R007-00

VOS Communications Software: Introduction

Stratus Computer, Inc.

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 COMPUTER, INC., 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 Computer, Inc., 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 Computer, Inc., 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 manuals document all of the subroutines and commands of the user interface. Any other operating-system commands and subroutines are intended solely for use by Stratus personnel and are subject to change without warning.

Stratus, the Stratus logo, Continuum, VOS, Continuous Processing, StrataNET, FTX, and SINAP are registered trademarks of Stratus Computer, Inc.

XA, XA/R, Stratus/32, Stratus/USF, StrataLINK, RSN, Continuous Processing, Isis, the Isis logo, Isis Distributed, Isis Distributed Systems, RADIO, RADIO Cluster, and the SQL/2000 logo are trademarks of Stratus Computer, Inc.

Apple and Macintosh are registered trademarks of Apple Computer, Inc.
IBM PC is a registered trademark of International Business Machines Corporation.
Sun is a registered trademark of Sun Microsystems, Inc.
Hewlett-Packard is a trademark of Hewlett-Packard Company.
UNIX is a registered trademark of X/Open Company, Ltd., in the U.S.A. and other countries.
HP-UX is a trademark of Hewlett-Packard Company.
Manual Name: *VOS Communications Software: Introduction*

Part Number: R007
Revision Number: 00
VOS Release Number:
Printing Date: July 1984

Stratus Computer, Inc.
55 Fairbanks Blvd.
Marlboro, Massachusetts 01752

Preface

The *VOS Communications Software: Introduction (R007)* first introduces the communications software facilities of VOS, the operating system of the Stratus Continuous Processing system, and then documents the VOS service subroutines that serve as interfaces to some of the communications software protocols.

This manual shows the subroutine declarations in VOS PL/I throughout the body of the text, then includes appendixes showing the declarations in each of the other VOS languages.

This manual is intended for system administrators.

Manual Version

This is a new manual.

Manual Organization

This manual contains three chapters and five appendixes.

Related Manuals

Refer to the following Stratus manuals for related documentation.

- *Guide to the Programmable StrataBUS Interface (R024)*
- *VOS Communications Software: Asynchronous Communications (R025)*
- *VOS Communications Software: 3270 Support and 3270 Emulation (R026)*
- *VOS Communications Software: Binary Synchronous Communications (R027)*
- *VOS Communications Software: X.25 and X.29 Programming (R028)*
- *VOS Communications Software: SNA Emulation (R029)*
- *VOS Communications Software: LAPB (R041)*
- *VOS Communications Software: SDLC (R043)*
- *VOS Communications Software: Poll/Select Terminal Support (R044)*

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	F18	* †	* †	5 † or * †	F18
CYCLE	F17	F12	Alt-C	4 †	F17
CYCLE BACK	Shift-F17	Shift-F12	Alt-B	7 †	Shift-F17
DISPLAY FORM	F19	– †	– †	6 † or – †	F19 or Shift–Help
HELP	Shift-F8	Shift-F2	Shift-F2	Shift-F8	Help
INSERT DEFAULT	Shift-F11	Shift-F10	Shift-F10	Shift-F11	F11
INSERT SAVED	F11	F10	F10	F11	Insert_Here
INTERRUPT	Shift-F20	Shift-Delete	Alt-I	1 †	Shift-F20
NO PAUSE	Shift-F18	Shift– * †	Alt-P	8 †	Shift-F18

† 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. It contains the latest information available, including updates and corrections to Stratus manuals and a glossary of terms.
- Stratus offers some of its manuals online, via StrataDOC, an online-documentation product that consists of online manuals and StrataDOC Viewer, delivered on a CD-ROM (note that you must order StrataDOC separately). StrataDOC Viewer allows you to access online manuals from an IBM PC or compatible PC, a Sun® or Hewlett-Packard™ workstation, or an Apple® Macintosh® computer. StrataDOC provides such features as hypertext links and, on the workstations and PCs, text search and retrieval across the manual collection. The online and printed versions of a manual are identical.

If you have StrataDOC, you can view this manual online.

For a complete list of the manuals that are available online as well as more information about StrataDOC, contact your Stratus account representative.

For more information about StrataDOC as well as a complete list of the manuals that are available online, contact your Stratus account representative.

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, `r007`, 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.

Contents

Figures

Tables

Chapter 1:
Overview of VOS Communications Software

Software Products

VOS communications software includes support for asynchronous devices and for a range of synchronous devices and protocols. Table 1-1 outlines the VOS communications software products.

