R015-04

OpenVOS PL/I
Transaction Processing Facility
Reference Manual

Stratus Technologies

Notice

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.

Manual Name: *OpenVOS PL/I Transaction Processing Facility Reference Manual*

Part Number: R015
Revision Number: 04
OpenVOS Release Number: 17.0.0
Publication Date: July 2008

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

Preface

The *OpenVOS PL/I Transaction Processing Facility Reference Manual (R015)* documents the commands, monitor requests, and subroutines needed to build and operate a transaction-processing application.

This manual is intended for programmers and administrators of transaction-processing applications.

Before using the *OpenVOS PL/I Transaction Processing Facility Reference Manual* (R015), you should be familiar with the *OpenVOS PL/I Subroutines Manual* (R005) and the *OpenVOS PL/I Language Manual* (R009).

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. Note, however, that change bars are not used in new chapters or appendixes.

This revision incorporates the following changes:

- The addition of the `s$get_tp_abort_code`, `s$task_setup_wait2`, and `s$task_wait_event2` subroutines

- Changes to the `s$msg_rewrite` and `s$set_transaction_file` subroutines

Related Manuals

Refer to the following Stratus manuals for related documentation.

- *VOS Transaction Processing Facility Guide* (R215)
- *OpenVOS Commands Reference Manual* (R098)
- *OpenVOS PL/I Language Manual* (R009)
- *OpenVOS PL/I Subroutines Manual* (R005)
- *OpenVOS System Administration: Administering and Customizing a System* (R281)
- *OpenVOS System Administration: Registration and Security* (R283)
- *OpenVOS System Administration: Disk and Tape Administration* (R284)
- *VOS System Administration: Administering the Spooler Facility* (R286)
- *OpenVOS System Administration: Configuring a System* (R287)

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  *.*
  ```

Format for Commands and Requests

Stratus manuals use the following format conventions for documenting commands and requests. (A *request* is typically a command used within a subsystem, such as `analyze_system`.) Note that the command and request descriptions do not necessarily include each of the following sections.

`name`
The name of the command or request is at the top of the first page of the description.

Privileged
This notation appears after the name of a command or request that can be issued only from a privileged process.

Purpose
Explains briefly what the command or request does.

Display Form
Shows the form that is displayed when you type the command or request name followed by `-form` or when you press the key that performs the `DISPLAY FORM` function. Each

field in the form represents a command or request argument. If an argument has a default value, that value is displayed in the form.

The following table explains the notation used in display forms.

The Notation Used in Display Forms

Notation	**Meaning**
██████	Required field with no default value.
█	The cursor, which indicates the current position on the screen. For example, the cursor may be positioned on the first character of a value, as in `all`.
`current_user` *`current_module`* *`current_system`* *`current_disk`*	The default value is the current user, module, system, or disk. The actual name is displayed in the display form of the command or request.

Command-Line Form

Shows the syntax of the command or request with its arguments. You can display an online version of the command-line form of a command or request by typing the command or request name followed by `-usage`.

The following table explains the notation used in command-line forms. In the table, the term *multiple values* refers to explicitly stated separate values, such as two or more object names. Specifying multiple values is **not** the same as specifying a star name. When you specify multiple values, you must separate each value with a space.

The Notation Used in Command-Line Forms

Notation	Meaning
argument_1	Required argument.
argument_1...	Required argument for which you can specify multiple values.
{ *element_1* *element_2* }	Set of arguments that are mutually exclusive; you must specify one of these arguments.
[*argument_1*]	Optional argument.
[*argument_1*]...	Optional argument for which you can specify multiple values.
[*argument_1* *argument_2*]	Set of optional arguments that are mutually exclusive; you can specify only one of these arguments.
Note: Dots, brackets, and braces are not literal characters; you should **not** type them. Any list or set of arguments can contain more than two elements. Brackets and braces are sometimes nested.	

Arguments

Describes the command or request arguments. The following table explains the notation used in argument descriptions.

The Notation Used in Argument Descriptions

Notation	Meaning
CYCLE	There are predefined values for this argument. In the display form, you display these values in sequence by pressing the key that performs the CYCLE function.
Required	You cannot issue the command or request without specifying a value for this argument. If an argument is required but has a default value, it is not labeled **Required** since you do not need to specify it in the command-line form. However, in the display form, a required field must have a value—either the displayed default value or a value that you specify.
(Privileged)	Only a privileged process can specify a value for this argument.

Explanation

Explains how to use the command or request and provides supplementary information.

Error Messages

Lists common error messages with a short explanation.

Examples

Illustrates uses of the command or request.

Related Information

Refers you to related information (in this manual or other manuals), including descriptions of commands, subroutines, and requests that you can use with or in place of this command or request.

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

The OpenVOS StrataDOC Web site is an online-documentation service provided by Stratus. It enables Stratus customers to view, search, download, print, and comment on OpenVOS technical manuals via a common Web browser. It also provides the latest updates and corrections available for the OpenVOS document set.

You can access the OpenVOS StrataDOC Web site, at no charge, at `http://stratadoc.stratus.com`. A copy of OpenVOS StrataDOC on supported media is included with this release. You can also order additional copies from Stratus.

This manual is available on the OpenVOS StrataDOC Web site.

For information about ordering OpenVOS StrataDOC on supported media, 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 `http://www.stratus.com/support/cac/index.htm` 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`. To use the `comment_on_manual` command, your system must be connected to the RSN. Alternatively, you can email comments on this manual to `comments@stratus.com`.

The `comment_on_manual` command is documented in the manual *OpenVOS System Administration: Administering and Customizing a System* (R281) and the *OpenVOS 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, `R015`, 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:
Transaction Processing Facility Commands

This chapter contains descriptions of the following transaction facility commands.

- display_lock_wait_time
- display_process_lock_wait_time
- display_tp_default_parameters
- display_tp_parameters
- list_messages
- set_lock_wait_time
- set_max_queue_depth
- set_process_lock_wait_time
- set_tp_default_parameters
- set_tp_parameters
- set_transaction_file
- tp_restore

display_lock_wait_time

Purpose

This command displays the default maximum time (in seconds) that a task or process running on a given module will wait to acquire an implicit lock during any I/O operation.

Display Form

```
--------------------------- display_lock_wait_time ---------------------------
-module: current_dir
```

Command Line Form

`display_lock_wait_time [-module module_name]`

Arguments

▶ `-module` *module_name*

Specifies a module or module star name. If you omit this option, OpenVOS displays the default wait time on the current module.

Explanation

The `display_lock_wait_time` command displays the default maximum time (in seconds) that a task or process will wait to acquire an implicit lock during an I/O operation. The default maximum wait time applies to all processes running on a given processing module or set of modules (unless the lock wait time for a process has been specifically changed from the default) and to all types of I/O. The default maximum wait time is set with the `set_lock_wait_time` command.

Related Information

`display_process_lock_wait_time`

Displays the maximum time (in seconds) that the current process (and any tasks that are part of that process) will wait to acquire an implicit lock during any I/O operation.

`set_lock_wait_time`

Sets the default maximum time (in seconds) that a task or process will wait to acquire an implicit lock during any I/O operation. The default wait time applies to all tasks and processes running on a given module or set of modules (unless the lock wait time for a task or process has been specifically changed from the default) and to all types of I/O operations. You must be privileged to use this command.

`set_process_lock_wait_time`

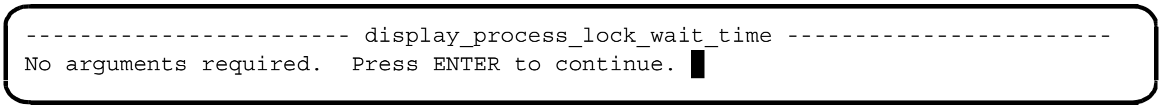
Sets the maximum time (in seconds) that a task or process will wait to acquire an implicit lock during any I/O operation.

display_process_lock_wait_time

Purpose

This command displays the maximum time (in seconds) that the current process (and any tasks that are part of that process) will wait to acquire an implicit lock during any I/O operation.

Display Form



```
----------------------- display_process_lock_wait_time -----------------------
No arguments required.  Press ENTER to continue. █
```

Command Line Form

```
display_process_lock_wait_time
```

Arguments

None.

Explanation

The `display_process_lock_wait_time` command displays the maximum lock wait time (in seconds) for the current process (and any tasks that are part of the current process).

Lock wait time is the amount of time a process or task will wait for an implicit lock on a file, record, or key. If the process or task has to wait longer, then it gives up and returns one of the following error codes depending on the type of lock sought:

```
e$file_in_use        (1084)
e$record_in_use      (2408)
e$key_in_use         (2918).
```

The default module lock wait time set by OpenVOS is 0 seconds. Issuing the command `set_process_lock_wait_time` or calling `s$set_lock_wait_time` allows you to reset the lock wait time for the current program or process. To change the lock wait time for the module, issue the command `set_lock_wait_time` or call the subroutine `s$set_default_lock_wait_time`. 10 seconds is a typical lock wait time. A task uses the program lock wait time if one has been set. Otherwise it looks for a process lock wait time, and if that has not been set it uses the module lock wait time.

The default maximum wait time applies to all processes running on a given module or set of modules (unless the lock wait time for a process has been specifically changed from the default).

Related Information

`display_lock_wait_time`
: Displays the default maximum time (in seconds) that a task or process running on a given module will wait to acquire an implicit lock during any I/O operation.

`set_lock_wait_time`
: Sets the default maximum time (in seconds) that a task or process will wait to acquire an implicit lock during any I/O operation. The default wait time applies to all tasks and processes running on a given module or set of modules (unless the lock wait time for a task or process has been specifically changed from the default) and to all types of I/O operations. You must be privileged to use this command.

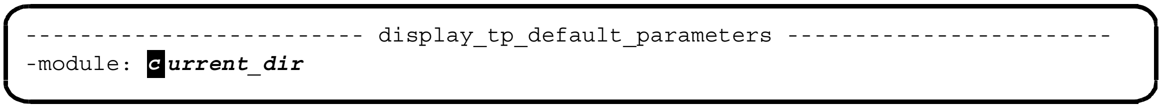
`set_process_lock_wait_time`
: Sets the maximum time (in seconds) that a task or process will wait to acquire an implicit lock during any I/O operation.

display_tp_default_parameters

Purpose

This command displays the default lock contention parameters and their values for transaction locking on a specified module.

Display Form



```
------------------------ display_tp_default_parameters -----------------------
-module: current_dir
```

Command Line Form

```
display_tp_default_parameters [-module module_name]
```

Arguments

▶ `-module` *`module_name`*

Specifies a module name or module star name for which you want the parameters displayed. If you omit this option, OpenVOS displays the default parameters for the current module.

Explanation

The `display_tp_default_parameters` command displays the current default values assigned by OpenVOS to all transactions started on the specified module. You can override these values for a specific process or task with the command `set_tp_parameters` or the subroutine `s$set_tp_parameters`. You can also change the module default values from their OpenVOS defaults with the command `set_tp_default_parameters` or the subroutine `s$set_tp_default_parameters`.

The following example shows output from the command `display_tp_default_parameters`:

```
The default transaction parameters on module_name are:
  priority:        0
  time value:      10 second(s)
  ignore priority  no
  ignore time:     no
  younger wins:    no
  allow deadlocks: no
```

The parameters above have the following meanings.

`priority`
: The default lock contention priority given a transaction by OpenVOS when you call `s$start_transaction`. The value can be 0 (lowest) to 9 (highest) inclusive. OpenVOS looks at this first when deciding which of two transactions should win a lock contention, unless `ignore_priority` has been set to `yes`.

`time_value`
: A number of seconds. If two transactions differ in the times that they started by less than `time_value`, then OpenVOS considers them to have started at the same time, and neither is considered younger or older.

`ignore_priority`
: If this switch is enabled (set to `yes`) for both transactions contending for a lock, then their priorities are ignored by OpenVOS in deciding which transaction wins.

`ignore_time`
: If this switch is enabled (set to `yes`) for both transactions contending for a lock, then the time that each transaction began is ignored by OpenVOS in deciding which transaction wins.

`younger_wins`
: If this switch is enabled (set to `yes`) for both transactions contending for a lock, then the one which began last wins.

`allow_deadlocks`
: If this switch is enabled (set to `yes`) for both of two transactions contending for a lock, then OpenVOS ignores any deadlock occurring between them.

If a deadlock occurs, OpenVOS does not abort either transaction; instead, it returns the error `e$record_in_use` (`2408`). When this happens, you must abort the transaction. If you do not do so, the deadlock could continue indefinitely, until one of the transactions is aborted. For this reason, the use of this option is not recommended.

> If `allow_deadlocks` is `no` (the default) for either transaction involved in a deadlock, OpenVOS breaks the deadlock by choosing a winner based on the value of an internal transaction identifier. In most cases the older transaction is the winner. However, if both transactions have `younger_wins` set to `true`, then the younger transaction is the winner. In either case, if one of the transactions has lost a deadlock immediately before, it wins this time.

Related Information

`display_tp_parameters`
: Displays the lock contention parameters and their values for transaction locking for the current process.

`set_tp_default_parameters`
: Sets the default lock contention parameters for transaction locking on a specified module.

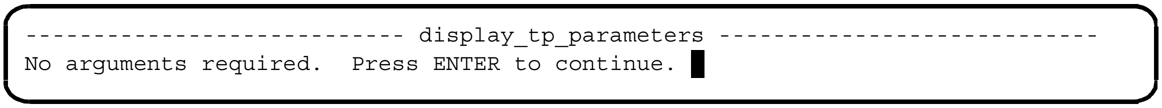
`set_tp_parameters`
Sets the lock contention parameters for transaction locking for the current process.

display_tp_parameters

Purpose

This command displays the lock contention parameters and their values for transaction locking for the current process.

Display Form



```
--------------------------- display_tp_parameters ---------------------------
No arguments required.  Press ENTER to continue. █
```

Command Line Form

```
display_tp_parameters
```

Arguments

None.

Explanation

The `display_tp_parameters` command displays the current values assigned by OpenVOS to all transactions started by the current process (see also the `display_tp_parameters` **request**, described in Chapter 2 that displays the parameters for the current program). You can change these values for a specific process with the command `set_tp_parameters` or the subroutine `s$set_tp_parameters`.

The following example shows output from the command `display_tp_parameters`:

```
The transaction parameters are:
  priority:        0
  time value:      10 second(s)
  ignore priority  no
  ignore time:     no
  younger wins:    no
  allow deadlocks: no
```

The parameters above have the following meanings.

`priority`
The default lock contention priority given a transaction by OpenVOS when you call `s$start_transaction`. The value can be 0 (lowest) to 9 (highest) inclusive. OpenVOS looks at this first when deciding which of two transactions should win a lock contention, unless `ignore_priority` has been set to `yes`.

`time_value`

A number of seconds. If two transactions differ in the times that they started by less than `time_value`, then OpenVOS considers them to have started at the same time, and neither is considered younger or older.

`ignore_priority`

If this switch is enabled (set to `yes`) for both transactions contending for a lock, then their priorities are ignored by OpenVOS in deciding which transaction wins.

`ignore_time`

If this switch is enabled (set to `yes`) for both transactions contending for a lock, then the time that each transaction began is ignored by OpenVOS in deciding which transaction wins.

`younger_wins`

If this switch is enabled (set to `yes`) for both transactions contending for a lock, then the one which began last wins.

`allow_deadlocks`

If this switch is enabled (set to `yes`) for both of two transactions contending for a lock, then OpenVOS ignores any deadlock occurring between them.

If a deadlock occurs, OpenVOS does not abort either transaction; instead, it returns the error `e$record_in_use` (`2408`). When this happens, you must abort the transaction. If you do not do so, the deadlock could continue indefinitely, until one of the transactions is aborted. For this reason, the use of this option is not recommended.

If `allow_deadlocks` is `no` (the default) for either transaction involved in a deadlock, OpenVOS breaks the deadlock by choosing a winner based on the value of an internal transaction identifier. In most cases the older transaction is the winner. However, if both transactions have `younger_wins` set to `true`, then the younger transaction is the winner. In either case, if one of the transactions has lost a deadlock immediately before, it wins this time.

Related Information

`display_tp_default_parameters`

Displays the default lock contention parameters and their values for transaction locking on a specified module.

`set_tp_default_parameters`

Sets the default lock contention parameters for transaction locking on a specified module.

`set_tp_parameters`

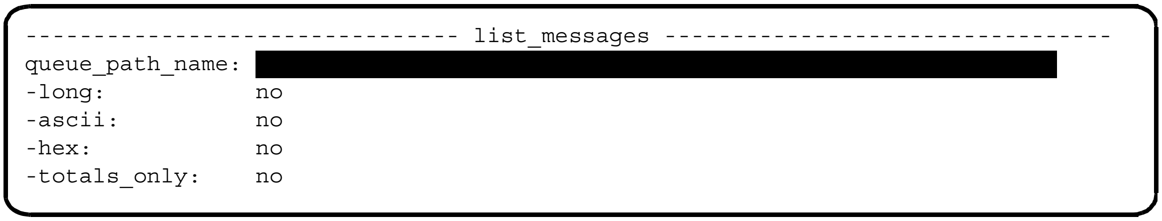
Sets the lock contention parameters for transaction locking for the current process.

list_messages

Purpose

This command lists the messages currently in a server queue or message queue.

Display Form



```
------------------------------ list_messages --------------------------------
queue_path_name:
-long:           no
-ascii:          no
-hex:            no
-totals_only:    no
```

Command Line Form

