal	LS12R	90 hexadecimal	A9 hexadecimal
SS13	8A hexadecimal	LS13R	90 hexadecimal	AA hexadecimal
SS14	8B hexadecimal	LS14R	90 hexadecimal	AB hexadecimal
SS15	8C hexadecimal	LS15R	90 hexadecimal	AC hexadecimal

*LS N R* indicates that subsequent bytes in the range 160 to 255 decimal (A0 to FF hexadecimal) represent characters in the supplementary graphic character set. A locking shift remains in effect until another locking shift, the end of a field, or the end of a record, whichever comes first. A locking shift has no effect on bytes in the range 0 to 159 decimal (00 to 9F hexadecimal).

**Word Processing Introducer**

The word processing introducer (WPI) is a prefix for the formatting control sequences created by the Stratus Word Processing Editor. Its rank is 145 decimal (91 hexadecimal). Each byte of a WPI sequence (except for *WPI* itself) must be less than 128 decimal (80 hexadecimal).

## Binary Data Introducer

The binary data introducer (BDI) indicates the presence of binary data in text files. Its rank is 147 decimal (93 hexadecimal). For terminals using the window terminal driver, `s$read_raw` uses the BDI to introduce function keys, generic input requests, and the first raw byte of a string that did not map to an internal character code, function key, or generic input request. The format of a BDI sequence is as follows:

```
BDI binary_data_type binary_data_length data
```

The value of *binary\_data\_type* determines the format of the binary data. For terminals, the value is 130 decimal (82 hexadecimal). (See [Chapter 4](#) for a description of the BDI sequences used for generic input requests and function keys.)

## Right-hand Supplementary Graphic Character Set

This set is one of the 15 supplementary graphic character sets, which are defined as Stratus internal character sets.

Character-Set Number	Graphic Character Set
1	Right-hand part of Latin alphabet No. 1
2	Japanese Kanji Graphic Character Set (double-byte)
3	Japanese Katakana Graphic Character Set
4	Korean Hangul Graphic Character Set (double-byte)
5	Simplified Chinese Graphic Character Set
6	Chinese Graphic Character Set - Part 1
7	Chinese Graphic Character Set - Part 2
8	User-Defined Double-Byte Character Set

The right-hand part of the Latin alphabet No. 1 character set includes characters that must be added to ASCII in order to complete the character set. (See [Appendix F](#) for a list of characters included in this character set.)

The Chinese Graphic Character Set is one standard character set, but occupies two character sets (Part 1 and Part 2). This is because it is structured as two double-byte character sets.

The User-Defined Double-Byte Character Set allows you to define your own double-byte character set. This character set can be used in conjunction with the Chinese Graphic Character Set to provide additional characters. (See the *National Language Support User's Guide* (R212) for more information about character sets.)

---

## Appendix C

# Terminal-Type Subroutines

This appendix identifies the subroutines that your application program can use to retrieve information about an installed terminal type (TTP) or to change a TTP. These subroutines are described in detail in the manual *VOS Communications Software: Defining a Terminal Type* (R096).

There are eight types of TTP subroutines.

- **Configuration subroutines.** These subroutines return information about the display size and other aspects of terminal configuration. See [Table C-1](#) for a list of these subroutines.
- **Keyboard subroutines.** These subroutines return information about the types of function keys or function-key/shift-modifier combinations that are defined in the keyboard database of the TTP. See [Table C-2](#) for a list of these subroutines.
- **Generic input subroutines.** These subroutines return information about the generic input requests supported by the TTP. See [Table C-3](#) for a list of these subroutines.
- **Character-translation subroutines.** These subroutines return information about the internal characters that are supported by a specific TTP. See [Table C-4](#) for a list of these subroutines.
- **Generic output subroutines.** These subroutines return information about the generic output requests that are supported by a specific TTP. See [Table C-5](#) for a list of these subroutines.
- **Output-capability subroutines.** These subroutines return information about the output capabilities of a specific TTP. Output capabilities implement generic output requests. See [Table C-6](#) for a list of these subroutines.
- **Display-attribute subroutines.** These subroutines return information about the display attributes (for example, boldface). See [Table C-7](#) for a list of these subroutines.
- **Database-cleanup subroutine.** The `s$http_free_db` subroutine cleans up the working copies of TTP databases produced by other subroutines. This subroutine frees the heap storage that is allocated for a TTP database. (An application program allocates heap storage by using one of the subroutines that returns database identifiers.)

**Table C-1. TTP Configuration Subroutines**

Subroutine	Description
<code>s\$http_get_configuration_db</code>	Given the device's port ID, returns the database ID of the configuration database of that port's TTP
<code>s\$http_get_configuration_info</code>	Returns global configuration information from the configuration database of a TTP
<code>s\$http_get_name_configuration_db</code>	Given a TTP name, returns the database ID of the configuration database of that TTP
<code>s\$http_get_setup_info</code>	Returns the height, width, and number of pages of a specified configuration setup of a TTP
<code>s\$http_list_setups</code>	Returns the names of the configuration setups in the configuration database of a TTP

**Table C-2. TTP Keyboard Subroutines**

Subroutine	Description
<code>s\$http_get_defined_keys</code>	Returns an array of bytes indicating which function keys and function-key/shift-modifier combinations are defined in the keyboard database of a TTP
<code>s\$http_get_key_legends</code>	Returns the key legends for the shift modifiers and function keys defined in a TTP
<code>s\$http_get_keyboard_db</code>	Given the device's port ID, returns the database ID of the keyboard database of that port's TTP
<code>s\$http_get_legend</code>	Given a key code, returns the key legend of the corresponding function key or shift modifier (if any)
<code>s\$http_get_name_keyboard_db</code>	Given a TTP name, returns the database ID of the keyboard database of that TTP



**Table C-3. TTP Generic Input Subroutines**

Subroutine	Description
<code>s\$http_get_defined_input_reqs</code>	Returns an array of bytes indicating which generic input requests are defined for a specified input section in the generic input database of a TTP
<code>s\$http_get_defined_reqs_flags</code>	Returns a string indicating which generic input requests are defined for a specified input section in the generic input database of a TTP
<code>s\$http_get_generic_req_components</code>	Returns a list of the function-key codes and raw bytes that form a generic input request
<code>s\$http_get_generic_req_legend</code>	Returns a string consisting of the key legends of the function keys that form a generic input request (legends in the string are separated by commas)
<code>s\$http_get_input_db</code>	Given the device's port ID, returns the database ID of the generic input database of that port's TTP
<code>s\$http_get_input_sections</code>	Returns the names of the input sections in the generic input database of a TTP
<code>s\$http_get_name_input_db</code>	Given a TTP name, returns the database ID of the generic input database of that TTP
<code>s\$http_input_sec_exists</code>	Verifies that a specified input section is defined for a TTP

**Table C-4. TTP Character-Translation Subroutines**

Subroutine	Description
<code>s\$http_get_character_support</code>	Determines whether a character in one of the operating system's internal character sets is supported by a TTP on input or on output
<code>s\$http_get_input_chars_db</code>	Given the device's port ID, returns the database ID of the input graphic character-translation database of that port's TTP
<code>s\$http_get_name_input_chars_db</code>	Given a TTP name, returns the database ID of the input graphic character-translation database of that TTP
<code>s\$http_get_name_output_chars_db</code>	Given a TTP name, returns the database ID of the output graphic character-translation database of that TTP
<code>s\$http_get_output_chars_db</code>	Given the device's port ID, returns the database ID of the output graphic character-translation database of that port's TTP
<code>s\$http_scan_supported</code>	Scans a character string to determine whether it contains one or more characters not supported by a TTP

**Table C-5. TTP Generic Output Subroutines**

Subroutine	Description
<code>s\$ttp_get_name_output_chars_db</code>	Given a TTP name, returns the database ID of the output graphic character-translation database of that TTP
<code>s\$ttp_get_name_output_db</code>	Given a TTP name, returns the database ID of the generic output database of that TTP
<code>s\$ttp_get_output_db</code>	Given the device's port ID, returns the database ID of the generic output database of that port's TTP
<code>s\$ttp_get_supported_output_seqs</code>	Returns an array of bytes indicating which generic output sequences are defined in the generic output database of a TTP. If the port is attached with the <code>hold_attached</code> switch and/or the <code>hold_open</code> switch turned on, your application program <b>must</b> call this subroutine again after calling any <code>s\$control</code> opcode that affects the arrangement of the subwindows (for example, the <code>s\$control</code> opcode <code>TERM_SET_SUBWIN_TO_TOP</code> (2025), <code>TERM_CREATE_SUBWIN</code> (2038), or <code>TERM_SET_SUBWIN_BOUNDS</code> (2067)). Any change (such as the arrangement of overlapping subwindows) may affect the information that is returned. If the current subwindow is at all obscured, requests such as <code>DELETE_LINES</code> will not be executed because the subroutine will not recognize them.
<code>s\$ttp_get_supported_seqs_flags</code>	Returns a string indicating which generic output sequences are defined in the generic output database of a TTP

**Table C-6. TTP Output-Capability Subroutines** (Page 1 of 2)

Subroutine	Description
<code>s\$ttp_get_area_attr_seq</code>	Returns the sequence of bytes that the operating system must send to the device to change the settings of specified area attributes
<code>s\$ttp_get_capability_db</code>	Given the device's port ID, returns the database ID of the capability database of that port's TTP
<code>s\$ttp_get_mixed_attr_seq</code>	Returns the sequence of bytes that the operating system must send to the device to change the settings of specified mode <b>and</b> area attributes
<code>s\$ttp_get_mode_attr_seq</code>	Returns the sequence of bytes that the operating system must send to the device to change the settings of specified mode attributes
<code>s\$ttp_get_name_capability_db</code>	Given a TTP name, returns the database ID of the capability database of that TTP
<code>s\$ttp_get_output_cap</code>	Returns the sequence of bytes that the operating system must send to the device to handle the specified output capability
<code>s\$ttp_get_output_cap_support</code>	Determines whether an output capability is supported by a TTP
<code>s\$ttp_graphics_to_ext</code>	Returns the sequence of bytes that the operating system must send to the device to display a line-graphics character
<code>s\$ttp_reset_output_state</code>	Reinitializes the output state for character translation and display attributes in the specified capability database
<code>s\$ttp_restore_output_state</code>	Restores the capability database's external-character-set-translation and mode-attribute output states from the temporary copy created by the most recent call to the subroutine <code>s\$ttp_save_output_state</code>
<code>s\$ttp_save_output_state</code>	Saves a temporary copy of the capability database's external-character-set-translation and mode-attribute output states for future use by the subroutine <code>s\$ttp_restore_output_state</code>

**Table C-6. TTP Output-Capability Subroutines** (Page 2 of 2)

Subroutine	Description
<code>s\$ttp_set_graphics_mode</code>	Sets a switch in the capability database that determines whether the subroutine <code>s\$ttp_translate_to_ext</code> interprets the single-byte characters whose codes are in the range 00 to FF hexadecimal as control characters or line-graphics characters

**Table C-7. TTP Display-Attribute Subroutines**

Subroutine	Description
<code>s\$ttp_get_attribute_info</code>	Returns attribute information from the attributes database of a TTP
<code>s\$ttp_get_attributes_db</code>	Given the device's port ID, returns the database ID of the attributes database of that port's TTP
<code>s\$ttp_get_generic_attr_db</code>	Given the device's port ID, returns the database ID of the generic attributes database of that port's TTP
<code>s\$ttp_get_name_attributes_db</code>	Given a TTP name, returns the database ID of the attributes database of that TTP
<code>s\$ttp_get_name_generic_attr_db</code>	Given a TTP name, returns the database ID of the generic attributes database of that TTP
<code>s\$ttp_set_output_state</code>	Updates the external-character-set-translation and mode-attribute output states for the specified port, to reflect the current state of a specified capability database
<code>s\$ttp_translate_to_ext</code>	Returns the sequence of bytes that the operating system must send to the device to display the characters represented by a specified string of internal character codes



---

## Appendix D

### Sample Program

This appendix contains a sample window terminal program named `chat`. The program, which is written in C, allows a local terminal user (`party_1`) to establish interactive communications with a remote terminal user (`party_2`) by creating multiple primary windows and subwindows.

The program uses no-wait mode and is designed to retrieve processed raw input (using `s$read_raw` and generic input mode) and send formatted sequential output (using `s$seq_write_partial`). The processed raw input/formatted sequential output combination is part of the application-managed I/O programming approach described in [Chapter 4](#). The combination allows the window terminal software to map all input requests and output requests to the appropriate terminal-type databases and to process the requests accordingly.

The sample program uses various input requests and output requests, as well as various `s$control` opcodes. Specifically, the program includes the following window terminal opcodes. [Chapter 8](#) describes these window terminal opcodes.

```
TERM_CREATE_SUBWIN (2038)
TERM_GET_SCREEN_HEIGHT (2024)
TERM_GET_SCREEN_WIDTH (2028)
TERM_MOVE_PWIN_TO_BOTTOM (2086)
TERM_MOVE_PWIN_TO_TOP (2087)
TERM_SET_CURRENT_SUBWIN (2091)
TERM_STATUS_MSG_CHANGE (2096)
TERM_WRITE_SYSTEM_MESSAGE (2101)
```

```
/* The program uses the following include files. Chapter 5
contains pertinent input requests from video_request_defs.incl.c
and output requests from output_sequence_codes.incl.c. The
s$control section of Chapter 8 lists the opcodes in
window_control_opcodes.incl.c and contains opcode structures
from window_term_info.incl.c. The get_port_info.incl.c file is a
system include file. The file <string.incl.c> is a C include file
used by memcpy and memset. */
```

```
#include "get_port_info.incl.c"
#include "window_control_opcodes.incl.c"
#include "output_sequence_codes.incl.c"
#include "video_request_defs.incl.c"
#include "window_term_info.incl.c"
#include <string.incl.c>
```

```
/* The program uses the following external entries. */
```

```
extern void  s$attach_port ();
extern void  s$close ();
extern void  s$control ();
extern void  s$detach_port ();
extern void  s$error ();
extern void  s$get_port_info ();
extern void  s$get_user_name ();
extern void  s$open ();
extern void  s$parse_command ();
extern void  s$read_event ();
extern void  s$read_raw ();
extern void  s$seq_write_partial ();
extern void  s$set_no_wait_mode ();
extern void  s$set_wait_mode ();
extern void  s$wait_event ();
```

```
/* The program uses the following external variables. */
```

```
extern short int  e$caller_must_wait;
extern short int  e$end_of_file;
```



```

/* The program includes the following local entries. */

void      establish_chat ();
void      look_for_chat_input ();
short int make_subwindows ();
void      process_input ();
void      store_party_gone_msg ();
void      store_signal_party_msg ();


/* The program uses the following constants. The s$control
opcodes are not defined here since they are defined in the
include file. */

#define    BDI                                147
#define    BDI_HEADER_LEN                    4
#define    BOTTOM                             0
#define    CHAR_VAR(MAXL)                    struct { short int length;
char text[MAXL]; }
#define    COL_ZERO                           0
#define    FALSE                             0
#define    HOLD_OPEN_ATTACHED                 3
#define    NO_HOLD_OPEN_ATTACHED              0
#define    NOT_TITLE                          0
#define    NUL_STR                            &(char_varying(1)) ""
#define    READ_SIZE                          10
#define    ROW_ZERO                           0
#define    SPACE_CHAR                         32
#define    TITLE                             1
#define    TOP                                1
#define    TRUE                               1
#define    WRITE_SIZE                         100


/* The program includes the following port states. */

#define    NONE                               0
#define    ATTACHED                           1
#define    OPEN                                2
#define    MY_READ                             3
#define    MY_WRITE                           4
#define    YOUR_WRITE                          5

```

```
/* The program uses the following structures for the two parties.
For the first structure, field names beginning with "my" refer
to each party's local terminal and field names beginning
with "your" refer to the remote party's terminal.
The second structure is used by s$read_raw to return
generic input requests. */
```

```
typedef struct chat_struct
{
    short int    my_port;
    short int    my_state;
    short int    my_subwin_id;
    short int    your_port;
    short int    your_state;
    short int    your_subwin_id;
    char         read_buffer[READ_SIZE];
    short int    read_len;
    char         write_buffer[WRITE_SIZE];
    short int    write_len;
} CHAT_STRUCT;
```

```
typedef struct bdi_header
{
    char         introducer;
    char         type;
    short int    len;
    short int    req_no;
} BDI_HEADER;
```

```
/* The beginning of the chat program. */
```

```
void chat()
{
    short int    code;
    char_varying(66) device_path = "";
    char_varying(27) device_str =
"device_path:device_name,req";
    char_varying(3) end_str      = "end";
    char_varying(6) my_name      = "chat";

    s$parse_command(    &my_name,
                        &code,
```

---

```

                                &device_str,
                                &device_path,
                                &end_str);

    if (code)
        return;

    establish_chat(              device_path,
                                &code);

    if (code)
        s$error(                &code,
                                &my_name,
                                NUL_STR);

    return;

}
/* End of chat */

/* This segment contains the establish_chat routine. */

void establish_chat (party_2_device_name, code_ptr)

char_varying(66)              party_2_device_name;
short int                      *code_ptr;

{
    short int                  access_mode          = 1;
    short int                  bottom_id;
    short int                  chat_flag            = FALSE;
    MESSAGE_CHANGE_V2          clear_status;
    short int                  code                  = 0;
    short int                  default_port          = 5;
    long int                   event_count[2];
    long int                   event_id[2];
    short int                  event_num             = 0;
    long int                   event_status[2];
    short int                  hold_sw;
    short int                  ignore_code           = 0;
    short int                  ignore_flag           = FALSE;
    short int                  io_type               = 4;
    CHAR_VAR(100)              message;
    short int                  num_events            = 2;