The term *device* is used in this manual to include both asynchronous devices (tapes, printers, and terminals) and synchronous communications lines.

Table 1-1. VOS Communications Software Products

Protocol Products	**Network Products**
BSC FTPS (Ticker) SDLC X.25 LAPB	X.25 virtual circuits X.25 StrataNET
Device-Support Products	**Emulation Products**
3270 terminals Programmable StrataBUS Interface (PSI) X.29 virtual terminals User-supplied terminals User-supplied printers Asynchronous devices	Using BSC: 3270 control units 2780 terminals 3780 terminals HASP workstations RJE facility VISA protocol Using SNA: 3270 control units

The User Program Interfaces

The interfaces to the communications products are the standard VOS I/O subroutines. However, different devices and protocols use different sets of subroutines. These differences depend upon:

- whether or not the device or protocol behaves as a sequential device, and
- whether or not the synchronous device is multiplexed.

The following protocols do not behave as sequential devices and are not multiplexed:

- Binary Synchronous Communications (BSC)
- Financial Ticker Protocol Support (FTPS)
- 2780/3780/HASP Emulation
- Synchronous Data Link Control (SDLC)

These protocols use as interfaces the subroutines that are documented in this manual. These are:

```
s$initialize_device
s$control
s$read_device
s$read_device_event
s$write_device
s$clear_device
```

The remaining protocols and devices use as interfaces only one of the subroutines -- `s$control` -- documented in this manual. In addition to `s$control`, they use subroutines (for example, `s$open`) that are documented in the *Subroutines Manuals*.

This manual provides the following information about the subroutines it documents:

- The purpose of the subroutine and information about how it functions.
- The argument declarations and the subroutine declaration. (These are shown in VOS PL/I throughout the body of the manual. Declarations for the other VOS languages are shown in Appendixes A through D.)
- A description of each of the subroutine's arguments.

This manual does *not* provide information specific to particular software products. For example, you must consult the manual on the BSC protocol for information about the control operations that are valid for that protocol. You must also consult the individual manuals for the declarations of structures (a term that is defined below).

Chapter 2:
Hardware Considerations

It is essential to the success of your application that each device be attached to a port on a line adapter that has the same transmission mode (i.e., asynchronous or synchronous) as the device.

Each synchronous line adapter contains two sockets, but you will use only one. If you are using an RS232C connector, you must plug it into the lower socket; if you are using an RS422 connector, you must plug it into the upper socket.

A synchronous line adapter occupies two channel numbers, although when you configure the device you refer only to the even channel number.

See the *VOS System Administrator's Guide (R012)* for descriptions of the communications controller, communications chassis, and communications line adapters.

Chapter 3:
Subroutine Declarations and Descriptions

This section describes the following subroutines:

- `s$initialize_device`
- `s$control`
- `s$read_device`
- `s$read_device_event`
- `s$write_device`
- `s$clear_device`

These are referred to as the *communications subroutines*.

The subroutine declarations and calls are given here in VOS PL/I. For declarations and calls in VOS BASIC, VOS COBOL, VOS FORTRAN, and VOS Pascal, see Appendixes A through D.

Declaring and Passing Data

Structures

A *structure* is an ordered collection of variables.

When a subroutine argument is a structure, the data type of the argument can differ from one product to another. In the case of the subroutine `s$control`, the data type of an argument can differ from one operation to another. Therefore, this manual does not include declarations for data in structures as part of the argument declarations. Instead, these declarations are shown in the manuals for particular products.

Most of the structures given as arguments to the communications subroutines contain declarations for two variables that are described below. They are:

- A version number for the structure.

- A variable named `switches`.

These variables are explained here, though they do not appear in the declarations shown in this manual. For instances of the version-number and `switches` variables, see the declarations of structures in the manuals describing particular products.

Version Numbers

Most of the subroutines described here have version numbers that the calling program must set to a specified value. The version number is given as the first variable in a structure, and it is used by the subroutine to determine the form of the structure.

The version-number variable is always declared as a two-byte integer. For the value that you must give for this variable in a particular situation, consult the manual that describes the product you are working with.

Switches

The variable named `switches` is used to pass information on certain conditions. For example: If you use the subroutine `s$initialize_device` with the BSC protocol, you must use `switches` to pass information on eight conditions that describe the way the device is to be initialized. In this case, the `switches` variable is a binary coding of eight logical variables and eight unused bits, with the bits laid out as:

32768	16384	8192	4096	2048	1024	512	256

128	64	32	16	8	4	2	1

Figure 3-1. An Example of the Binary Coding of a `switches` Variable

The eight least-significant bits are unused, and each of the eight most-significant bits sets one of the conditions. Specifically:

- If the “256” bit (`receive_only`) is 1, the BSC software will operate only as a receiver;
- If the “512” bit (`send_only`) is 1, the BSC software will operate only as a transmitter;
- If the “1024” bit (`transparency`) is 1, transparent records will be allowed; and so forth.