```
list_messages queue_path_name
         [-long]
         [-ascii]
         [-hex]
         [-totals_only]
```

Arguments

- ▶ *`queue_path_name`* **Required**

  The path name of a server queue or message queue. The messages in the queue are listed.

- ▶ `-long` CYCLE

  Displays the full message headers.

- ▶ `-ascii` CYCLE

  Displays the text of the messages in ASCII characters.

- ▶ `-hex` CYCLE

  Displays the text of the messages in hexadecimal characters.

- ▶ `-totals_only` CYCLE

  Displays only the number of the messages in the queue.

Explanation

The `list_messages` command displays information about the messages in a given queue. To get information about messages in a queue, you must have execute, read, or write access to it. If you have execute or read access to the queue, you can get information about your messages only. If you have write access, you can get information about all of the messages.

The information displayed about each message is the information contained in the message header:

- the message's identifier
- the time of day the message was sent
- the message's priority
- the message's subject
- the requester's process identifier
- the requester's user name
- whether the queue is a server queue
- whether the message is busy (being serviced)
- whether the message has been busy
- whether the message has been aborted.

If the queue is a server queue, `list_messages` lists only those messages that have not yet been replied to by the server. Messages that have been replied to but have not yet been removed from the queue by `s$msg_receive_reply` are not listed.

If you supply the `-totals_only` argument, `list_messages` displays only the number of the messages in the queue.

Related Information

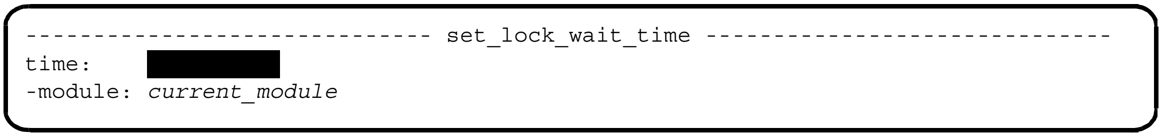
None.

`set_lock_wait_time` *Privileged*

Purpose

This command sets the default maximum time (in seconds) that a task or process will wait to acquire an implicit lock during any I/O operation. The default wait time applies to all tasks and processes running on a given module or set of modules (unless the lock wait time for a task or process has been specifically changed from the default) and to all types of I/O operations. You must be privileged to use this command.

Display Form



```
----------------------------- set_lock_wait_time -----------------------------
time:
-module: current_module
```

Command Line Form

```
set_lock_wait_time time
         [-module module_name]
```

Arguments

- ▶ `time` **Required**

  A number of seconds. The operating system sets the default maximum wait time to this value. `time` must be between 0 and 1000 inclusive.

- ▶ `-module` *`module_name`*

  Specifies a module or module star name. If you omit this option, OpenVOS sets the lock wait time on the current module.

Explanation

The `set_lock_wait_time` command sets the default maximum time that a task or process will wait to acquire a lock during an I/O operation. The default maximum wait time applies to all processes running on the specified module or set of modules (unless the lock wait time for a process has been specifically changed from the default).

The default lock wait time set by OpenVOS is 0 seconds.

Related Information

`display_lock_wait_time`

Displays the default maximum time (in seconds) that a task or process running on a given module will wait to acquire an implicit lock during any I/O operation.

`display_process_lock_wait_time`
Displays the maximum time (in seconds) that the current process (and any tasks that are part of that process) will wait to acquire an implicit lock during any I/O operation.

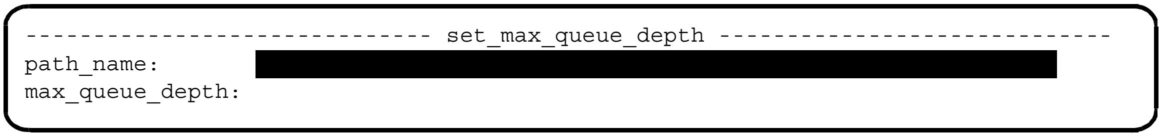
`set_process_lock_wait_time`
Sets the maximum time (in seconds) that a task or process will wait to acquire an implicit lock during any I/O operation.

set_max_queue_depth

Purpose

This command sets the maximum queue depth (or maximum number of messages) of a server or one-way server queue.

Display Form



```
---------------------------- set_max_queue_depth ----------------------------
path_name:
max_queue_depth:
```

Command Line Form

```
set_max_queue_depth path_name
          [max_queue_depth]
```

Arguments

- `path_name` **Required**

  The name of the server queue or one-way server queue. Only server queue and one-way server queue files are valid.

- *`max_queue_depth`*

  The maximum queue depth for the server queue or one-way server queue. Valid values range from 1 to 32767.

Explanation

The default `max_queue_depth` for any server queue or one-way server queue is 256.

The maximum queue depth is reached when the total number of messages equals `max_queue_depth`. At this point, the distribution of messages in the queue determines whether a message may be added.

A priority level is full if the number of messages at that priority is greater than or equal to:

```
max (number_of_servers, 4)
```

Once the `max_queue_depth` has been reached, new messages may be added to the server queue only at priority levels higher than the highest filled priority. If priority level 19 (the highest level) is filled, no more messages can be added to the queue.

> **Note:** The only case in which the number of messages at any given priority is restricted is when the number of messages in the server queue is greater than or equal to

`max_queue_depth`. Thus, if the queue has fewer than `max_queue_depth` messages in it, the number of messages at a given priority level is unlimited.

The action taken for a message that cannot be added to the queue depends on whether the caller is in wait or no-wait mode, or if an I/O time limit is set:

- If the caller is in wait mode, the application waits until the message can be added to the queue.

- If the caller is in no-wait mode, the error `e$caller_must_wait` (`1277`) is returned to the application.

- If an I/O time limit is set, the error `e$timeout` (`1081`) is returned to the application, and all messages sent on the port with the I/O time limit are removed from the queue.

For more information about I/O time limits, see the description of `s$set_io_time_limit` in the OpenVOS Subroutines manuals.

You can get the current `max_queue_depth` for a server or one-way server queue using `display_file_status`. (See the *OpenVOS Commands Reference Manual* (R098).

Related Information

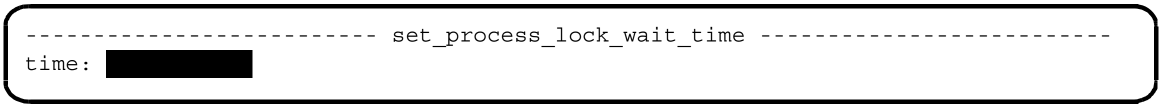
None.

set_process_lock_wait_time

Purpose

This command sets the maximum time (in seconds) that a task or process will wait to acquire an implicit lock during any I/O operation.

Display Form



```
------------------------- set_process_lock_wait_time -------------------------
time:
```

Command Line Form

`set_process_lock_wait_time` *`time`*

Arguments

▶ *`time`* **Required**

A number of seconds. The operating system sets the process lock wait time to this value. The `time` value must be between 0 and 1000 inclusive.

Explanation

The `set_process_lock_wait_time` command sets the lock wait time (in seconds) for the current process (and any tasks that are part of the current process).

Lock wait time is the amount of time a process or task will wait for an implicit lock on a file, record, or key. If the process or task has to wait longer, then it gives up and returns one of the following error codes depending on the type of lock sought:

```
e$file_in_use          (1084)
e$record_in_use        (2408)
e$key_in_use           (2918).
```

The default module lock wait time set by OpenVOS is 0 seconds. Calling `s$set_lock_wait_time` allows you to reset the lock wait time for the current program or process. To change the lock wait time for the module, issue the command `set_lock_wait_time` or call the subroutine `s$set_default_lock_wait_time`. 10 seconds is a typical lock wait time. A program uses the program lock wait time if one has been set. Otherwise, it looks for a process lock wait time, and if that has not been set it uses the module lock wait time.

The default maximum wait time applies to all processes running on a given module or set of modules (unless the lock wait time for a process has been specifically changed from the default).

Related Information

`display_lock_wait_time`
Displays the default maximum time (in seconds) that a task or process running on a given module will wait to acquire an implicit lock during any I/O operation.

`display_process_lock_wait_time`
Displays the maximum time (in seconds) that the current process (and any tasks that are part of that process) will wait to acquire an implicit lock during any I/O operation.

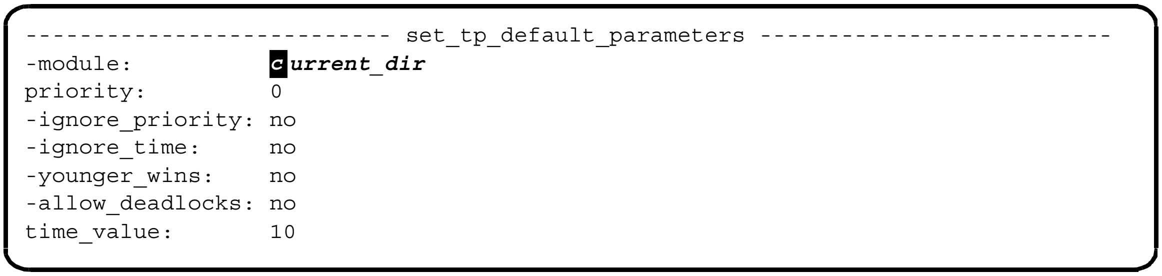
`set_lock_wait_time`
Sets the default maximum time (in seconds) that a task or process will wait to acquire an implicit lock during any I/O operation. The default wait time applies to all tasks and processes running on a given module or set of modules (unless the lock wait time for a task or process has been specifically changed from the default) and to all types of I/O operations. You must be privileged to use this command.

set_tp_default_parameters *Privileged*

Purpose

This command sets the default lock contention parameters for transaction locking on a specified module.

Display Form



```
-------------------------- set_tp_default_parameters -------------------------
-module:            current_dir
priority:           0
-ignore_priority:   no
-ignore_time:       no
-younger_wins:      no
-allow_deadlocks:   no
time_value:         10
```

Command Line Form

```
set_tp_default_parameters [priority]
          [-module module_name]
          [-ignore_priority]
          [-ignore_time]
          [-younger_wins]
          [-allow_deadlocks]
          [time_value]
```

Arguments

- `-module` *`module_name`*

  Specifies the name or star name of the module for which the parameters are to be set. The default value set by OpenVOS is the current module.

- *`priority`*

  The default lock contention priority to be given a transaction by `s$start_transaction`. The value can be 0 to 9 inclusive. The default value set by OpenVOS is 0.

- `-ignore_priority` CYCLE

  Tells OpenVOS whether to ignore the priorities of transactions competing for locks. The default value set by OpenVOS is `no`. In this case, transaction priorities are considered in determining which transaction wins.

- `-ignore_time` CYCLE

  Tells OpenVOS whether to ignore the start times of competing transactions. The default value set by OpenVOS is `no`. In this case, the starting times are considered in deciding which transaction wins.

- `-younger_wins` CYCLE

  Specifies whether the transaction with a later start time will win the contention. The default value set by OpenVOS is `no`. In this case, then the transaction which started sooner (the older transaction) will win.

- `-allow_deadlocks` CYCLE

  Tells OpenVOS whether to ignore any deadlock between two transactions of equal precedence. The default value set by OpenVOS is `no`.

  > **Caution:** If `-allow_deadlocks` is `yes` for two transactions of equal precedence involved in a conflict, OpenVOS does not abort either transaction; instead, OpenVOS returns the error `e$record_in_use` (`2408`). When this happens, you must abort the transaction. If you do not do so, it is possible for the deadlock to continue indefinitely, until one of the transactions is aborted. For this reason, the use of this option is not recommended.

  If `-allow_deadlocks` is `no` (the default), OpenVOS breaks the deadlock. See the Explanation below.

- *`time_value`*

  A number of seconds between 0 and 1024, inclusive. If two transactions differ in the time they were started by less than *`time_value`*, then OpenVOS considers them to have started at the same time. The default value set by OpenVOS is 10.

Explanation

The lock arbitration and resolution sequence proceeds as follows: (Assume that transaction A holds a lock and transaction B wants a lock, but they are conflicting locks — both A and B cannot have their locks at the same time.)

Arbitration:

1. If one transaction, and only one, has `priority` set to `-1` (`always_lose`) by `s$start_priority_transaction`, then that one loses; go to **Resolution**. Otherwise, go to **2**.

2. If both transactions have `-ignore_priorities` set to `yes` or their priorities are equal, then go to **3**. Otherwise the higher priority wins; go to **Resolution**.

3. If both transactions have `-ignore_time` set to `yes` or if their starting times differ by less than B's *`time_value`*, then go to **Deadlocks**. Otherwise go to **4**.

4. If both transactions have `-younger_wins` set to `yes`, then the younger wins; go on to **Resolution**. Otherwise, the older wins; go on to **Resolution**.

Deadlocks:

If neither transaction has won by the rules of arbitration and at least one has `-allow_deadlocks` set to `no`, OpenVOS breaks the deadlock by choosing a winner based on the value of an internal transaction identifier. In most cases the older transaction is the winner. However, if both transactions have `-younger_wins` set to `yes`, then the younger transaction is the winner. In either case, if one of the transactions has lost a deadlock immediately before, it wins this time.

Resolution:

If B (wants lock) wins then abort A (has lock); otherwise, B waits for `lock_wait_time` (set by `set_lock_wait_time`, `s$set_lock_wait_time`, `s$set_default_lock_wait_time`, or by OpenVOS) and then tries again. If the lock has not become available during that time, then abort B. The error code `e$tp_aborted (2931)` is returned for the transaction that loses.

Related Information

`display_tp_default_parameters`
Displays the default lock contention parameters and their values for transaction locking on a specified module.

`display_tp_parameters`
Displays the lock contention parameters and their values for transaction locking for the current process.

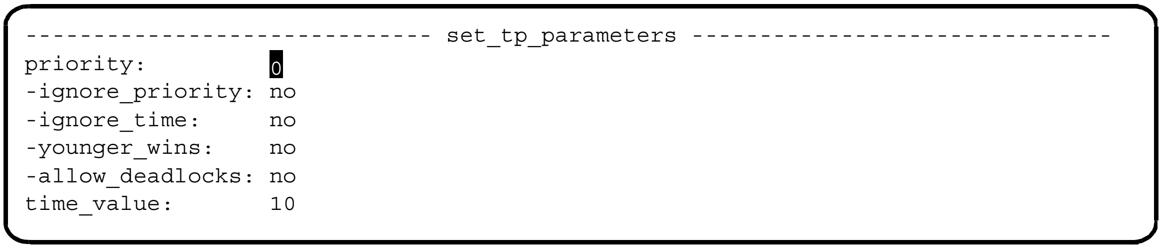
`set_tp_parameters`
Sets the lock contention parameters for transaction locking for the current process.

set_tp_parameters

Purpose

This command sets the lock contention parameters for transaction locking for the current process.

Display Form



```
----------------------------- set_tp_parameters -----------------------------
priority:          0
-ignore_priority:  no
-ignore_time:      no
-younger_wins:     no
-allow_deadlocks:  no
time_value:        10
```

Command Line Form

```
set_tp_parameters [priority]
         [time_value]
         [-ignore_priority]
         [-ignore_time]
         [-younger_wins]
         [-allow_deadlocks]
```

Arguments

▶ *`priority`*

The default lock contention priority to be given a transaction by `s$start_transaction`. The value can be from 0 to 9 inclusive. The default value set by OpenVOS (unless changed by `set_tp_default_parameters` or `s$set_tp_default_parameters`) is 0.

▶ `-ignore_priority` CYCLE

Tells OpenVOS whether to ignore the priorities of transactions competing for locks. The default value set by OpenVOS (unless changed by `set_tp_default_parameters` or `s$set_tp_default_parameters`) is `no`. In this case, transaction priorities are considered in determining which transaction wins.

▶ `-ignore_time` CYCLE

Tells OpenVOS whether to ignore the start times of competing transactions. The default value set by OpenVOS (unless changed by `set_tp_default_parameters` or `s$set_tp_default_parameters`) is `no`. In this case, the starting times are considered in deciding which transaction wins. TP

▶ `-younger_wins` CYCLE

Specifies whether the transaction with a later start time will win the contention. The default value set by OpenVOS (unless changed by `set_tp_default_parameters` or `s$set_tp_default_parameters`) is `no`. In this case, then the transaction which started sooner (the older transaction) will win.

▶ `-allow_deadlocks` CYCLE

Tells OpenVOS whether to ignore any deadlock between two transactions of equal precedence. The default value set by OpenVOS (unless changed by `set_tp_default_parameters` or `s$set_tp_default_parameters`) is `no`.

> **Caution:** If `-allow_deadlocks` is `yes` for two transactions of equal precedence involved in a conflict, OpenVOS does not abort either transaction; instead, OpenVOS returns the error `e$record_in_use` (`2408`). When this happens, you must abort the transaction. If you do not do so, it is possible for the deadlock to continue indefinitely until one of the transactions is aborted. For this reason, we do not recommend the use of this option.
>
> If `-allow_deadlocks` is `no` (the default) OpenVOS breaks the deadlock. See the Explanation below.

▶ *`time_value`*

A number of seconds between 0 and 1024, inclusive. If two transactions differ in the time they were started by less than *`time_value`*, then OpenVOS considers them to have started at the same time. The default value set by OpenVOS (unless changed by `set_tp_default_parameters` or `s$set_tp_default_parameters`) is 10.

Explanation

The lock arbitration and resolution sequence proceeds as follows: (Assume that transaction A holds a lock and transaction B wants a lock, but they are conflicting locks—both A and B cannot have their locks at the same time.)

Arbitration:

1. If one transaction, and only one, has `priority` set to `-1` (`always_lose`) by `s$start_priority_transaction`, then that one loses; go to **Resolution**. Otherwise, go to **2**.

2. If both transactions have `-ignore_priorities` set to `yes` or their priorities are equal, then go to **3**. Otherwise, the higher priority wins; go to **Resolution**.

3. If both transactions have `-ignore_time` set to `yes` or if their starting times differ by less than B's *`time_value`*, then go to **Deadlocks**. Otherwise, go to **4**.

4. If both transactions have `-younger_wins` set to `yes`, then the younger wins; go on to **Resolution**. Otherwise, the older wins; go on to **Resolution**.

Deadlocks:

If neither transaction has won by the rules of arbitration and at least one has `-allow_deadlocks` set to `no`, OpenVOS breaks the deadlock by choosing a winner based on the value of an internal transaction identifier. In most cases the older transaction is the winner. However, if both transactions have `-younger_wins` set to `yes`, then the younger transaction is the winner. In either case, if one of the transactions has lost a deadlock immediately before, it wins this time.

Resolution:

If B (wants lock) wins then abort A (has lock); otherwise, B waits for `lock_wait_time` (set by `set_lock_wait_time`, `s$set_lock_wait_time`, `s$set_default_lock_wait_time`, or by OpenVOS) and then tries again. If the lock has not become available during that time, then abort B. The error code `e$tp_aborted (2931)` is returned for the transaction that loses.

Related Information

`display_tp_default_parameters`
Displays the default lock contention parameters and their values for transaction locking on a specified module.

`display_tp_parameters`
Displays the lock contention parameters and their values for transaction locking for the current process.

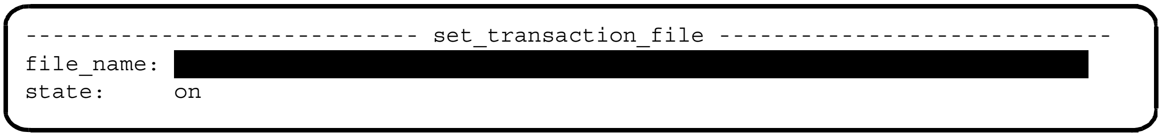
`set_tp_default_parameters`
Sets the default lock contention parameters for transaction locking on a specified module.

set_transaction_file *Privileged*

Purpose

This command converts one or more specified ordinary files into transaction files or converts one or more specified transaction files into ordinary files.

Display Form