```

```
short int          opcode;
CHAT_STRUCT        party_1;
short int          party_1_bottom_title_id;
char_varying(66)   party_1_device_name;
short int          party_1_top_title_id;
CHAT_STRUCT        party_2;
short int          party_2_bottom_title_id;
short int          party_2_top_title_id;
short int          port;
struct get_port_info port_info;
short int          status_msg_port;
short int          status_msg_state      = NONE;
long int           timeout               = -1;
short int          top_id;
short int          zero                  = 0;

/* This segment initializes the structures to be used by both
parties. */

party_1.my_state = NONE;
party_1.your_state = NONE;
party_1.read_len = 0;
party_1.write_len = 0;
memset(&party_1.read_buffer[0], 0, READ_SIZE);
memset(&party_1.write_buffer[0], 0, WRITE_SIZE);
party_2.my_state = NONE;
party_2.your_state = NONE;
party_2.read_len = 0;
party_2.write_len = 0;
memset(&party_2.read_buffer[0], 0, READ_SIZE);
memset(&party_2.write_buffer[0], 0, WRITE_SIZE);

/* The program attaches a port to the remote terminal (party_2).
The port is opened with the hold_open and hold_attached switches
turned off, which allows the window terminal software to
redisplay any formatted sequential output that becomes
overlapped temporarily (for example, by another subwindow or
primary window). */

hold_sw = NO_HOLD_OPEN_ATTACHED;
s$attach_port(      NUL_STR,
                   &party_2_device_name,
                   &hold_sw,
```

```
                                &port,
                                &code);
    if (!code)
    {
        party_2.my_port = port;
        party_2.my_state = ATTACHED;
        party_1.your_port = port;
        party_1.your_state = ATTACHED;
    }
    else
        goto ERROR_RETURN;

/* The program opens the remote terminal port,
thereby creating a primary window for the remote terminal. */

    s$open(
                                &port,
                                &zero,
                                &zero,
                                &io_type,
                                &zero,
                                &access_mode,
                                NUL_STR,
                                &code);
    if (!code)
    {
        party_2.my_state = OPEN;
        party_1.your_state = OPEN;
    }
    else
        goto ERROR_RETURN;

/* The program moves the new remote primary window to the
bottom of the primary window stack so that the new window does
not interfere with the remote user's current activity. */

    opcode = TERM_MOVE_PWIN_TO_BOTTOM;
    s$control(
                                &port,
                                &opcode,
                                &zero,
                                &code);
    if (code)
        goto ERROR_RETURN;
```

```
/* The program attaches another port to the local terminal.
This port is opened with the hold_open and hold_attached
switches turned off, which allows the window terminal
software to redisplay any formatted sequential output
that becomes overlapped (for example, by another subwindow
or primary window). */
```

```
port_info.version = 1;
s$get_port_info(    &default_port,
                   &port_info,
                   &code);

if (code)
    goto ERROR_RETURN;

party_1_device_name =
    (char_varying(66)) port_info.path_name;

hold_sw = NO_HOLD_OPEN_ATTACHED;
s$attach_port(      NUL_STR,
                   &party_1_device_name,
                   &hold_sw,
                   &port,
                   &code);

if (!code)
{
    party_1.my_port = port;
    party_1.my_state = ATTACHED;
    party_2.your_port = port;
    party_2.your_state = ATTACHED;
}
else
    goto ERROR_RETURN;
```

```
/* The program opens the port it just attached.
This gives the local terminal a chat primary window when it
wants to begin communicating with the remote terminal. */
```

```
s$open(            &port,
                  &zero,
                  &zero,
                  &io_type,
                  &zero,
```

```
                                &access_mode,
                                NUL_STR,
                                &code);

    if (!code)
    {
        party_1.my_state = OPEN;
        party_2.your_state = OPEN;
    }
    else
        goto ERROR_RETURN;

/* The program creates four local subwindows within the
new primary window (a party_2 title subwindow that
identifies the remote terminal, a party_2 subwindow to
contain messages from the remote terminal, a party_1 title
subwindow that identifies the local terminal, and a party_1
subwindow that echoes all messages sent to the remote
terminal). */

    code = make_subwindows (party_1.my_port, TOP,
                            party_2_device_name,
                            &party_1_top_title_id, &top_id);

    if (!code)
        party_2.your_subwin_id = top_id;
    else
        goto ERROR_RETURN;

    code = make_subwindows (party_1.my_port, BOTTOM,
                            party_1_device_name,
                            &party_1_bottom_title_id,
                            &bottom_id);

    if (!code)
        party_1.my_subwin_id = bottom_id;
    else
        goto ERROR_RETURN;
```

```
/* The program creates four remote subwindows within the
primary window (a party_1 title subwindow that
identifies the local terminal, a party_1 subwindow to
contain messages from the local terminal, a party_2 title
subwindow that identifies the remote terminal, and a party_2
subwindow that echoes all messages sent to the local
terminal). */
```

```
code = make_subwindows (party_2.my_port, TOP,
                        party_1_device_name,
                        &party_2_top_title_id, &top_id);
if (!code)
    party_1.your_subwin_id = top_id;
else
    goto ERROR_RETURN;
```

```
code = make_subwindows (party_2.my_port, BOTTOM,
                        party_2_device_name,
                        &party_2_bottom_title_id,
                        &bottom_id);
if (!code)
    party_2.my_subwin_id = bottom_id;
else
    goto ERROR_RETURN;
```

```
/* The program puts the local and remote ports into
no-wait mode. */
```

```
s$set_no_wait_mode(      &party_1.my_port,
                        &event_id[0],
                        &code);
if (code)
    goto ERROR_RETURN;

s$set_no_wait_mode(      &party_2.my_port,
                        &event_id[1],
                        &code);
if (code)
    goto ERROR_RETURN;
```



---

```

/* The program gets the event counts for both ports. */

    s$read_event(          &event_id[0],
                           &event_count[0],
                           &event_status[0],
                           &code);

    if (code)
        goto ERROR_RETURN;

    s$read_event(          &event_id[1],
                           &event_count[1],
                           &event_status[1],
                           &code);

    if (code)
        goto ERROR_RETURN;

/* The main processing loop begins. */

    while (TRUE)
    {

/* The program handles remote (party_2) processing. */

        if (party_1.your_state == OPEN)
            look_for_chat_input (&party_2, FALSE, &ignore_flag,
                                &code);

        if (code == e$end_of_file)
        {

/* If party_2 presses the CANCEL key to end the session with
party_1, the program clears the party_2 status message before
closing and detaching the port. */

            s$set_wait_mode(    &party_2.my_port,
                                &code);

            opcode = TERM_STATUS_MSG_CHANGE;
            clear_status.version = 2;
            clear_status.status_flag = 0;
            s$control(          &party_2.my_port,
                                &opcode,
                                &clear_status,
                                &code);

```

```
s$close(                &party_2.my_port,
                        &code);

s$detach_port(          &party_2.my_port,
                    &code);

party_2.my_state = NONE;
party_1.your_state = NONE;

opcode = TERM_SET_CURRENT_SUBWIN;
s$control(              &party_1.my_port,
                        &opcode,
                        &party_1_top_title_id,
                        &code);

if (code)
    goto ERROR_RETURN;

store_party_gone_msg (&message);

/* The program sends a message to notify party_1 that party_2 is
gone. */

s$seq_write_partial( &party_1.my_port,
                    &message.length,
                    &message.text[0],
                    &code);

if (code)
    goto ERROR_RETURN;
}
/* End of  if (code == e$end_of_file). */

if (code && code != e$caller_must_wait)
    break;

/* The program handles local (party_1) work. */

look_for_chat_input (&party_1, TRUE, &chat_flag, &code);

if (chat_flag)
{
```

---

/\* If party\_2 has not made the chat primary window the current primary window, the program attaches and opens another remote primary window. The sole purpose of this new primary window is to allow the local terminal to send a request-to-chat message to the remote terminal's status line. After the message is sent, the primary window is deleted. \*/

```

hold_sw = HOLD_OPEN_ATTACHED;
s$attach_port(      NUL_STR,
                   &party_2_device_name,
                   &hold_sw,
                   &status_msg_port,
                   &code);

if (code)
    goto ERROR_RETURN;

status_msg_state = ATTACHED;
s$open(            &status_msg_port,
                  &zero,
                  &zero,
                  &io_type,
                  &zero,
                  &access_mode,
                  NUL_STR,
                  &code);

if (code)
    goto ERROR_RETURN;

status_msg_state = OPEN;
store_signal_party_msg (&message);
opcode = TERM_WRITE_SYSTEM_MESSAGE;
s$control(         &status_msg_port,
                  &opcode,
                  &message,
                  &code);

if (code)
    goto ERROR_RETURN;

s$close(           &status_msg_port,
                  &code);
status_msg_state = ATTACHED;
if (code)
    goto ERROR_RETURN;

```

```

        s$detach_port(      &status_msg_port,
                           &code);
        if (code)
            goto ERROR_RETURN;
        status_msg_state = NONE;
        chat_flag = FALSE;
    }
/* End of  if (chat_flag). */

/* If there is no input available, s$read_raw returns the error
e$caller_must_wait.  The program then waits on the event. */

        if (code && code != e$caller_must_wait)
            break;

        if (party_1.your_state == OPEN)
            s$wait_event(      &num_events,
                              &event_id,
                              &event_count,
                              &timeout,
                              &event_num,
                              &code);

/* If party_2 is gone, the program waits for any keystroke to
terminate
the session. */

        else
        {
            num_events = 1;
            event_num = 0;
            s$wait_event(      &num_events,
                              &event_id,
                              &event_count,
                              &timeout,
                              &event_num,
                              &code);

        }
        if (code)
            goto ERROR_RETURN;
    }
/* End of  while (TRUE). */

```

---

```

/* The program then closes and detaches the ports. */

ERROR_RETURN:
    if (code == e$end_of_file)
        *code_ptr = 0;
    else
        *code_ptr = code;

    switch (party_2.my_state)
    {
    case MY_READ:
    case MY_WRITE:
    case YOUR_WRITE:
    case OPEN:
        s$set_wait_mode(    &party_2.my_port,
                           &ignore_code);
        s$close(           &party_2.my_port,
                           &ignore_code);

    case ATTACHED:
        s$detach_port(      &party_2.my_port,
                           &ignore_code);

    default: ;
    }

    switch (party_1.my_state)
    {
    case MY_READ:
    case MY_WRITE:
    case YOUR_WRITE:
    case OPEN:
        s$set_wait_mode(    &party_1.my_port,
                           &ignore_code);
        s$close(           &party_1.my_port,
                           &ignore_code);

    case ATTACHED:
        s$detach_port(      &party_1.my_port,
                           &ignore_code);

    default: ;
    }

```

```
        switch (status_msg_state)
        {
        case OPEN:
            s$close(                &status_msg_port,
                                &ignore_code);

        case ATTACHED:
            s$detach_port(          &status_msg_port,
                                &ignore_code);

        default: ;
        }

        return;

    }
    /* End of establish_chat. */

    /* This segment contains the look_for_chat_input routine. */

    void look_for_chat_input (party_ptr, signal_allowed_flag,
                            signal_party_ptr, code_ptr)

    CHAT_STRUCT          *party_ptr;
    short int            signal_allowed_flag;
    short int            *signal_party_ptr;
    short int            *code_ptr;

    {
        short int        code        = 0;
        short int        read_len    = READ_SIZE;
        short int        opcode;

        while (TRUE)
        {
            switch (party_ptr->my_state)
            {
            case OPEN:
                party_ptr->my_state = MY_READ;
            }
        }
    }
}
```

---

```
/* The program calls s$read_raw to retrieve input from the
party's local terminal port. */
```

```
case MY_READ:
    s$read_raw(                &party_ptr->my_port,
                              &read_len,
                              &party_ptr->read_len,
                              &party_ptr->read_buffer[0],
                              &code);

    if (code)
        break;

    if (party_ptr->your_state != OPEN)
    {
        code = e$end_of_file;
        break;
    }

    process_input              (&party_ptr->read_buffer[0],
                              party_ptr->read_len,
                              &party_ptr->write_buffer[0],
                              &party_ptr->write_len,
                              signal_allowed_flag,
                              signal_party_ptr, &code);

    if (code)
        break;

    party_ptr->my_state = MY_WRITE;
```

```
/* The program echoes the input to the party's local terminal. */
```

```
case MY_WRITE:
    if (party_ptr->write_len > 0)
    {
        opcode = TERM_SET_CURRENT_SUBWIN;
        s$control(    &party_ptr->my_port,
                      &opcode,
                      &party_ptr->my_subwin_id,
                      &code);

        if (code)
            break;
```

```
        s$seq_write_partial(&party_ptr->my_port,
                           &party_ptr->write_len,
                           &party_ptr->write_buffer[0],
                           &code);
    if (code)
        break;
}

party_ptr->my_state = YOUR_WRITE;

/* The program writes the data to the party's remote terminal. */

case YOUR_WRITE:
    if ((party_ptr->your_state == OPEN)
        && (party_ptr->write_len > 0))
    {
        opcode = TERM_SET_CURRENT_SUBWIN;
        s$control(    &party_ptr->your_port,
                     &opcode,
                     &party_ptr->your_subwin_id,
                     &code);

        if (code)
            break;

        s$seq_write_partial(&party_ptr->your_port,
                           &party_ptr->write_len,
                           &party_ptr->write_buffer[0],
                           &code);

        if (code)
            break;

        party_ptr->write_len = 0;
    }
    party_ptr->my_state = MY_READ;

default: ;

}
/* End of  switch (party_ptr->my_state). */
```



---

```

        if (code)
            break;

    }
/* End of while (TRUE). */
*code_ptr = code;
    return;

}
/* End of look_for_chat_input. */

/* The program makes each bottom subwindow in the local and
remote chat window the current subwindow. For every
keystroke processed, the program performs a read from the
current subwindow, prepares the output, echoes it, then
changes to the other party's top subwindow to write the
output. */

short make_subwindows (port, top_flag, dev_name,
                      title_id_ptr, id_ptr)

short int port;
short int top_flag;
CHAR_VAR(66) dev_name;
short int *title_id_ptr;
short int *id_ptr;

{

    short int code          = 0;
    short int height;
    short int id            = 0;
    short int len;
    short int num_horiz;
    short int opcode;
    char      str[100];
    short int str_len       = 0;
    short int title_id      = 0;
    short int upper_left_v;
    short int width;

    SUBWINDOW_INFO_V1      subwin;

```

```
/* The program gets the height and width of the window. */

opcode = TERM_GET_SCREEN_HEIGHT;
s$control(      &port,
                &opcode,
                &height,
                &code);

if (code)
    goto ERROR_RETURN;

height = height >> 1;

opcode = TERM_GET_SCREEN_WIDTH;
s$control(      &port,
                &opcode,
                &width,
                &code);

if (code)
    goto ERROR_RETURN;

opcode = TERM_CREATE_SUBWIN;

subwin.version = 1;
subwin.upper_left_corner_h = 0;
subwin.width = width;

/* The program creates the title subwindow. */

if (top_flag)
    upper_left_v = 0;
else
    upper_left_v = height;

subwin.upper_left_corner_v = upper_left_v;
subwin.height = 1;

s$control(      &port,
                &opcode,
                &subwin,
                &code);

if (code)
    goto ERROR_RETURN;

title_id = subwin.subwin_id;
```

```

/* The program writes the title. */

    num_horiz = (width - (dev_name.length + 13)) >> 1;

    str[str_len++] = GSI;
    str[str_len++] = ENTER_GRAPHICS_MODE_SEQ;
    len = num_horiz;
    memset(&str[str_len], 11 /* horizontal bar */, len);
    str_len += len;
    str[str_len++] = GSI;
    str[str_len++] = LEAVE_GRAPHICS_MODE_SEQ;
    str[str_len++] = ' ';

    len = dev_name.length;
    memcpy (&str[str_len], &dev_name.text[0], len);
    str_len += len;

    str[str_len++] = ' ';
    str[str_len++] = GSI;
    str[str_len++] = ENTER_GRAPHICS_MODE_SEQ;
    len = num_horiz;
    memset(&str[str_len], 11 /* horizontal bar */, len);
    str_len += len;
    str[str_len++] = GSI;
    str[str_len++] = LEAVE_GRAPHICS_MODE_SEQ;
    memcpy(&str[str_len], " use {help}", 11);
    str_len += 11;

    s$seq_write_partial(      &port,
                              &str_len,
                              &str,
                              &code);

    if (code)
        goto ERROR_RETURN;

/* The program creates the appropriate subwindow. */

    if (top_flag)
        upper_left_v = 1;
    else
        upper_left_v = height + 1;

    subwin.upper_left_corner_v = upper_left_v;
    subwin.height = height - 1;
vfileject

```

```
s$control(          &port,
                  &opcode,
                  &subwin,
                  &code);

if (code)
    goto ERROR_RETURN;

id = subwin.subwin_id;

/* The program positions the cursor on row 0, column 0 of the
subwindow. */

str_len = 0;
str[str_len++] = GSI;
str[str_len++] = POSITION_CURSOR_SEQ;
str[str_len++] = ROW_ZERO;
str[str_len++] = COL_ZERO;

s$seq_write_partial(&port,
                  &str_len,
                  &str,
                  &code);

if (code)
    goto ERROR_RETURN;

*title_id_ptr = title_id;
*id_ptr = id;

ERROR_RETURN:
    return (code);

}
/* End of make_subwindows. */
```

---

```

/* This segment contains the process_input routine. */

void process_input (in_buf, in_len, out_buf, out_len_ptr,
                   signal_allowed_flag, signal_party_ptr,
                   code_ptr)

char            in_buf[READ_SIZE];
short int       in_len;
char            out_buf[WRITE_SIZE];
short int       *out_len_ptr;
short int       signal_allowed_flag;
short int       *signal_party_ptr;
short int       *code_ptr;

{
    BDI_HEADER    bdi;
    unsigned char  c;
    short int      i          = 0;
    char           *out_ptr    = out_buf;
    short int      out_len     = 0;
    short int      request;

    *out_len_ptr = 0;
    *code_ptr = 0;

    for (i = 0;
         (i < in_len) && (out_len < WRITE_SIZE);)
    {
        c = in_buf[i];

        switch (c)
        {
        case BDI:
            memcpy (&bdi, &in_buf[i],
                   BDI_HEADER_LEN + sizeof(request));
            request = bdi.req_no;
            i += bdi.len + BDI_HEADER_LEN;

            if (request == RETURN_REQ ||
                request == ENTER_REQ)
            {
                if ((out_len + 2) > WRITE_SIZE)
                    break;
                *out_ptr++ = GSI;
                *out_ptr++ = NEW_LINE_SEQ;
            }
        }
    }
}

```

```
        out_len += 2;
    }
    else if (request == ERASE_CHAR_REQ)
    {
        if ((out_len + 5) > WRITE_SIZE)
            break;
        *out_ptr++ = GSI;
        *out_ptr++ = LEFT_SEQ;
        *out_ptr++ = SPACE_CHAR;
        *out_ptr++ = GSI;
        *out_ptr++ = LEFT_SEQ;
        out_len += 5;
    }
    else if (request == LEFT_REQ)
    {
        if ((out_len + 2) > WRITE_SIZE)
            break;
        *out_ptr++ = GSI;
        *out_ptr++ = LEFT_SEQ;
        out_len += 2;
    }
    else if (request == RIGHT_REQ)
    {
        if ((out_len + 2) > WRITE_SIZE)
            break;
        *out_ptr++ = GSI;
        *out_ptr++ = RIGHT_SEQ;
        out_len += 2;
    }
    else if (request == CYCLE_REQ)
    {
        if (signal_allowed_flag)
            *signal_party_ptr = TRUE;
    }
    else if (request == REDISPLAY_REQ)
    {

```

---

```

/* To refresh the display of the primary window, the program must
make a sequential I/O call.  This program calls
s$seq_write_partial to do a harmless position cursor sequence. */

```

```

        *out_ptr++ = GSI;
        *out_ptr++ = POSITION_CURSOR_SEQ;
        *out_ptr++ = 255;
        *out_ptr++ = 255;
        out_len += 4;
    }
    else if (request == HELP_REQ)
    {
        if ((out_len + 26) > WRITE_SIZE)
            break;
        *out_ptr++ = GSI;
        *out_ptr++ = NEW_LINE_SEQ;
        out_len += 2;
        memcpy (out_ptr, "Help: {cancel} to exit",
                22);
out_len += 22;

        out_ptr += 22;
        *out_ptr++ = GSI;
        *out_ptr++ = NEW_LINE_SEQ;
        out_len += 2;
        if (signal_allowed_flag)
        {
            if ((out_len + 37) > WRITE_SIZE)
                break;
            memcpy (out_ptr,
                    "        {cycle} to signal other party",
                    35);
            out_len += 35;
            out_ptr += 35;
            *out_ptr++ = GSI;
            *out_ptr++ = NEW_LINE_SEQ;
            out_len += 2;
        }
    }
    else if (request == CANCEL_REQ
        || request == END_OF_FILE_REQ)
    {
        *code_ptr = e$end_of_file;
    }

    break;

```

```
        default:
            *out_ptr++ = in_buf[i++];
            out_len++;

    }
/* End of  switch (c). */

}
/* End of  for (i= . */

    *out_len_ptr = out_len;

}
/* End of process_input. */

/* This segment contains the store_party_gone_msg routine.
It indicates that party_2 has ended the session. */

void store_party_gone_msg (p_msg_ptr)

CHAR_VAR(100) *p_msg_ptr;
{
    short int          msg_len          = 0;

    p_msg_ptr->text[msg_len++] = GSI;
    p_msg_ptr->text[msg_len++] = NEW_LINE_SEQ;
    p_msg_ptr->text[msg_len++] = GSI;
    p_msg_ptr->text[msg_len++] = INVERSE_VIDEO_ON_SEQ;
    memcpy(&p_msg_ptr->text[msg_len],
        " Other party no longer there. ", 30);
    msg_len +=30;
    memcpy(&p_msg_ptr->text[msg_len],
        "CHAT will terminate with your next keystroke...",
        47);
    msg_len += 47;
    p_msg_ptr->text[msg_len++] = GSI;
    p_msg_ptr->text[msg_len++] = INVERSE_VIDEO_OFF_SEQ;
    p_msg_ptr->length = msg_len;

}
/* End of store_party_gone_msg. */
```



---

```

/* This segment contains the store_signal_party_msg routine.
It notifies party_2 that a new primary window is available. */

void store_signal_party_msg (p_msg_ptr)

CHAR_VAR(100) *p_msg_ptr;
{
    CHAR_VAR(66)      user_name;
    short int         msg_len      = 0;

    s$get_user_name(    &user_name);
    while ((user_name.text[msg_len] != '.')
           && (msg_len < user_name.length))
    {
        p_msg_ptr->text[msg_len] = user_name.text[msg_len];
        msg_len++;
    }
    p_msg_ptr->text[msg_len++] = ':';
    p_msg_ptr->text[msg_len++] = ' ';

    memcpy (&p_msg_ptr->text[msg_len],
           "Hello!  Feel like chatting?  (window mgr and cycle)",
51);
    msg_len += 51;
    p_msg_ptr->length = msg_len;
}
/* End of store_signal_party_msg. */

/* End of program. */

```



---

## Appendix E

### Tools and Commands

The operating system provides a variety of tools and commands that support asynchronous device communications. This appendix lists and briefly describes the tools and commands that allow you to configure, operate, and monitor asynchronous channels or subchannels and devices.