The subroutines `s$encode_flags` and `s$decode_flags` (described in the *Subroutines Manuals*) ease the construction and interpretation of `switches`. `s$encode_flags` converts a character string of length 16 to an integer with precision 15. It translates each zero character (`0`) in the string to a zero bit in the integer, and each nonzero character to a one bit. `s$decode_flags` performs the inverse conversion.

In VOS PL/I, `switches` is generally the name of a substructure containing several members of data type `bit(1)` or `bit(`*`N`*`)`. See the appendixes for information about how the other VOS languages declare `switches`.

s$initialize_device

Purpose

`s$initialize_device` initializes the I/O device connected to a specified port.

Use `s$initialize_device` for devices to be initialized as BSC channels, SDLC channels, ticker-protocol channels, or 2780, 3780, or HASP emulators. Otherwise, use `s$open`, which is documented in the *Subroutines Manuals*.

Usage

```
declare  port_id                     binary(15);
declare  device_type                 binary(15);
declare  time_out                    binary(31);
declare  1 initialize_device_info    ,
          2 version                  binary(15),
            . . .
declare  status_code                 binary(15);

declare  s$initialize_device entry ( binary(15),
                                     binary(15),
                                     binary(31),
                                     binary(15),
                                     binary(15));

          call s$initialize_device(  port_id,
                                     device_type,
                                     time_out,
                                     initialize_device_info.version
                                     status_code);
```

Arguments

▶ `port_id` (input)

The identifier of a port attached to a device. The port must be attached to a synchronous communications line.

▶ `device_type (input)`

A code specifying the particular type of protocol or terminal emulator to be run on the device. The allowed values are:

- `12` for 2780
- `13` for 3780
- `14` for HASP
- `15` for BSC and Ticker

- ▶ `time_out` (input)

  A time period, given in VOS time units (1/1024 second). When the port is in `wait` mode, `s$initialize_device` waits this amount of time for the device to become ready before returning to the caller. If the device fails to become ready in the time period, `s$initialize_device` returns the error status code `e$timeout` (`1081`). If `time_out` is `-1`, then the subroutine waits indefinitely. VOS disregards this argument if the port is in `no_wait` mode.

- ▶ `initialize_device_info (input-output)`

  A structure with a total size of 256 bytes that contains information about the device being initialized. The form of the structure depends upon the device type. For declarations of the variables in `initialize_device_info`, see the manual that describes the type of device you are working with.

- ▶ `status_code` (output)

  A returned status code. See Appendix E.

s$control

Purpose

The subroutine `s$control` sets the control parameters of the I/O device connected to the specified port and returns information about the current values of the control parameters.

The data type of `control_info`, which is the third argument to `s$control`, can vary with different calls to the subroutine. The declarations below show this argument declared in the form you will use when the value of `control_info` is a structure. However, in many cases the data type of the argument is a variable. The notation *`data_type`* stands for one of several predefined data types.

Usage

```
declare  port_id           binary(15);
declare  opcode            binary(15);
declare  1 control_info    ,
          2 first_field    @data_type@,
            . . .
declare  status_code       binary(15);

declare  s$control entry(  binary(15),
                           binary(15),
                           @data_type@,
                           binary(15));

          call s$control(  port_id,
                           opcode,
                           control_info.first_field,
                           status_code);
```

Arguments

▶ `port_id` (input)
The identifier of a port attached to a device. The port must be attached to a synchronous communications line.

▶ `opcode (input)`
The operation performed on the I/O device. The values for `opcode` are grouped into categories according to device type or synchronous protocol. See Table 3-2.

▶ `control_info` (input-output)
The data type of this argument is a structure or variable. The form depends upon the device type and the value given for `opcode`. For the specific values given for this argument, refer to the manual that describes the type of device that you are working with.

In a VOS PL/I program that calls an external service subroutine, you must declare the data type of the subroutine's parameters in a `declare` statement with the `entry` attribute. When you call `s$control` more than once with different argument types in different calls, you are faced with the problem of how to declare the data type of the `control_info` parameter in the subroutine declaration.

One solution is to arrange your program so that it is divided into subroutines in which you redeclare the service subroutines. In this way, `s$control` is called with only one argument type in the scope of its declaration.