```
---------------------------- set_transaction_file ----------------------------
file_name: 
state:     on
```

Command Line Form

```
set_transaction_file star_name
          [state]
```

Arguments

- ▶ *star_name* **Required**

  The name of a file or files.

- ▶ *state* CYCLE

  An option telling OpenVOS whether to convert the files indicated by *star_name* into transaction files or into ordinary files. The value `on` tells OpenVOS to make them transaction files; the value `off` means ordinary files. If you omit this option, OpenVOS converts the files into transaction files.

Explanation

A `set_transaction_file` command converts a file or set of files into transaction files or into ordinary files.

The operating system can back out the effects of an uncommitted transaction on a file only if the file is a transaction file.

You cannot rename a transaction file. You cannot create or delete an index to a transaction file. You cannot truncate a transaction file, including the truncation that normally occurs when a file is opened in output mode. If you open an existing transaction file in output mode, the file status is changed to a nontransaction file and the file is truncated. Output to the file is **not** transaction-protected.

You cannot convert a stream file to a transaction file. You must first convert it to a sequential file. The following sequence of commands shows one way to do this:

```
create_file new_file_name
copy_file stream_file new_file_name -truncate
```

You can then rename *new_file_name* to *stream_file*.

You must be privileged to issue the set_transaction_file command.

If the file is a log-protected file, the error code e$invalid_log_protected_op (7149) is returned. The file is a log-protected file. It indicates an invalid operation on a log protected object. It cannot also be a transaction-protected file. Log protection and transaction protection are mutually exclusive file attributes.

Related Information

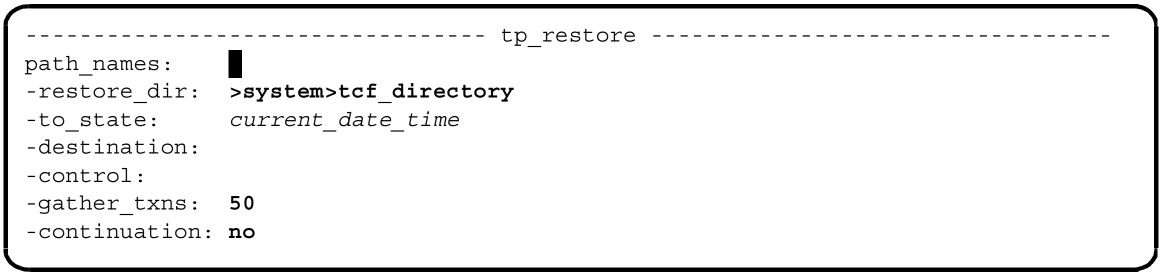
None.

tp_restore

Purpose

This command uses a transaction log to reconstruct a later state of a file from an earlier saved state.

Display Form



```
-------------------------------- tp_restore --------------------------------
path_names:     █
-restore_dir:   >system>tcf_directory
-to_state:      current_date_time
-destination:
-control:
-gather_txns:   50
-continuation:  no
```

Command Line Form

```
tp_restore path_names ...
           [-restore_dir path_name]
           [-to_state stop_time]
           [-destination path_name]
           [-control control_file]
           [-gather_txns number]
           [-continuation]
```

Arguments

▶ `path_names` **Required**

The path names of the files to be restored. For each file, use the path name it had when the transaction log files were last updated. The path names may be relative path names or star names. This argument is incompatible with `-control`.

▶ `-restore_dir` *`path_name`*

Specifies the directory containing the transaction log file(s) to be used by `tp_restore`. The default is `>system>tcf_directory`.

▶ `-destination` *`path_name`*

Specifies a directory in which the files to be restored are to be found. If this option is used, then files with the same object names as the files to be restored must be in the destination directory specified by *`path_name`*. This ensures that the files to be restored will have the same object names as the names in the log. In this case, all object names in the list of files to be restored must be unique.

If `-destination` is not given, then saved files with the expanded path names must exist in their original locations.

- `-to_state` *`stop_time`*

  Indicates the point to which `tp_restore` should restore the files.

  If *`stop_time`* is not specified, the restoration continues until the current time.

- `-control` *`control_file`*

  Specifies a file containing the path names of the files to be restored. Each path name must be listed on a separate line in the control file. Blank lines are ignored. These path names must be full path names; they should not contain any links. It is recommended that each path name be followed by a module name, in the form of `%`*`system`*`#`*`module`* or *`system_number`*`-`*`module_number`* on the same line, separated from the path name by one or more spaces. The module name indicates the module on which the file designated by the path name resided when used in transaction processing.

  Example of a control file:

  ```
  %sts#m7>fio_qa>test_tp_restore>make_files_mod1>file1  2-7
  %st2#m18>fio_qa>test_tp_restore>make_files_mod2>file3 %st2#m18
  ```

- `-gather_txns` *`number`*

  Specifies how many transactions the `tp_restore` command should process before clearing the user heap. The value must be an integer between `10` and `50`, inclusive. The default value is `50`.

  In general, use the default value. If a `tp_restore` process terminates abnormally because of a shortage of user heap space, reconstruct the destination directory and then reissue the `tp_restore` command, specifying a lower number for this argument.

- `-continuation`

  Starts the `tp_restore` session at the point where a previous `tp_restore` session stopped. This argument supports serial executions of the `tp_restore` command.

  Omit this argument if you are not doing serial executions of the `tp_restore` command.

Explanation

The tp_restore command is used when a transaction file has been corrupted or lost. This could happen, for example, if an incorrect transaction was committed, or if a user mistakenly deleted a transaction file.

To use tp_restore, the TPOverseer **must have been started with the option** -keep_transaction_log enabled, before the transactions involved were executed. This ensures that:

- the transaction log files will not be deleted when transaction log switching occurs
- files to be processed will have sufficient information stored in them during transaction processing to allow tp_restore to work.

Before invoking tp_restore, the user must place all transaction logs to be used during the restoration in the directory specified by -restore_dir. If the user wants to restore files on multiple modules, all transaction logs from the different modules must be placed in the same directory. The default is *master_disk*>system>tcf_directory.

The recovery may continue well past *stop_time*. For example, in cases where the transactions being restored from the log files access many files, tp_restore may stop long after *stop_time*. All transactions committed before or at *stop_time* will be restored, but others committed later may have to be included due to the mechanisms used to guarantee the integrity of transactions.

> **Caution:** The copy_file command should not be used with any transaction files as a method of backup and retrieval of files to be rolled forward by the tp_restore command. The copy_file command does not maintain transaction information across the copy. You should use the save and restore commands for backup and retrieval of files to be used with tp_restore. For information about backup and retrieval of files to be used with tp_restore, see Chapter 6 of the *VOS Transaction Processing Facility Guide* (R215).

The tp_restore command uses the module name following the path names in the -control control_file to determine the module ID for each file to be restored. It is good practice to always include the module name. When the restore takes place on a machine in the same OpenVOS network as the one on which the transaction processing originally occurred and when tp_restore can determine the module IDs from the information stored in network configuration .tin files, it is not necessary to specify the module name. When the restore takes place on a machine that does not have the information about the machines on which the original transaction processing took place, tp_restore will not be able to determine the module ID for the files. (For example, a backup system that is isolated from the primary system.) In this case, it is necessary to specify the module name and it is preferable to use the *system_number-module_number* format.

The -gather_txns argument provides the flexibility necessary to handle large transactions. In most cases, you can allow the -gather_txns argument to use its default value. For processing efficiency, the tp_restore command gathers information about multiple transactions and stores this information in the user heap area of memory. When it accumulates the number of transactions specified in the -gather_txns argument, the tp_restore command applies those transactions to the appropriate files and clears the user heap before starting to gather more transactions.

Depending on the number of transaction files involved in the transactions, as well as the number of modules, the number of log files involved, and the available resources in the user heap, `tp_restore` might run out of user heap space before it completes its information-gathering step. In that case, reconstruct the destination directory and then reissue the `tp_restore` command using a lower value in the `-gather_txns` argument.

The `-continuation` argument provides support for serial executions of the `tp_restore` command. Whenever the `tp_restore` command executes, it saves information about its stopping point in a file named `tp_restore_continuation_info` in the restore directory. The command also adds a message to its `.out` file naming the starting log for the next `tp_restore` session. To perform serial `tp_restore` sessions, use the log file identified in the previous session's `.out` file as the starting log for the next session and issue the `tp_restore` command using the `-continuation` argument. The `-continuation` argument causes the `tp_restore` command to use the `tp_restore_continuation_info` file to determine the starting point for each of the data files in the destination directory.

For serial `tp_restore` sessions, you must use log files with names whose numbers sequentially follow the logs used in the previous `tp_restore` session, with some overlap. Since the overlap requirement varies, always start with the log file named in the previous session's `.out` file. For example, assume that the log files used in a `tp_restore` session were named as follows:

```
tlf.128-1.0000000003.98-01-27
tlf.128-1.0000000004.98-01-27
tlf.128-1.0000000005.98-01-27
```

Now assume that the `tp_restore .out` file included the following message:

```
To restore files from module 1 in the next tp_restore -continuation
session, you should start with log tlf.128-1.0000000003.98-01-27
```

In this case, the next `tp_restore` session must include the first three files in the following list, and could have any number of subsequently numbered log files as well. For example, the next `tp_restore` session might use the following log files.

```
tlf.128-1.0000000003.98-01-27
tlf.128-1.0000000004.98-01-27
tlf.128-1.0000000005.98-01-27
tlf.128-1.0000000006.98-01-27
tlf.128-1.0000000007.98-01-27
```

You should use the same restore directory for all executions of the `tp_restore` command in the series. Otherwise, you risk accessing incorrect continuation information.

If you specify the `-continuation` argument with a restore directory that does not contain a `tp_restore_continuation_info` file, `tp_restore` executes as if the `-continuation` argument were not specified. If you are in the middle of a series of `tp_restore` operations, you **do not** want this to happen. To avoid this problem, create an OpenVOS command macro that executes the tp_restore command with the -continuation argument. The macro could check for the existence of the tp_restore_continuation_info file and if the file is not found, exit before executing tp_restore.

You can use the -continuation argument even if some of the transaction data files in the destination directory were not included in the previous tp_restore session. However, you must research the correct starting log file for the newly introduced transaction file and include the required log files in the restore directory. Follow the procedures described in "Explanation" to research the starting log file for a transaction data file. When the destination directory contains a transaction data file that was not included in the previous tp_restore session, the next tp_restore session knows that it must read **all** log files in the restore directory. However, when the transaction data files in the restore directory are identical to the files used in the previous session, tp_restore starts with the continuation log it identified in the previous session's .out file.

If you do not specify the -continuation argument or if you are performing the first tp_restore command in a planned series, tp_restore looks for certain information in the log files that ensures a proper starting point for rolling forward transactions for each of the files in the destination directory. You must research which log files to move to the restore directory. Follow the procedures described in "Explanation."

Messages in the tp_restore Output File

This section describes some situations that might cause the tp_restore command to produce messages in its output file. Since the tp_restore messages are lengthy and mostly self-explanatory, exact message text is not included here. Some situations that produce messages are as follows:

- If you specify the -continuation argument but the tp_restore command cannot find the tp_restore_continuation_info file in the restore directory, tp_restore produces a warning message and executes as if the -continuation argument were set to no. This situation could cause you to lose the ability to update an infrequently updated file using serial tp_restore methodology. To prevent this situation, follow the recommendations in the Explanation section for creating a macro that executes the tp_restore command.

- If the tp_restore command cannot find a log file it needs in the restore directory, it produces an error message stating that a log file is missing. The message tells you the name of the missing log file. After moving the missing log file into the restore directory, reissue the tp_restore command. (The process terminates before any files in the destination directory are updated.)

- If multiple tp_restore commands execute using the same restore directory, the second tp_restore command terminates and produces an error message stating "file in use." The message refers to work files that the tp_restore command uses. Wait until the first command finishes before issuing the next one. (The command terminates before any files in the destination directory are updated.)

- The tp_restore command generates an information message naming the starting log file for the next tp_restore session. If you are doing serial restore sessions, use this information when preparing the restore directory for the next execution.

- If the logs in the restore directory do not contain any updates for one of the files in the destination directory, the `tp_restore` command generates an information message stating that a file was not updated because no transactions affected it.
- If the restore directory does not have the correct starting log for some files in the destination directory, the `tp_restore` command rolls forward transactions into the other files in the destination directory. It also generates an information message telling you that a transaction file was not updated because the correct starting point was not seen in any logs. The message identifies the transaction file that was not updated. When this message is returned, your transaction files are not synchronized. Under this situation, you have the following three options.

  Option 1

  Research the correct starting log file for the unrestored transaction files, using the procedures described in "Explanation." Move the additional log files to the restore directory, and rerun `tp_restore`, leaving the destination directory unchanged. Use the `-continuation` argument if you are performing serial restores.

  If you are dealing with many transaction files or transaction files that are very large, rerunning `tp_restore` for all of the transaction files might be too time-consuming. In this case, perform the steps in Option 2, which reduces the runtime required by `tp_restore`.

  Option 2

  The following steps describe how to run a `tp_restore` session for an out-of-sync transaction file while preserving the ability to continue with serial restores later for the entire set of transaction files.

  1. Research the correct starting log file for the unrestored transaction file, using the procedures described in "Explanation." Copy the starting log file and all subsequent log files into a temporary restore directory.
  2. Use the `save` and `restore` commands to move the unrestored transaction file into a temporary destination directory. You must use the `save` and `restore` commands for this operation to obtain the required system information about the transaction file.
  3. Run `tp_restore` using the temporary restore and destination directories. The `-continuation` argument is optional for this run.
  4. Use the `save` and `restore` commands to move the newly restored transaction file back to the original destination directory.
  5. Examine the `.out` file in the temporary restore directory for the message that identifies the starting log for the next serial `tp_restore`. Compare this starting log to the starting log identified in the .out file in the permanent restore directory. The starting log for the next serial `tp_restore` session must be the earlier (older) of these two logs. If

necessary, copy log files from the temporary restore directory into the permanent restore directory.

6. You are now ready to continue normally with serial `tp_restore` sessions.

Option 3

If you are not performing serial restores, you can run a separate `tp_restore` session for the unrestored file in the current destination directory, using the following steps.

1. Research the correct starting log file for the unrestored transaction file, using the procedures described in the next section. Move the missing log files into the restore directory.

2. Move the transaction data files that were successfully rolled forward out of the destination directory.

3. Issue the `tp_restore` command.

Determining Which Log Files to Use in the Restore Process

The `tp_restore` command has been enhanced to require fewer log files than it formerly required in certain situations. Therefore, the recommended strategy for determining the starting log file for a `tp_restore` command has changed. Note that you can successfully restore files using more logs than necessary; however, when you use extra log files, the `tp_restore` command might take longer to execute.

Previously, you determined which log file should be the starting log file based on the saved transaction file's last modified date. Now, you can base your decisions on the saved transaction file's save date. In situations where files are saved more frequently than they are updated, this enhancement can eliminate a large number of log files. For example, if a transaction file is updated monthly but is saved during weekly backups, you can restore that file using log files made just prior to the save.

To determine the starting log file for a nonserial `tp_restore` command, perform the following steps.

1. Find the date/time the transaction data file was saved. If you are restoring multiple files, be sure to find the date/time for the transaction data file that was saved first (earliest).

2. Find the log file whose date/time last modified is closest to the date found in step 1.

3. Go back two log files prior to the log file found in step 2. This should be your starting log file.

To make step 2 easier, you might want to establish a record-keeping procedure that keeps track of which log file is active at the time transaction files are saved. A suggested form with a sample entry is shown in Figure 1-1. In the second column, record the log file name that is the current target of the `transaction_log` link in the `>system>transaction_logs` directory when you start the save procedure. To find the log file name, go to the `transaction_logs` directory and execute the `list -links` command.

Transaction File Saved	**Active `transaction_log` at Start of Save**	**Minus 2 Logs to Restore**
1. `file1`	`tlf.1-19.0000069219.98-04-05`	`...69217...`
2.		
3.		

Figure 1-1. Active Log Files During Transaction File Saves

Related Information

See the *VOS Transaction Processing Facility Guide* (R215) for a detailed explanation on how to use the `tp_restore` command.

Chapter 2:
Transaction Processing Facility Requests

Requests, also called *monitor requests*, are special commands that you can type on a terminal connected to a *monitor task*, a task in which either the subroutine `s$monitor` or the subroutine `s$monitor_full` is executing. There can be only one such task per process, and it must be the *primary* task, which has a task identifier of `1`.

Requests are typically used to control or obtain information about tasks interactively. However, a monitor task can execute with no terminal attached, accepting requests from a file or queue.

Request lines are typed after the monitor prompt. They have the same syntax as OpenVOS command lines, except that request names are used instead of command names.

You can invoke OpenVOS internal commands from the monitor task by beginning the request line with two periods.

See the subroutines `s$monitor` and `s$monitor_full` in Chapter 3 for further information on invoking requests and OpenVOS internal commands.

This chapter describes the following transaction processing facility requests.

- `cleanup_task`
- `control_task`
- `create_task`
- `delete_message`
- `delete_task`
- `display_task_info`
- `display_tp_parameters`
- `help`
- `initialize_configuration`
- `initialize_task`
- `list_messages`
- `quit`
- `set_no_wait_mode`
- `set_task_priority`
- `set_tp_parameters`
- `set_wait_mode`
- `start_task`
- `stop_task`

Making Programs Available As Requests

Requests can include the OpenVOS-supplied programs described in this chapter and programs written by the user. To be recognized as requests to the monitor task, both OpenVOS-supplied and user-written programs must be listed in the `retain:` directive in an application's binder control file. The two exceptions to this rule are the `help` and `quit` requests, which are built into the monitor subroutines. The name of the object file containing a program must be listed in the `modules:` directive of the binder control file. For OpenVOS-supplied programs, this is the same as the request name.

Figure 2-1 shows a sample binder control file. The `open_station` and `close_station` modules are user-written programs, and the other requests are OpenVOS-supplied. See the `bind` command in the *OpenVOS Commands Reference Manual* (R098) for further information on binding.