Note that the commands described in this appendix are either privileged commands (that is, you must have system-administration access in order to issue them) or general-user commands (that is, you can issue them if you have general-user access). [Table E-1](#) lists these commands and identifies the manual in which each command is fully documented.

**Table E-1. Tools and Commands Supporting Asynchronous Device Communications**

<b>Tool or Command</b>	<b>Manual Containing Complete Description</b>
compile_terminal_type	<i>VOS Communications Software: Defining a Terminal Type (R096)</i>
install_terminal_type	<i>VOS Communications Software: Defining a Terminal Type (R096)</i>
configure_comm_protocol	<i>VOS System Administration: Configuring a System (R287)</i>
create_table	<i>VOS System Administration: Configuring a System (R287)</i>
broadcast_file	<i>VOS System Administration: Configuring a System (R287)</i>
configure_devices	<i>VOS System Administration: Configuring a System (R287)</i>
change_terminal	<i>VOS System Administration: Configuring a System (R287)</i>
update_channel_info	<i>VOS System Administration: Configuring a System (R287)</i>
configure_async_lines	<i>VOS System Administration: Configuring a System (R287)</i>
set_terminal_parameters	<i>VOS Commands Reference Manual (R098)</i>
display_terminal_parameters	<i>VOS Commands Reference Manual (R098)</i>
list_terminal_types	<i>VOS Commands Reference Manual (R098)</i>
list_devices	<i>VOS Commands Reference Manual (R098)</i>
display_device_info	<i>VOS Commands Reference Manual (R098)</i>
analyze_system	<i>VOS System Analysis Manual (R073)</i>

## Commands That Set Up and Configure the Device

Each asynchronous device that is connected to a module must be associated with an installed driver and have an installed terminal type (TTP) that defines its raw ASCII byte sequences and individual capabilities. The device must also have an installed device table (`devices.table`) entry that defines the device connection to the module. You use the following commands to define the asynchronous device (including its byte sequences, capabilities, operating parameters, and connection type) to both the module and the system.

- **`compile_terminal_type`**. This general-user command takes the source terminal-type definition file (`.ttp`) written for the device and translates it into the binary format used by the TTP database. The command produces an output file that contains the TTP specifications in the format required by the TTP database.
- **`install_terminal_type`**. This privileged command installs in the TTP database the contents of the output file produced by the `compile_terminal_type` command.
- **`configure_comm_protocol`**. This privileged command loads a communications device driver on the current module. It must be issued once per bootload (or included in the module startup file using uncommented command lines) to load the various layers of the window terminal driver.
- **`create_table`**. This privileged command creates the device table by using the `devices.dd` data definition file and the edited `devices.tin` table input file.
- **`broadcast_file`**. This privileged command installs the device table on the current module and on each module in the system.
- **`configure_devices`**. This privileged command allows the operating system to immediately recognize additions, deletions, and changes to entries in the device table. All additions to the device table are recognized automatically; all deletions are recognized if you specify the `-flush` argument of this command. (Changes to existing entries take effect if the device is not currently being used for I/O.) The changes remain in effect until you change the corresponding entries in the device table. Note that before you issue this command with the `-flush` argument on a terminal or printer device, you must make that device a null device by using the `change_terminal` command.
- **`change_terminal`**. This privileged command allows you to change the device type of the device (for example, from a standard asynchronous device to a window terminal device) during the current bootload. It also allows you to change the device type to `null` so that the device can be removed from service temporarily to update the configuration with the `configure_devices` command. Do **not** use this command for remote connections or network connections (such as OS TELNET connections).

- **update\_channel\_info.** This privileged command changes the characteristics of a device for the duration of the current bootload. You can use this command to make a temporary change that is not reflected in the device table. For RS-232-C devices, you can use this command to change characteristics such as the default TTP or the baud rate. For asynchronous device connections that do not use the RS-232-C interface (for example, OS TELNET connections), you may not be able to change certain characteristics. For example, the baud-rate characteristic is not relevant to OS TELNET connections.
- **configure\_async\_lines.** This privileged command, when issued with the `-update` argument, sets or changes the configuration of each RS-232-C subchannel on a K118 16-Line Asynchronous I/O Adapter, as specified in the device table.
- **set\_terminal\_parameters.** This general-user command sets or changes various terminal parameters (for example, the terminal type, pause message, pause-lines limit, and cursor format). The new parameter settings take effect immediately for the current terminal, but only for the duration of the current process. When the port closes, the channel or subchannel reverts to the default parameters specified in the device table or specified by the `update_channel_info` command.

## Commands That Display Device Information

The following commands allow you to display information about the asynchronous devices in the system.

- **display\_terminal\_parameters.** This general-user command identifies the terminal's current parameter settings (for example, the terminal type, pause message, pause-lines limit, and cursor format).
- **list\_terminal\_types.** This general-user command lists the names of the installed terminal types (TTPs) on the current or specified module.
- **list\_devices.** This general-user command lists the names (path names) of all devices of the specified device type (for example, `window_term`) on a module.
- **display\_device\_info.** This general-user command lists the values defined in the device table for the specified device.

## System Analysis Tool

The `analyze_system` subsystem is a tool that allows privileged users to monitor and analyze different parts of the VOS run-time system. The subsystem consists of various administrative requests that aid in debugging and performance tuning and can display information about communications buffers and devices. To access the subsystem and its requests, you must issue the privileged command `analyze_system`.

Some of the `analyze_system` requests allow you to monitor and analyze different aspects of RS-232-C asynchronous communications. You can use the following requests to obtain information about window terminal communications specifically.

- `dump_acb`
- `dump_channel_info`
- `dump_channels`
- `dump_dvt`
- `dump_scb`
- `dump_tcbh`
- `dump_wcb`
- `terminal_meters`

See the *VOS System Analysis Manual* (R073) for a general explanation of how to use the `analyze_system` subsystem and a subset of its requests.





---

## Appendix F

### Default Internal Character Sets

This appendix identifies the default internal character sets used by the operating system. The following default character sets are part of the internal character coding system (see [Appendix B](#)) and are presented in [Table F-1](#):

- the ASCII control character set
- the ASCII graphic character set (the left-hand side of Latin alphabet No. 1)
- the right-hand control character set that is specific to Stratus
- the right-hand side of Latin alphabet No. 1.

**Table F-1. VOS Internal Character Code Set** *(Page 1 of 11)*

Decimal Code	Hex Code	Symbol	Name
0	00	NUL	Null
1	01	SOH	Start of Heading
2	02	STX	Start of Text
3	03	ETX	End of Text
4	04	EOT	End of Transmission
5	05	ENQ	Enquiry
6	06	ACK	Acknowledge
7	07	BEL	Bell
8	08	BS	Backspace
9	09	HT	Horizontal Tabulation
10	0A	LF	Linefeed
11	0B	VT	Vertical Tabulation
12	0C	FF	Form Feed
13	0D	CR	Carriage Return

**Table F-1. VOS Internal Character Code Set** (Page 2 of 11)

<b>Decimal Code</b>	<b>Hex Code</b>	<b>Symbol</b>	<b>Name</b>
14	0E	SO	Shift Out
15	0F	SI	Shift In
16	10	DLE	Data Link Escape
17	11	DC1	Device Control 1
18	12	DC2	Device Control 2
19	13	DC3	Device Control 3
20	14	DC4	Device Control 4
21	15	NAK	Negative Acknowledge
22	16	SYN	Synchronous Idle
23	17	ETB	EOT Block
24	18	CAN	Cancel
25	19	EM	End of Medium
26	1A	SUB	Substitute
27	1B	ESC	Escape
28	1C	FS	File Separator
29	1D	GS	Group Separator
30	1E	RS	Record Separator
31	1F	US	Unit Separator
32	20	SP	Space
33	21	!	Exclamation Mark
34	22	“	Quotation Marks
35	23	#	Number Sign
36	24	\$	Dollar Sign
37	25	%	Percent Sign
38	26	&	Ampersand
39	27	'	Apostrophe

**Table F-1. VOS Internal Character Code Set** (Page 3 of 11)

Decimal Code	Hex Code	Symbol	Name
40	28	(	Opening Parenthesis
41	29	)	Closing Parenthesis
42	2A	*	Asterisk
43	2B	+	Plus Sign
44	2C	,	Comma
45	2D	-	Hyphen, Minus Sign
46	2E	.	Period
47	2F	/	Slant
48	30	0	Zero
49	31	1	One
50	32	2	Two
51	33	3	Three
52	34	4	Four
53	35	5	Five
54	36	6	Six
55	37	7	Seven
56	38	8	Eight
57	39	9	Nine
58	3A	:	Colon
59	3B	;	Semicolon
60	3C	<	Less-Than Sign
61	3D	=	Equals Sign
62	3E	>	Greater-Than Sign
63	3F	?	Question Mark
64	40	@	Commercial “at” Sign
65	41	A	Uppercase A

**Table F-1. VOS Internal Character Code Set** (Page 4 of 11)

Decimal Code	Hex Code	Symbol	Name
66	42	B	Uppercase B
67	43	C	Uppercase C
68	44	D	Uppercase D
69	45	E	Uppercase E
70	46	F	Uppercase F
71	47	G	Uppercase G
72	48	H	Uppercase H
73	49	I	Uppercase I
74	4A	J	Uppercase J
75	4B	K	Uppercase K
76	4C	L	Uppercase L
77	4D	M	Uppercase M
78	4E	N	Uppercase N
79	4F	O	Uppercase O
80	50	P	Uppercase P
81	51	Q	Uppercase Q
82	52	R	Uppercase R
83	53	S	Uppercase S
84	54	T	Uppercase T
85	55	U	Uppercase U
86	56	V	Uppercase V
87	57	W	Uppercase W
88	58	X	Uppercase X
89	59	Y	Uppercase Y
90	5A	Z	Uppercase Z
91	5B	[	Opening Bracket

**Table F-1. VOS Internal Character Code Set** (Page 5 of 11)

<b>Decimal Code</b>	<b>Hex Code</b>	<b>Symbol</b>	<b>Name</b>
92	5C	\	Reverse Slant
93	5D	]	Closing Bracket
94	5E	^	Circumflex
95	5F	_	Underline
96	60	`	Grave Accent
97	61	a	Lowercase a
98	62	b	Lowercase b
99	63	c	Lowercase c
100	64	d	Lowercase d
101	65	e	Lowercase e
102	66	f	Lowercase f
103	67	g	Lowercase g
104	68	h	Lowercase h
105	69	i	Lowercase i
106	6A	j	Lowercase j
107	6B	k	Lowercase k
108	6C	l	Lowercase l
109	6D	m	Lowercase m
110	6E	n	Lowercase n
111	6F	o	Lowercase o
112	70	p	Lowercase p
113	71	q	Lowercase q
114	72	r	Lowercase r
115	73	s	Lowercase s
116	74	t	Lowercase t
117	75	u	Lowercase u

**Table F-1. VOS Internal Character Code Set** (Page 6 of 11)

Decimal Code	Hex Code	Symbol	Name
118	76	v	Lowercase v
119	77	w	Lowercase w
120	78	x	Lowercase x
121	79	y	Lowercase y
122	7A	z	Lowercase z
123	7B	{	Opening Brace
124	7C		Vertical Line
125	7D	}	Closing Brace
126	7E	~	Tilde
127	7F	DEL	Delete
128	80	SS1	Single-Shift 1
129	81	SS4	Single-Shift 4
130	82	SS5	Single-Shift 5
131	83	SS6	Single-Shift 6
132	84	SS7	Single-Shift 7
133	85	SS8	Single-Shift 8
134	86	SS9	Single-Shift 9
135	87	SS10	Single-Shift 10
136	88	SS11	Single-Shift 11
137	89	SS12	Single-Shift 12
138	8A	SS13	Single-Shift 13
139	8B	SS14	Single-Shift 14
140	8C	SS15	Single-Shift 15
141	8D		(Not Assigned)
142	8E	SS2	Single-Shift 2
143	8F	SS3	Single-Shift 3

**Table F-1. VOS Internal Character Code Set** (Page 7 of 11)

<b>Decimal Code</b>	<b>Hex Code</b>	<b>Symbol</b>	<b>Name</b>
144	90	LSI	Locking-Shift Introducer
145	91	WPI	Word Processing Introducer
146	92	XCI	Extended-Control Introducer
147	93	BDI	Binary-Data Introducer
148	94		(Not Assigned)
149	95		(Not Assigned)
150	96		(Not Assigned)
151	97		(Not Assigned)
152	98		(Not Assigned)
153	99		(Not Assigned)
154	9A		(Not Assigned)
155	9B		(Not Assigned)
156	9C		(Not Assigned)
157	9D		(Not Assigned)
158	9E		(Not Assigned)
159	9F		(Not Assigned)
160	A0	NBSP	No Break Space
161	A1	¡	Inverted Exclamation Mark
162	A2	¢	Cent Sign
163	A3	£	British Pound Sign
164	A4	¤	Currency Sign
165	A5	¥	Yen Sign
166	A6		Broken Bar
167	A7	§	Paragraph Sign
168	A8	¨	Dieresis
169	A9	©	Copyright Sign

**Table F-1. VOS Internal Character Code Set** (Page 8 of 11)

Decimal Code	Hex Code	Symbol	Name
170	AA	ª	Feminine Ordinal Indicator
171	AB	«	Left-Angle Quote Mark
172	AC	¬	“Not” Sign
173	AD	SHY	Soft Hyphen
174	AE	®	Registered Trademark Sign
175	AF	—	Macron
176	B0	°	Degree Sign, Ring Above
177	B1	±	Plus-Minus Sign
178	B2	²	Superscript 2
179	B3	³	Superscript 3
180	B4	´	Acute Accent
181	B5	µ	Micro Sign
182	B6	¶	Pilcrow Sign
183	B7	·	Middle Dot
184	B8	¸	Cedilla
185	B9	¹	Superscript 1
186	BA	º	Masculine Ordinal Indicator
187	BB	»	Right-Angle Quote Mark
188	BC	¼	One-Quarter
189	BD	½	One-Half
190	BE	¾	Three-Quarters
191	BF	¿	Inverted Question Mark
192	C0	À	A with Grave Accent
193	C1	Á	A with Acute Accent
194	C2	Â	A with Circumflex
195	C3	Ã	A with Tilde



**Table F-1. VOS Internal Character Code Set** (Page 9 of 11)

<b>Decimal Code</b>	<b>Hex Code</b>	<b>Symbol</b>	<b>Name</b>
196	C4	Ä	A with Dieresis
197	C5	Å	A with Ring Above
198	C6	Æ	Diphthong A with E
199	C7	Ç	C with Cedilla
200	C8	È	E with Grave Accent
201	C9	É	E with Acute Accent
202	CA	Ê	E with Circumflex
203	CB	Ë	E with Dieresis
204	CC	Ì	I with Grave Accent
205	CD	Í	I with Acute Accent
206	CE	Î	I with Circumflex
207	CF	Ï	I with Dieresis
208	D0	Ð	D with Stroke
209	D1	Ñ	N with Tilde
210	D2	Ò	O with Grave Accent
211	D3	Ó	O with Acute Accent
212	D4	Ô	O with Circumflex
213	D5	Õ	O with Tilde
214	D6	Ö	O with Dieresis
215	D7	×	Multiplication Sign
216	D8	Ø	O with Oblique Stroke
217	D9	Ù	U with Grave Accent
218	DA	Ú	U with Acute Accent
219	DB	Û	U with Circumflex
220	DC	Ü	U with Dieresis
221	DD	Ý	Y with Acute Accent

**Table F-1. VOS Internal Character Code Set** (Page 10 of 11)

Decimal Code	Hex Code	Symbol	Name
222	DE	Þ	Uppercase Thorn
223	DF	ß	Sharp s
224	E0	à	a with Grave Accent
225	E1	á	a with Acute Accent
226	E2	â	a with Circumflex
227	E3	ã	a with Tilde
228	E4	ä	a with Dieresis
229	E5	å	a with Ring Above
230	E6	æ	Diphthong a with e
231	E7	ç	c with Cedilla
232	E8	è	e with Grave Accent
233	E9	é	e with Acute Accent
234	EA	ê	e with Circumflex
235	EB	ë	e with Dieresis
236	EC	ì	i with Grave Accent
237	ED	í	i with Acute Accent
238	EE	î	i with Circumflex
239	EF	ï	i with Dieresis
240	F0	ö	Lowercase Eth
241	F1	ñ	n with Tilde
242	F2	ò	o with Grave Accent
243	F3	ó	o with Acute Accent
244	F4	ô	o with Circumflex
245	F5	õ	o with Tilde
246	F6	ö	o with Dieresis
247	F7	÷	Division Sign

**Table F-1. VOS Internal Character Code Set** (Page 11 of 11)

<b>Decimal Code</b>	<b>Hex Code</b>	<b>Symbol</b>	<b>Name</b>
248	F8	ø	o with Oblique Stroke
249	F9	ù	u with Grave Accent
250	FA	ú	u with Acute Accent
251	FB	û	u with Circumflex
252	FC	ü	u with Dieresis
253	FD	ý	y with Acute Accent
254	FE	þ	Lowercase Thorn
255	FF	ÿ	y with Dieresis



---

# Glossary

## **access layer**

The layer of the window terminal driver that contains all code specific to the communications medium and performs all data-link tasks. It incorporates the communications I/O hardware subsystem, which is responsible for making the physical connection between the module and the communications device. The asynchronous access layer incorporates all code needed to handle RS-232-C communications, as well as the communications I/O hardware required to make terminal connections. The window terminal driver also supports access layers such as OS TELNET, which handles OS TELNET connections over an OS TCP/IP Ethernet-based network.

## **access mode**

The method that the I/O system uses to access records for reading or writing. For asynchronous communications, the access mode specified with `s$open` should be sequential (represented by the value 1).

## **allocate**

In those programming languages that support dynamic memory allocation, to set aside an area of storage for a particular purpose.

## **American National Standards Institute (ANSI)**

A group that promotes standards for computer languages and devices. An ANSI terminal is a terminal that conforms to those standards.

## **American Standard Code for Information Interchange (ASCII)**

In the VOS internal character coding system, the half of the 8-bit code page with code values in the range 00 to 7F hexadecimal.

## **ANSI**

See **American National Standards Institute**.

## **application-managed I/O**

A window terminal programming approach that allows an application program to manage the display of the subwindow. Applications using this approach can retrieve formatted sequential input, processed raw input, or unprocessed raw input, but must send formatted sequential output. With this approach, an application program must use a subwindow in formatted I/O mode to control the processing of

sequential input or output. See also **formatted sequential input**, **formatted sequential output**, **processed raw input**, and **unprocessed raw I/O**.

### **argument**

1. A character string that specifies how a command, request, subroutine, or function is to be executed.
2. A variable or value explicitly passed to a called procedure and corresponding to a parameter of that procedure. Arguments are sometimes referred to as actual parameters.

### **array**

1. Several fields that are grouped together and treated as a unit. The fields that make up an array have the same length, and common field characteristics. Any change to the specifications of a field in an array changes the whole array.
2. A grouping of multiple objects all of the same type.

### **ASCII**

See **American Standard Code for Information Interchange**.

### **ASCII string**

A character string that contains only ASCII data. It therefore contains no shift characters and no characters from the right-hand graphic sets. However, it may contain generic input or generic output sequences.

### **asynchronous communications**

A type of communication that is characterized by a start/stop transmission mode. Each character transmitted in start/stop mode is preceded by a start bit and followed by one or more stop bits. Since there is no clocking mechanism, the interval between characters varies. This method of transmitting data is designed to accommodate irregular transmissions (for example, from terminals) where data is sent sporadically.

### **attach**

1. To associate a port with a terminal, creating the port, if necessary, and generating a port ID. The port ID can then be used to refer to the terminal.
2. To associate an event with a process.