You must declare this argument so that it is allocated on an even byte boundary. Refer to the appropriate VOS language manual for the storage-allocation rules.

▶ `status_code (output)`
A returned status code. See Appendix E.

The Categories of Opcodes

Table 3-1 shows the categories into which the opcodes are divided and lists where you will find explanations of the specific control operations for each category of opcode.

Table 3-1. Categories of Opcodes

Opcodes	Device Type or Protocol	Opcode Documentation
1-100	Any device	This manual
101-200	Magnetic tape drive	*VOS Tape Processing User's Guide and Programmer's Reference (R052)*
201-300	Asynchronous terminal	*VOS Communications Software: Asynchronous Communications (R025)*
301-400	BSC line	*VOS Communications Software: Binary Synchronous Communications (R027)*
401-500	2780/3780/HASP emulator	*VOS Communications Software: Binary Synchronous Communications (R027)*
501-600	Link access protocol (X.25)	Reserved for system use
601-700	Internal interfaces	Reserved for system use
701-800	Synchronous line	Various manuals
801-900	3270 emulator	*VOS Communications Software: 3270 Support and 3270 Emulation (R026)*
901-1000	3270 terminal support	*VOS Communications Software: 3270 Support and 3270 Emulation (R026)*
1101-1400	User-defined devices	*Guide to the Programmable StrataBUS Interface (R024)*
1401-1500	SDLC line	*VOS Communications Software: SDLC (R043)*

The Global Control Operations

The *global control operations* are those that are defined for any device (`opcode` numbers 1-100). Table 3-2 shows the `opcode` values for those global control operations that are presently defined. For each of these control operations, the `control_info` argument must be declared as a two-byte integer.

Table 3-2. The Global Control Operations

Opcode	Control Operation
1	`get_line_length_opcode`
2	`runout_opcode`
3	`abort_opcode`

`get_line_length_opcode`

Returns the device's line length in the output argument `control_info`.

`runout_opcode`

Flushes the device's I/O buffers, or writes out buffers. The `control_info` argument must be set to `0`.

`abort_opcode`

Aborts the current I/O operation. The `control_info` argument must be set to `0`.

s$read_device

Purpose

s$read_device reads records from the I/O device connected to a specified port. The operation of the subroutine and the declaration of the argument read_device_info depend upon the type of device.

Before using this subroutine, you must use s$initialize_device to initialize the device.

If the port specified in the subroutine is in wait mode, then s$read_device returns only after it has read an entire record into the buffer.

In both wait and no_wait modes, if the buffer supplied by the user program is not large enough to hold the entire received record, then VOS reads in only that part of the record that will fit into the buffer. The part of the record that does not fit is discarded, and the status code e$data_truncated (1363) is returned. Therefore, you must be sure that the buffers supplied by your program are large enough to hold the largest anticipated record.

The number of characters read into the buffer is returned in the record_length argument.

Usage

```
declare  port_id              binary(15);
declare  buffer_length        binary(15);
declare  1 read_device_info   ,
          2 read_version      binary(15),
           . . .
declare  record_length        binary(15);
declare  buffer               char(N);
declare  status_code          binary(15);

declare  s$read_device entry( binary(15),
                              binary(15),
                              binary(15),
                              binary(15),
                              char(N),
                              binary(15));

          call s$read_device( port_id,
                              buffer_length,
                              read_device_info.read_version,
                              record_length,
                              buffer,
                              status_code);
```

Arguments

- `port_id` (input)

  The identifier of a port attached to a device. The port must be attached to a synchronous communications line.

- `buffer_length (input)`

  The number of available character positions in the string `buffer`. This value must be large enough to hold the largest anticipated record. If the record to be read into `buffer` is longer than the value given for `buffer_length`, VOS reads in only the number of characters that will fit, discards the rest, and returns the status code `e$data_truncated` (`1363`).

- `read_device_info` (input-output)

  A structure with a total size of 16 bytes that contains information about the record returned in `buffer`. The form of the structure depends upon the device type. For declarations of the variables in `read_device_info`, see the manual that describes the type of device you are working with.

- `record_length (output)`

  The length of the record returned in `buffer`.

- `buffer` (output)

  The data in the record that the subroutine reads. (See the explanation of `buffer_length` above.)

- `status_code (output)`

  A returned status code. See Appendix E.

s$read_device_event

Purpose

`s$read_device_event` returns the identifier and current count of the event that is notified when the I/O device connected to the given port finishes an I/O operation.