```
name:               order_system;

number_of_tasks:    10;

modules:            open_station,
                    close_station,
                    control_task,
                    display_task_info,
                    initialize_configuration;

retain:             open_station,
                    close_station,
                    control_task,
                    display_task_info,
                    initialize_configuration;

end;
```

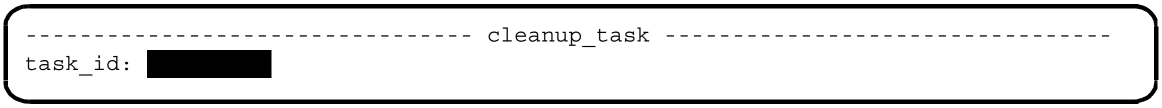
Figure 2-1. A Sample Binder Control File

cleanup_task

Purpose

This request closes the terminal assigned to each task specified and detaches the port attached to the terminal.

Display Form



```
------------------------------- cleanup_task -------------------------------
task_id:
```

Command Line Form

`cleanup_task` *`task_id`*

Arguments

- ▶ *`task_id`* **Required**

  The identifier of the task to be cleaned up. It must be either the valid identifier of a task in the process or an asterisk. If you specify an asterisk, `cleanup_task` cleans up all the tasks in the process except the current task.

Explanation

A `cleanup_task` request closes the terminals of the specified tasks and detaches the terminal ports. It puts the given tasks into the `uninitialized` state. See the description of the request `initialize_task`.

The identifier of a task is an integer from 1 to the number of tasks in the process inclusive.

Related Information

`control_task`
Changes the state of each specified task in the current process.

`create_task`
Creates a dynamic task.

`delete_task`
Deletes a dynamic task.

`display_task_info`
Displays information about one or all tasks.

`initialize_task`
: Initializes a specified `uninitialized` task.

`initialize_configuration`
: Initializes a set of tasks described in a task configuration file. The request attaches a specified terminal for each task and starts the tasks.

`set_no_wait_mode`
: Puts a specified port into `no_wait` mode.

`set_wait_mode`
: Puts a specified port into `wait` mode.

`start_task`
: Makes each specified `initialized` or `stopped` task ready to run.

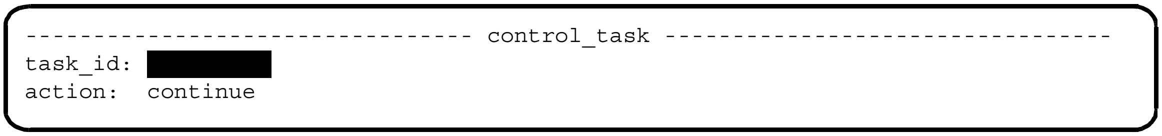
`stop_task`
: Stops each specified task.

control_task

Purpose

This request changes the state of each specified task in the current process.

Display Form



```
------------------------------- control_task --------------------------------
task_id:
action:  continue
```

Command Line Form

```
control_task task_id
        [action]
```

Arguments

- *task_id* **Required**

  The identifier of the task to be controlled. It must be either the valid identifier of a task in the process or an asterisk. If you specify an asterisk, `control_task` performs the action on all the tasks in the process except the current task.

- *action* CYCLE

  A control action. The choices for *action* are `continue_task`, `enable_metering`, `disable_metering`, and `pause_task`. If you omit this argument, the default is `continue_task`.

Explanation

A `control_task` request carries out a given control action on a set of tasks.

The control action `continue_task` puts a `paused_ready` task into the `ready` state and puts a `paused_waiting` task into the `waiting` state. Applying this control action to a `stopped` task is an error.

The control actions `enable_metering` and `disable_metering` enable and disable metering of the execution of a task.

The control action `pause_task` puts a `ready` or `running` task into the `paused_ready` state and puts a `waiting` task into the `paused_waiting` state. It is an error to apply this control action to a `stopped` task. A `paused_ready` or `paused_waiting` task does not run until you continue it with the control action `continue_task`.

The identifier of a task is an integer from 1 to the number of tasks in the process inclusive.

Related Information

`cleanup_task`
Detaches the terminal assigned to each specified task.

`create_task`
Creates a dynamic task.

`delete_task`
Deletes a dynamic task.

`display_task_info`
Displays information about one or all tasks.

`initialize_task`
Initializes a specified `uninitialized` task.

`initialize_configuration`
Initializes a set of tasks described in a task configuration file. The request attaches a specified terminal for each task and starts the tasks.

`set_no_wait_mode`
Puts a specified port into `no_wait` mode.

`set_wait_mode`
Puts a specified port into `wait` mode.

`start_task`
Makes each specified `initialized` or `stopped` task ready to run.

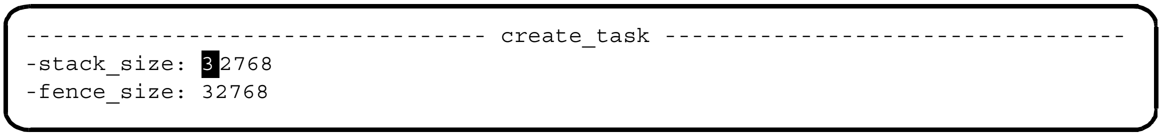
`stop_task`
Stops each specified task.

create_task

Purpose

This request creates a dynamic task.

Display Form



```
--------------------------------- create_task ---------------------------------
-stack_size: 32768
-fence_size: 32768
```

Command Line Form

```
create_task [-stack_size stack_size]
          [-fence_size fence_size]
```

Arguments

- `-stack_size` *`stack_size`*

  Specifies the number of bytes of storage to allocate for the stack in this task. If you specify less than the minimum required by OpenVOS (currently 1024), `create_task` uses the minimum. The default value set by OpenVOS is 32768.

- `-fence_size` *`fence_size`*

  Specifies the minimum number of bytes of storage to allocate for the fence following the stack in this task. The `create_task` request rounds `fence_size` up to the next page boundary (unit of 4096 bytes). The default value set by OpenVOS is 32768.

Explanation

The `create_task` request creates a dynamic task. *Dynamic tasks* differ from tasks defined in a program module's binder control file in that:

- you can define and create them at run time, as needed
- you can delete them at run time, using `delete_task`, when they are no longer needed
- they have the added security of a fence.

The `create_task` request allocates space in the user heap for the new task's static storage, stack, and fence, and places the task in the `uninitialized` state. You can then start the new task with the initialize_task and start_task requests.

The *fence* is a region of storage following the stack frame. It allows OpenVOS to detect most references beyond the end of the stack without any corruption of data in adjacent regions.

Related Information

`cleanup_task`
Detaches the terminal assigned to each specified task.

`control_task`
Changes the state of each specified task in the current process.

`delete_task`
Deletes a dynamic task.

`display_task_info`
Displays information about one or all tasks.

`initialize_task`
Initializes a specified `uninitialized` task.

`initialize_configuration`
Initializes a set of tasks described in a task configuration file. The request attaches a specified terminal for each task and starts the tasks.

`set_no_wait_mode`
Puts a specified port into `no_wait` mode.

`set_wait_mode`
Puts a specified port into `wait` mode.

`start_task`
Makes each specified `initialized` or `stopped` task ready to run.

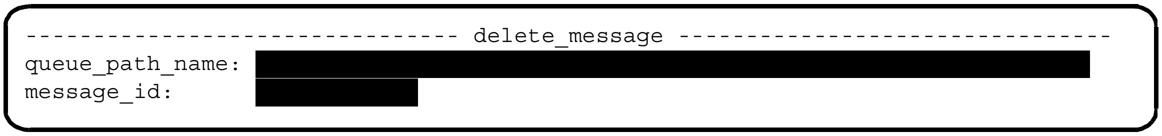
`stop_task`
Stops each specified task.

delete_message

Purpose

This request removes a message from a message queue.

Display Form



```
------------------------------- delete_message -------------------------------
queue_path_name:
message_id:
```

Command Line Form

`delete_message` *`queue_path_name message_id`*

Arguments

- ▶ *`queue_path_name`* **Required**

  The path name of a message queue.

- ▶ *`message_id`* **Required**

  The identifier of the message to be removed.

Explanation

The `delete_message` request removes a message from a message queue.

The argument *`queue_path_name`* must be the path name of the queue. The argument *`message_id`* must be the identifier of a message in the queue.

You can remove any message that you sent to the queue. To remove any other message, you need write access to the queue.

Related Information

`list_messages`

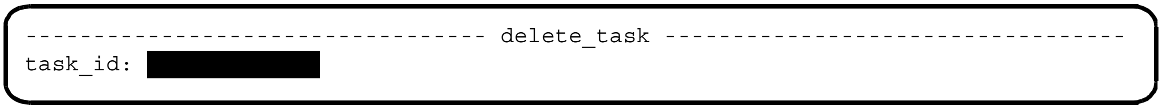
Lists the messages currently in a server queue or message queue.

delete_task

Purpose

This request deletes a dynamic task.

Display Form



```
---------------------------------- delete_task ----------------------------------
task_id:
```

Command Line Form

`delete_task` *`task_id`*

Arguments

- ▶ *`task_id`* **Required**

  The task identifier of the task to be deleted.

Explanation

The `delete_task` request deletes a dynamic task, created by `create_task` or `s$create_task`. (A *static* task, created by the binder, cannot be deleted.) The task to be deleted must be in the `uninitialized` state.

The task monitor releases the memory allocated for the task's stack, fence, and internal static. It does **not**, however, free any other heap storage the task has allocated, or detach or close any ports the task has attached or opened.

Related Information

`cleanup_task`
: Detaches the terminal assigned to each specified task.

`control_task`
: Changes the state of each specified task in the current process.

`create_task`
: Creates a dynamic task.

`display_task_info`
: Displays information about one or all tasks.

`initialize_task`
: Initializes a specified `uninitialized` task.

`initialize_configuration`
: Initializes a set of tasks described in a task configuration file. The request attaches a specified terminal for each task and starts the tasks.

`set_no_wait_mode`
: Puts a specified port into `no_wait` mode.

`set_wait_mode`
: Puts a specified port into `wait` mode.

`start_task`
: Makes each specified `initialized` or `stopped` task ready to run.

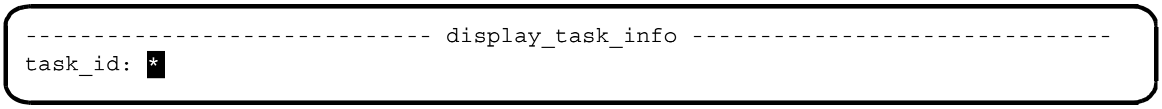
`stop_task`
: Stops each specified task.

display_task_info

Purpose

This request displays information about one or all tasks.

Display Form



```
------------------------------ display_task_info -------------------------------
task_id: *
```

Command Line Form

```
display_task_info [task_id]
```

Arguments

▶ `task_id`

The identifier of a task about which you want information. It must be either a valid identifier of a task in the process or an asterisk (the default). If you omit this argument, or if you give an asterisk as the argument, the monitor displays information about all of the tasks in the process.

Explanation

A `display_task_info` request displays the following information about one or all tasks in a process:

- task identifier
- task state
- address of the task stack
- address of the task internal static storage region
- identifier of the task terminal port
- task expended CPU time (if metered)
- number of page faults taken by the task (if metered).

Related Information

`cleanup_task`

Detaches the terminal assigned to each specified task.

`control_task`

Changes the state of each specified task in the current process.

`create_task`
Creates a dynamic task.

`delete_task`
Deletes a dynamic task.

`initialize_task`
Initializes a specified `uninitialized` task.

`initialize_configuration`
Initializes a set of tasks described in a task configuration file. The request attaches a specified terminal for each task and starts the tasks.

`set_no_wait_mode`
Puts a specified port into `no_wait` mode.

`set_wait_mode`
Puts a specified port into `wait` mode.

`start_task`
Makes each specified `initialized` or `stopped` task ready to run.

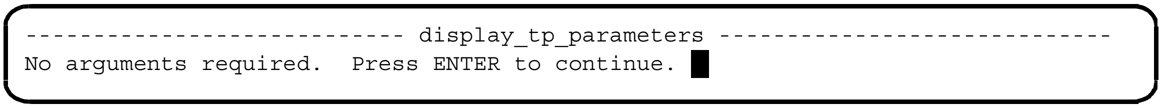
`stop_task`
Stops each specified task.

display_tp_parameters

Purpose

This request displays the lock contention parameters for transaction locking for the current program or process.

Display Form



```
--------------------------- display_tp_parameters ----------------------------
No arguments required.  Press ENTER to continue. █
```

Command Line Form

```
display_tp_parameters
```

Arguments

None.

Explanation

The `display_tp_parameters` request displays the current values assigned by OpenVOS to all transactions started by the current program.

The following example shows output from the request `display_tp_parameters`:

```
The transaction parameters are:
  priority:        0
  time value:      10 second(s)
  ignore priority  no
  ignore time:     no
  younger wins:    no
  allow deadlocks: no
```

Related Information

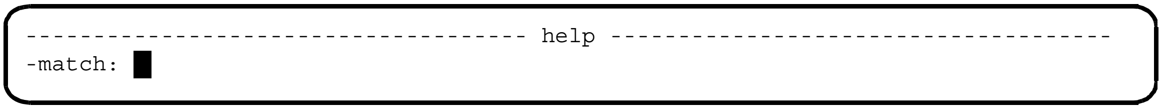
`set_tp_parameters`
: Sets the lock contention parameters for transaction locking for the current program.

help

Purpose

This request lists all of the requests you can make from the current monitor program.

Display Form



```
------------------------------------ help ------------------------------------
-match: █
```

Command Line Form

```
help
          [-match string]
```

Arguments

- ▶ `-match` *`string`*

  An option to list only those requests containing the specified string.

Explanation

The `help` request lists all the requests bound with the monitor program and therefore available to the administrator.

Related Information

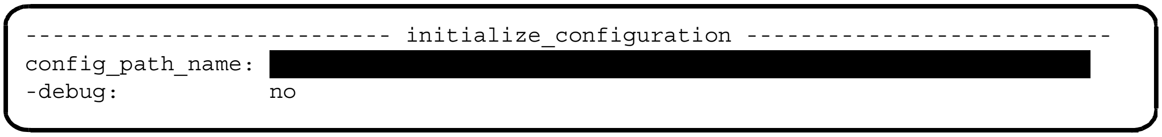
None.

initialize_configuration

Purpose

This request initializes a set of tasks described in a task configuration file. The request attaches a specified terminal for each task and starts the tasks.

Display Form



```
-------------------------- initialize_configuration --------------------------
config_path_name:
-debug:           no
```

Command Line Form

```
initialize_configuration config_path_name
          [-debug]
```

Arguments

- `config_path_name` **Required**

  The path name of a task configuration file.

- `-debug` CYCLE

  An option to call the debugger before starting each task.

Explanation

The `initialize_configuration` request reads the configuration file *`config_path_name`* and starts the specified tasks. The format of a task configuration file follows.

If the task configuration file contains a new specification for the primary task (`task_id 1`), `initialize_configuration` restarts it with the specified entry point and terminal port attachment.

Task Configuration Files

Use the command `create_table`, documented in the *OpenVOS System Administration: Configuring a System* (R287), to create a task configuration file. The `create_table` command requires two input files: a data description file and a table input file. The data description file, which must have suffix `.dd`, specifies the format of the table being created. The table input file, which must have suffix `.tin`, contains the data to be stored in the table.

The configuration file must contain a record for each task that is to be initialized. The declaration of a task's record in the configuration file must be equivalent to the following:

```
declare 1 task_record,
          2 task_id           binary(15),
          2 entry_name        char(32) varying,
          2 terminal_name     char(66) varying;
```

The `task_id` argument is the identifier of a new task. It must be either a positive integer or 0. If it is 0, `initialize_configuration` uses the next available positive integer for the identifier of the task.

The `entry_name` argument is the name of the entry point in the program where the task is to begin executing. You must list the name of the entry in the `retain:` directive of the binder control file for the program module into which `initialize_configuration` is bound. See the beginning of this chapter for an example of a binder control file.

The `terminal_name` argument must be either an empty string or the full path name of an I/O device to which the task is to be attached. If it is the empty string, the task will not be attached to a terminal.

Figures 2-2 and 2-3 show the data description file and a sample table input file. The contents of the data description file must be as shown.

```
fields:
          task_id             bin(15),
          entry_name          char(32) varying,
          terminal_name       char(66) varying;
end;
```

Figure 2-2. The Data Description File for the `initialize_configuration` Request

```
/ =task_id      2
=entry_name     personnel_administration
=terminal_name  %Riverside#t2.5

/ =task_id      3
=entry_name     personnel_administration
=terminal_name  %Riverside#t2.7

/ =task_id      5
=entry_name     batch_metering
=terminal_name  ''
```

Figure 2-3. A Sample Table Input File for the `initialize_configuration` Request

The format of a record in the `create_table` command input file therefore is the following:

```
/=task_id            task_id
 =entry_name         entry
 =terminal_name      %system#terminal
```

task_id is the identifier of the task, *entry* is the name of the entry and %*system*#*terminal* is the full path name of the terminal or other I/O device to be connected to the task.

Using the data description and table input files above, the following command would produce a file *config_path_name*`.table`.