### **attribute**

A video parameter of a terminal (for example, high intensity, low intensity, blinking, blanked, inverse video, and underlined). With the window terminal software, all attributes are treated as *mode attributes*. A mode attribute affects every character transmitted after the attribute is set, and stays in effect until the attribute is reset.

**attribute request**

A request from a program to the operating system to change a terminal's attributes.

**attribute sequence**

The sequence of characters the operating system sends to a terminal to change the terminal's attributes.

**binary data introducer (BDI)**

For Stratus asynchronous communications, the first byte of an `s$read_raw` input sequence containing a generic input request, function key, or raw byte. This type of sequence is called a BDI sequence. The BDI is 93 hexadecimal (147 decimal).

**buffer**

For a terminal, a temporary data storage location in the terminal's memory. The buffer can be used to compensate for differences in transmission rates or temporarily store characters until the computer can accept them.

**calling process**

The process that invoked the program currently being executed.

**calling program**

The program that invoked the subroutine currently executing.

**capability**

An output function that the operating system can request a terminal device to perform. Capabilities include actions such as clearing the scrolling region, inserting and deleting characters, and moving the cursor. The terminal driver uses the capabilities implemented by a terminal type, as specified in the terminal-type table, to carry out generic output requests. The terminal-type table also specifies the sequence of bytes that must be transmitted to the terminal to invoke the function defined for each capability.

**channel**

A hardware-addressable connection point to which a device (such as a terminal or printer) can be connected via a cable. The channel is associated with a particular connector, which is in turn associated with a particular connector, which is in turn associated with a particular electrical interface (such as RS-232-C). The channel should be clearly differentiated from the device connected to the channel. (The device may be a terminal, a printer, another computer, or another channel on the same computer.)

**character**

A symbol, such as a letter of the alphabet or a numeral, or a control signal, such as a carriage return or a backspace. Characters are represented in electronic media by character codes.

**character code**

1. A numeric value used to represent a character according to a specified system. For example, the ASCII (character) code for A is 41 (hexadecimal). In the internal character coding system, the bit representation of a character code can occupy one or two bytes, depending on the character set.
2. A particular system used to assign numeric values to characters (for example, ASCII).

**character string**

An ordered set of characters from one or more character sets. The length of a character string depends on the size of each character (one or more bytes, depending on the character set), the number of characters in the string, and the use of single-shift and locking-shift characters within the string. Character strings are evaluated from left to right.

**clear a device**

To release a device and make it ready for another I/O operation. Clearing a device is similar to closing a file.

**close**

In communications, to disconnect an application program from a device.

**communications chassis**

A card cage that is connected to a C-series communications controller and located at the rear of a module. Every C-series communications chassis has eight slots, each designed to hold a communications line adapter. See also **I/O adapter chassis**.

**communications controller**

A main chassis controller board that connects terminals, printers, and synchronous devices to a module using line adapters. See also **I/O processor**.

**communications line**

The physical medium connecting one location to another for the purpose of transmitting or receiving data.



**communications line adapter**

See **line adapter**.

**complete input record**

In sequential I/O, an input record that is terminated with any of the following input requests: RETURN, ENTER, FORM, HELP, BREAK\_CHAR (depending on the break action), END\_OF\_FILE, ESI, EGI, or EMI. Once an input record is complete, it can be read by `s$seq_read`. In a simple sequential I/O subwindow, an input record entered before a read operation resides in the typeahead area.

**configuration table**

One of the table files that the operating system uses to identify the elements of a system or a network. For example, the file `devices.table` contains information about each device present in the system.

**control character**

A character in the internal character coding system that can perform a specified control function. There are two sets of control characters: ASCII and Stratus-specific. The ASCII control characters are represented by the hexadecimal character codes 00 to 1F. The Stratus-specific control characters are represented by the hexadecimal character codes 80 to 9F. Unlike graphics characters, control characters are non-printing characters. See also **control function** and **graphics character**.

**control function**

An action that affects the recording, processing, transmission, or interpretation of data. A control function has a coded representation consisting of a 1-byte bit combination in the VOS internal character coding system. See also **control character** and **VOS internal character coding system**.

**current I/O subwindow**

The subwindow receiving all input and output.

**cursor**

A marker on the display (typically a blinking rectangular block or an underline) that shows where the next character typed will appear.

**cursor addressing**

To position the cursor at a specific line and column position on the screen.

**cursor coordinates**

One-character representations of each row and column on the screen. Cursor coordinates are primarily used in addressing and reading the cursor.

**data circuit-terminating equipment (DCE)**

In asynchronous communications, refers to a modem.

**data set leads**

The individual wires or pin signals that make up a device cable and implement a standard interface. A standard interface definition (for example, RS-232-C) specifies the number of data set leads in a cable, as well as their names and electrical behavior.

**data terminal equipment (DTE)**

In asynchronous communications, refers to modules, terminals, and printers.

**DCE**

See **data circuit-terminating equipment**.

**default**

The value or attribute used when a necessary value or attribute is omitted.

**default character set**

In National Language Support (NLS) strings or in text files, the supplementary graphic character set that, in the absence of single- or locking-shift characters, is represented by the character codes in the range A0 to FF hexadecimal. The default character set for a file can be set by the `create_file`, `set_text_file`, or `emacs` command. Some of the NLS built-in functions in VOS programming languages permit specification of the default character set as an argument. If a default character set is not specified, the default is assumed to be Latin alphabet No. 1. See also **Latin alphabet No. 1 character set** and **National Language Support (NLS)**.

**default value**

The value that the operating system uses if a specific value is not supplied.

**deleting data**

On a terminal display, when data is deleted, it is removed and the remaining data moves accordingly.

**detach**

1. To disassociate a port from a device.
2. To disassociate an event from a process.

**device**

Any hardware component that can be referenced and used by the system or users of the system and that is defined in the device configuration table. Terminals, printers, tape drives, and communications lines are devices.

**device configuration table**

The file `devices.table`, which contains information about each device present in a system. See also **configuration table**.

**device name**

The name of a device. Device names are specified by the system administrator in the `devices.tin` file, which is used to create the device configuration table. (See **device configuration table**.) The path name of a device has two components: (1) the name of the system, prefixed by a percent sign, and (2) the name of the device, prefixed by a number sign (for example, `% s1# sales_printer`).

**device table**

A file with the name `devices.table` that contains information about each device present in a system. See also **configuration table**.

**dial-out**

A type of line configuration in which more advanced modem protocols are available to allow for both the dial-up capability and the dial-out capability. The dial-out capability is based on the CCITT V.25bis standard. The line can be configured for dial-out only if it is connected to a K-series I/O adapter.

**dial-up line**

A switched communications line that connects a device at a remote site to a host computer via two modems. Dial-up lines connect devices that are not wired directly to the host.

**DTE**

See **data terminal equipment**.

**echoing**

The process by which a character is sent by a terminal keyboard to the computer and then by the computer back to the terminal screen.

**error code**

An arithmetic value (typically a 2-byte integer) indicating what, if any, error has occurred. The value is typically a unique VOS status code. An error-code argument is often included in subroutines.

**error message**

A character string that is associated with an error code.

**ESC character**

The ASCII character with decimal rank 27 (1B hexadecimal). See also **generic sequence introducer (GSI)**.

**escape character**

A user-defined character that signals the beginning of an escape sequence. The default escape character is set in the terminal-type definition file (TTP) for the terminal device. The escape character is also known as the **hexadecimal notation character**, and is defined in the configuration section of the TTP using the `hex-notation-char` variable.

**event**

1. In a Stratus system, a data structure, associated with a file or device, that is used by processes to communicate with one another. When one process notices that some action has occurred that is of potential interest to one or more processes, the first process notifies the event. This notification causes the operating system to inform all processes that have been waiting for the action. (These processes are said to be waiting on the event.) Each time an event is notified, its event count is incremented.
2. An occurrence of significance to a task; for example, the completion of an asynchronous input/output operation.

**event attachment**

The association of an event with a process.

**event count**

An integer value that is associated with an event. Each time the event is notified, the event count is incremented.

**event status**

An integer value, associated with an event, that can be used to tell waiting processes the reason for the event's notification.

**flow control**

The process of controlling the rate at which data is transmitted by the sender, according to the capacity of the receiver to receive additional data. Flow control is based on authorizations from the receiver, and is managed separately for each direction of transmission.

**formatted sequential input**

A type of input that is part of the window terminal software's application-managed I/O approach. It offers character echoing and line-editing support when a read is active, but does not process or display any typeahead until a read is active. Unlike applications using simple sequential I/O, applications using formatted sequential input can position the input line anywhere in the subwindow.

**formatted sequential output**

A type of output that is part of the window terminal software's application-managed I/O approach. It allows an application program to fully control the contents of the subwindow and does not perform automatic line wrapping or pause processing. An application program can use formatted sequential output with either raw input or formatted sequential input.

**Forms Management System (FMS)**

A set of tools, system software, and programming language extensions that gives application programs a consistent interface for the entry and display of data on a video display terminal.

**function key**

One of the keys on the keyboard that generally produces a nonprinting character or a sequence of nonprinting and printing characters. Examples include the numbered function keys, the arrow keys, and the <BACK\_SPACE> key.

**function-key input mode**

An input mode in which terminal-specific keystrokes for a particular set of keys (function keys) are mapped to terminal-independent key codes by the operating system. In this mode, the stream of characters and function-key codes is passed directly to the application program without being mapped to generic input requests.

**generic input mode**

An input mode in which terminal-specific keystrokes are mapped to terminal-independent generic requests by the operating system.

**generic input request**

A request that the operating system either executes on the terminal screen or passes to a program when it recognizes the input characters that define the request in the TTP. The `TAB` and `ERASE_CHAR` requests are generic input requests.

### **generic input sequence**

A series of bytes, beginning with the binary data introducer (BDI), that form the operating system's internal representation of a generic input request. See also **binary data introducer (BDI)**.

### **generic output request**

A terminal-independent output request that can be inserted in the output stream. The `CLEAR_SCROLLING_REGION` and `ENTER_GRAPHICS_MODE` requests are generic output requests. The terminal driver carries out this request for a particular terminal type by using the capabilities implemented by that terminal type. See also **capability**.

### **generic output sequence**

A series of bytes, beginning with the generic sequence introducer (GSI), that form the operating system's internal representation of a generic output request. See also **generic sequence introducer (GSI)**.

### **generic sequence introducer (GSI)**

For Stratus asynchronous communications, the first character in every generic output sequence. The GSI is the ESC character (1B hexadecimal). See also **ESC character**.

### **graphics character**

A symbol, such as a letter of the alphabet, a numeral, or a punctuation mark, as opposed to a control character. In the internal character coding system, graphics characters are represented by codes in the ranges 21 to 7E and A0 to FF (hexadecimal).

### **graphics mode**

A terminal mode in which the terminal converts alphanumeric characters to special graphics characters.

### **GSI**

See **generic sequence introducer (GSI)**.

### **hexadecimal notation**

Notation of numbers in base 16.

### **hexadecimal notation character**

See **escape character**.

**I/O access type**

A type of I/O operation performed on a file or device. For Stratus asynchronous communications, the I/O type specified with `s$open` should be updated (represented by the value 4). This type allows the device connection to support both input and output.

**I/O adapter**

A K-series adapter (card) that is connected to an I/O processor pair. Asynchronous devices connect to the module via full-modem or null-modem MultiCommunications I/O adapters.

**I/O adapter chassis**

A card cage containing 16 slots that houses the K-series I/O adapters.

**I/O processor**

A K-series controller board that manages communications I/O adapters, tape I/O adapters, disk I/O adapters, and Ethernet I/O adapters. It resides in the main chassis of an equipment cabinet.

**include file**

A file that the compiler includes in the source module used by the compilation process. The name of the include file must be specified in a language-specific directive within the source module.

**input line area**

In simple sequential I/O, the area of the subwindow containing the current (incomplete) input record.

**input request**

A request that the operating system sends to a program in response to sequences received from the terminal.

**input sequence**

A series of one or more characters that is generated by the terminal user and represents a specific input request.

**insert mode**

A terminal mode in which characters written to the terminal are inserted at the current location. Existing characters that follow are pushed to the right, not overwritten.

### **internal character set**

One of the sets of characters used to represent data internally in the operating system. Internal character sets include the following:

- the ASCII control character set
- the ASCII graphic character set
- a Stratus-specific control character set containing characters used primarily for National Language Support (NLS)
- supplementary sets of graphics characters, used primarily to provide NLS.

### **interrupt**

A control signal that temporarily diverts the attention of the operating system or program from its current processing.

### **interrupt character**

Input characters that cause the communications hardware to generate an interrupt. With the window terminal software, interrupt characters are interpreted according to the setting of the unprocessed raw input mode. In normal raw input mode, every input character is an interrupt character. In raw record mode and raw table input mode, each interrupt character must be defined in an interrupt table.

### **interrupt table**

A 256-byte buffer in which each buffer position (0 to 255) represents the rank assigned to a character in the internal character coding system. If an application program uses the window terminal software's raw table or raw record mode, it must set its own interrupt table. Creating an interrupt table allows an application program to specify which characters cause interrupts and which do not.

### **inverse video**

A display attribute that reverses the character color and screen background color (for example, if the terminal normally displays amber characters on a black background, enabling this attribute makes the characters black and the screen background amber). Support of the inverse video attribute depends on the individual terminal and its TTP.

### **jump scroll**

A scrolling method in which the terminal scrolls lines as fast as data is received. See also **smooth scroll**.

### **katakana, kanji, or hangul default string**

A string with a default character set of katakana, kanji, or hangul. In a string with a katakana default, a single-shift character precedes each non-katakana character. In a string with a kanji default, a single-shift character precedes each non-kanji



character. In a string with a hangul default, a single-shift character precedes each non-hangul character. These strings contain no locking-shift characters, but may contain generic input or generic output sequences.

**Latin alphabet No. 1 character set**

A character set located in the right-hand side of the internal character coding system in the range A0 to FF hexadecimal. For Stratus, the left-hand side is occupied by ASCII.

**left control character**

An ASCII control character. See also **control character**.

**left graphics character**

A character located in the range 21 to 7E hexadecimal in the internal character coding system. Left graphics characters compose the ASCII character set.

**line adapter**

A small board containing electrical interfaces that allow communications devices to be connected to the module.

**line mode I/O**

See **simple sequential I/O**.

**locking-shift character**

A user-transparent character in an array or string, indicating that the remaining characters in the key or record are from a character set other than the default character set.

**login terminal**

A terminal from which a valid user can execute, interrupt, or stop execution of VOS commands, command macros, or program modules.

**metacharacter**

A type of marker that the communications driver places in the input stream to signal certain conditions (usually error conditions) to the application program.

**modem**

A device that converts data from a form that is compatible with data processing (digital) to a form that is compatible with transmission facilities (analog), and vice versa.

**module**

A single Stratus computer. A module is the smallest hardware unit of a system capable of executing a user's process.

**National Language Support (NLS)**

The ability of the operating system to represent text in languages other than English.

**newline character**

A nonprinting character that you insert in a file by pressing the <RETURN> key; an ASCII LF (line feed) character whose hexadecimal code is 0A.

**NLS**

See **National Language Support**.

**NLS string**

A general term for any of the string formats, combined or used alone. Therefore, an NLS string contains characters from one or more NLS character sets and may contain locking- or single-shift characters.

**no-wait mode**

A port setting that allows a task or process to continue executing if a requested I/O operation cannot be completed immediately.

**non-ASCII**

In the VOS internal character coding system, the half of the 8-bit code page with code values in the range 80 to FF hexadecimal representing Stratus-specific control characters and supplementary graphic character sets.

**normal raw mode**

An unprocessed raw input mode that causes every incoming character to be treated as an interrupt character. When a character is designated an interrupt character, its arrival signals an interrupt.

**opcode**

For VOS, an operation code (opcode) is a 2-byte integer passed to `s$control`. Each opcode instructs `s$control` to execute a different control operation on the device.

**open**

To initialize a device that is attached to a specified port.

**output area**

In simple sequential I/O, the portion of the subwindow that displays output (for example, command output). The output area extends from the top of the subwindow down to the typeahead and input line areas (if there is completed typeahead and/or a current input record). The number of lines in the output area increases and decreases in response to activity in the typeahead and input line areas.

**output request**

A request that a program sends to VOS to perform a specific action on the device.

**output sequence**

A sequence of characters that VOS sends to a device to perform a specific action.

**overlay mode**

A terminal mode in which each character entered at the keyboard overwrites the character at the current cursor position. See also **insert mode**.

**parity**

Used to verify the accuracy of data passed through the channel or subchannel. It involves the addition of a bit to a data byte. For example, ASCII uses 7 or 8 data bits to represent a character and uses an additional bit for parity. The window terminal driver supports six types of parity: even, odd, mark, space, Baudot, and no parity.

**pause**

To temporarily suspend the display of output. Pausing is a mechanism that allows the terminal user to view long displays of output at certain intervals so that data does not scroll up and out of the subwindow before the user can read it. The window terminal software supports pausing in a simple sequential I/O subwindow.

**port**

A data structure, identified by a name or an ID, that VOS assigns to your process when you perform an `s$attach_port` operation. The port associates your program with a device. Therefore, to perform I/O on a device, you must supply the appropriate port name or ID.

**port attachment**

The creation of a port for the purpose of accessing the terminal.

**port ID**

A 2-byte integer used to identify a port.

**port name**

The character-string name of a port.

**primary window**

The portion of the terminal screen that displays the output of a single process or program running the window terminal driver. When the user logs in, a primary window appears that comprises the entire terminal screen. Either the user or the application program can create additional independent primary windows and move among all existing primary windows.

Primary windows overlap. Conceptually, they are ordered from top to bottom, where the primary window currently displayed is on top of the other primary windows.

**printable characters**

Those characters that can be printed by most standard printers and displayed on most standard terminals (ASCII characters whose rank is greater than 31 and less than 127).

**processed raw input**

A type of raw input that uses several input modes to provide terminal independence through various degrees of character mapping. There are three types of processed raw input: translated input, function-key input, and generic input. With each type of processed raw input, there is no automatic character echoing.

**raw I/O**

In the operating system, the transfer of input and output characters between the user's buffer space and a device, without any translation. The system subroutines `s$read_raw` and `s$write_raw` perform raw I/O.

**raw record mode**

An unprocessed raw input mode that allows an application program to impose a record structure on incoming data. It allows an application program to use its own interrupt table, in which each interrupt character specified is also a record terminator.

**raw table mode**

An unprocessed raw input mode that allows an application program to use its own interrupt table, in which each character specified as an interrupt character causes an interrupt.

**right control character**

A control character with a hexadecimal character code between 80 and 9F. See also **control character**.

**right graphics character set**

A character set located in the range A0 to FF hexadecimal in the VOS internal character coding system. Latin alphabet No. 1, katakana, kanji, and hangul are right graphics character sets.

**RS-232-C**

The standard interface for communications between a modem and the associated data terminal equipment. A 25-position connector is used. A voltage level between +5 and +15 indicates a 0 data bit or an ON control lead, while a voltage level between -5 and -15 indicates a 1 data bit or an OFF control lead. In addition to defining electrical characteristics, RS-232-C defines signals and their use in providing half- or full-duplex transmission on asynchronous or synchronous circuits at rates of up to 19,200 bps. This standard is equivalent to a combination of CCITT Recommendations V.24 and V.28. It is gradually being superseded by its latest revision, EIA-232-D.

**scrolling**

1. The movement of data across the screen.
2. A characteristic that vertically extends a field or scrolling region to contain more data than can normally be displayed at one time.

**scrolling region**

The area of the subwindow in which the user or application program can scroll data.

**sequential file**

A sequentially organized file that contains records of varying sizes stored in a disk or tape region holding approximately the same number of bytes as the record. Thus, the record-storage regions in a sequential file vary from record to record. Sequential files can only be accessed on a record basis, using `ssseq_read`, which reads the next record from the file.

**shift mode**

A specification that indicates which shift characters, if any, are allowed in a text file. The shift modes specify the following:

- no shift characters
- only single-shift characters

- only locking-shift characters
- a combination of single-shift and locking-shift characters.

### **simple sequential I/O**

A window terminal programming approach that allows the application program to read/write records of text to a terminal in a sequential manner. With this approach, there is an output area, a typeahead area, and an input line area. All processed output records reside in the output area, all completed unread input records reside in the typeahead area, and the current input record is echoed to the input line area automatically. The approach supports full line editing and handles pause processing. Simple sequential I/O is sometimes referred to as line mode I/O.