If a subroutine that starts an I/O operation returns to the caller before the operation is completed, and if the port specified by the port ID in the subroutine is in `no_wait_mode`, the subroutine will return the status code `e$caller_must_wait` (`1277`). Then, when the operation completes, an event is notified. The purpose of `s$read_device_event` is to provide information for a given port about the event that is notified.

In the simplest case, the caller can wait on the event. However, since it is possible to wait on more than one event at the same time, the caller can start several I/O operations and still be prepared to respond to other I/O requests before any of them is completed.

See the descriptions of `s$set_no_wait_mode` and `s$set_wait_mode` in the *Subroutines Manuals*.

Usage

```
declare  port_id                    binary(15);
declare  event_id                   binary(31);
declare  event_count                binary(31);
declare  status_code                binary(15);

declare  s$read_device_event entry( binary(15),
                                    binary(31),
                                    binary(31),
                                    binary(15));

          call s$read_device_event( port_id,
                                    event_id,
                                    event_count,
                                    status_code);
```

Arguments

- `port_id` (input)

  The identifier of a port attached to an I/O device. The port must be attached to a synchronous communications line.

- `event_id (output)`

  The identifier of the event associated with the I/O device.

- ▶ `event_count` (output)

  The current count of the event attached to the I/O device.

- ▶ `status_code (output)`

  A returned status code. See Appendix E.

s$write_device

Purpose

s$write_device writes records to the I/O device connected to a specified port. Before using this subroutine, you must use s$initialize_device to initialize the device.

If the port specified in the subroutine is in wait mode, then s$write_device returns only after it has written the entire record.

If the port is in no_wait mode and if s$write_device cannot write the entire record in the buffer, then the subroutine returns the status code e$caller_must_wait (1277). In order to write the rest of the record, you must call s$write_device again, with the buffer containing only those characters not previously transmitted. (The number of characters transmitted by the previous call is returned in the buffer_length argument.)

Usage

```
declare  port_id               binary(15);
declare  buffer_length         binary(15);
declare  1 write_device_info   ,
          2 write_version      binary(15),
            . . .
declare  buffer                char(N);
declare  status_code           binary(15);

declare  s$write_device entry( binary(15),
                               binary(15),
                               binary(15),
                               char(N),
                               binary(15));

          call s$write_device( port_id,
                               buffer_length,
                               write_device_info.write_version,
                               buffer,
                               status_code);
```

Arguments

▶ port_id (input)

The identifier of a port attached to a device. The port must be attached to a synchronous communications line.

▶ buffer_length (input-output)

The length of the record in buffer.

Upon return, `buffer_length` is the length of the record written. The output value differs from the input value only when the port is in `no_wait` mode and the subroutine does not write the entire record.

- ▶ `write_device_info` (input)
  A structure with a total size of 16 bytes that contains information about the record to be written. The form of the structure depends upon the device type. For declarations of the variables in `write_device_info`, see the manual that describes the type of device you are working with.

- ▶ `buffer (input)`
  The data in the record written.

- ▶ `status_code` (output)
  A returned status code. See Appendix E.

s$clear_device

Purpose

`s$clear_device` clears (resets) the device connected to a given I/O port and closes the port. The device must have been initialized by `s$initialize_device`.

Clearing a device is analogous to closing a file.

Usage

```
declare  port_id               binary(15);
declare  status_code           binary(15);

declare  s$clear_device entry( binary(15),
                               binary(15));

          call s$clear_device( port_id,
                               status_code);
```

Arguments

- `port_id` (input)

  The identifier of a port attached to a device. The port must be attached to a synchronous communications line.

- `status_code (output)`

  A returned status code. See Appendix E.

Appendix A: BASIC Usage

Structures in BASIC Programs

The term *structure*, used throughout the body of this manual, refers in BASIC to a `map` statement.

Switches in BASIC Programs

The variable named `switches` is used to pass information on various conditions. In BASIC, the logical (Boolean) variables represented by `switches` must be passed to the subroutines encoded as integers, since BASIC does not have a bit string data type. Generally, the least significant bits in the integer correspond to the logical variables and the most significant bits are unused (although this order is sometimes reversed, as in the example given in Figure 3-1). A bit in the integer is set to `1` when the value of the corresponding logical variable is `true`.

Subroutine Declarations in BASIC Programs

The remainder of this appendix lists the subroutine declarations in BASIC programs.