```
create_table config_path_name -description_path
task_description
```

Subsequently invoking `initialize_configuration` with the name of this file as its argument would initialize and start three tasks.

Related Information

`cleanup_task`
Detaches the terminal assigned to each specified task.

`control_task`
Changes the state of each specified task in the current process.

`create_task`
Creates a dynamic task.

`delete_task`
Deletes a dynamic task.

`display_task_info`
Displays information about one or all tasks.

`initialize_task`
Initializes a specified `uninitialized` task.

`set_no_wait_mode`
: Puts a specified port into `no_wait` mode.

`set_wait_mode`
: Puts a specified port into `wait` mode.

`start_task`
: Makes each specified `initialized` or `stopped` task ready to run.

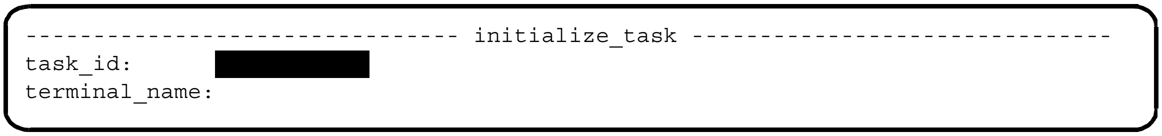
`stop_task`
: Stops each specified task.

`initialize_task`

Purpose

This request initializes a specified `uninitialized` task.

Display Form



```
------------------------------- initialize_task -------------------------------
task_id:
terminal_name:
```

Command Line Form

```
initialize_task task_id
          [terminal_name]
```

Arguments

- ▶ `task_id` **Required**

  The identifier of the task to be initialized. It must be a valid identifier of a task in the process.

- ▶ *`terminal_name`*

  The path name of a terminal to which the task's default ports are to be attached.

Explanation

The `initialize_task` request changes the state of the task specified by *`task_id`* from `uninitialized` to `initialized`. It is an error to attempt to initialize a task in any other state.

If you supply a terminal path name, the monitor attaches the task's default ports (`default_input`, `default_output`, `command_input`, and `terminal_output`) to the given terminal. If you omit this argument, the task's ports are not attached to any port. In this case, you should not write output from the task.

Related Information

`cleanup_task`

Detaches the terminal assigned to each specified task.

`control_task`

Changes the state of each specified task in the current process.

`create_task`
Creates a dynamic task.

`delete_task`
Deletes a dynamic task.

`display_task_info`
Displays information about one or all tasks.

`initialize_configuration`
Initializes a set of tasks described in a task configuration file. The request attaches a specified terminal for each task and starts the tasks.

`set_no_wait_mode`
Puts a specified port into `no_wait` mode.

`set_wait_mode`
Puts a specified port into `wait` mode.

`start_task`
Makes each specified `initialized` or `stopped` task ready to run.

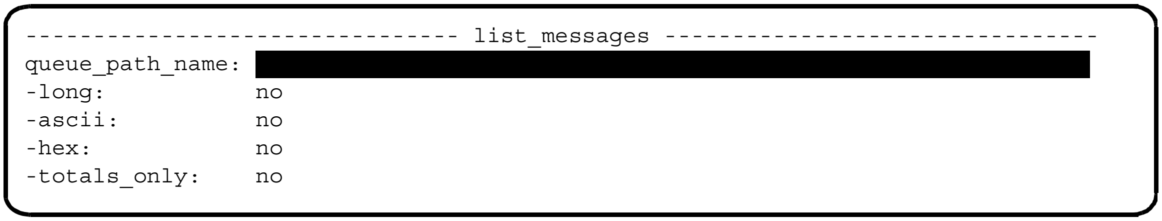
`stop_task`
Stops each specified task.

list_messages

Purpose

This request lists the messages currently in a server queue or message queue.

Display Form



```
------------------------------- list_messages -------------------------------
queue_path_name:
-long:           no
-ascii:          no
-hex:            no
-totals_only:    no
```

Command Line Form

```
list_messages queue_path_name
        [-long]
        [-ascii]
        [-hex]
        [-totals_only]
```

Arguments

- `queue_path_name` **Required**

  The path name of a server queue or message queue. The messages in the queue are listed.

- `-long` CYCLE

  Displays the full message headers.

- `-ascii` CYCLE

  Displays the text of the messages in ASCII characters.

- `-hex` CYCLE

  Displays the text of the messages in hexadecimal characters.

- `-totals_only` CYCLE

  Displays only the number of the messages in the queue.

Explanation

The `list_messages` request displays information about the messages in a given queue. To get information about messages in a queue, you must have execute, read, or write access to it.

If you have execute or read access to the queue, you can get information about your messages only. If you have write access, you can get information about all of the messages.

The information displayed about each message is the information contained in the message header:

- the message's identifier
- the time of day the message was sent
- the message's priority
- the message's subject
- the requester's process identifier
- the requester's user name
- whether the queue is a server queue
- whether the message is busy (being serviced)
- whether the message has been busy
- whether the message has been aborted.

If the queue is a server queue, `list_messages` lists only those messages that have not yet been replied to by the server. Messages that have been replied to but have not yet been removed from the queue by `s$msg_receive_reply` are not listed.

If you supply the `-totals_only` argument, `list_messages` displays only the number of the messages in the queue.

Related Information

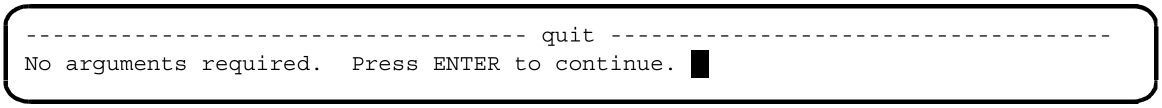
`delete_message`
Removes a message from a message queue.

quit

Purpose

This request tells the subroutine `s$monitor` or the subroutine `s$monitor_full`, whichever is running, to stop accepting requests and return to its caller.

Display Form



```
------------------------------------- quit -------------------------------------
No arguments required.  Press ENTER to continue. █
```

Command Line Form

```
quit
```

Arguments

None.

Explanation

The `quit` request tells `s$monitor` or `s$monitor_full`, whichever is running, to stop accepting requests and return to its caller.

Related Information

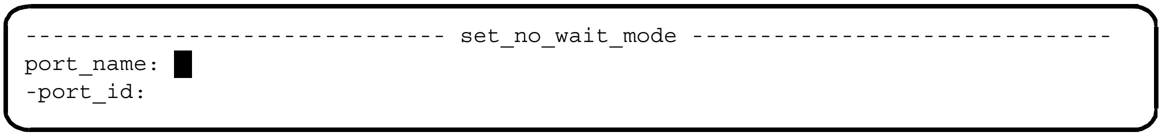
None.

set_no_wait_mode

Purpose

This request puts a specified port into no-wait mode.

Display Form



```
------------------------------ set_no_wait_mode ------------------------------
port_name: █
-port_id:
```

Command Line Form

```
set_no_wait_mode {port_name -port_id port_id}
```

Arguments

- ▶ *port_name*

  The name of a port attached in the current process.

- ▶ -port_id *port_id*

  The identifier of a port attached in the current process.

Explanation

The `set_no_wait_mode` request puts a given port into no-wait mode.

You must specify either the name of a port or the identifier of a port, but you cannot specify both.

A port can be in wait mode or in no-wait mode. When a port is created, it is in wait mode. The operating system makes a process wait for the completion of an I/O request made through a port in wait mode before returning control to the process. The operating system returns control immediately to a process when the process makes an I/O request through a port that is in no-wait mode. Therefore, a process must explicitly wait for the completion of such I/O. The operating system switches tasks when the port is in wait mode, but it does not switch if the port is in no-wait mode. When a port is in wait mode, OpenVOS makes the calling task wait, but if other tasks are ready to run, OpenVOS gives control to one of them. See the explanations of the subroutines `s$set_no_wait_mode` and `s$set_wait_mode` in the OpenVOS Subroutines manuals.

Related Information

`cleanup_task`
Detaches the terminal assigned to each specified task.

`control_task`
Changes the state of each specified task in the current process.

`create_task`
Creates a dynamic task.

`delete_task`
Deletes a dynamic task.

`display_task_info`
Displays information about one or all tasks.

`initialize_task`
Initializes a specified `uninitialized` task.

`initialize_configuration`
Initializes a set of tasks described in a task configuration file. The request attaches a specified terminal for each task and starts the tasks.

`set_wait_mode`
Puts a specified port into `wait` mode.

`start_task`
Makes each specified `initialized` or `stopped` task ready to run.

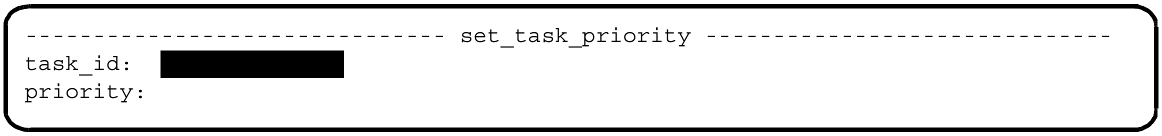
`stop_task`
Stops each specified task.

set_task_priority

Purpose

This request sets the scheduling priority for a task.

Display Form



```
------------------------------ set_task_priority -----------------------------
task_id:
priority:
```

Command Line Form

`set_task_priority` *`task_id priority`*

Arguments

- *`task_id`* **Required**

  The task identifier of the task whose priority is to be set.

- *`priority`* **Required**

  A priority value in the range 0 to 255 inclusive.

Explanation

The `set_task_priority` request sets the scheduling priority of a task. You can set the priority of both static and dynamic tasks.

A task with a higher-numbered priority is scheduled before a task with a lower-numbered priority.

Related Information

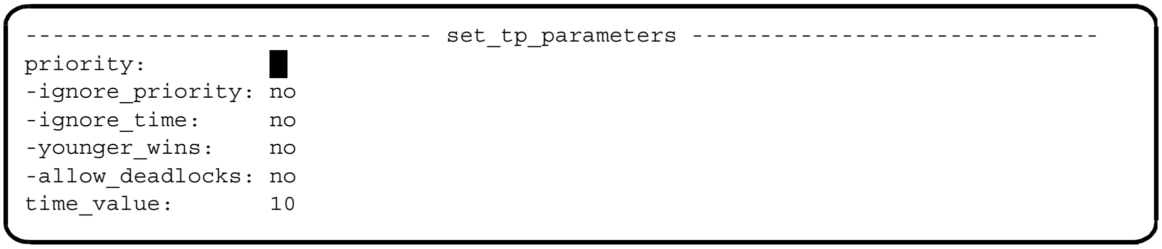
None.

set_tp_parameters

Purpose

This request sets the lock contention parameters for transaction locking for the current program.

Display Form



```
----------------------------- set_tp_parameters -----------------------------
priority:          █
-ignore_priority:  no
-ignore_time:      no
-younger_wins:     no
-allow_deadlocks:  no
time_value:        10
```

Command Line Form

```
set_tp_parameters [priority]
          [time_value]
          [-ignore_priority]
          [-ignore_time]
          [-younger_wins]
          [-allow_deadlocks]
```

Arguments

- ▶ *`priority`*

  The default lock contention priority to be given a transaction by `s$start_transaction` for the current program. The value can be 0 to 9 inclusive. The default value set by OpenVOS (unless changed by set_tp_default_parameters or `s$set_tp_default_parameters`) is 0.

- ▶ `-ignore_priority` CYCLE

  Tells OpenVOS whether to ignore the priorities of transactions competing for locks. The default value set by OpenVOS (unless changed by set_tp_default_parameters or `s$set_tp_default_parameters`) is `no`. In this case, transaction priorities are considered in determining which transaction wins.

- ▶ `-ignore_time` CYCLE

  Tells OpenVOS whether to ignore the start times of competing transactions. The default value set by OpenVOS (unless changed by set_tp_default_parameters or `s$set_tp_default_parameters`) is `no`. In this case, the starting times are considered in deciding which transaction wins.

▶ `-younger_wins` CYCLE

Specifies whether the transaction with a later start time will win the contention. The default value set by OpenVOS (unless changed by set_tp_default_parameters or `s$set_tp_default_parameters`) is `no`. In this case, then the transaction which started sooner (the older transaction) will win.

▶ `-allow_deadlocks` CYCLE

Tells OpenVOS whether to ignore any deadlock between two transactions of equal precedence. The default value set by OpenVOS (unless changed by set_tp_default_parameters or `s$set_tp_default_parameters`) is `no`.

> **Caution:** If `-allow_deadlocks` is `yes` for two transactions of equal precedence involved in a conflict, OpenVOS does not abort either transaction; instead, OpenVOS returns the error `e$record_in_use` (`2408`). When this happens, you must abort the transaction. If you do not do so, it is possible for the deadlock to continue indefinitely, until one of the transactions is aborted. For this reason, the use of this option is not recommended.
>
> If `-allow_deadlocks` is `no` (the default) OpenVOS breaks the deadlock. See the Explanation below.

▶ *`time_value`*

A number of seconds between 0 and 1024, inclusive. If two transactions differ in the time they were started by less than *`time_value`*, then OpenVOS considers them to have started at the same time. The default value set by OpenVOS (unless changed by set_tp_default_parameters or `s$set_tp_default_parameters`) is 10.

Explanation

The lock arbitration and resolution sequence proceeds as follows: (Assume that transaction A holds a lock and transaction B wants a lock, but they are conflicting locks—both A and B cannot have their locks at the same time.)

Arbitration:

1. If one transaction, and only one, has `priority` set to `-1` (`always_lose`) by `s$start_priority_transaction`, then that one loses; go to **Resolution**. Otherwise, go to **2**.

2. If both transactions have `-ignore_priorities` set to `yes` or their priorities are equal, then go to **3**. Otherwise, the higher priority wins; go to **Resolution**.

3. If both transactions have `-ignore_time` set to `yes` or if their starting times differ by less than B’s *`time_value`*, then go to **Deadlocks**. Otherwise, go to **4**.

4. If both transactions have `-younger_wins` set to `yes`, then the younger wins; go on to **Resolution**. Otherwise, the older wins; go on to **Resolution**.

Deadlocks:

If neither transaction has won by the rules of arbitration and at least one has `-allow_deadlocks` set to `no`, OpenVOS breaks the deadlock by choosing a winner based on the value of an internal transaction identifier. In most cases the older transaction is the winner. However, if both transactions have `-younger_wins` set to `yes`, then the younger transaction is the winner. In either case, if one of the transactions has lost a deadlock immediately before, it wins this time.

Resolution:

If B (wants lock) wins then abort A (has lock); otherwise, B waits for `lock_wait_time` (set by set_lock_wait_time, `s$set_lock_wait_time`, `s$set_default_lock_wait_time`, or by OpenVOS) and then tries again. If the lock has not become available during that time, then abort B. The error code `e$tp_aborted (2931)` is returned for the transaction that loses.

Related Information

`display_tp_parameters`

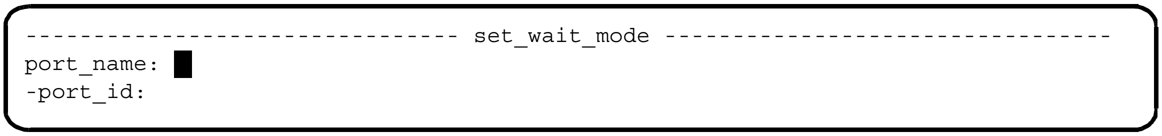
Displays the lock contention parameters for transaction locking for the current program or process.

set_wait_mode

Purpose

This request puts a specified port into wait mode.

Display Form



```
------------------------------ set_wait_mode -------------------------------
port_name: █
-port_id:
```

Command Line Form

```
set_wait_mode {port_name -port_id port_id}
```

Arguments

▶ `port_name`

The name of a port attached in the current process.

▶ `-port_id` *`port_id`*

The identifier of a port attached in the current process.

Explanation

The `set_wait_mode` request puts the specified port into wait mode.

You must specify either the name of a port or the identifier of a port but you cannot specify both.

A port can be in wait mode or in no-wait mode. When created, a port is in wait mode. The operating system makes a process wait for the completion of an I/O request made through a port in wait mode before returning control to the process. The operating system returns control immediately to a process when the process makes an I/O request through a port in no-wait mode. Therefore, a process must explicitly wait for the completion of such I/O. The operating system switches tasks when the port is in wait mode, but it does not switch if the port is in no-wait mode. When a port is in wait mode, OpenVOS makes the calling task wait, but if other tasks are ready to run, OpenVOS gives control to one of them. See the explanations of the subroutines `s$set_no_wait_mode` and `s$set_wait_mode` in the OpenVOS Subroutines manuals.

Related Information

`cleanup_task`
Detaches the terminal assigned to each specified task.

`control_task`
Changes the state of each specified task in the current process.

`create_task`
Creates a dynamic task.

`delete_task`
Deletes a dynamic task.

`display_task_info`
Displays information about one or all tasks.

`initialize_task`
Initializes a specified `uninitialized` task.

`initialize_configuration`
Initializes a set of tasks described in a task configuration file. The request attaches a specified terminal for each task and starts the tasks.

`set_no_wait_mode`
Puts a specified port into `no_wait` mode.

`start_task`
Makes each specified `initialized` or `stopped` task ready to run.

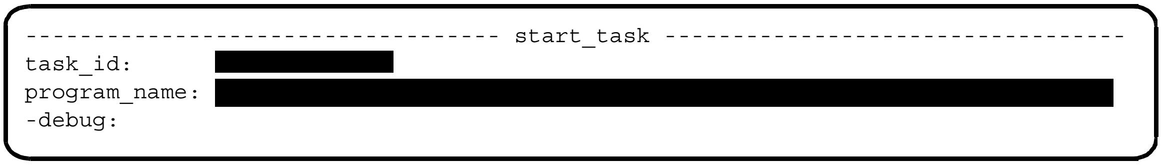
`stop_task`
Stops each specified task.

start_task

Purpose

This request makes each specified initialized or stopped task ready to run.

Display Form



```
---------------------------------- start_task ----------------------------------
task_id:
program_name:
-debug:
```

Command Line Form

```
start_task task_id program_name
        [-debug]
```

Arguments

- *task_id* **Required**

  The identifier of the task to be started. It must be either the valid identifier of a task in the process or an asterisk. If you specify an asterisk, `start_task` starts all the tasks in the current process except the current task.

- *program_name* **Required**

  The name of an entry point in the current program. The monitor starts the specified task or tasks at this entry point.

- `-debug` CYCLE

  Starts the execution of the program *program_name* in the task under the control of the debugger. If you omit this option, the task does not run under the debugger's control.

Explanation

The `start_task` request puts the task or tasks specified by *task_id* into the `ready` state. The specified tasks must be in the `initialized` or `stopped` state.

The identifier of a task is an integer from 1 to the number of tasks in the process inclusive.

The name of the program that the task is to execute is *program_name*. This name must be in the entry map created for the application by the binder. To get the entry name into the entry map for the program, list it in the `retain:` directive of the binder control file for the application. See the beginning of this chapter for an example of a binder control file.

Related Information

`cleanup_task`
Detaches the terminal assigned to each specified task.

`control_task`
Changes the state of each specified task in the current process.

`create_task`
Creates a dynamic task.

`delete_task`
Deletes a dynamic task.

`display_task_info`
Displays information about one or all tasks.

`initialize_task`
Initializes a specified `uninitialized` task.

`initialize_configuration`
Initializes a set of tasks described in a task configuration file. The request attaches a specified terminal for each task and starts the tasks.

`set_no_wait_mode`
Puts a specified port into `no_wait` mode.

`set_wait_mode`
Puts a specified port into `wait` mode.

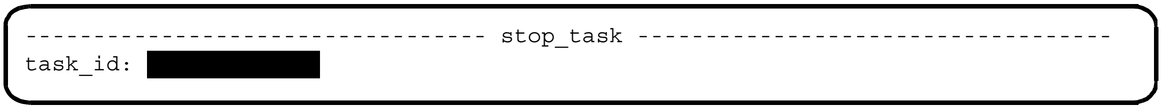
`stop_task`
Stops each specified task.

stop_task

Purpose

This request stops each specified task.

Display Form