### **single-shift character**

A user-transparent character in a key or record, indicating that the next character is from a character set other than the default character set.

### **slave state**

For a tasking terminal, a nonlogin state that prevents the user from breaking out of the program, terminating the program, or doing anything not explicitly allowed by the program.

### **smooth scroll**

A scrolling method in which the text scrolls up or down smoothly at a slower rate than usual. For example, the V102 terminal has three speeds of smooth scroll: normal, slow, and fast. See also **jump scroll**.

### **standout attribute**

A display attribute used to highlight and emphasize text. The style used to highlight the text can vary, depending on the type of terminal. On some terminals, low intensity may be suitable to emphasize the text. On other terminals, high intensity may be the best choice. This attribute must be defined in the TTP.

### **status area**

A line that can display information about the current status of the primary window, error messages, and messages describing the status of the system. The status area is usually the bottom line on the screen.

### **structure**

1. In PL/I, a collection of hierarchically arranged variables.
2. In C, an object that comprises an ordered sequence of named members, possibly of different types.

**subchannel**

A K-series hardware-addressable connection point to which a device (such as a terminal or printer) can be connected via a cable. The subchannel is associated with a particular connector, which is in turn associated with a particular electrical interface (such as RS-232-C). The number of subchannels available depends on the model of I/O adapter being used.

**subroutine**

A sequence of statements that can be invoked as a set at one or more points in a program to execute a specific operation.

**subwindow**

A portion of a primary window that is controlled by the application program executing in the primary window. There are three types of subwindows: simple sequential I/O, formatted sequential I/O, and forms. All I/O goes to the subwindow that is designated as the current I/O subwindow.

Subwindows may or may not overlap and may vary in size, depending on how the application program creates and arranges them within the primary window.

**terminal interface layers**

The layers of the window terminal driver that supply the application program interface and all terminal interface code for character-mode terminals. They are organized to separate application-specific tasks from terminal-specific tasks.

**terminal type**

In VOS, a software table-driven mapping facility that provides a certain degree of terminal independence to the Stratus asynchronous terminal software. Terminal types are typically defined for all terminals that use the system.

**terminal-type definition file**

In VOS, a file that defines the relationship between generic input and output requests and terminal-specific input and output sequences. This file is the input that is used to create the terminal-type tables for a terminal type. The suffix for the file is `.ttp`.

**terminal-type table**

A table created from the terminal-type definition file. Terminal-type tables reside in system storage and provide a mapping between input and output sequences and generic input and output requests.

### **translated input mode**

A window terminal raw input mode that causes raw bytes to be compared with the character-translation database established by the current terminal type. When raw bytes map to the database, they are replaced by internal character codes and passed to the application program without further processing.

### **typeahead**

Unread input. In a simple sequential I/O subwindow, typeahead can be either completed unread input or incomplete unread input. Completed unread input records reside in the typeahead area, while the current (incomplete) input record resides in the input line area.

### **typeahead area**

The area of a simple sequential I/O subwindow containing completed unread input records. The typeahead in this area is typically displayed in half intensity (if the terminal supports half intensity). The application program can limit the number of completed typeahead records permitted in the subwindow at a time by setting a typeahead limit.

### **undisplayable notation character mode**

A window terminal mode that allows your application program to determine the handling of characters for which there is no graphic representation (control characters and non-ASCII characters).

### **unprocessed raw I/O**

A programming approach that allows an application program to control the asynchronous communications line as a communications device. Unprocessed raw I/O handles raw data; characters or bytes are transmitted without being interpreted in any way. There are three window terminal raw input modes associated with unprocessed raw I/O: normal raw, raw table, and raw record. With each type of unprocessed raw input, there is no automatic character echoing. See also **normal raw mode**, **raw record mode**, and **raw table mode**.

### **VOS internal character coding system**

The system used internally for encoding character data on Stratus systems. This system, based on the international standard ISO-2022-1986, encodes the multiple character sets needed for National Language Support (NLS). See also **American Standard Code for Information Interchange (ASCII)**, **character code**, **graphics character**, **internal character set**, and **non-ASCII**.

### **wait mode**

A port setting that requires a task or process to wait until a requested I/O operation completes before it can continue executing.



**window manager**

A mechanism offered by the window terminal driver that provides a means for manipulating primary windows. Among the actions the user can request are opening additional primary windows, moving among a group of primary windows, and interrupting activities in primary windows in various ways.

**window terminal software**

A loadable operating-system device driver that supports character-mode terminals (for example, video display terminals) in a window environment. Its layered architecture separates the application program interface and terminal code from the code associated with the communications medium.



---

# Index

8-bit transmission, 1-6, 1-7, 1-22, 1-38, 1-39,  
3-2, 3-4  
compatibility issues concerning, A-1  
parity and hardware requirements when  
using, 8-34

## A

ABORT (3) opcode, 6-6, 8-24  
ABORT\_OUTPUT input request, 2-8, 8-72, A-4  
Access layers, 1-5  
asynchronous, 1-6, 3-4, 3-5, 3-6, 8-26  
Console Controller, 1-7  
opcodes and, 8-27, 8-37  
OS TELNET, 1-7, 3-4, 3-5, 3-6, 8-26  
STCP TELNET, 1-8  
types of, 1-3, 1-5  
Adapters  
asynchronous, 1-25  
baud rates and, 1-37  
considerations when selecting, 1-22  
errors caused by, 6-10, 6-13, 7-30, 7-34  
K101, 1-31  
K110, 1-32  
K111, 1-31  
K118, 1-32, E-4  
list of K-Series multicom munications I/O  
adapters, 1-29  
null-modem versus full-modem, 1-33  
Alternative I/O subroutines, 1-9, 7-1  
analyze\_system subsystem, E-2, E-5  
dump\_acb request, E-5  
dump\_channel\_info request, E-5  
dump\_channels request, E-5  
dump\_dvt request, E-5  
dump\_scb request, E-5  
dump\_tcbh request, E-5  
dump\_wcb request, E-5  
terminal\_meters request, E-5  
Application program  
compiling and binding, 1-24

debugging, 1-24  
design considerations for, 1-10  
programming approaches for, 1-12  
subroutine interface for, 1-3, 1-8  
tasks for, 1-9  
Application programming approaches, 1-12  
Application-managed I/O, 1-12, 4-1  
characteristics of, 4-1  
formatted sequential input and, 1-13, 4-1,  
4-8  
formatted sequential output and, 1-13, 4-1,  
4-4  
types of input for, 4-2  
Architecture of the window terminal driver, 1-3,  
1-5  
Area attributes, 5-22, A-12  
Areas in a simple sequential I/O  
subwindow, 2-4  
ASCII character codes, F-1  
ASCII character set, 1-17, 1-38, 5-30, B-1, F-1  
ascii terminal type, 2-4  
ASYNC\_HANGUP (234) opcode, A-15  
Asynchronous access layer (async\_al), 1-3,  
1-6, 3-4, 3-5, 3-6  
Asynchronous communications  
overview of Stratus, 1-25  
overview of the window terminal driver, 1-2  
tools and commands for, E-1  
Asynchronous connections  
direct, 1-25  
modem, 1-27  
types of, 1-25  
Asynchronous devices  
channel/subchannel characteristics, 1-36  
configuring, 1-30, E-3  
creating TTPs for, 5-6, E-3  
displaying information about, E-4  
line speeds for, 8-27  
RS-232-C interface for, 1-25, 1-33  
tools and commands for, E-1  
Attaching a port, 6-1, 6-2

**Attributes**

- area, 5-22
- display, 5-24
- list of, 5-24
- mode, 5-22
- output requests for setting, 5-22, A-12

Attributes section of a TTP, 5-6, 5-7, 5-22, 5-24

Auto-baud detection, 1-37

Automatic frame detection, 1-6, 1-7, 1-22, 3-2, 3-6

**B**

Baud rates, configuring, 1-37, 8-27

Baudot character set, 1-38

BDI data type, 4-17, B-5

BDI header, 4-16

BDI sequences, 4-16, 5-5, 7-4, A-7, A-8, B-5

- for a raw byte, 4-18, 4-24
- for function keys, 4-17, 4-23
- for generic input requests, 4-17, 4-24
- structure for, 4-16

BEEP output request, 5-11, 5-29, 7-26

BEL control character, 5-31, 7-26, 7-27, A-13, A-14

Bidirectional flow control, 1-6, 1-7, 1-19, 8-32, A-2

rules for using, 8-29

Binary data introducer sequences. *See* BDI sequences

Binary data length, 4-17

bind command, 1-24

Binding an application program, 1-24

Bit mask, setting the, 8-59

Bits-per-character configuration, 1-39, 8-28, A-1, A-22

Black-on-white screen option, 8-43

BLANK\_OFF output request, 5-11, 5-22

BLANK\_ON output request, 5-11, 5-22

BLINK\_OFF output request, 5-11, 5-23

BLINK\_ON output request, 5-11, 5-23

BOLDFACE\_OFF output request, 5-11, 5-23

BOLDFACE\_ON output request, 5-11, 5-23

Break action, 3-2, 7-10, 7-11, 7-17, 8-43, 8-53

IGNORE option, 8-53

RETURN\_ERROR option, 8-53, 8-54

RETURN\_NUL\_CHAR option, 8-53

SIGNAL\_AND\_DISCARD option, 8-53

SIGNAL\_ONLY option, 8-53

Break condition, 7-18, 8-46

Break handling, support of, 1-22, 1-23, 1-39

Break table input mode, A-5

Break table record input mode, A-5

BREAK\_CHAR input request, 3-2, 7-15, 8-43, 8-53

BREAK\_DISABLE (223) opcode, A-15

BREAK\_ENABLE (222) opcode, A-15

broadcast\_file command, E-2, E-3

BS control character, 5-31, 7-26, 7-27, A-13, A-14

Buffers, internal input, 6-8, 6-9, 7-18, 8-38

Bulk raw retrieval, A-5

**C**

Cables, selecting, 1-34

CANCEL input request, 2-5, 2-10, A-4

CANCEL\_DOESNT\_ABORT screen option, 2-8, 8-43, 8-44, A-3

Capabilities, 2-14, 4-4, 4-8, 5-6

reset-terminal, 5-29

*See also* Terminal type (TTP)

terminal, 5-7

change\_terminal command, E-2, E-3

Channels

characteristics of asynchronous, 1-36

pinouts on full-modem asynchronous, 1-34

pinouts on null-modem

asynchronous, 1-35

Character codes

ASCII, F-1

VOS internal, F-1

Character echoing, 2-2, 7-4, 7-14, 8-50

enabling and disabling, 8-50

Character sets, 1-17

ASCII, B-1

default internal, F-1

supplementary graphic, B-5

Character-mode terminals, 1-2, 1-3

Characters

ASCII, F-1

VOS internal, F-1

Character-translation section of a TTP, 5-6

Clear to send (CTS) signal, 1-34, 1-35

CLEAR\_SCREEN output request, 5-9, A-12

CLEAR\_SCROLLING\_REGION output request, 4-8, 5-9, 5-18, 8-48, A-12

CLEAR\_TO\_EOL output request, 5-9, 5-18

- 
- CLEAR\_TO\_EOR output request, 4-8, 5-9, 5-18, A-12
  - CLEAR\_TO\_EOS output request, 5-9, A-12
  - Clearing operations, 9-1
    - output requests performing, 5-18
  - Closing a port, 9-1, 9-2
  - Command level, VOS operating system, 2-4
  - Command modes, resetting, 2-8, 8-72
  - Commands
    - analyze\_system, E-2, E-5
    - bind, 1-24
    - broadcast\_file, E-2, E-3
    - change\_terminal, E-2, E-3
    - compile\_terminal\_type, E-2, E-3
    - configure\_async\_lines, 1-32, E-2, E-4
    - configure\_comm\_protocol, 1-2, E-2, E-3
    - configure\_devices, E-2, E-3
    - create\_table, E-2, E-3
    - display\_device\_info, E-2, E-4
    - display\_line, 5-13
    - display\_terminal\_parameters, E-2, E-4
    - install\_terminal\_type, E-2, E-3
    - list\_devices, E-2, E-4
    - list\_terminal\_types, E-2, E-4
    - logout -hold, 8-39
    - set\_terminal\_parameters, 2-2, 2-9, E-2, E-4
    - update\_channel\_info, E-2, E-4
  - Communications I/O hardware, 1-5, 1-29, 8-29, 8-34, 8-59
    - full-modem, 1-27
    - null-modem, 1-25
  - Communications media, opcodes affecting all, 8-15, 8-20, 8-37
  - Communications medium, opcodes affecting the RS-232-C, 8-15, 8-20, 8-27
  - Communications subsystem, C-Series
    - communications controllers, 3-8, 8-34
  - Communications, overview of
    - asynchronous, 1-25
  - Compatibility issues, 1-2, A-1
    - break actions, 8-54
    - complete-write option, A-21
    - control characters, 5-32
    - handling of s\$control opcodes, A-14
    - line length (width), 8-45
    - screen height, 8-42
  - compile\_terminal\_type command, E-2, E-3
  - Compiling
    - a TTP, E-2, E-3
    - an application program, 1-24
    - using shortmap versus longmap, 1-24
  - Complete-write option, compatibility issues concerning, A-21
  - Configuration
    - ASCII or Baudot, 1-38
    - asynchronous device, 1-36, A-22, E-3
    - baud-rate, 1-37, 8-27
    - bits-per-character, 1-39, 8-28
    - force-listen, 1-39
    - full-duplex, 1-37
    - full-modem, 1-30, 1-32, 1-33
    - guidelines for the K118 I/O adapter, 1-32
    - K-Series hardware and, 1-30
    - login or slave, 1-37
    - null-modem, 1-30, 1-32, 1-34
    - parity, 1-38, 8-34
    - privileged-terminal, 1-39
    - stop-bits, 1-39
  - Configuration section of a TTP, 5-6, 8-24, 8-42, 8-45
    - specifying the name of, 8-45
  - configure\_async\_lines command, 1-32, E-2, E-4
  - configure\_comm\_protocol command, 1-2, E-2, E-3
  - configure\_devices command, E-2, E-3
  - Connection states, 8-37
  - Connections
    - direct, 1-25
    - holding, 8-39
    - modem, 1-26, 1-27
    - monitor terminal, 1-29
    - null-modem versus full-modem
      - adapters, 1-33
    - OS TELNET, 1-7
    - Remote Service Network (RSN), 1-29, 1-33, 1-39
    - RS-232-C, 1-25, 1-33
    - STCP TELNET, 1-8
    - support for different types of, 1-3
  - Console Controller access layer (recc\_al), 1-7
  - Continue characters, 2-2, 8-54

Control characters, 7-26, 7-27, B-3  
    compatibility issues concerning, A-13  
    defining the handling of, 5-31  
    undisplayable notation character mode  
        and, 8-68

Control operations, 8-1  
    getting event information, 8-12  
    setting device-specific parameters, 8-80  
    setting no-wait mode, 8-7  
    setting terminal parameters, 8-15  
    setting time limits, 8-2  
    setting wait mode, 8-5

Control sets, B-1  
    left-hand, B-1, B-3  
    right-hand, B-1, B-4

CR control character, 5-31, 7-26, 7-27, A-13,  
    A-14

`create_table` command, E-2, E-3

Creating  
    a device configuration table, E-2, E-3  
    an interrupt table, 3-7, 8-57  
    formatted sequential I/O subwindows, 4-3,  
        8-48

C-Series hardware, 8-34

Current I/O subwindow, 4-2, 4-5, 4-9, 8-47,  
    8-73, 8-74  
    *See also* Subwindows

Cursor format  
    changing, 8-54, A-11  
    changing with the `SET_CURSOR_FORMAT`  
        output request, 5-17

Cursor operations, output requests  
    performing, 5-13

`CURSOR_OFF` output request, 5-10, 5-13

`CURSOR_ON` output request, 5-10, 5-13

## D

Data carrier detect (DCD) signal, 1-27, 1-28,  
    1-34, 1-35

Data communications equipment (DCE)  
    devices, 1-25

Data path  
    7-bit, 1-38  
    8-bit, 1-6, 1-7, 1-38, 3-2

Data set lead (DSL) flow control, 1-6, 1-7, 1-19,  
    1-20, 8-29, 8-30, 8-32, 8-33

Data set ready (DSR) signal, 1-27, 1-28, 1-34,  
    1-35

Data terminal equipment (DTE) devices, 1-25

Data terminal ready (DTR) signal, 1-25, 1-27,  
    1-28, 1-34, 1-35

DCE devices. *See* Data communications  
    equipment (DCE) devices

Debugging  
    using the `analyze_system`  
        subsystem, E-5  
    using the VOS debugger, 1-24

Default internal character sets, 1-17, F-1

DEL character, B-1

`DELETE_CHARS` output request, 5-10, 5-19

`DELETE_LINES` output request, 5-10, 5-19,  
    8-46

Deleting subwindows, 8-49

Design considerations, application, 1-10

Detaching a port, 9-1, 9-2, 9-5

Device configuration, 1-36  
    compatibility issues concerning, A-21  
    guidelines for the K118 I/O adapter, 1-32  
    K-Series hardware and, 1-30

Device driver  
    organization of, 1-3, 1-5  
    standard asynchronous, A-1, A-22  
    window terminal, 1-2, 1-3, 1-5, A-1

Device types, A-22  
    `device_type` field of `devices.tin` file, A-22

Device-interface layer, 1-5

`devices.tin` file, device-type values of, A-22

Dial-out connections, 1-27

Dial-up connections, 1-27, 7-11, 7-18, 8-39

Direct connections, 1-25

DISCARD function, 8-72

`DISCARD_INPUT` (204) opcode, A-15

`DISCARD_OUTPUT` (205) opcode, A-15

Discarding  
    input, 2-5, 8-52  
    output, 2-8, 8-52, A-4

`DISPLAY_BLOCK` output request, 5-11, 5-29

`display_device_info` command, E-2, E-4

`display_line` command, 5-13

`DISPLAY_OFF` (221) opcode, A-15

`DISPLAY_ON` (220) opcode, A-15

`display_terminal_parameters`  
    command, E-2, E-4

`DOWN` output request, 5-10, 5-14

`DOWNARROW` key, 2-7, 2-8, A-4

Drivers  
    old asynchronous, 1-1, 1-2, 1-18

window terminal, 1-1, 1-2, 1-18  
 DSL flow control. *See* Data set lead (DSL) flow control  
 DTE devices. *See* Data terminal equipment (DTE) devices  
 Duration switches. *See* hold\_attached and hold\_open switches