$initialize_device

Usage

```
dimension   port_id%=15
dimension   device_type%=15
dimension   time_out%=31
map         (initialize_device_info_map) &
            initialize_version%=15, &
  . . .
dimension   status_code%=15

subprogram  s$initialize_device(        port_id%=15, &
                                        device_type%=15, &
                                        time_out%=31, &
                                        initialize_version%=15, &
                                        status_code%=15) external

            call s$initialize_device(   port_id%, &
                                        device_type%, &
                                        time_out%, &
                                        initialize_version%, &
                                        status_code%)
```

$control

The notation *data_type_indicator* stands for one of the symbols used in VOS BASIC to indicate the data type of an argument.

Usage

```
dimension   port_id%=15
dimension   opcode%=15
map         (control_info_map) &
            control_infodata_type_indicator &
            . . .
dimension   status_code%=15

subprogram  s$control(                  port_id%=15, &
                                        opcode%=15, &
                                        control_infodata_type_indicator, &
                                        status_code%=15) external

            call s$control(             port_id%, &
                                        opcode%, &
                                        control_infodata_type_indicator, &
                                        status_code%)
```

$read_device

Usage

```
dimension   port_id%=15
dimension   buffer_length%=15
map         (read_device_info_map) &
            read_version%=15, &
            . . .
dimension   record_length%=15
dimension   buffer$=*
dimension   status_code%=15

subprogram  s$read_device(              port_id%=15, &
                                        buffer_length%=15, &
                                        read_version%=15, &
                                        record_length%=15, &
                                        buffer$=*, &
                                        status_code%=15) external

          call s$read_device(           port_id%, &
                                        buffer_length%, &
                                        read_version%, &
                                        record_length%, &
                                        buffer$, &
                                        status_code%)
```

$read_device_event

Usage

```
dimension   port_id%=15
dimension   event_id%=31
dimension   event_count%=31
dimension   status_code%=15

subprogram  s$read_device_event(        port_id%=15, &
                                        event_id%=31, &
                                        event_count%=31, &
                                        status_code%=15) external

            call s$read_device_event(   port_id%, &
                                        event_id%, &
                                        event_count%, &
                                        status_code%)
```

$write_device

Usage

```
dimension   port_id%=15
dimension   buffer_length%=15
map         (write_device_info_map) &
            write_version%=15, &
            . . .
dimension   buffer$=*
dimension   status_code%=15

subprogram  s$write_device(             port_id%=15, &
                                        buffer_length%=15, &
                                        write_version%=15, &
                                        buffer$=*, &
                                        status_code%=15) external

            call s$write_device(        port_id%, &
                                        buffer_length%, &
                                        write_version%, &
                                        buffer$, &
                                        status_code%)
```

$clear_device

Usage

```
dimension   port_id%=15
dimension   status_code%=15

subprogram  s$clear_device(             port_id%=15, &
                                        status_code%=15) external

            call s$clear_device(        port_id%, &
                                        status_code%)
```

Appendix B:
COBOL Usage

Structures in COBOL Programs

The term *structure*, used throughout the body of this manual, refers in COBOL to a group item.

Switches in COBOL Programs

The variable named `switches` is used to pass information on various conditions. In COBOL, the logical (Boolean) variables represented by `switches` must be passed to the subroutines encoded as integers, since COBOL does not have a bit string data type. Generally, the least significant bits in the integer correspond to the logical variables and the most significant bits are unused (although this order is sometimes reversed, as in the example given in Figure 3-1). A bit in the integer is set to `1` when the value of the corresponding logical variable is `true`.

Subroutine Declarations in COBOL Programs

The remainder of this appendix lists the subroutine declarations in COBOL programs.

$initialize_device

Usage

```
01  port_id                           computational-4.
01  device_type                       computational-4.
01  time_out                          computational-5.
01  initialize_device_info.
    . . .
01  status_code                       computational-4.

    call “s$initialize_device” using  port_id,
                                      device_type,
                                      time_out,
                                      initialize_device_info,
                                      status_code.
```

$control

Usage

```
        01  port_id                 computational-4.
        01  opcode                  computational-4.
        01  control_info.
            . . .
        01  status_code             computational-4.

            call “s$control” using  port_id,
                                    opcode,
                                    control_info,
                                    status_code.
```

$read_device

Usage

```
01  port_id                     computational-4.
01  buffer_length               computational-4.
01  read_device_info.
    . . .
01  record_length               computational-4.
01  buffer                      picture x(N) display.
01  status_code                 computational-4.

    call “s$read_device” using  port_id,
                                buffer_length,
                                record_length,
                                buffer,
                                status_code.
```