```
-------------------------------- stop_task ----------------------------------
task_id:
```

Command Line Form

`stop_task` *`task_id`*

Arguments

▶ *`task_id`* **Required**

The identifier of a task. It must be either the valid identifier of a task in the process, or an asterisk. The monitor stops the task. If you specify an asterisk, `stop_task` stops all the tasks in the process except the current task.

Explanation

The `stop_task` request puts a task or tasks, specified by *`task_id`* into the `stopped` state. A task to be stopped must not be in the `uninitialized` or `initialized` state.

The identifier of a task is an integer from 1 to the number of tasks in the process inclusive.

Related Information

`cleanup_task`
Detaches the terminal assigned to each specified task.

`control_task`
Changes the state of each specified task in the current process.

`create_task`
Creates a dynamic task.

`delete_task`
Deletes a dynamic task.

`display_task_info`
Displays information about one or all tasks.

`initialize_task`
: Initializes a specified `uninitialized` task.

`initialize_configuration`
: Initializes a set of tasks described in a task configuration file. The request attaches a specified terminal for each task and starts the tasks.

`set_no_wait_mode`
: Puts a specified port into `no_wait` mode.

`set_wait_mode`
: Puts a specified port into `wait` mode.

`start_task`
: Makes each specified `initialized` or `stopped` task ready to run.

Chapter 3:
Transaction Processing Facility Subroutines

This chapter documents the following transaction processing facility subroutines.

s$abort_transaction
s$add_task_epilogue_handler
s$call_server
s$cleanup_task
s$commit_transaction
s$control_task
s$create_task
s$delete_task
s$delete_task_epilogue_handler
s$enable_tasking
s$get_default_lock_wait_time
s$get_free_task_id
s$get_lock_wait_time
s$get_max_task_id
s$get_server_queue_info
s$get_task_id
s$get_task_info
s$get_tp_abort_code
s$get_tp_default_parameters
s$get_tp_parameters
s$init_task
s$init_task_config
s$monitor
s$monitor_full
s$msg_cancel_receive
s$msg_delete
s$msg_open
s$msg_open_direct
s$msg_open_virtual
s$msg_read
s$msg_receive
s$msg_receive_reply
s$msg_rewrite
s$msg_send
s$msg_send_reply
s$reschedule_task
s$set_default_lock_wait_time
s$set_lock_wait_time
s$set_max_queue_depth
s$set_process_terminal
s$set_task_priority
s$set_task_terminal
s$set_task_wait_info
s$set_tp_default_parameters
s$set_tp_parameters
s$set_transaction_file
s$start_priority_transaction
s$start_task
s$start_task_full†
s$start_transaction
s$task_setup_wait
s$task_setup_wait2
s$task_wait_event
s$task_wait_event2
s$truncate_queue

† s$start_task_full is not available in FORTRAN.

s$abort_transaction

Purpose

This subroutine aborts the current transaction of the calling task or process.

Usage

```
declare error_code                 binary(15);

declare s$abort_transaction entry( binary(15));

         call s$abort_transaction( error_code);
```

Arguments

▶ `error_code` (output)
A returned error code.

Explanation

The subroutine `s$abort_transaction` aborts the current transaction of the calling task or process. The operating system does none of the postponed output operations requested during the transaction. There will be no trace of the transaction in any transaction protected file involved in the transaction.

Error Codes

The following is an error code this subroutine might return.

`e$tp_no_tid`
`(2872)`
	There is no transaction in progress. The task or process has no current transaction.

Related Information

`s$commit_transaction`
Commits the current transaction of the calling task or process.

`s$set_transaction_file`
Turns transaction protection on or off for a specified file.

`s$start_priority_transaction`
: Starts a transaction with priority different from the default priority given by OpenVOS when `s$start_transaction` is used.

`s$start_transaction`
: Starts a transaction for the calling task or process.

s$add_task_epilogue_handler

Purpose

This subroutine adds a routine to a list of epilogue handlers for the current task. When the task stops, OpenVOS runs the epilogue handlers.

Usage

```
declare epilogue_handler                   entry;
declare error_code                         binary(15);

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

         call s$add_task_epilogue_handler( epilogue_handler,
                                           error_code);
```

Arguments

- epilogue_handler (input)

  The entry value of an external task epilogue handler.

  The subroutine s$add_task_epilogue_handler adds epilogue_handler to the list of epilogue handlers for the current task.

- error_code (output)

  A returned error code.

Explanation

A *task epilogue handler* is an external routine that executes the next time the task stops (enters the stopped state).

The subroutine s$add_task_epilogue_handler enables a task to specify an epilogue routine to clean up after itself, for example, closing files it has opened. A task can have as many as 10 task epilogue handlers at one time.

The entry epilogue_handler cannot have parameters.

You can obtain entry values from several sources:

- entry statements
- external entry constants
- external entry variables
- entry parameters
- calls to `s$find_entry`.

The subroutine `s$find_entry`, which returns the entry value corresponding to an entry point name, is documented in the *OpenVOS PL/I Subroutines Manual* (R005).

Entry values cannot be shared among tasks. An entry value consists of three components: a display pointer, a code pointer, and a static pointer. The static pointer points to the static region, which is different for each task. Consequently, each task must obtain separately the entry value for a particular routine.

The subroutine `s$add_task_epilogue_handler` does not add a routine that is already in the list of epilogue handlers for the current task, but it is not an error to attempt this.

Error Codes

The following is an error code this subroutine might return.

`e$too_many_epilogue_handlers`
`(1126)`
	Attempt to define too many epilogue handlers. You have attempted to establish more than 10 task epilogue handlers.

Related Information

`s$add_task_epilogue_handler`
Adds a routine to a list of epilogue handlers for the current task. When the task is stopped, OpenVOS runs the epilogue handlers. This routine may only be called from a program written in C, COBOL, FORTRAN, Pascal, or PL/I.

`s$cleanup_task`
Detaches the terminal assigned to a specified task in the current process.

`s$control_task`
Performs one of a variety of functions on a specified task, depending on the value of `action_code` given.

`s$create_task`
Creates a dynamic task.

`s$delete_task`
Deletes a dynamic task.

`s$delete_task_epilogue_handler`

Deletes a specified entry from the list of task epilogue routines that the current task had previously established with calls to the subroutine `s$add_task_epilogue_handler`. This routine may only be called from a program written in C, COBOL, FORTRAN, Pascal, or PL/I.

`s$enable_tasking`

Enables or disables tasking in the calling process.

s$call_server

Purpose

This subroutine is used by a requester to put a message in a two-way queue and to receive a reply from the queue.

Usage

```
declare port_id              binary(15);
declare msg_priority         binary(15);
declare msg_subject          char(32) varying;
declare msg_length_in        binary(31);
declare msg                  --several-types--;
declare reply_length_in      binary(31);
declare reply_length_out     binary(31);
declare reply                --several-types--;
declare error_code           binary(15);

declare s$call_server entry( binary(15),
                             binary(15),
                             char(32) varying,
                             binary(31),
                             --several-types--,
                             binary(31),
                             binary(31),
                             --several-types--,
                             binary(15));
```

(Continued on next page)

(Continued)

```
call s$call_server( port_id,
                    msg_priority,
                    msg_subject,
                    msg_length_in,
                    msg,
                    reply_length_in,
                    reply_length_out,
                    reply,
                    error_code);
```

Arguments

▶ `port_id` (input)

The identifier of a port attached to the queue. `s$call_server` puts a message into the queue.

▶ `msg_priority` (input)

The priority of the message. The value must be between 0 and 19 inclusive.

▶ `msg_subject` (input)

The subject of the message. This value is put into the message header.

▶ `msg_length_in` (input)

The length of the message, in bytes. This value cannot be greater than the length of the input buffer `msg`. The maximum length of a message is 2^{23} bytes for a queue on the same module as the calling process, 2^{20} bytes for a queue on a different module, or the value of `maximum_msg_size` specified to `s$msg_open_direct` for a direct queue.

▶ `msg` (input)

The input buffer containing the message to be put into the queue. The length of `msg`, in bytes, must be at least `msg_length_in`.

▶ `reply_length_in` (input)

The length of the output buffer `reply`, in bytes.

▶ `reply_length_out` (output)

The actual length of the reply that the subroutine puts into `reply`.

▶ `reply` (output)

The output buffer containing the reply to the message. The maximum length of a reply is 2^{23} bytes for a queue on the same module as the requester process, 2^{20} bytes for a queue on a different module, or the value of `maximum_msg_size` specified to `s$msg_open_direct` for a direct queue.

▶ `error_code` (output)

A returned error code.

Explanation

For both DIRECT and SERVER queues:

The s$call_server subroutine combines the actions of s$msg_send and s$msg_receive_reply. It puts a message in a two-way queue and returns to the caller with the reply.

Only a requester can call this subroutine. The queue connected to port_id must be a two-way queue.

The msg_priority argument tells the server or servers at a queue the priority of this request. 0 is the lowest priority and 19 is the highest. Thus, for example, all messages with priority 8 will be placed in the queue ahead of all messages with priority 7, regardless of their times of arrival at the queue.

If the port port_id is in no-wait mode, then the subroutine returns immediately, and if the reply had not yet been received at the queue, it returns the error code e$caller_must_wait (1277).

If the port is in wait mode, then s$call_server does not return until the server has replied to the message.

If the port has a time limit set on it, the subroutine waits for the time limit, tries again, and if it does not succeed, returns the error code e$timeout (1081).

If a server at the queue receives a request but aborts or is stopped while servicing the request, s$call_server returns the error code e$server_aborted (2759). You can immediately resubmit the request, and, if another server is active at the queue, it can service the request. (If the transaction is not protected, however, the aborted server may have already performed part of the request.)

For SERVER queues only:

If msg_priority is between 0 and 9, and if no server is active at the queue, s$call_server waits at the queue if the port is in wait mode. If the port is in no-wait mode, s$call_server returns the error code e$caller_must_wait. s$call_server should not be used in no-wait mode; s$msg_send and s$msg_receive_reply should be called directly.

If msg_priority is between 10 and 19 and if no server is active at the queue or if all servers at the queue are stopped before any can receive the request by taking it out of the queue, s$call_server returns the error code e$no_msg_server_for_queue (2817).

If reply is longer than reply_length_in, then s$call_server:

- returns as much of reply as fits in the space allocated
- returns the true length of reply in reply_length_out
- returns the error code e$long_record (1026)
- removes reply from the queue; therefore, the caller will not be able to get the rest of reply.

For DIRECT queues only:

If no server is active at the queue or if all servers at the queue are stopped before any can receive the request by taking it out of the queue, `s$call_server` returns the error code `e$no_msg_server_for_queue` (`2817`).

If you try to call `s$call_server` and a reply to a previous message has not yet been returned, the subroutine returns the error code `e$invalid_io_operation` (`1040`).

The original message is removed when the server receives it. No other message may be put into the queue until the reply has been received.

When the subroutine receives a reply, it removes `reply` from the queue.

If `reply` is longer than `reply_length_in`, then `s$call_server`:

- returns none of `reply`
- returns the true length of `reply` in `reply_length_out`
- returns the error code `e$buffer_too_small` (`1133`).

This problem can be avoided, because the maximum length of messages was specified when the queue was opened. There is an absolute maximum of 3072 bytes.

Error Codes

The following lists some error codes this subroutine might return.

`e$long_record` (`1026`)	Record is too long. For a server queue, the reply was longer than `reply_length_in`.
`e$invalid_io_operation` (`1040`)	Invalid I/O operation for current port state or attachment. `s$call_server` was called for a direct queue when the reply to the previous message had not yet been returned.
`e$timeout` (`1081`)	Timeout period has expired. `s$call_server` failed a second time to receive a reply, after waiting the length of the time limit for port `port_id`.
`e$buffer_too_small` (`1133`)	The specified buffer is too small. For a direct queue, the reply was longer than `reply_length_in`.
`e$caller_must_wait` (`1277`)	Caller must wait for operation to complete. The port `port_id` is in no-wait mode and, for either server or direct queues, the reply has not been received at the queue. Or, for server queues only, no server is active at the queue.
`e$task_wrong_state` (`2576`)	Task is in wrong state to perform operation. The specified task is not `initialized` or `stopped`.
`e$server_aborted` (`2759`)	Server closed queue file without replying to message. The server aborted or was stopped while servicing a request.

`e$no_msg_server_for_queue`
`2817`
	There is no message server for queue. `msg_priority` was between 10 and 19, or the queue is a direct queue; no server is active at the queue or all servers were stopped before any could receive the message.

Related Information

`s$call_server`
: Is used by a requester to put a message into a two-way queue and to receive a reply from the queue.

`s$msg_cancel_receive`
: Can be used by a server to cancel the receipt of a message contained in a two-way server queue, or by a server or a requester to cancel the receipt of a message contained in a message queue.

`s$msg_delete`
: Can be used by a requester or a server to delete a message from a message queue.

`s$msg_open`
: Is used by a process to open (to connect to) a server queue, a message queue, or a one-way server queue.

`s$msg_open_direct`
: Is used by a process to open (to connect to) a direct queue or a one-way direct queue.

`s$msg_read`
: Can be used by a requester or a server to read a message contained in a queue.

`s$msg_receive`
: Can be used by a requester or a server to receive a message contained in a queue.

`s$msg_receive_reply`
: Can be used by a requester to receive a reply from a two-way server queue or a two-way direct queue.

`s$msg_rewrite`
: Can be used by a requester or a server to rewrite a message contained in a message queue.

`s$msg_send`
: Can be used by a `send_only_requester` to put a message into any one-way queue (message, server, or direct) or by a two-way requester to put a message into any two-way queue.

`s$msg_send_reply`
: Can be used by a server to put a reply into a two-way server queue or a two-way direct queue.

`s$truncate_queue`
: Can be used by a server to truncate an empty message queue.

s$cleanup_task

Purpose

This subroutine closes the terminal assigned to a specified task in the current process and detaches the port attached to the terminal.

Usage

```
declare task_id               binary(15);
declare error_code            binary(15);

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

         call s$cleanup_task( task_id,
                              error_code);
```

Arguments

- `task_id` (input)
  Either the identifier of a task or the value 0. `s$cleanup_task` cleans up the indicated task.

- `error_code` (output)
  A returned error code.

Explanation

The subroutine `s$cleanup_task` aborts the current operation on the task's terminal, closes and detaches the port attached to the terminal, and makes the task `uninitialized` (state 0).

If `task_id` is 0, then `s$cleanup_task` cleans up the current task. Otherwise, `task_id` must be the valid identifier of a task in the current process.

The specified task must be `stopped` (state 7) or `initialized` (state 1) when this subroutine is called.

Note that `s$cleanup_task` does not free any storage that may have been allocated by the task. You must do that explicitly before calling `s$cleanup_task`. Be sure that the storage is not being shared with any active tasks. Also, if you are using FMS, include an `accept clear;` statement to free any storage allocated by FMS.

Error Codes

The following is an error code this subroutine might return.

`e$task_wrong_state`
`(2576)`
	Task is in wrong state to perform operation. The task was not `stopped` (state 7) or `initialized` (state 1) when you called `s$cleanup_task`.

Related Information

`s$add_task_epilogue_handler`
Adds a routine to a list of epilogue handlers for the current task. When the task is stopped, OpenVOS runs the epilogue handlers. This routine may only be called from a program written in C, COBOL, FORTRAN, Pascal, or PL/I.

`s$cleanup_task`
Detaches the terminal assigned to a specified task in the current process.

`s$control_task`
Performs one of a variety of functions on a specified task, depending on the value of `action_code` given.

`s$create_task`
Creates a dynamic task.

`s$delete_task`
Deletes a dynamic task.

`s$delete_task_epilogue_handler` Deletes a specified entry from the list of task epilogue routines that the current task had previously established with calls to the subroutine `s$add_task_epilogue_handler`. This routine may only be called from a program written in C, COBOL, FORTRAN, Pascal, or PL/I.

`s$enable_tasking`
Enables or disables tasking in the calling process.

s$commit_transaction

Purpose

This subroutine commits the current transaction of the calling task or process.

Usage

```
declare error_code                  binary(15);

declare s$commit_transaction entry( binary(15));

         call s$commit_transaction( error_code);
```

Arguments

- ▶ `error_code` (output)
  A returned error code.

Explanation

`s$commit_transaction` commits the current transaction of the calling task or process.

If the calling task or process has no current transaction, `s$commit_transaction` returns the error code `e$tp_no_tid` (`2872`).

If the subroutine cannot commit the transaction for any other reason, it aborts the transaction and returns the error code `e$tp_aborted` (`2931`). See the `set_tp_default_parameters` command for an explanation of how conflicts between transactions are decided.

When `s$commit_transaction` returns a zero error code, OpenVOS guarantees that all affected transaction files will be up to date with respect to any I/O done within the transaction before any further access is possible. However, the updating process is not necessarily completed by the time `s$commit_transaction` returns.

Error Codes

The following lists some error codes this subroutine might return.

`e$tp_no_tid` (`2872`)	There is no transaction in progress. The calling task or process has no current transaction.
`e$tp_aborted` (`2931`)	Transaction has been aborted. `s$commit_transaction` could not commit the transaction.

Related Information

`s$abort_transaction`
Aborts the current transaction of the calling task or process.

`s$set_transaction_file`
Turns transaction protection on or off for a specified file.

`s$start_priority_transaction`
Starts a transaction with a priority different from the default priority given by OpenVOS if `s$start_transaction` were used.

`s$start_transaction`
Starts a transaction for the calling task or process.

s$control_task

Purpose

This subroutine stops, pauses, or continues the execution of one or more tasks in the current process. It can also enable or disable metering of CPU time and page faults on a per task basis.

Usage

```
declare task_id               binary(15);
declare action_code           binary(15);
declare error_code            binary(15);