## E

Echoing, character, 2-2, 7-4, 8-50  
 Edited-output option, compatibility issues concerning, 5-32, A-14  
 EGI input request, 2-2, 7-15  
 EGI\_DELIMITER output request, A-11  
 8-bit transmission, 1-6, 1-7, 1-22, 1-38, 1-39, 3-2, 3-4  
     compatibility issues concerning, A-1  
     parity and hardware requirements when using, 8-34  
 EMI input request, 2-2, 7-15  
 EMI\_DELIMITER output request, A-11  
 END\_25TH\_LINE output request, A-10  
 END\_OF\_FILE input request, 7-15  
 ENTER input request, 2-2, 7-15  
 Enter key, 2-9  
 ENTER\_GRAPHICS\_MODE output request, 5-11, 5-26  
 ENTER\_INSERT\_MODE output request, A-9  
 ENTER\_MONITOR\_MODE output request, A-9  
 Errors  
     channel, 6-10, 6-13, 7-29  
     common asynchronous, 1-22, 7-9, 7-10, 7-11, 7-17  
     e\$abort\_output (1279), 2-8, 7-27, 7-29, 7-33  
     e\$bad\_port\_number (1029), 9-6  
     e\$block\_overrun (2917), 7-9, 7-11, 7-18  
     e\$break\_signalled (1086), 7-10, 7-11, 7-17, 7-18, 8-54  
     e\$buffer\_too\_small (1133), 7-11, 7-18, 7-24, 7-29, 7-33, 8-83, 8-84  
     e\$caller\_must\_wait (1277), 2-13, 3-5, 3-6, 4-10, 6-6, 6-9, 6-13, 7-5, 7-7, 7-11, 7-15, 7-18, 7-22, 7-24, 7-27, 7-29, 7-33, 8-7, 8-8, 8-12, 8-25, 8-77, 8-80, 8-84, 9-2, 9-4, A-5

e\$close\_invalid\_on\_sys\_port (3425), 9-4  
 e\$device\_already\_assigned (1155), 6-5, 6-13  
 e\$device\_not\_assigned\_to\_you (1158), 8-77, 8-84  
 e\$device\_not\_found (1220), 6-5, 6-13  
 e\$egi (2858), 7-15, 7-19  
 e\$emi (2857), 7-15, 7-19  
 e\$end\_of\_file (1025), 7-15, 7-19  
 e\$esi (2856), 7-15, 7-19  
 e\$feature\_unavailable (1687), 7-19, 7-29, 7-33  
 e\$form\_cant\_delete\_orig\_subwin (3956), 8-49, 8-77  
 e\$form\_invalid\_subwindow\_id (3964), 8-77  
 e\$form\_requested (1225), 7-15, 7-19  
 e\$frame\_error (2309), 7-9, 7-10, 7-12, 7-17, 7-19  
 e\$help\_requested (4085), 7-15, 7-19  
 e\$invalid\_access\_mode (1071), 6-9  
 e\$invalid\_baud\_rate (4761), 8-77  
 e\$invalid\_bits\_per\_char (4762), 8-77  
 e\$invalid\_break\_action (4770), 8-77  
 e\$invalid\_buffer\_size (1215), 6-9, 8-77  
 e\$invalid\_comm\_model (3144), 8-77, 8-84  
 e\$invalid\_control\_operation (1366), 8-77, 8-84  
 e\$invalid\_cursor\_format (4767), 8-78  
 e\$invalid\_form\_id (3794), 8-70, 8-78, A-16  
 e\$invalid\_hardware\_type (2998), 6-9, 6-13  
 e\$invalid\_input\_mode (4760), 8-78  
 e\$invalid\_io\_operation (1040), 7-19, 8-4, 8-6, 8-11, 8-78, 8-84, 9-4, 9-6  
 e\$invalid\_io\_type (1070), 6-9, 6-13  
 e\$invalid\_key (4769), 8-78  
 e\$invalid\_open\_action (4768), 8-78  
 e\$invalid\_parity (4763), 8-78

`e$invalid_stop_bits` (4764), 8-78  
`e$invalid_string_size` (4776), 8-78  
`e$invalid_subwindow_bounds` (4771), 8-78  
`e$invalid_tab_settings` (1499), 8-78  
`e$invalid_undisp_mode` (4774), 8-79  
`e$line_hangup` (1365), 6-9, 6-13, 7-9, 7-11, 7-12, 7-17, 7-19, 7-24, 7-30, 7-34  
`e$long_record` (1026), 2-9, 3-7, 7-6, 7-7, 7-11, 7-12, 7-14, 7-20, 8-79, 8-80, 8-83, 8-84, A-5  
`e$name_not_found` (3084), 8-57, 8-79  
`e$no_alloc_wired_heap` (3077), 6-10, 6-13  
`e$no_ports_available` (1006), 6-5, 6-13  
`e$not_yet_implemented` (1062), 8-79, 8-84  
`e$out_of_range` (1038), 8-79  
`e$out_of_service` (2535), 6-10, 6-13, 7-9, 7-12, 7-20, 7-24, 7-30, 7-34  
`e$overflow_error` (2919), 7-9, 7-13, 7-17, 7-20  
`e$parity_error` (2916), 7-9, 7-10, 7-13, 7-17, 7-21  
`e$port_already_attached` (1008), 6-5  
`e$port_not_attached` (1021), 7-13, 7-21, 8-4, 8-6, 8-11, 8-14, 9-4, 9-6  
`e$redisplay_form` (3363), 7-21, 7-30, 7-34  
`e$short_record` (1299), A-5  
`e$timeout` (1081), 7-13, 7-21, 7-22, 7-24, 7-30, 7-34, 8-2  
`e$unknown_terminal_type` (1221), 8-79  
`e$wrong_version` (1083), 8-79  
for record terminators, 7-15, 7-19  
ESC character, B-3  
Escape character, 5-6  
Escaped notation, 5-31, 5-32  
ESI input request, 2-2, 7-15  
ESI\_DELIMITER output request, A-11  
Even parity, 1-38

Events, 8-7  
count for, 8-12  
ID for, 8-7, 8-8, 8-12

## F

FF control character, 5-31, 7-26, 7-27, A-13, A-14  
Flow control  
bidirectional, 1-6, 1-7, 1-19, 8-29, 8-32  
characters designating, 1-19, 8-30, 8-32  
compatibility issues concerning, A-2  
DSL, 1-6, 1-7  
input, 1-6, 1-7, 1-18, 1-19, 8-29  
output, 1-6, 1-7, 1-19, 8-32, 8-33  
-rts\_cts\_flow\_control option, 1-20, 8-30, 8-33  
structure for, 8-30, 8-33  
using XON/XOFF keys at a pause, 2-8  
XON/XOFF, 1-6, 1-7, 1-20, 8-29, 8-30, 8-32, 8-33  
Force-listen configuration, 1-39, 8-29, 8-33  
FORM input request, 2-2, 7-15  
Formatted I/O mode, 1-11, 4-5  
disabling, 2-16, 8-51  
enabling, 2-16, 8-50, 8-51  
formatted sequential input and, 4-9  
raw input and, 4-2  
switching the original subwindow to, 8-51  
using, 4-4, 7-14, 7-26, 8-48  
using generic output requests to switch to, 5-7, 5-15, 5-18, 5-26, 7-26, A-11, A-12  
Formatted sequential input, 1-11, 1-13, 4-1, 4-2, **4-8**, 7-14  
character echoing and, 8-50  
generic input requests available with, 4-10, 5-1  
input line and, 4-8, 4-9  
Formatted sequential output, 1-11, **4-4**, 7-25  
generic output requests available with, 4-5, 4-8, 5-7  
subwindow updates and, 4-8, 7-26, 8-52  
Forms  
aborting, 8-70  
primary-window ID for, 8-60, 8-69  
Forms I/O programming approach, 1-13  
Frame detection, 1-6, 1-7, 3-2, 3-6  
Full-duplex configuration, 1-37



Function keys, 4-20  
     BDI sequence for, 4-17  
     include file for, 1-23  
     list of, 4-21, 4-22  
     mapping of, 4-15  
     return of, 4-17, 4-23  
     TTP and, 4-22  
 FUNCTION\_KEY input request, 4-17, 4-23, 5-5, A-8  
 Function-key input mode, 4-2, 4-13, 4-15, 4-20, 7-4, 8-5, 8-7, 8-56  
     compatibility issues concerning, A-7

## G

Generic input mode, 4-2, 4-15, 4-16, 4-24, 7-4, 8-56  
     compatibility issues concerning, A-7  
     s\$read\_raw and, 4-13, 7-3, 8-5, 8-7  
     sequential I/O and, 7-14  
 Generic input requests, 2-2, 5-1  
     ABORT\_OUTPUT, 2-8, 7-29, 8-72, A-4  
     bit mask for, 8-59  
     CANCEL, 2-10  
     compatibility issues concerning, A-8  
     for forms input and Forms Editor, 5-1  
     for sequential I/O, 2-10, 2-11, 4-10, 5-1  
     for window manager mode, 5-2  
     FUNCTION\_KEY, A-8  
     include file for, 1-23, 2-10, 5-1  
     introducing a function key, 4-23, 5-5  
     introducing a raw byte, 4-24, 5-5  
     INVALID\_SEQUENCE, 5-2  
     list of line-editing, 5-2  
     mapping of, 4-15  
     NEXT\_SCREEN, 2-7, 2-10  
     RAW\_INPUT, A-8  
     REDISPLAY, 4-16  
     return of, 4-17, 4-24  
     TTP request specifiers for, 5-2  
     TWIDDLE, A-8  
 Generic output, **5-6**, 7-1, 7-25, 7-31, 8-52  
     characteristics of, 5-6  
     handling control characters, 5-31  
     handling generic output requests, 5-7  
     handling NLS characters, 5-30  
 Generic output requests, 2-2, 2-13, 2-16, 5-1, 5-6  
     BEEP, 5-11, 5-29

BLANK\_OFF, 5-11, 5-22  
 BLANK\_ON, 5-11, 5-22  
 BLINK\_OFF, 5-11, 5-23  
 BLINK\_ON, 5-11, 5-23  
 BOLDFACE\_OFF, 5-11, 5-23  
 BOLDFACE\_ON, 5-11, 5-23  
 CLEAR\_SCREEN, 5-9  
 CLEAR\_SCROLLING\_REGION, 4-8, 5-9, 5-18, 8-48, A-12  
 CLEAR\_TO\_EOL, 5-9, 5-18  
 CLEAR\_TO\_EOR, 4-8, 5-9, 5-18, A-12  
 CLEAR\_TO\_EOS, 5-9  
     compatibility issues concerning, A-8, A-12  
     creating sequences for, 5-12  
 CURSOR\_OFF, 5-10, 5-13  
 CURSOR\_ON, 5-10, 5-13  
 DELETE\_CHARS, 5-10, 5-19  
 DELETE\_LINES, 5-10, 5-19, 8-46  
 DISPLAY\_BLOCK, 5-11, 5-29  
 DOWN, 5-10, 5-14  
     enabling formatted I/O mode with, 5-7  
     ENTER\_GRAPHICS\_MODE, 5-11, 5-26  
     for formatted sequential output, 2-15, 2-17, 4-4, 4-5, 4-8, 7-26, 8-52  
     for simple sequential I/O, 2-13  
 HALF\_INTENSITY\_OFF, 5-11, 5-23  
 HALF\_INTENSITY\_ON, 5-11, 5-23  
 HI\_INTENSITY\_OFF, 5-11, 5-23  
 HI\_INTENSITY\_ON, 5-11, 5-23  
     include file for, 1-23, 2-13, 5-8  
 INSERT\_CHARS, 5-10, 5-20  
 INSERT\_LINES, 5-10, 5-21  
 INVERSE\_VIDEO\_OFF, 5-11, 5-23  
 INVERSE\_VIDEO\_ON, 5-11, 5-23  
 KEY\_CLICK\_ON, 5-9  
 LEAVE\_GRAPHICS\_MODE, 5-11, 5-27  
 LEFT, 5-10, 5-14  
 NEW\_LINE, 5-9, 5-14  
 POSITION\_CURSOR, 5-9, 5-12, 5-15, A-12  
 RESET\_TERMINAL, 5-11, 5-29  
 RIGHT, 5-10, 5-15  
 SCREEN\_OFF, 4-8, 5-9, 5-11, 5-29  
 SCREEN\_ON, 4-8, 5-9, 5-11, 5-29, 5-30  
 SCROLL\_DOWN, 5-10, 5-16, 5-22  
 SCROLL\_UP, 5-10, 5-16, 5-22  
 SET\_ATTRIBUTE\_CHAR, 5-9  
 SET\_ATTRIBUTES, 5-9  
 SET\_CURSOR\_FORMAT, 5-10, 5-17, A-11  
 SET\_MODE\_ATTRIBUTES, 5-11, 5-23,

- 5-24, A-12
- SET\_SCROLLING\_REGION, 5-10, 5-13,  
5-21, A-11
- SET\_SMOOTH\_SCROLL, 5-9
- STANDOUT\_OFF, 5-11, 5-25
- STANDOUT\_ON, 5-11, 5-25
- testing, 5-13
- TTP and, 5-8
- UNDERSCORE\_OFF, 5-11, 5-26
- UNDERSCORE\_ON, 5-11, 5-26
- UP, 5-10, 5-18
- Generic output sequences
  - changing how the subwindow is  
updated, 4-8, 5-9, 7-26
  - creating, 5-12
  - specifying, 4-5
- Generic sequence introducer (GSI), 2-14, 4-5,  
5-12, B-3
- Get operation, 8-16
- GET\_BREAK\_TABLE (243) opcode, A-15
- GET\_DEFINED\_KEYS (251) opcode, A-15
- GET\_INFO (201) opcode, A-15
- GET\_INPUT\_SECTION (268) opcode, A-15
- GET\_INTERRUPT\_TABLE (267)  
opcode, A-15
- GET\_KEY\_LEGEND (252) opcode, A-15
- GET\_KNOCKDOWN\_ID (272) opcode, A-16
- GET\_LINE\_LENGTH (1) opcode, 5-7, 8-24,  
8-45
- GET\_TERMINAL\_SETUP (270) opcode, A-16
- Glass TTYS, 2-4, 4-8
- Global opcodes, 8-15, 8-19, 8-24
- GOTO\_PAGE output request, A-10, A-11
- Graphic sets
  - left-hand, B-1, B-3
  - of the internal character coding  
system, B-1
  - supplementary, B-1, B-5
- Graphics characters, 4-22, 5-26, 5-27
- Graphics mode, 5-27
  - entering, 5-26
  - TTP and, 5-26
- GSI. *See* Generic sequence introducer (GSI)

## H

- HALF\_INTENSITY\_OFF output request, 5-11,  
5-23
- HALF\_INTENSITY\_ON output request, 5-11,

- 5-23
- Hang-up, line, 8-29, 8-33, 8-39
- Hardware
  - communications I/O, 1-5
  - C-Series, 8-34
  - features for an application program, 1-21
  - K-Series, 1-29, 8-29, 8-34, 8-59
  - requirements, 1-6, 1-7, 3-8
  - support for, 1-3
- height variable, 8-42
- HELP input request, 2-2, 7-15
- hex-notation-char variable, 5-6, 5-31,  
5-32
- HI\_INTENSITY\_OFF output request, 5-11,  
5-23
- HI\_INTENSITY\_ON output request, 5-11, 5-23
- hold\_attached and hold\_open
  - switches, 2-16, 4-5, 6-2, 6-4, 6-11,  
7-25, 8-26, 8-46, 8-48, 8-51, 8-72
- HOLD\_CONNECTION (244) opcode, A-16
- HT control character, 5-31, 7-26, 7-27, A-13,  
A-14

## I

- I/O
  - application-managed, 1-12, 1-13, 4-1, 4-2
  - formatted sequential, 4-4
  - operations, 7-1
  - simple sequential, 2-1
  - time limits, 8-2
  - unprocessed raw, 1-12, 3-1
- I/O hardware, A-22
  - communications I/O adapters, 1-29
  - compatibility and, A-15, A-16, A-17, A-18,  
A-22
  - I/O adapter chassis, 1-29
  - I/O processor, 1-29, 3-8, 8-29, 8-34, 8-59,  
A-2
- ID, subwindow, 1-14, 8-48, 8-49, 8-50, 8-73,  
8-74
- Include files
  - for function keys, 1-23, 4-20
  - for generic input requests, 1-23, 2-10,  
4-10, 4-23, 5-1
  - for generic output requests, 1-23, 4-6, 5-8,  
A-9
  - for `s$control` opcodes, 1-23, 8-16
  - for structures, 1-23, 8-16

list of available, 1-23  
 INITIAL\_STRING output request, A-10  
 Input  
   discarding, 8-52  
   for application-managed I/O, 4-2  
   formatted sequential, 1-13, 4-1, 4-2, 4-8, 4-9  
   function-key, 4-17, 4-20  
   generic, 4-24  
   mapping, 4-14, 4-15  
   processed raw, 4-2, 4-13  
   raw, 4-15  
   time limits, 8-2  
   types of, 1-11  
   unprocessed raw, 1-12, 3-3, 4-2  
   using flow control, 1-19  
 Input buffers  
   allocation for s\$seq\_open, 6-11  
   internal, 6-9  
   maximum length of, 6-9, 8-38  
 Input database, 4-15, 5-5  
 Input flow control, 1-6, 1-7, 1-19, 8-29, A-2  
   rules for using, 8-29  
   window terminal driver and, 1-18  
 Input line area, 2-4  
 Input modes  
   changing, 3-3, 8-55, 8-56  
   normal raw, 3-5  
   processed raw, 7-4  
   raw record, 3-6  
   raw table, 3-5  
   saving and restoring, 3-3  
   setting, 8-55, 8-56  
   unprocessed raw, 3-4, 7-3  
 Input processing  
   compatibility issues concerning, A-7  
   for simple sequential I/O, 2-10  
 Input record, 2-2, 3-6, 7-14  
 Input requests. *See* Generic input requests  
 Input section of a TTP, 5-5, 8-56  
 Insert mode, 8-43  
 INSERT\_CHARS output request, 5-10, 5-20  
 INSERT\_LINES output request, 5-10, 5-21  
 install\_terminal\_type command, E-2, E-3  
 Interface  
   application program, 1-8  
   RS-232-C, 1-6, 1-7, 1-25, 1-33  
 Internal character coding system, 1-17, 4-15,

4-19, B-1, B-3  
 Interrupt characters, 1-22, 3-4, 3-7, 8-57  
   interrupt character plus one, 3-8, 8-59  
   interrupt character plus two, 3-8, 8-59  
   noninterrupt, 3-7, 8-59  
   normal, 3-8, 8-59  
   s\$read\_raw and, 7-5  
 Interrupt table, 1-22, 3-4  
   creating an, 3-7, 8-57, 8-59  
   deleting with TERM\_LISTEN (2023)  
     opcode, 3-9, 8-36, 8-59  
   deleting with  
     TERM\_RESET\_COMMAND\_MODES  
     (2088) opcode, 3-9, 8-59  
   hardware requirements for, 8-59  
   restoring, 8-58  
   structure for, 8-57  
 INTERRUPT\_KEY\_DISABLE (246)  
   opcode, A-16  
 INTERRUPT\_KEY\_ENABLE (245)  
   opcode, A-16  
 Interrupts, 3-2, 3-4  
   minimum time between, 3-4, 8-37  
   raw input and, 7-3, 7-4, 7-7  
 INVALID\_SEQUENCE input request, 5-2  
 INVERSE\_VIDEO\_OFF output request, 5-11, 5-23  
 INVERSE\_VIDEO\_ON output request, 5-11, 5-23

## K

Key bit mask, structure for, 8-59  
 Key numbers, 4-18, 4-20  
 KEY\_CLICK\_OFF output request, A-10  
 KEY\_CLICK\_ON output request, 5-9, A-10  
 Key-click option, 8-43  
 KNOCK\_DOWN\_FORM (240) opcode, A-16  
 KNOCK\_DOWN\_FORM\_OK (264) opcode, A-16  
 K-Series hardware, 1-29, **1-29**, 3-8, 8-29, 8-34, 8-59, A-2, A-15, A-16, A-17, A-18, A-22  
   device configuration using, 1-30  
   features for an application program, 1-22  
   interrupt table for, 1-22  
   K101 I/O adapter, 1-31  
   K110 I/O adapter, 1-32  
   K111 I/O adapter, 1-31  
   K118 I/O adapter, 1-32, E-4

- list of multicomunications adapters, 1-29
- support for dial-out, 1-27

## L

### Languages

- setting, 5-30
- support for multinational, 1-17

- Latin alphabet No. 1 character set, 1-17, 5-30, B-5, F-1

### Layers

- access, 1-3, 1-5
- asynchronous access, 1-6
- Console Controller access, 1-7
- terminal-interface, 1-3, 1-5

- LEAVE\_GRAPHICS\_MODE output

- request, 5-11, 5-27

- LEAVE\_INSERT\_MODE output request, A-9

- LEAVE\_MONITOR\_MODE output request, A-9

- LEFT input request, 4-24

- LEFT output request, 5-10, 5-14, 7-26

- LF control character, 5-31, 7-26, 7-27, A-13, A-14

### Limits

- pause-lines, 2-8, 8-61
- primary-window, 8-61
- typeahead, 2-6, 8-68

- Line editing, 2-2, 5-1, 7-14

- Line length, terminal screen, 5-7, 8-24, 8-45, 8-76

- Line operations, output requests
- performing, 5-19

- Line truncating, 5-7

- Line wrapping, 2-3, 4-4, 5-7, 7-25, 8-54

- Line-mode I/O, 2-1

### Lines

- dial-out, 1-27
- dial-up, 1-26, 1-27, 8-36
- recovering from hang-up on, 8-36

- list\_devices command, E-2, E-4

- list\_terminal\_types command, E-2, E-4

- LISTEN (203) opcode, A-16

- Location of primary window, 8-62

- Locking shift introducer (LSI), B-4

- Login or slave configuration, 1-37

- logout -hold command, 8-39

## M

- Mapping, input, 1-11, 4-14, 4-15

- Mark parity, 1-38

- maximum\_length argument of s\$open
- subroutine, 6-8

- Message, writing a system, 8-76

- Metacharacters, 1-23, 1-39

- Mode attributes, 5-22, A-12

- Modem connections, 1-26, 1-27

### Modes

- formatted I/O, 1-11

- graphics, 5-27

- output requests for, 5-26

- processed raw input, 4-15

- undisplayable notation character, 5-32

- unprocessed raw input, 1-22, 4-15

- Modifiers, shift, 4-18, 4-22

- Monitor terminal connection, 1-29

- Monitoring asynchronous device

- channels/subchannels, E-5

- Moving primary windows, 1-14, 8-70

- Multibyte characters, 2-10, 7-15

- Multicomunications I/O adapters, 1-29

## N

- National Language Support (NLS), 1-3, 1-17, 5-30, B-5

- NEW\_LINE output request, 5-9, 5-14, 5-17

- s\$seq\_write and, 7-25

- s\$seq\_write\_partial and, 7-31

- NEXT\_SCREEN input request, 2-7, A-4

- NLS. *See* National Language Support (NLS)