$read_device_event

Usage

```
    01  port_id                           computational-4.
    01  event_id                          computational-5.
    01  event_count                       computational-5.
    01  status_code                       computational-4.

        call “s$read_device_event” using  port_id,
                                          event_id,
                                          event_count,
                                          status_code.
```

$write_device

Usage

```
01  port_id                      computational-4.
01  buffer_length                computational-4.
01  write_device_info.
    . . .
01  buffer                       picture x(N) display.
01  status_code                  computational-4.

    call “s$write_device” using  port_id,
                                 buffer_length,
                                 write_device_info,
                                 buffer,
                                 status_code.
```

$clear_device

Usage

```
01  port_id                       computational-4.
01  status_code                   computational-4.

    call “s$clear_device” using  port_id,
                                  status_code.
```

Appendix C:
FORTRAN Usage

Structures in FORTRAN Programs

The term *structure*, used throughout the body of this manual, refers in FORTRAN to a common block.

Switches in FORTRAN Programs

The variable named `switches` is used to pass information on various conditions. In FORTRAN, the logical (Boolean) variables represented by `switches` must be passed to the subroutines encoded as integers, since FORTRAN does not have a bit string data type. Generally, the least significant bits in the integer correspond to the logical variables and the most significant bits are unused (although this order is sometimes reversed, as in the example given in Figure 3-1). A bit in the integer is set to `1` when the value of the corresponding logical variable is `true`.

Subroutine Declarations in FORTRAN Programs

The remainder of this appendix lists the subroutine declarations in FORTRAN programs.

$initialize_device

Usage

```
 external  s$initialize_device

 integer*2 port_id
 integer*2 device_type
 integer*4 time_out

 common    /initialize_device_info_block/
1          initialize_version,
2          . . .

 integer*2 initialize_version
           . . .
 integer*2 status_code

           call s$initialize_device(   port_id,
1                                      device_type,
2                                      time_out,
3                                      initialize_version,
4                                      status_code)
```

$control

The notation *data_type* stands for one of several predefined data types.

Usage

```
 external     s$control

 integer*2    port_id
 integer*2    opcode

 common       /control_info_block/
1 first_field,
2 . . .

 data_type    first_field
              . . .
 integer*2    status_code

              call s$control(  port_id,
1                              opcode,
2                              first_field,
3                              status_code)
```

$read_device

Usage

```
 external     s$read_device

 integer*2    port_id
 integer*2    buffer_length

 common       /read_device_info_block/
1             read_version,
2             . . .

 integer*2    read_version
              . . .
 integer*2    record_length
 character*N  buffer
 integer*2    status_code

              call s$read_device(  port_id,
1                                  buffer_length,
2                                  read_version,
3                                  record_length,
4                                  buffer,
5                                  status_code)
```

$read_device_event

Usage

```
 external  s$read_device_event

 integer*2 port_id
 integer*4 event_id
 integer*4 event_count
 integer*2 status_code

           call s$read_device_event(  port_id,
1                                     event_id,
2                                     event_count,
3                                     status_code)
```

$write_device

Usage

```
 external     s$write_device

 integer*2    port_id
 integer*2    buffer_length

 common       /write_device_info_block/
1             write_version,
2             . . .

 integer*2    write_version
                   . . .
 character*N  buffer
 integer*2    status_code

              call s$write_device(  port_id,
1                                   buffer_length,
2                                   write_version,
3                                   buffer,
4                                   status_code)
```

$clear_device

Usage

```
 external  s$clear_device

 integer*2 port_id
 integer*2 status_code

           call s$clear_device(  port_id,
1                                status_code)
```

Appendix D:
Pascal Usage

Structures in Pascal Programs

The term *structure*, used throughout the body of this manual, refers in Pascal to a record variable.

Switches in Pascal Programs

The variable named `switches` is used to pass information on various conditions. In Pascal, `switches` consists of variables of data type `boolean` or `array[1..`*`N`*`] of boolean`.

The `short` Data Type

The Pascal data type `short` is defined as a two-byte integer type. To use this data type in a program, you must include the following type definition in the program:

```
type short = -32768..32767;
```

Subroutine Declarations in Pascal Programs

The remainder of this appendix lists the subroutine declarations in Pascal programs.

$initialize_device

Usage

```
var        port_id:                 short;
           device_type:             short;
           time_out:                integer;
           initialize_device_info:  record
                                     initialize_version: short;
                                     . . .
                                    end;
           status_code:             short;

procedure  s$initialize_device(     port_id: short;
                                    device_type: short;
                                    time_out: integer;
                                    var initialize_version: short;
                                    var status_code: short);
           external;

           s$initialize_device(     port_id,
                                    device_type,
                                    time_out,
                                    initialize_device_info.
                                      initialize_version,
                                    status_code);
```

$control

The notation *data_type_indicator* stands for one of the symbols used in VOS BASIC to indicate the data type of an argument.