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

         call s$control_task( task_id,
                              action_code,
                              error_code);
```

Arguments

- `task_id` (input)

  The identifier of a task in the current process, or the value -1, or the value 0. `s$control_task` acts on the indicated task or tasks. See the Explanation.

- `action_code` (input)

  A numerical code for the action to be taken.

- `error_code` (output)

  A returned error code

Explanation

`s$control_task` performs the action on the task indicated by `task_id`.

If `task_id` is -1, then `s$control_task` acts on all the tasks in the current process. If `task_id` is 0, then `s$control_task` acts on the current task. Otherwise, `task_id` must be the valid identifier of a task in the current process.

The argument `action_code` specifies the action. Table 3-1 lists the code for each action and the result of each action.

Table 3-1. Action Codes *(Page 1 of 2)*

Code	Action	Old State	New State
1	stop_task	uninitialized	uninitialized
		initialized	initialized
		ready	stopped
		running	stopped
		waiting	stopped
		paused_waiting	stopped
		paused_ready	stopped
		stopped	stopped
2	stop_task_return	uninitialized	uninitialized
		initialized	initialized
		ready	stopped
		running	stopped
		waiting	stopped
		paused_waiting	stopped
		paused_ready	stopped
		stopped	stopped
3	pause_task	uninitialized	(error)
		initialized	(error)
		ready	paused_ready
		running	paused_ready
		waiting	paused_waiting
		paused_waiting	paused_waiting
		paused_ready	paused_ready
		stopped	(error)
4	continue_task	uninitialized	(error)
		initialized	(error)
		ready	ready
		running	running
		waiting	waiting
		paused_waiting	waiting
		paused_ready	ready
		stopped	(error)
5	enable_metering	uninitialized	uninitialized
		initialized	initialized
		ready	ready
		running	running
		waiting	waiting
		paused_waiting	paused_waiting
		paused_ready	paused_ready
		stopped	stopped

Table 3-1. Action Codes *(Page 2 of 2)*

Code	Action	Old State	New State
6	disable_metering	uninitialized initialized ready running waiting paused_waiting paused_ready stopped	uninitialized initialized ready running waiting paused_waiting paused_ready stopped
7	stop_task_cleanup	uninitialized initialized ready running waiting paused_waiting paused_ready stopped	uninitialized uninitialized uninitialized uninitialized uninitialized uninitialized uninitialized uninitialized

`stop_task_return` was designed specifically to allow a task to move itself from one terminal to another. However, this capability is now provided by the subroutine `s$set_task_terminal`.

> **Caution:** Use of `stop_task_return` in new applications is not recommended. It is incompatible with dynamic tasks, setting task priorities and other new tasking capabilities.

If you call `s$control_task` with an `action_code` value of `1` (`stop_task`) to stop a running task, the subroutine sets the task to call `stop_task` on itself. When the task resumes execution, it executes `stop_task`, which then does the following:

- If there are any task epilogue handlers, the task runs these. The epilogue handlers may do I/O which makes the task wait.
- The task closes its terminal. This, too, may make the task wait.

The task stops only after these actions have finished.

However, if you stop the running task with `stop_task_return`, the task manager returns control to the running task and does not stop the task until it next gives up control. (The task runs until it makes a call at which the task manager normally switches tasks.)

Any task which calls `s$control_task(stop_task_return)` and subsequently calls either `s$sleep` or `s$wait_event` will cause the process to suspend execution rather than the calling task.

If no task is ready or waiting when the running task stops or pauses, then the task manager stops the program by calling `s$stop_program`.

Metering measures CPU time and page faults for individual tasks. If you enable metering, you can use `s$get_task_info` at any time to get the current totals. Metering is enabled or disabled for all tasks at once. Therefore, `task_id` can be any valid number when you enable or disable metering.

`stop_task_cleanup`, in addition to stopping the task, aborts any work in progress on the task's terminal, and closes and detaches the terminal port.

Error Codes

The following lists some error codes this subroutine might return.

`e$invalid_task_id`
`(2575)`
	Invalid task ID. The value specified for `task_id` does not correspond to a task that currently exists in the process.

`e$task_wrong_state`
`(2576)`
	Task is in wrong state to perform operation.

Related Information

`s$add_task_epilogue_handler`
: Adds a routine to a list of epilogue handlers for the current task. When the task is stopped, OpenVOS runs the epilogue handlers. This routine may only be called from a program written in C, COBOL, FORTRAN, Pascal, or PL/I.

`s$cleanup_task`
: Detaches the terminal assigned to a specified task in the current process.

`s$control_task`
: Performs one of a variety of functions on a specified task, depending on the value of `action_code` given.

`s$create_task`
: Creates a dynamic task.

`s$delete_task`
: Deletes a dynamic task.

`s$delete_task_epilogue_handler`
: Deletes a specified entry from the list of task epilogue routines that the current task had previously established with calls to the subroutine `s$add_task_epilogue_handler`. This routine may only be called from a program written in C, COBOL, FORTRAN, Pascal, or PL/I.

`s$enable_tasking`
: Enables or disables tasking in the calling process.

s$create_task

Purpose

This subroutine creates a dynamic task.

Usage

```
declare task_id               binary(15);
declare stack_size            binary(31);
declare fence_size            binary(31);
declare error_code            binary(15);

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

         call s$create_task( task_id,
                             stack_size,
                             fence_size,
                             error_code);
```

Arguments

- `task_id` (output)
  The task identifier assigned to the newly created task.

- `stack_size` (input)
  The number of bytes of storage to allocate for the stack in this task. If you specify less than the minimum required by OpenVOS (currently 1024), `s$create_task` uses the minimum.

- `fence_size` (input)
  The minimum number of bytes of storage to allocate for the fence following the stack in this task. `s$create_task` rounds `fence_size` up to the next page boundary (unit of 4096 bytes).

- `error_code` (output)
  A returned error code.

Explanation

The subroutine s$create_task creates a dynamic task. *Dynamic tasks* differ from tasks defined in a program module's binder control file in that:

- you can define and create them at run time, as needed
- you can delete them at run time, using s$delete_task, when they are no longer needed
- they have the added security of a fence.

The subroutine s$create_task allocates space in the user heap for the new task's static storage, stack, and fence, and places the task in the uninitialized state. You can then start the new task by calling s$init_task and s$start_task or s$start_task_full.

The *fence* is a region of storage following the stack frame. It allows OpenVOS to detect most references beyond the end of the stack without any corruption of data in adjacent regions.

Error Codes

The following lists some error codes this subroutine might return.

e$no_alloc_user_heap (3080)	No room in user heap for allocation. There is insufficient space in memory to allocate the static, stack, and fence for another task.
e$invalid_task_data_region (4011)	The task data region has been corrupted. This code is not returned in error_code but an error handler can access it via the oncode built-in function.
e$invalid_fence_size (4068)	The size of the fence must be zero or more bytes. You specified a negative value for fence_size.

Related Information

s$control_task

Performs one of a variety of functions on a specified task, depending on the value of action_code given.

s$delete_task

Deletes a dynamic task.

s$enable_tasking

Enables or disables tasking in the calling process.

s$init_task

Initializes a specified uninitialized task.

s$init_task_config

Reads a task configuration file, initializes the set of tasks described in the configuration file, and puts the tasks into the `ready` state. If a primary task (a task with identifier 1) is defined in the configuration file, then that task is given control.

`s$monitor`
Reads and carries out requests for the administration of a tasking process.

`s$monitor_full`
Like `s$monitor`, reads and carries out requests for the administration of a tasking process. It provides additional control, including allowing the caller to specify an initial request, with an option to return from the subroutine immediately after executing the request.

`s$reschedule_task`
Tells the task manager to dispatch another ready task.

`s$set_process_terminal`
Attaches a task's five predefined ports (`default_input`, `default_output`, `command_input`, `terminal_output`, and `terminal`), either to the process terminal or to the task's terminal.

`s$set_task_priority`
Sets the scheduling priority for a task.

`s$set_task_terminal`
Changes the terminal port attachment for the running task.

`s$start_task`
Makes an initialized or stopped task ready to run.

`s$start_task_full`
Makes an initialized or stopped task ready to run, passing it one or more initial arguments.

s$delete_task

Purpose

This subroutine deletes a dynamic task.

Usage

```
declare task_id              binary(15);
declare error_code           binary(15);

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

         call s$delete_task( task_id,
                             error_code);
```

Arguments

▶ `task_id` (input)
The task identifier of the task to be deleted.

▶ `error_code` (output)
A returned error code.

Explanation

The subroutine `s$delete_task` deletes a dynamic task, created by `create_task` or `s$create_task`. (A *static* task, created by the binder, cannot be deleted.) The task to be deleted must be in the `uninitialized` state.

The task monitor releases the memory allocated for the task's stack, fence, and internal static. It does **not**, however, free any other heap storage the task has allocated, nor does it detach or close any ports the task has attached or opened.

Error Codes

The following lists some error codes this subroutine might return.

`e$invalid_task_id`
`(2575)`
	Invalid task ID. The value specified for `task_id` does not correspond to a task that currently exists in the process.

`e$task_wrong_state`
`(2576)`
	Task is in wrong state to perform operation. The specified task is not `uninitialized`.

`e$task_not_dynamic`
`(4069)`
	This operation is valid only on dynamically created tasks. Because the task is not a dynamic task, it cannot be deleted with `s$delete_task`.

Related Information

`s$add_task_epilogue_handler`
Adds a routine to a list of epilogue handlers for the current task. When the task is stopped, OpenVOS runs the epilogue handlers. This routine may only be called from a program written in C, COBOL, FORTRAN, Pascal, or PL/I.

`s$cleanup_task`
Detaches the terminal assigned to a specified task in the current process.

`s$control_task`
Performs one of a variety of functions on a specified task, depending on the value of `action_code` given.

`s$create_task`
Creates a dynamic task.

`s$delete_task`
Deletes a dynamic task.

`s$delete_task_epilogue_handler`
Deletes a specified entry from the list of task epilogue routines that the current task had previously established with calls to the subroutine `s$add_task_epilogue_handler`. This routine may only be called from a program written in C, COBOL, FORTRAN, Pascal, or PL/I.

`s$enable_tasking`
Enables or disables tasking in the calling process.

s$delete_task_epilogue_handler

Purpose

This subroutine deletes a specified entry from the list of epilogue routines for the current task.

Usage

```
declare epilogue_handler                      entry;
declare error_code                            binary(15);

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

         call s$delete_task_epilogue_handler( epilogue_handler,
                                              error_code);
```

Arguments

▶ `epilogue_handler` (input)

An entry value in the list of epilogue handlers for the current task. The calling program must have previously established this handler by calling the subroutine `s$add_task_epilogue_handler`.

`s$delete_task_epilogue_handler` deletes the entry from the list.

▶ `error_code` (output)

A returned error code.

Explanation

The subroutine `s$delete_task_epilogue_handler` deletes the specified entry in the list of pending routines OpenVOS is to call when the task is next stopped. See the Explanation section of the subroutine `s$add_task_epilogue_handler`.

Related Information

`s$add_task_epilogue_handler`

Adds a routine to a list of epilogue handlers for the current task. When the task is stopped, OpenVOS runs the epilogue handlers. This routine may only be called from a program written in C, COBOL, FORTRAN, Pascal, or PL/I.

`s$cleanup_task`
: Detaches the terminal assigned to a specified task in the current process.

`s$control_task`
: Performs one of a variety of functions on a specified task, depending on the value of `action_code` given.

`s$create_task`
: Creates a dynamic task.

`s$delete_task`
: Deletes a dynamic task.

`s$delete_task_epilogue_handler`
: Deletes a specified entry from the list of task epilogue routines that the current task had previously established with calls to the subroutine `s$add_task_epilogue_handler`. This routine may only be called from a program written in C, COBOL, FORTRAN, Pascal, or PL/I.

`s$enable_tasking`
: Enables or disables tasking in the calling process.

s$enable_tasking

Purpose

This subroutine enables or disables tasking in the calling process.

Usage

```
declare enable_tasking_switch   binary(15);

declare s$enable_tasking entry( binary(15));

         call s$enable_tasking( enable_tasking_switch);
```

Arguments

- `enable_tasking_switch` (input-output)

  On input, a switch telling the subroutine whether to enable tasking in the calling process. If `enable_tasking_switch` is 1, `s$enable_tasking` enables tasking for the process. If `enable_tasking_switch` is 0, the subroutine disables tasking.

  On output, the value of the switch is the state of the process immediately before the call to the subroutine.

Explanation

The subroutine `s$enable_tasking` enables and disables tasking in the calling process.

This subroutine allows a task to disable task switching in its process, so that it can exclusively use a device that would otherwise be shared among a set of tasks. After disabling tasking and using the device, the caller should re-enable tasking so that other tasks can run.

This subroutine has an effect only when called in a process that has ports in `wait` mode. It does not change the modes of the ports, but it prevents task switching when the running task must wait for device I/O.

Related Information

`s$add_task_epilogue_handler`
Adds a routine to a list of epilogue handlers for the current task. When the task is stopped, OpenVOS runs the epilogue handlers. This routine may only be called from a program written in C, COBOL, FORTRAN, Pascal, or PL/I.

`s$cleanup_task`
Detaches the terminal assigned to a specified task in the current process.

`s$control_task`
Performs one of a variety of functions on a specified task, depending on the value of `action_code` given.

`s$create_task`
Creates a dynamic task.

`s$delete_task`
Deletes a dynamic task.

`s$delete_task_epilogue_handler`
Deletes a specified entry from the list of task epilogue routines that the current task had previously established with calls to the subroutine `s$add_task_epilogue_handler`. This routine may only be called from a program written in C, COBOL, FORTRAN, Pascal, or PL/I.

`s$enable_tasking`
Enables or disables tasking in the calling process.

s$get_default_lock_wait_time

Purpose

This subroutine returns the default maximum time (in units of 1/1024 of a second) that a task or process running on a given module will wait to acquire an implicit lock during any I/O operation.

Usage

```
declare module_name                         char(66) varying;
declare lock_wait_time                      binary(31);
declare error_code                          binary(15);

declare s$get_default_lock_wait_time entry( char(66) varying,
                                            binary(31),
                                            binary(15));

         call s$get_default_lock_wait_time( module_name,
                                            lock_wait_time,
                                            error_code);
```

Arguments

- `module_name` (input)

  The path name of the module. The default is the current module.

- `lock_wait_time` (output)

  The default lock wait time for the specified module, expressed in 1/1024 of a second.

- `error_code` (output)

  A returned error code.

Explanation

The subroutine `s$get_default_lock_wait_time` returns the default maximum time (in 1/1024 of a second) that a task or process will wait to acquire an implicit lock during an I/O operation. The default maximum wait time applies to all processes running on a given processing module or set of modules (unless the lock wait time for a process has been specifically changed from the default) and to all types of I/O. The default maximum wait time is set with the subroutine `s$set_default_lock_wait_time`.

Related Information

`s$get_default_lock_wait_time`
: Returns the default maximum time (in units of 1/1024 of a second) that a task or process will wait to acquire an implicit lock during any I/O operation. The default time applies to all tasks and processes running on the specified module (unless the lock wait time has been specifically changed from the default) and to all types of I/O.

`s$get_lock_wait_time`
: Returns the maximum time (in units of 1/1024 of a second) that the current program or process will wait to acquire an implicit lock during any I/O operation.

`s$set_default_lock_wait_time`
: Sets the default maximum time (in units of 1/1024 of a second) that a task or process will wait to acquire an implicit lock during any I/O operation. The default time applies to all tasks and processes running on the specified module (unless the lock wait time has been specifically changed from the default) and to all types of I/O.

`s$set_lock_wait_time`
: Sets the maximum time (in units of 1/1024 of a second) that the current program or process will wait to acquire an implicit lock during any I/O operation.

s$get_free_task_id

Purpose

This subroutine returns the task identifier of an available `uninitialized` task.

Usage

```
declare last_used_task_id         binary(15);
declare free_task_id              binary(15);
declare error_code                binary(15);

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

         call s$get_free_task_id( last_used_task_id,
                                  free_task_id,
                                  error_code);
```

Arguments

- `last_used_task_id` (input-output)

  On input, a task identifier or the value 0. On output, the identifier of the next `uninitialized` task.

- `free_task_id` (output)

  The identifier of the next `uninitialized` task.

- `error_code` (output)

  A returned error code.

Explanation

The subroutine `s$get_free_task_id` returns the identifier of an `uninitialized` task in the current program.

Starting at one more than `last_used_task_id`, `s$get_free_task_id` searches the list of allocated tasks in order of increasing identifiers for a task that is `uninitialized`. If it finds one, it returns the identifier in both `last_used_task_id` and `free_task_id`.

The first time you call `s$get_free_task_id`, you should set `last_used_task_id` to 0. On each subsequent call, you should use the value `s$get_free_task_id` returned in the previous call.

Error Codes

The following is an error code this subroutine might return.

`e$no_more_free_tasks`
`(2863)`
	No more free tasks. No allocated but `uninitialized` task exists whose identifier is greater than `last_used_task_id`.

Related Information

`s$get_free_task_id`
Returns the task identifier of an available `uninitialized` task.

`s$get_max_task_id`
Returns the maximum task identifier among tasks currently in use or `uninitialized` and available for use.

`s$get_task_id`
Returns the identifier of the calling task.

`s$get_task_info`
Returns information about a specified task in the current process.

s$get_lock_wait_time

Purpose

This subroutine returns the maximum time (in units of 1/1024 of a second) that the current program or process will wait to acquire an implicit lock during any I/O operation.

Usage

```
declare program_or_process          binary(15);
declare lock_wait_time              binary(31);
declare error_code                  binary(15);