- No parity (parity none), 1-38

- NO\_PAUSE input request, 2-8, 8-72

- Normal raw mode, 3-4, 4-2, 7-3, 7-5, 7-7, 8-5, 8-7, 8-56, A-5

- No-wait mode, 1-21, 8-7

- s\$open and, 6-6

- s\$read\_raw and, 3-5, 3-6, 3-7, 7-4, 7-7

- s\$seq\_read and, 2-9, 7-15

- s\$seq\_write and, 2-13, 7-27

- s\$write\_raw and, 7-22

## O

- Odd parity, 1-38

- Old asynchronous driver, 1-1, 1-2, 1-39
- flow control and, 1-18

### Opcodes

- affecting all communications media, 1-6, 8-15, 8-20, 8-37

- affecting the primary window and subwindow, 8-15, 8-20, 8-21
    - affecting the RS-232-C communications medium, 1-6, 8-15, 8-20, 8-27
    - affecting the terminal, 8-15, 8-20, 8-40
      - global, 8-15, 8-19, 8-24
      - include file for, 1-23, 8-16
      - standard asynchronous, 8-15
      - window terminal, 8-15, 8-19
  - Open actions, 8-39
  - Opening a port, 6-1, 6-6, 6-11
  - Operating values, structure for, 8-32
  - Original subwindow, 1-14, 2-1, 2-3, 4-2, 8-47, 8-48, 8-49, 8-50, 8-51, 8-72, 8-73
  - OS TELNET. *See* `telnet_al` access layer
  - Output
    - discarding, 8-52
    - formatted sequential, 4-2, 4-4
    - generic, 7-1, 7-25, 7-31
    - graphics mode, 5-26
    - pending, 8-25
    - raw, 3-9, 7-22
    - resetting, 2-8, 8-72
    - time limits, 8-2
    - translated, 5-6
    - types of, 1-11
    - using output flow control, 1-19
  - Output area, 2-4
    - maximum size of, 2-6
    - minimum size of, 2-6
  - Output buffers, system, 3-10
  - Output flow control, 1-6, 1-7, 1-19, 8-32, 8-33, A-2
  - Output requests. *See* Generic output requests
  - Output section of a TTP, 5-6, 5-7
  - Overlay mode, 8-43
  - Overrun errors, 1-29, 7-11, 7-13, 7-18, 7-20
- P**
- Parameters, primary-window, 8-63, 8-73, 8-82
  - Parity
    - compatibility issues concerning, A-1
    - configuring, 1-38, A-22
    - types of, 1-38, 8-34
  - Partial input, 8-8
  - Partial output, 3-10, 8-8
  - Pause processing, 2-7, 4-4, 7-14, 7-25
    - avoiding the use of flow-control keys
      - with, 2-8
    - pause characters, 2-3, 8-60
    - pause-lines limit, 2-8, 8-61
    - screen options controlling, 2-8, 8-43, 8-44, A-3
  - Pending output, 8-25
  - Performance, monitoring system, E-5
  - Ports
    - attaching, 6-1, 6-2
    - attaching and opening, 6-1
    - closing, 9-1, 9-2
    - detaching, 9-1, 9-5
    - no-wait mode, 1-20, 8-7
    - opening, 6-1, 6-6, 6-11
    - reading from, 7-3, 7-14
    - wait mode, 1-20, 8-5
    - writing to, 7-22, 7-25, 7-31
  - `POSITION_CURSOR` output request, 5-9, 5-12, 5-15, A-12
  - POSIX mode, 7-3, 7-22
  - Primary windows, **1-13**, 2-1, 2-3
    - current, 1-14
    - current parameters of, 8-63
    - default parameters of, 8-41
    - opcodes for, 8-15, 8-20, 8-21, 8-40, 8-46
    - restoring default parameter settings
      - of, 8-73
    - size limits of, 8-61
    - structure for default parameters, 8-41
    - structure for size and location, 8-62
    - title (name) of, 8-64
    - top, 8-72
  - Printers, controlling with the VOS spooler, 8-26
  - Privileged-terminal configuration, 1-39
  - Processed I/O programming approach, 1-13, 4-2
  - Processed raw input, 1-11, 4-15
    - BDI sequences returning, 4-16
    - compatibility issues concerning, A-6
    - enabling a mode, 4-13, 8-55
    - function-key input mode, 4-15, 4-20, 7-4
    - generic input mode, 4-15, 4-16, 4-23, 7-4
    - translated input mode, 4-15, 4-19, 7-4
    - types of, **4-13**, 8-56
  - Prompt character string, 4-9, 8-61
- R**
- Raw bytes, return of, 4-18

- Raw input, 4-15
    - application-managed I/O approach for, 4-2
    - collecting with `s$read_raw`, 7-3
    - processed, 4-13
    - unprocessed, 1-12, **3-3**
  - Raw input modes, 8-55, 8-56
    - changing, 8-55, 8-56
    - compatibility issues concerning, A-4
    - no-wait mode and, 7-4, 7-7, 8-7
    - setting, 8-55, 8-56
    - wait mode and, 7-4, 7-5, 8-5
  - Raw output, 1-11, 1-12, 3-9, 7-22
    - no-wait mode and, 7-22
    - wait mode and, 7-22
    - warning about using with sequential (command-line) input, 1-12
  - Raw record mode, 1-22, 3-4, **3-6**, 4-2, 7-3, 7-5, 7-7, 8-5, 8-7, 8-56, A-5
  - Raw table mode, 3-4, **3-5**, 4-2, 7-3, 7-5, 7-7, 8-5, 8-7, 8-56, A-5
  - `RAW_INPUT` request, 4-18, 4-24, 5-5, A-8
  - `recc_al` access layer, 1-7
  - Record terminators, 2-9
    - for raw I/O, 3-4, 7-5
    - for sequential I/O, 7-15
  - Records, reading and writing, 2-1, 3-6, 7-1, 7-14, 7-25
  - `REDISPLAY` input request, 4-16
  - Remote Service Network (RSN)
    - connections, 1-29, 1-33, 1-39
  - Request to send (RTS) signal, 1-28, 1-34, 1-35
  - Requests
    - `analyze_system`, E-5
    - generic input, 2-2, 4-15, 4-16, 4-17, 4-23, **5-1**, 5-5
    - generic output, 2-2, 4-4, 4-5, 5-6, 5-7, 5-12
  - `RESET_25TH_LINE` output request, A-10
  - `RESET_COMMAND_INPUT` (226)
    - opcode, A-17
  - `RESET_OUTPUT` (224) opcode, A-17
  - `RESET_TERMINAL` (230) opcode, A-17
  - `RESET_TERMINAL` output request, 5-11, 5-29
  - `reset-terminal` capability, 5-29
  - Resetting
    - command modes, 2-8, 8-72
    - output, 2-8
  - `RETURN` input request, 2-2, 7-15
  - Return key, 2-9, A-4
  - `RETURN_DOESNT_TAB` screen option, 8-43, 8-44, A-3
  - `RETURN_DOESNT_UNPAUSE` screen
    - option, 2-8, 8-43, 8-44, A-3
  - `RIGHT` output request, 5-10, 5-13, 5-15
  - RS-232-C interface, 1-2, 1-6, 1-7, 1-25, 1-33
    - 8-bit support and, 3-2
    - connection states and, 8-37
    - errors affecting the, 6-9, 6-10, 6-13, 7-9, 7-10, 7-11, 7-12, 7-13, 7-17, 7-18, 7-20, 7-24, 7-30, 7-34
    - holding a connection, 8-39
    - listen operations and, 8-36
    - opcodes affecting the, 8-15, 8-20, 8-26
    - `s$open` and, 6-6
  - RSN connections. *See* Remote Service Network (RSN) connections, 1-33
  - `RUNOUT` (2) opcode, **8-25**, 8-48, 8-72, 9-2
- ## S
- `s$attach_port` subroutine, 1-9, 6-2
  - `s$close` subroutine, 1-9, 8-24, **9-2**
  - `s$control` opcodes, 1-14, 8-15
    - affecting all communications media, 8-37
    - affecting the primary window and subwindow, 8-46
    - affecting the RS-232-C communications medium, 8-27
    - affecting the terminal, 8-40
    - categories of, 8-15, 8-19
    - global, 5-7, 6-6, 8-24, 8-25, 8-48, 8-72, 9-2
    - include file for, 1-23, 8-16
    - issuing before `s$open`, 6-6
    - list of, 8-19, 8-20, 8-21
    - window terminal. *See the* individual `TERM_` opcodes
  - `s$control` subroutine, 1-13, 1-14, 8-1, **8-15**, 8-24
    - `control_info` argument of, 8-16
    - include files for `control_info`, 8-16
  - `s$control_device` subroutine, 8-1, **8-80**
    - opcodes for, 8-82
  - `s$detach_port` subroutine, 1-9, 9-2, 9-5
  - `s$open` subroutine, 1-9, 1-13, 1-14, 6-6, 8-24, 8-36
  - `s$read` subroutine, 1-9, 7-1, 8-9
  - `s$read_code` subroutine, 1-9, 7-1
  - `s$read_device_event` subroutine, 8-1, 8-8, **8-12**

- s\$read\_event subroutine, 8-8, 8-12
- s\$read\_form subroutine, 8-24
- s\$read\_raw subroutine, 1-9, 1-12, 3-1, 3-3, 4-13, 4-15, 7-1, **7-3**, 8-2, 8-24
  - BDI sequences and, 4-16
  - compatibility issues concerning, A-5
  - function-key input mode and, 4-20, 7-3
  - generic input mode and, 4-24, 7-3
  - handling processed raw input, 4-15, 4-16
  - no-wait mode and, 3-3, 3-5, 3-6, 3-7, 7-4, 7-7, 8-7
  - POSIX mode, 7-3, 7-22
  - translated input mode and, 4-19, 7-3
  - wait mode and, 3-3, 3-5, 3-6, 3-7, 7-4, 7-5, 8-5
- s\$seq\_open subroutine, 1-9, 6-11
- s\$seq\_read subroutine, 1-9, 1-13, 2-1, 4-1, 7-1, **7-14**, 8-24
  - formatted sequential input and, 4-9
  - multibyte characters and, 2-10
  - no-wait mode and, 2-9, 7-15, 8-7
  - simple sequential I/O and, 2-9
  - wait mode and, 2-9, 7-15, 8-5
  - warning about using with
    - s\$write\_raw, 1-12
- s\$seq\_write subroutine, 1-9, 1-12, 2-1, 5-6, 7-1, **7-25**, 8-9, 8-24
  - formatted sequential output and, 4-5, 7-25
  - no-wait mode and, 2-13, 7-27, 8-7
  - simple sequential I/O and, 2-13, 7-25
  - wait mode and, 2-13, 7-27, 8-5
- s\$seq\_write\_partial subroutine, 1-9, 1-12, 2-1, 2-13, 4-5, 5-6, 5-26, 7-1, **7-31**, 8-5, 8-9, 8-24
- s\$set\_io\_time\_limit subroutine, 1-21, **1-21**, 3-10, 7-22, 7-27, 8-1, **8-2**, 8-5, 8-8
- s\$set\_no\_wait\_mode subroutine, 1-21, 8-1, 8-2, **8-7**
- s\$set\_wait\_mode subroutine, 1-21, 8-1, 8-2, **8-5**
- s\$http\_get\_output\_cap subroutine, 5-8, A-10, A-11, C-6
- s\$http\_get\_supported\_output\_seqs subroutine, 8-47, C-5
- s\$wait\_event subroutine, 8-2, 8-8, 8-12
- s\$write subroutine, 1-9, 7-2
- s\$write\_code subroutine, 1-9, 7-2
- s\$write\_partial subroutine, 7-2
- s\$write\_partial\_code subroutine, 7-2
- s\$write\_raw subroutine, 1-9, 1-12, 3-1, 3-9, 7-1, **7-22**, 8-2, 8-8, 8-9, 8-24
  - no-wait mode and, 7-22, 8-7
  - wait mode and, 7-22, 8-5
  - warning about using with
    - s\$seq\_read, 1-12
- s\$write\_wrap subroutine, 7-2
- s\$write\_wrap\_indent subroutine, 7-1
- s\$write\_wrap\_partial subroutine, 7-1
- Sample window terminal program, D-1
- Screen
  - height, 8-41
  - preferences, 8-44
    - compatibility issues concerning, A-10
    - turning the display on and off, 5-30
    - width, 8-45
- SCREEN\_OFF output request, 4-8, 5-9, 5-11, 5-29
- SCREEN\_ON output request, 4-8, 5-9, 5-11, 5-29, 5-30
- SCROLL\_DOWN output request, 5-10, 5-16, 5-22
- SCROLL\_UP output request, 5-10, 5-16, 5-22
- Scrolling
  - down, 5-16
  - options, 8-43
  - regions, 5-13, 5-21
  - up, 5-16
- SEND\_BREAK (275) opcode, A-14
- Sequences
  - BDI, 4-16
  - for specifying generic output requests, 4-5
- Sequential I/O
  - application-managed I/O approach
    - for, 1-12, 4-1
  - no-wait mode and, 2-9, 2-13, 7-15, 7-27
  - simple sequential I/O approach for, 1-12, 2-1
  - types of, 7-14
  - wait mode and, 2-9, 7-15, 7-27
- Sequential input, warning about using with raw output, 1-12
- Set operation, 8-16
- SET\_ATTRIBUTE\_CHAR output request, 5-9, A-12
- SET\_ATTRIBUTES output request, 5-9, A-12
- SET\_BLACK\_ON\_WHITE output request, A-10
- SET\_BREAK\_TABLE (225) opcode, A-6, A-17
- SET\_CONTINUE\_CHARS (214) opcode, A-18

- SET\_CURSOR\_BLINKING\_BLOCK output request, A-11
- SET\_CURSOR\_BLINKING\_UNDLIN output request, A-11
- SET\_CURSOR\_FORMAT output request, 5-10, 5-17, A-11
- SET\_CURSOR\_INVISIBLE output request, A-11
- SET\_CURSOR\_STEADY\_BLOCK output request, A-11
- SET\_CURSOR\_STEADY\_UNDERLINE output request, A-11
- SET\_ESCAPE\_CHAR (217) opcode, A-18
- SET\_INFO (202) opcode, 4-13, 8-56, A-6, A-8, A-18
- SET\_INPUT\_SECTION (269) opcode, A-17
- SET\_INSERT\_CHAR output request, A-9
- SET\_INTERRUPT\_TABLE (266) opcode, A-6, A-18
- SET\_JUMP\_SCROLL output request, A-10
- SET\_LANGUAGE (258) opcode, A-17
- SET\_LINE\_LENGTH (209) opcode, A-18
- SET\_MAX\_BUFFER\_SIZE (247) opcode, A-18
- SET\_MODE\_ATTRIBUTES output request, 5-11, 5-23, 5-24, A-12
- SET\_MODES (207) opcode, 4-13, 8-56, A-6, A-8, A-18
- SET\_OUTPUT\_FLOW (216) opcode, A-18
- SET\_PAUSE\_CHARS (215) opcode, A-19
- SET\_PAUSE\_LINES (211) opcode, A-19
- SET\_PRIMARY\_USER (231) opcode, A-19
- SET\_PROMPT\_CHARS (208) opcode, A-19
- SET\_SCREEN\_PAGE\_SIZE output request, A-9
- SET\_SCREEN\_SIZE (210) opcode, A-19
- SET\_SCROLLING\_REGION output request, 5-10, 5-13, 5-21, A-11
- SET\_SMOOTH\_SCROLL output request, 5-9, A-10
- SET\_SPOOLER (4) opcode, 8-26
- SET\_TAB (213) opcode, A-20
- set\_terminal\_parameters command, E-2, E-4
- SET\_TERMINAL\_SETUP (271) opcode, A-20
- SET\_TERMINAL\_TYPE (218) opcode, A-20
- SET\_WHITE\_ON\_BLACK output request, A-10
- Setup and initialization operations, **6-1**
- Setup, configuration, 8-24, 8-42, 8-45
- Shift modifiers, 4-18, 4-22
- Simple sequential I/O, 1-10, **2-1**, 7-14, 7-25
  - character echoing and, 8-50
  - characteristics of, 2-1
  - compatibility issues concerning, A-3
  - generic input requests available with, 5-1, 5-5
  - pause characters and, 8-60
  - pause-lines limit and, 8-61
  - prompt character string and, 8-61
  - reenabling after formatted I/O mode, 8-51
  - retrieving simple sequential input, 2-9
  - sending simple sequential output, 2-13
  - switching the subwindow to formatted I/O mode, 8-50
- Single-shift characters, B-4
- Size of primary window, 8-62
- SP character, B-1
- Space parity, 1-38
- SPECIAL\_N output requests, A-10
- STANDOUT\_OFF output request, 5-11, 5-25
- STANDOUT\_ON output request, 5-10, 5-11, 5-21, 5-23, 5-24, 5-25, 5-29
- START\_25TH\_LINE output request, A-10
- Status area, 2-4, 8-74
- Status message, 2-4
  - clearing and displaying, 8-75
  - compatibility issues concerning, A-10
  - structure for appending error message to, 8-75
- Stop bits, configuring, 1-39, 8-35
- Structures
  - for appending error message to status message, 8-75
  - for flow control, 8-30, 8-33
  - for interrupt table, 8-57
  - for key bit mask, 8-59
  - for operating values, 8-32
  - for primary-window defaults, 8-41
  - for primary-window parameters, 8-64
  - for primary-window size and location, 8-62
  - for primary-window size limits, 8-62
  - for screen preferences, 8-43, 8-44
  - for subwindow size, 8-66
  - for subwindow values, 8-48
  - for tab settings, 8-67
  - for toggling display of status message, 8-75
  - include file for, 1-23, 8-16
  - longmap, 1-24, 8-17