Usage

```
var        port_id:       short;
           opcode:        short;
           control_info:  record
                           first_field: data_type;
                           . . .
                          end;
           status_code:   short;

procedure  s$control(     port_id: short;
                          opcode: short;
                          var first_field: data_type;
                          var status_code: short);
           external;

           s$control(     port_id,
                          opcode,
                          control_info.first_field,
                          status_code);
```

$read_device

Usage

```
type       buffer_type        = array[1..N] of char;

var        port_id:           short;
           buffer_length:     short;
           read_device_info:  record
                               read_version: short;
                               . . .
                              end;
           record_length:     short;
           buffer:            buffer_type;
           status_code:       short;

procedure  s$read_device(     port_id: short;
                              buffer_length: short;
                              var read_version: short;
                              var record_length: short;
                              var buffer: buffer_type;
                              var status_code: short);
           external;

           s$read_device(     port_id,
                              buffer_length,
                              read_device_info.read_version,
                              record_length,
                              buffer,
                              status_code);
```

$read_device_event

Usage

```
var        port_id:              short;
           event_id:             integer;
           event_count:          integer;
           status_code:          short;

procedure  s$read_device_event(  port_id: short;
                                 var event_id: integer;
                                 var event_count: integer;
                                 var status_code: short);
           external;

           s$read_device_event(  port_id,
                                 event_id,
                                 event_count,
                                 status_code);
```

$write_device

Usage

```
type       buffer_type =       array [1..N] of char;

var        port_id:            short;
           buffer_length:      short;
           write_device_info:  record
                                write_version: short;
                                 . . .
                               end;
           buffer:             buffer_type;
           status_code:        short;

procedure  s$write_device(     port_id: short;
                               var buffer_length: short;
                               var write_version: short;
                               buffer: buffer_type;
                               var status_code: short);
           external;

           s$write_device(     port_id,
                               buffer_length,
                               write_device_info.write_version,
                               buffer,
                               status_code);
```

$clear_device

Usage

```
var        port_id:         short;
           status_code:     short;

procedure  s$clear_device(  port_id: short;
                            var status_code: short);
           external;

           s$clear_device(  port_id,
                            status_code);
```

Appendix E: Status Codes

Some of the status codes that can be returned by the VOS communications software are:

`1005` `$no_alloc`
Returned during initialization if VOS does not have enough room to allocate storage to run the channel. Try again later to initialize the device.

`1026` `$long_record`
Returned during BSC, 2780, 3780, or HASP operations. This condition is fatal to the record that was being written.

`1040` `$invalid_io_operation`
This condition is fatal to the continued operation of the channel. Contact the Stratus Customer Assistance Center.

`1081` `$timeout`
Returned during initialization if the device fails to become ready in the time specified in `s$initialize_device`. For BSC, returned for a control operation that does not complete within ten seconds.

`1083` `$wrong_version`
Returned if the version-number variable in the structure was set incorrectly. Supply the correct number and repeat the call.

`1229` `$short_record`
Returned during execution of an RJE protocol that requires 80-byte records in transparent mode. Returned during BSC operations if the record is less than one byte long.

`1277` `$caller_must_wait`
Returned in `no_wait` mode to inform the user that VOS will notify when the operation can be completed.

`1363` `$data_truncated`
Returned during read operations if the user's buffer is not large enough to hold the received record. The portion of the record that does not fit into the buffer is discarded.

`1365` `$line_hangup`
Returned during any operation when line hang-up is noted. No data is returned. You must finish reading any pending input, then clear the device and initialize it again.

`1364      $line_status`

Returned during 3270 support operations, to indicate that you must call `s$control` with `r$3270_get_ss_opcode`.

`1429      $inconsistent_device_state`

Returned during operations on a BSC, 2780, 3780, or HASP line if:

- The state of the transmission line was not set correctly by a previous control operation (for example, if the station tried a write operation without requesting permission to send).

- The other station sent a line bid during a read operation.

`1439      $invalid_data_in_record`

Returned during write operations on a BSC, 2780, 3780, or HASP line if VOS finds a data-link control character in data to be transmitted in non-transparent mode.

`2833      $block_check_error`

Returned during a read operation when the received record contains a block check error or a parity error.

`2950      $block_discarded`

Returned during ticker operations if the user program is not reading records as fast as they are received.