- shortmap, 1-24, 8-17, 8-41, 8-64
- SUB character, 2-10, 7-15, B-3
- Subchannels, 1-30
  - characteristics of asynchronous, 1-36
  - pinouts on full-modem asynchronous, 1-34
  - pinouts on null-modem
    - asynchronous, 1-35
- Subroutines
  - alternative I/O, 1-9, 7-1
  - for attaching and opening a port, 1-9
  - for closing and detaching a port, 1-9, 9-1
  - for control operations, 1-9, 8-1, 8-15, 8-80
  - for I/O operations, 1-9, 7-1
  - for setup and initialization, 6-1
  - s\$attach\_port, 1-9, 6-2
  - s\$close, 1-9, **9-2**
  - s\$control, 1-13, 8-1, **8-15**
  - s\$control\_device, 8-1, **8-80**
  - s\$detach\_port, 1-9, 9-2, 9-5, **9-5**
  - s\$open, 1-9, 1-13, 1-14, 6-6, 8-36
  - s\$read, 1-9, 7-1, 8-9
  - s\$read\_code, 1-9, 7-1
  - s\$read\_device\_event, 8-1, 8-8, **8-12**, 8-12
  - s\$read\_event, 8-8
  - s\$read\_raw, 1-9, 1-12, 3-1, 3-3, 4-13, 4-14, 4-15, 7-1, **7-3**, 7-3, 8-2, 8-5, 8-24
  - s\$seq\_open, 1-9, 6-11
  - s\$seq\_read, 1-9, 1-12, 2-1, 2-9, 7-1, **7-14**, 7-14, 8-5, 8-24
  - s\$seq\_write, 1-9, 1-12, 2-1, 2-13, 5-6, 7-1, **7-25**, 7-25, 8-5, 8-9, 8-24
  - s\$seq\_write\_partial, 1-9, 1-12, 2-1, 2-13, 5-6, 5-26, 7-1, **7-31**, 7-31, 8-5, 8-9, 8-24
  - s\$set\_io\_time\_limit, 1-21, 3-10, 8-1, **8-2**
  - s\$set\_no\_wait\_mode, 1-21, 8-1, 8-7
  - s\$set\_wait\_mode, 1-21, 8-1, 8-5
  - s\$http\_get\_output\_cap, 5-8, A-10, A-11, C-6
  - s\$http\_get\_supported\_output\_seqs, 8-47, C-5
  - s\$wait\_event, 8-2, 8-8
  - s\$write, 1-9, 7-1
  - s\$write\_code, 1-9, 7-1
  - s\$write\_partial, 7-1
  - s\$write\_partial\_code, 7-1
  - s\$write\_raw, 1-9, 1-12, 3-1, 3-9, 7-1, **7-22**, 8-2, 8-5, 8-7, 8-8, 8-9, 8-24
  - s\$write\_wrap, 7-1
  - s\$write\_wrap\_indent, 7-1
  - s\$write\_wrap\_partial, 7-1
  - TTP, 3-2, 8-47, C-1
  - Subwindows, **1-14**, 4-2, 4-3, 7-25
    - arranging, 1-14
      - issuing TTP subroutine after, C-5
    - bottom, 1-14, 1-15, 8-74
    - creating, 1-14
    - creating to use formatted sequential I/O, 4-3, 8-47
    - current I/O, 1-14, 1-15, 4-2, 4-5, 4-9, 8-47, 8-74
    - deleting, 8-49
    - display of, 2-1
    - flow control and, 2-8
    - ID for, 1-14, 8-48, 8-49, 8-50, 8-74
    - management of, 4-2
    - opcodes for, 8-15, 8-20, 8-21, 8-46
    - organization for simple sequential I/O, 2-3
    - original, 1-14, 2-1, 2-3, 4-2, 8-47, 8-48, 8-49, 8-50, 8-51, 8-72, 8-73
    - redisplay/refresh of, 2-3, 8-26, 8-48, 8-72
      - See also* hold\_attached and hold\_open switches
    - s\$seq\_write and, 7-25
    - s\$write\_raw and, 7-22
    - simple sequential I/O, 2-1, 2-3
    - structure for, 8-48
    - structure for size of, 8-66
    - switching to use formatted I/O mode, 2-15, 2-17, 4-3, 5-7, 8-50
    - top, 1-14, 1-15, 8-74
    - updating the display of, 8-52
  - Switches, 2-16, 4-5, 6-2, 6-4, 6-11, 7-25, 8-47, 8-48, 8-51, 8-71
    - See also* hold\_attached and hold\_open switches

## T

- TAB input request, 8-52
- Tab settings, 8-67
- TELNET. *See* telnet\_al access layer
- telnet\_al access layer, 1-3, 1-6, 1-7, 3-4, 3-5, 3-6, 8-26
- TERM\_CONFIGURE (227) opcode, A-20

- TERM\_CREATE\_SUBWIN (2038)
  - opcode, 2-17, 4-3, 8-17, **8-47**, 8-49, 8-74, C-5
- TERM\_DELETE\_SUBWIN (2039) opcode, 4-3, **8-49**
- term\_dil device driver, 1-3, 1-5
- TERM\_DISABLE\_CACHED\_IO\_OPCODE (2109) opcode, **8-50**
- TERM\_DISABLE\_ECHO (2043) opcode, 2-2, 4-9, **8-50**, A-15
- TERM\_DISABLE\_FMT\_IO\_MODE (2047) opcode, 2-17, 4-4, **8-50**, 8-51
- TERM\_DISABLE\_IMMEDIATE\_PAINT (2098) opcode, 8-52
- TERM\_DISABLE\_KEY (2045) opcode, **8-52**, A-16
- TERM\_DISCARD\_INPUT (2040) opcode, **8-52**, A-15
- TERM\_DISCARD\_OUTPUT (2041) opcode, 7-27, **8-52**, A-15
- TERM\_ENABLE\_CACHED\_IO\_OPCODE (2108) opcode, **8-50**, 8-71
- TERM\_ENABLE\_ECHO (2042) opcode, 4-9, **8-50**, A-15
- TERM\_ENABLE\_FMT\_IO\_MODE (2046) opcode, 2-16, 4-3, 6-2, 8-47, **8-50**, 8-51, A-12
- TERM\_ENABLE\_IMMEDIATE\_PAINT (2097) opcode, 4-8, 5-8, 5-9, **8-52**
- TERM\_ENABLE\_KEY (2044) opcode, **8-52**, A-16
- TERM\_GET\_BAUD\_RATE (2003) opcode, **8-27**, 8-27
- TERM\_GET\_BITS\_PER\_CHAR (2005) opcode, 8-28
- TERM\_GET\_BREAK\_ACTION (2050) opcode, 8-53
- TERM\_GET\_CONFIG (265) opcode, A-20
- TERM\_GET\_CONNECTION\_STATE (2002) opcode, 8-36, **8-37**
- TERM\_GET\_CONTINUE\_CHARS (2052) opcode, 8-54
- TERM\_GET\_CURSOR\_FORMAT (2054) opcode, 8-54
- TERM\_GET\_INPUT\_FLOW\_INFO (2007) opcode, 1-19, **8-29**, A-2
- TERM\_GET\_INPUT\_MODE (2056) opcode, 3-3, **8-55**, 8-56
- TERM\_GET\_INPUT\_SECTION (2058) opcode, 5-5, **8-56**, A-16
- TERM\_GET\_INTERRUPT\_TABLE (2060) opcode, 1-22, 3-9, **8-57**, 8-59, A-15
- TERM\_GET\_KEY\_BIT\_MASK (2062) opcode, **8-59**, A-15
- TERM\_GET\_KNOCKDOWN\_ID (2105) opcode, **8-60**, 8-69, A-16
- TERM\_GET\_MAX\_BUFFER\_SIZE (2009) opcode, 8-38
- TERM\_GET\_OPEN\_ACTION (2011) opcode, 8-39
- TERM\_GET\_OPERATING\_VALUES (2013) opcode, 8-27, 8-28, **8-31**, 8-34, 8-35, A-20
- TERM\_GET\_OUTPUT\_FLOW\_INFO (2015) opcode, 1-19, **8-32**, 8-33, A-2
- TERM\_GET\_PARITY (2017) opcode, 8-28, **8-34**
- TERM\_GET\_PAUSE\_CHARS (2068) opcode, 8-60
- TERM\_GET\_PAUSE\_LINES (2070) opcode, 8-61
- TERM\_GET\_PROMPT\_CHARS (2072) opcode, **8-61**, A-15
- TERM\_GET\_PWIN\_BOUND\_LIMS (2080) opcode, 8-61
- TERM\_GET\_PWIN\_BOUNDS (2078) opcode, 8-62
- TERM\_GET\_PWIN\_DEFAULTS (2034) opcode, 1-24, 8-17, **8-41**
- TERM\_GET\_PWIN\_PARAMS (2082) opcode, 1-24, 8-17, **8-63**
- TERM\_GET\_PWIN\_TITLE (2084) opcode, 8-65
- TERM\_GET\_SCREEN\_PREF (2026) opcode, **8-42**, **8-43**
- TERM\_GET\_SCREEN\_WIDTH (2028) opcode, 5-7, 8-24, **8-44**, 8-76
- TERM\_GET\_SESSION\_ID\_OPCODE (2114) opcode, **8-65**
- TERM\_GET\_STOP\_BITS (2019) opcode, 8-35
- TERM\_GET\_SUBWIN\_BORDER\_OPCODE (2112) opcode, **8-65**
- TERM\_GET\_SUBWIN\_BOUNDS (2066) opcode, 8-17, **8-66**
- TERM\_GET\_TABS (2074) opcode, 8-67
- TERM\_GET\_TERMINAL\_SETUP (2030) opcode, 5-7, 8-42, **8-45**, A-16

- 
- TERM\_GET\_TERMINAL\_TYPE (2032)  
opcode, 8-46
  - TERM\_GET\_TYPEAHEAD\_LINES (2076)  
opcode, 8-68
  - TERM\_GET\_UNDISP\_MODE (2064)  
opcode, 5-32, **8-68**
  - TERM\_HANGUP (2001) opcode, 1-28, 7-12,  
7-20, 7-24, 7-30, 7-34, 8-36, **8-39**,  
A-15
  - TERM\_HOLD\_CONNECTION (2021)  
opcode, **8-39**, A-16
  - TERM\_KNOCK\_DOWN\_FORM (2102)  
opcode, 8-60, **8-69**, A-16
  - TERM\_KNOCK\_DOWN\_FORM\_OK (2104)  
opcode, 8-60, **8-70**, A-17
  - TERM\_LISTEN (2023) opcode, 1-28, 3-9,  
7-12, 7-20, 7-24, 7-30, 7-34, **8-36**,  
8-39, 8-59, A-17  
See also Force-listen configuration
  - TERM\_MOVE\_PWIN\_TO\_BOTTOM (2086)  
opcode, 8-70
  - TERM\_MOVE\_PWIN\_TO\_TOP (2087)  
opcode, **8-70**, **8-71**, A-19
  - TERM\_OPEN\_EXISTING\_WINDOW\_OPCODE  
(2107) opcode, 6-6, **8-71**
  - TERM\_POSIX\_CLOSE\_OPCODE (2110)  
opcode, **8-71**
  - TERM\_POSIX\_CLOSE\_OPCODE (2111)  
opcode, **8-71**
  - TERM\_POSIX\_GETATTR\_OPCODE (2115)  
opcode, **8-72**
  - TERM\_POSIX\_SETATTR\_OPCODE (2116)  
opcode, **8-72**
  - TERM\_RESET\_COMMAND\_MODES (2088)  
opcode, 2-8, 3-9, 4-3, 8-51, 8-52,  
8-59, **8-72**, A-17
  - TERM\_RESET\_OUTPUT (2089) opcode, 2-8,  
4-3, 8-51, 8-52, **8-72**, A-17
  - TERM\_RESTORE\_PWIN\_PARAMS\_DEVOP  
(202) opcode, 8-83
  - TERM\_SAVE\_COMMAND\_INPUT (2090)  
opcode, **8-73**, A-17
  - TERM\_SAVE\_PWIN\_PARAMS\_DEVOP (201)  
opcode, 8-83
  - TERM\_SEND\_BREAK (2106) opcode, **8-46**,  
A-14
  - TERM\_SET\_BAUD\_RATE (2004)  
opcode, **8-27**, 8-27
  - TERM\_SET\_BITS\_PER\_CHAR (2006)  
opcode, **8-28**, A-2
  - TERM\_SET\_BREAK\_ACTION (2051)  
opcode, 3-2, 7-10, 7-11, 7-17, 7-18,  
8-43, **8-53**, A-15
  - TERM\_SET\_CONTINUE\_CHARS (2053)  
opcode, 2-2, **8-54**, A-17
  - TERM\_SET\_CURRENT\_SUBWIN (2091)  
opcode, **8-73**, A-10
  - TERM\_SET\_CURSOR\_FORMAT (2055)  
opcode, 5-17, **8-54**, 8-82, A-11
  - TERM\_SET\_FORWARDING\_TIMER (2095)  
opcode, 3-4, **8-37**
  - TERM\_SET\_INPUT\_FLOW\_INFO (2008)  
opcode, 1-19, 7-18, 7-20, **8-29**, 8-33,  
A-2
  - TERM\_SET\_INPUT\_MODE (2057)  
opcode, 1-12, 3-3, 3-5, 3-7, 4-13, 7-3,  
**8-55**, 8-56, A-6, A-8, A-18
  - TERM\_SET\_INPUT\_SECTION (2059)  
opcode, 5-5, **8-56**, A-18
  - TERM\_SET\_INTERRUPT\_TABLE (2061)  
opcode, 1-22, 3-9, 8-36, **8-57**, 8-59,  
A-6, A-17, A-18
  - TERM\_SET\_KEY\_BIT\_MASK (2063)  
opcode, 8-59
  - TERM\_SET\_MAX\_BUFFER\_SIZE (2010)  
opcode, 6-8, 7-11, 7-18, **8-38**, A-18
  - TERM\_SET\_OPEN\_ACTION (2012)  
opcode, 6-6, **8-39**
  - TERM\_SET\_OPERATING\_VALUES (2014)  
opcode, 3-4, 8-27, 8-28, **8-31**, 8-34,  
8-35, A-2, A-20
  - TERM\_SET\_OUTPUT\_FLOW (2016)  
opcode, A-18
  - TERM\_SET\_OUTPUT\_FLOW\_INFO (2016)  
opcode, 1-19, 8-30, **8-32**, 8-33, A-2
  - TERM\_SET\_PARITY (2018) opcode, **8-34**,  
A-2
  - TERM\_SET\_PAUSE\_CHARS (2069)  
opcode, 2-3, **8-60**, A-19
  - TERM\_SET\_PAUSE\_LINES (2071)  
opcode, 2-9, **8-61**, A-19
  - TERM\_SET\_PROMPT\_CHARS (2073)  
opcode, 2-4, **8-61**, A-19
  - TERM\_SET\_PWIN\_DEFAULTS (2035)  
opcode, 1-24, 8-17, **8-41**
  - TERM\_SET\_PWIN\_PARAMS (2083)  
opcode, 1-24, 8-17, **8-63**, 8-82
  - TERM\_SET\_PWIN\_TITLE (2085)

- opcode, 8-64
  - TERM\_SET\_PWIN\_TO\_DEFAULTS (2092)
    - opcode, 8-73
  - TERM\_SET\_SCREEN\_PREF (2027)
    - opcode, 2-8, 3-2, **8-42**, **8-43**, A-3, A-10
  - TERM\_SET\_STOP\_BITS (2020)
    - opcode, 8-35
  - TERM\_SET\_SUBWIN\_BORDER\_OPCODE (2113) opcode, **8-65**
  - TERM\_SET\_SUBWIN\_BOUNDS (2067)
    - opcode, 2-1, 5-13, 8-17, **8-66**, C-5
  - TERM\_SET\_SUBWIN\_TO\_BOTTOM (2029)
    - opcode, 8-73
  - TERM\_SET\_SUBWIN\_TO\_TOP (2025)
    - opcode, **8-74**, C-5
  - TERM\_SET\_TABS (2075) opcode, **8-67**, A-20
  - TERM\_SET\_TERMINAL\_SETUP (2031)
    - opcode, 5-7, 8-24, 8-42, **8-45**, A-18, A-19, A-20
  - TERM\_SET\_TERMINAL\_TYPE (2033)
    - opcode, **8-46**, A-20
  - TERM\_SET\_TYPEAHEAD\_LINES (2077)
    - opcode, 2-6, **8-68**
  - TERM\_SET\_UNDISP\_MODE (2065)
    - opcode, 5-32, 7-26, **8-68**, A-14
  - TERM\_STATUS\_MSG\_CHANGE (2096)
    - opcode, 2-4, 8-16, **8-74**, A-10
  - TERM\_SWITCH\_TO\_WIN\_MGR (2036)
    - opcode, 8-46
  - TERM\_WRITE\_SYSMSG\_NOBEEP (2103)
    - opcode, 8-74, **8-76**, A-10, A-20
  - TERM\_WRITE\_SYSTEM\_MESSAGE (2101)
    - opcode, 8-74, **8-76**, A-10, A-20
  - Terminal independence, 1-3, 2-2, 3-2, 4-12
  - Terminal opcodes, 8-40, A-15, A-16, A-17, A-18
  - Terminal type (TTP), 1-3, 1-5, 1-17, 2-2
    - ascii terminal type, 2-4
    - attributes section of, 2-14, 4-8, 5-6, 5-7, 5-22, 5-24
    - character-translation section of, 4-15, 4-19, 5-6
    - commands for setting up, E-3
    - compiling, E-2, E-3
    - configuration section of, 5-6
    - configuration setup of, 8-42, 8-45
    - defining capabilities in, 5-8
    - handling of function keys, 4-22
    - handling of generic input, 4-23
    - handling of translated input, 4-19
    - height configuration variable, 8-42
    - input section of, 2-10, 4-15, 5-5, 8-56
    - installing, E-2, E-3
    - keyboard section of, 4-15, 4-22
    - National Language Support (NLS) and, 1-17
    - output section of, 2-14, 4-8, 5-6, 5-7
      - line-graphics entry, 5-26
    - overview of input mapping, 4-14, 4-15
    - purpose of, 1-3
    - request specifiers for generic input requests, 5-2
    - specifying the name of, 8-46
    - subroutines, 3-2, 5-8, 8-46, 8-47, A-10, A-11, C-1
    - width configuration variable, 8-24, 8-45
  - Terminal-independent layer, 1-5
  - Terminal-interface layers, 1-3, 1-5
  - Terminals
    - attaching and opening, 6-2, 6-6, 6-11
    - character-mode, 1-2, 1-3
    - closing and detaching, 9-1
    - controlling, 8-15
    - flow control and, 1-18
    - opcodes affecting, 8-15, 8-20, 8-40
    - resetting, 5-29
  - Terminal-type definition file. *See* Terminal type (TTP), 1-3
  - Terminators, record, 2-9, 7-15
  - Time limits for I/O operations, 8-2
  - Timer, forwarding, 8-37
  - tli\_term1 access layer, 1-8
  - Tools for asynchronous communications, E-1
  - Trailer bytes, 3-6, 3-7, **3-8**
  - Translated input mode, 4-2, 4-13, 4-15, **4-19**, 7-4, 8-5, 8-7, 8-56
    - compatibility issues concerning, A-7
  - TTP. *See* Terminal type (TTP)
  - TWIDDLE input request, A-8
  - Typeahead, 2-2, 7-14, A-3
    - formatted sequential input and, 4-9
  - Typeahead area, 2-4, 2-5, 2-6
  - Typeahead limit, 2-6, 8-68
- ## U
- UNDERSCORE\_OFF output request, 5-11, 5-26

UNDERSCORE\_ON output request, 5-11, 5-26  
 Undisplayable notation character mode, 5-32, 7-26  
 UNFREEZE\_LINES output request, A-11  
 Unprocessed raw I/O, 1-11, 1-12, **3-1**, 4-15  
     characteristics of, 3-1  
     compatibility issues concerning, A-4  
     programming approach for, 1-12  
 Unprocessed raw input modes, 3-4  
     enabling, 8-55  
     normal raw, 3-5, 7-3, 7-5  
     raw record, 3-6, 7-3, 7-5  
     raw table, 3-5, 7-3, 7-5  
 UP output request, 5-10, 5-18  
 update\_channel\_info command, E-2, E-4

## V

Version numbers, 8-16  
 Virtual terminals and printers, input flow control and, 1-18  
 VOS internal character codes, **F-1**  
 VT control character, 5-31, 7-26, 7-27, A-13, A-14

## W

Wait mode, 1-21, 8-5  
     s\$read\_raw and, 3-5, 3-6, 3-7, 7-4, 7-5  
     s\$seq\_read and, 2-9, 7-15  
     s\$seq\_write and, 2-13, 7-27  
     s\$write\_raw and, 7-22  
 White-on-black screen option, 8-43  
 width configuration variable, 8-24, 8-45  
 Window manager, 1-13  
     break to window manager mode, 8-43, 8-46  
     generic input requests available with, 5-2  
 Window terminal driver, 1-2, 1-39  
     application program interface for, 1-8  
     design considerations for, 1-10  
     device interface for, 1-25, 1-33  
     flow control and, 1-18  
     loading, 1-1  
     organization of, 1-3, 1-5  
     overview of, 1-2  
 window\_term device driver, 1-3, 1-5  
 Windows. *See* Primary windows  
 Wired heap, 6-10, 6-13  
 Word processing introducer (WPI), B-4

Wrapping, line, 2-2, 4-4, 7-25, 8-54  
 WRITE\_SYSTEM\_MESSAGE (206)  
     opcode, A-20  
 WRITE\_SYSTEM\_MSG\_NO\_BEEP (259)  
     opcode, A-20

## X

XON/XOFF flow control, 1-6, 1-7, 1-19, 1-20, 8-29, 8-30, 8-32, 8-33