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

         call s$get_lock_wait_time( program_or_process,
                                    lock_wait_time,
                                    error_code);
```

Arguments

▶ `program_or_process` (input)
The possible values follow:

`0`	get lock wait time for program
`1`	get lock wait time for process.

▶ `lock_wait_time` (output)
The time, in 1/1024 of a second, that the program or process will wait for a lock before giving up.

▶ `error_code` (output)
A returned error code.

Explanation

The subroutine `s$get_lock_wait_time` returns the maximum lock wait time (in units of 1/1024 of a second) for the current program or process (and any tasks that are part of the current program or process).

Lock wait time is the amount of time a process or task will wait for an implicit lock on a file, record, or key. If the process or task has to wait longer, then it gives up and returns one of the following error codes depending on the type of lock sought:

`e$file_in_use`	(1084)
`e$record_in_use`	(2408)
`e$key_in_use`	(2918).

The default module lock wait time set by OpenVOS is 0 seconds. Issuing the command `set_process_lock_wait_time` or calling `s$set_lock_wait_time` allows you to reset the lock wait time for the current program or process. To change the lock wait time for the module, issue the command `set_lock_wait_time` or call the subroutine `s$set_default_lock_wait_time`. 10 seconds is a typical lock wait time. A task uses the program lock wait time if one has been set. Otherwise, it looks for a process lock wait time, and if that has not been set it uses the module lock wait time.

The default maximum wait time applies to all processes running on a given module or set of modules (unless the lock wait time for a process has been specifically changed from the default).

Related Information

`s$get_default_lock_wait_time`

Returns the default maximum time (in units of 1/1024 of a second) that a task or process will wait to acquire an implicit lock during any I/O operation. The default time applies to all tasks and processes running on the specified module (unless the lock wait time has been specifically changed from the default) and to all types of I/O.

`s$get_lock_wait_time`

Returns the maximum time (in units of 1/1024 of a second) that the current program or process will wait to acquire an implicit lock during any I/O operation.

`s$set_default_lock_wait_time`

Sets the default maximum time (in units of 1/1024 of a second) that a task or process will wait to acquire an implicit lock during any I/O operation. The default time applies to all tasks and processes running on the specified module (unless the lock wait time has been specifically changed from the default) and to all types of I/O.

`s$set_lock_wait_time`

Sets the maximum time (in units of 1/1024 of a second) that the current program or process will wait to acquire an implicit lock during any I/O operation.

s$get_max_task_id

Purpose

This subroutine returns the maximum task identifier among tasks currently in use or `uninitialized` and available for use.

Usage

```
declare max_task_id              binary(15);

declare s$get_max_task_id entry( binary(15));

         call s$get_max_task_id( max_task_id);
```

Arguments

▶ `max_task_id` (output)
The maximum task identifier of the tasks in the current program.

Explanation

The subroutine `s$get_max_task_id` returns the maximum task identifier among tasks currently in use or `uninitialized` and available for use. The sequence of task identifiers starts at 1, the identifier of the primary task.

If the program has not created dynamic tasks with the `create_task` request or the subroutine `s$create_task`, the maximum task identifier is also the maximum number of tasks that the program can run. This number is specified by the `number_of_tasks:` directive in the binder control file for the program.

However, if the program **has** created dynamic tasks, the maximum number of tasks that the program can run may be less than the maximum task identifier. This is because one or more dynamic tasks with smaller task identifiers may have been deleted with the `delete_task` request or the subroutine `s$delete_task`.

Related Information

`s$get_free_task_id`
Returns the task identifier of an available `uninitialized` task.

`s$get_max_task_id`
Returns the maximum task identifier among tasks currently in use or `uninitialized` and available for use.

`s$get_task_id`
Returns the identifier of the calling task.

`s$get_task_info`
Returns information about a specified task in the current process.

s$get_server_queue_info

Purpose

The `s$get_server_queue_info` subroutine returns information about the number of messages in an open one-way or two-way server queue attached to a specified port.

Usage

```
declare port_id                     fixed bin (15);

declare 1 queue_info                longmap,
     2 version                      fixed bin(15),
     2 file_organization            fixed bin(15),
     2 num_messages                 fixed bin(31),
     2 num_non_busy_messages        fixed bin(31),
     2 highest_num_messages         fixed bin(31),
     2 total_num_messages           fixed dec(18),
     2 max_messages                 fixed bin(31);

declare error_code                  fixed bin(15);

declare s$get_server_queue_info entry (fixed bin(15),
                                    1 like queue_info,
                                    fixed bin(15));

call s$get_server_queue_info (port_id, queue_info, error_code);
```

Arguments

▶ `port_id (input)`

The identifier of a port attached to a one-way or two-way server queue.

▶ `queue_info (input-output)`

Information about the number of messages in the queue.

▶ `version (input)`

The version number of the `queue_info` structure. Assign it the value `1`. This is the only input argument in `queue_info`.

- ▶ `file_organization (output)`

  The organization of the file. Possible values are shown below.

Value	**Organization**
`5`	two-way server queue
`31`	one-way server queue

- ▶ `num_messages (output)`

  The number of messages currently in the queue.

  For one-way server queues, this is the number of messages which have not been received by the receiver.

  For two-way server queues, this is the number of messages for which replies have not been received by the requester.

  In both cases, `num_messages` is reset to zero when all openers of the queue have closed their attached ports.

- ▶ `num_non_busy_messages (output)`

  The number of messages that have been sent and not yet received.

  For one-way server queues, once a message has been received, it is no longer in the queue. When a message is received, `num_non_busy_messages` is decremented.

  For two-way server queues, a message can be in the queue and can be nonbusy. When a message is received, `num_non_busy_messages` is decremented. If `s$msg_cancel_receive` is called, `num_non_busy_messages` is incremented.

- ▶ `highest_num_messages (output)`

  The highest number of messages that have been in the queue at one time. This is the highest value that `num_messages` has reached. If the value of `num_messages` is reduced, the value of `highest_num_messages` does not change.

- ▶ `total_num_messages (output)`

  The total number of messages that have been sent during the opening of the queue.

- ▶ `max_messages (output)`

  The maximum number of messages that can be sent to this queue. This is the same as `max_queue_depth`. See additional information in the descriptions of the `set_max_queue_depth` command and the `s$set_max_queue_depth` subroutine.

- ▶ `error_code (output)`

  A returned error code.

Explanation

The `s$get_server_queue_info` subroutine returns information about the number of messages in an open one-way or two-way server queue attached to a specified port. The attached port is identified by the `port_id` argument. The `queue_info` data structure returns information about this identified port.

Error Codes

The following lists some error codes this subroutine might return.

`e$portname_not_found` (`1028`)	The specified port has not been attached.
`e$invalid_io_operation` (`1040`)	You are attempting to get server queue information via a port that is not attached to an open one-way or two-way server queue.
`e$wrong_version` (`1083`)	The caller specified the wrong version number in `queue_info.version`. The only correct number is `1`.

s$get_task_id

Purpose

This subroutine returns the identifier of the calling task.

Usage

```
declare task_id               binary(15);

declare s$get_task_id entry( binary(15));

         call s$get_task_id( task_id);
```

Arguments

► `task_id` (output)
The identifier of the calling task.

Explanation

The subroutine `s$get_task_id` returns the identifier of the calling task.

Related Information

`s$get_free_task_id`
Returns the task identifier of an available `uninitialized` task.

`s$get_max_task_id`
Returns the maximum task identifier among tasks currently in use or `uninitialized` and available for use.

`s$get_task_id`
Returns the identifier of the calling task.

`s$get_task_info`
Returns information about a specified task in the current process.

s$get_task_info

Purpose

This subroutine returns information about a specified task in the current process.

Usage

```
declare task_id                   binary(15);

declare 1 get_task_info           shortmap,
          2 version               binary(15),
          2 info_task_id          binary(15),
          2 reserved1             binary(31),
          2 task_stack_length     binary(31),
          2 reserved2             binary(31),
          2 task_static_length    binary(31),
          2 task_terminal_port_id binary(15),
          2 task_state            binary(15),
          2 reserved3             binary(15),
          2 task_cpu_time         decimal(15),
          2 task_page_faults      binary(31),
          2 reserved4             (2) binary(31),
          2 task_priority         binary(15),
          2 task_fence_length     binary(15),
          2 reserved5             binary(31);

declare error_code                binary(15);

declare s$get_task_info entry(    binary(15),
                                  1 like get_task_info,
                                  binary(15));

         call s$get_task_info(    task_id,
                                  get_task_info,
                                  error_code);
```

Arguments

- ▶ `task_id` (input)

  The identifier of a task. `s$get_task_info` returns information about the task.

- ▶ `get_task_info` (input-output)

  Information about the task.

- `version` (input)
  The version number of the `get_task_info` structure. You must assign the value 3 to this argument. It is the only input argument in the structure `get_task_info`.

- `info_task_id` (output)
  The identifier of the given task; it will have the same value as `task_id`, unless `task_id` is 0, in which case `info_task_id` will be the identifier of the current task.

- `reserved1` (output)
  Reserved for use by OpenVOS.

- `task_stack_length` (output)
  The length in bytes of the task's stack.

- `reserved2` (output)
  Reserved for use by OpenVOS.

- `task_static_length` (output)
  The length in bytes of the task's internal static region.

- `task_terminal_port_id` (output)
  The identifier of the port to which the task's terminal is attached.

- `task_state` (output)
  The current state of the task. See the Explanation.

- `reserved3` (output)
  Reserved for future use.

- `task_cpu_time` (output)
  The accumulated time, exclusive of page faults, that the CPU has spent running the task. The units are 1/65536 of a second (jiffies).

- `task_page_faults` (output)
  The number of page faults that the task has taken since it started.

- `reserved4` (output)
  Reserved for use by OpenVOS.

- `task_priority` (output)
  The current priority of the task.

- `task_fence_length` (output)
  The size in **pages** (units of 4096 bytes) of the fence for the task. The size of the fence for static tasks is always zero.

- `reserved5` (output)
  Reserved for use by OpenVOS.

- `error_code` (output)
  A returned error code.

Explanation

The subroutine s$get_task_info returns information about the task task_id.

If task_id is 0, s$get_task_info returns information about the caller (the running task). Otherwise, task_id must be the valid identifier of a task in the current process.

Table 3-2 shows the values of task_state and their codes.

Table 3-2. task_state Values

Task State	Code
uninitialized	0
initialized	1
ready	2
running	3
waiting	4
paused_waiting	5
paused_ready	6
stopped	7

Error Codes

The following is an error code this subroutine might return.

e$invalid_task_id (2575)	Invalid task ID. The value specified for task_id does not correspond to a task that currently exists in the process.

Related Information

s$get_free_task_id
Returns the task identifier of an available uninitialized task.

s$get_max_task_id
Returns the maximum task identifier among tasks currently in use or uninitialized and available for use.

s$get_task_id
Returns the identifier of the calling task.

s$get_task_info
Returns information about a specified task in the current process.

s$get_tp_abort_code

Purpose

The `s$get_tp_abort_code` subroutine returns information about why the most recent transaction was aborted.

Usage

```
declare abort_reason                 binary(15);
declare error_code                   binary(15);

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

           call s$get_tp_abort_code (abort_reason, error_code);
```

Arguments

▶ `abort_reason` (output)

Returns `0` if the most recent transaction was not aborted. Returns one of the following values if the most recent transaction was aborted for any reason:

- `e$user_abort` (`7418`): User initiated an abort.
- `e$lock_conflict_abort` (`7419`): Abort due to a lock conflict.
- `e$in_use_abort` (`7420`): Attempt by two processes to access the same transaction.
- `e$memory_abort` (`7421`): Ran out of memory.
- `e$cant_resv_disk_blks_abort` (`7422`): Unable to reserve disk blocks.
- `e$off_module_abort` (`7423`): Abort by a remote module.
- `e$phase1_off_module_abort` (`7424`): Abort by a remote module during phase 1.
- `e$phase2_off_module_abort` (`7425`): Abort by a remote module before phase 2 and after phase 1.
- `e$process_terminate_abort` (`7426`): Process is terminating.
- `e$module_down_abort` (`7427`): Remote module is down.

- e$no_tpo_on_remote_abort (7428): There is no TPOverseer on the remote module.
- e$error_setting_tid_abort (7429): Error occurred while setting a transaction ID.
- e$error_writing_to_log_abort (7430): Error occurred while writing to a transaction log.
- e$error_setting_recd_lck_abort (7431): Error while setting a TP record lock.
- e$phase1_off_mixed_abort (7432): Global abort by remote module during phase 1.
- e$phase2_off_mixed_abort (7433): Global abort by remote module before phase 2 and after phase 1.

▶ error_code (output)
Returns 0 if the call succeeds. Returns the error message e$out_of_range (1038) if an unknown error code is stored as the value of abort_reason.

Explanation

If a subroutine returns the error message e$tp_aborted (2931), call the s$get_tp_abort_code subroutine to determine the reason why the transaction was aborted. Typically, transactions are aborted by the TPOverseer or the user.

Access Requirements

None.

s$get_tp_default_parameters

Purpose

This subroutine returns the default lock contention parameters for transaction locking on a specified module.

Usage

```
declare module_name                        char(66) varying;

declare 1 parameter_info                   shortmap,
          2 version                        binary(15),
          2 transaction_priority           char(1),
          2 switches                       char(1),
          2 time_value                     binary(15);

declare error_code                         binary(15);

declare s$get_tp_default_parameters entry( char(66) varying,
                                           1 like parameter_info,
                                           binary(15));

         call s$get_tp_default_parameters( module_name,
                                           parameter_info,
                                           error_code);
```

Arguments

▶ `module_name` (input)
The name of the module for which you want the parameters to be returned.

▶ `parameter_info` (output)
Information about how the parameters are set.

▶ `version` (output)
The version number of the `parameter_info` structure. This is always set to 1.

▶ `transaction_priority` (output)
The default priority given to a transaction by `s$start_transaction`. `s$get_tp_default_parameters` returns `transaction_priority` as a hexadecimal value. The value can be 0 up to 9.

- ▶ `switches` (output)
  Information about how the lock contention parameters are set. See the Explanation for details.

- ▶ `time_value` (output)
  A number of seconds. If two transactions differ in the times that they started by less than `time_value`, then OpenVOS considers them to have started at the same time, and therefore neither is considered younger or older than the other.

- ▶ `error_code` (output)
  A returned error code.

Explanation

The subroutine `s$get_tp_default_parameters` returns information about the current default values assigned by OpenVOS to all transactions started on the specified module. You can override these values for a specific process or task with the command `set_tp_parameters` or the subroutine `s$set_tp_parameters`. You can also change the module default values from their OpenVOS defaults with the command `set_tp_default_parameters` or the subroutine `s$set_tp_default_parameters`. The parameters and their default values are:

```
transaction_priority   0
ignore_priority        false
ignore_time            false
younger_wins           false
allow_deadlocks        false
time_value             10 seconds.
```

The `ignore_priority`, `ignore_time`, `younger_wins`, and `allow_deadlocks` parameters are contained in the argument `switches`. The value of switches is a binary coding of four variables. The variables are coded in the “128” bit, the “64” bit, the “16” bit, and the “8” bit switches as shown in Table 3-4. In addition, three bits are reserved for future use.

Table 3-4. Value of `switches` Argument for `s$get_tp_default_parameters` *(Page 1 of 2)*

Bit	Switch Name	Explanation
128	`ignore_priority`	If the bit is set to `true` for both transactions contending for a lock, their priorities are ignored by OpenVOS in deciding which transaction wins.
64	`ignore_time`	If the bit is set to `true` for both transactions contending for a lock, then the time that each transaction began is ignored by OpenVOS in deciding which transaction wins.
32	---	The bit is reserved for use by OpenVOS.
16	`younger_wins`	If the bit is set to `true` for both transactions contending for a lock, then the one which began last wins.

Table 3-4. Value of `switches` Argument for `s$get_tp_default_parameters` *(Page 2 of 2)*

Bit	Switch Name	Explanation
8	`allow_deadlocks`	If the bit is set to `true` for both of two transactions contending for a lock, then OpenVOS ignores any deadlock occurring between them. If a deadlock occurs, OpenVOS does not abort either transaction; instead, it returns the error `e$record_in_use` (`2408`). When this happens, you must abort the transaction. If you do not do so, the deadlock could continue indefinitely, until one of the transactions is aborted. For this reason, the use of this option is not recommended. If `allow_deadlocks` is `false` (the default) for either transaction involved in a deadlock, OpenVOS breaks the deadlock by choosing a winner based on the value of an internal transaction identifier. In most cases the older transaction is the winner. However, if both transactions have `younger_wins` set to `true`, then the younger transaction is the winner. In either case, if one of the transactions has lost a deadlock immediately before, it wins this time.

See `s$get_tp_parameters` for information on how to decode the bit values in `switches`.

Related Information

`s$get_tp_default_parameters`
Returns the default lock contention parameters for transaction locking on a specified module.

`s$get_tp_parameters`
Returns the lock contention parameters for transaction locking for the current program or process.

`s$set_tp_default_parameters`
Sets the default lock contention parameters for transaction locking for a specified module.

`s$set_tp_parameters`
Sets the lock contention parameters for transaction locking for a specified program or process.

s$get_tp_parameters

Purpose

This subroutine returns the lock contention parameters for transaction locking for the current program or process.

Usage

```
declare program_or_process         binary(15);

declare 1 parameter_info           shortmap,
          2 version                binary(15),
          2 transaction_priority   char(1),
          2 switches               char(1),
          2 time_value             binary(15);

declare error_code                 binary(15);

declare s$get_tp_parameters entry( binary(15),
                                   1 like parameter_info,
                                   binary(15));

         call s$get_tp_parameters( program_or_process,
                                   parameter_info,
                                   error_code);
```

Arguments

- `program_or_process` (input)

  If set to 0, then the subroutine gets the parameters for the current program. If set to anything but 0, then the subroutine gets the parameters for the current process.

- `parameter_info` (output)

  Information about how the parameters are set.

- `version` (output)

  The version number of the `parameter_info` structure. This is always set to 1.

- `transaction_priority` (output)

  The default priority given to a transaction by `s$start_transaction`. `s$get_tp_parameters` returns `transaction_priority` as a hexadecimal value. The value can be from 0 to 9.

- `switches` (output)
  Information about how the lock contention parameters are set. See the Explanation section for details.

- `time_value` (output)
  A number of seconds. If two transactions differ in the times that they started by less than `time_value`, then OpenVOS considers them to have started at the same time, and therefore neither is considered younger or older than the other.

- `error_code` (output)
  A returned error code.

Explanation

The subroutine `s$get_tp_parameters` returns the current values assigned by OpenVOS to all transactions started by the current program or process. The parameters and their default values are:

```
transaction_priority   0
ignore_priority        false
ignore_time            false
younger_wins           false
allow_deadlocks        false
time_value             10 seconds.
```

The `ignore_priority`, `ignore_time`, `younger_wins`, and `allow_deadlocks` parameters are contained in the argument `switches`. The value of switches is a binary coding of four variables. The variables are coded in the “128” bit, the “64” bit, the “16” bit, and the “8” bit switches as shown in Table 3-5. In addition, three bits are reserved for future use.

Table 3-5. Value of `switches` Argument for `s$get_tp_parameters` *(Page 1 of 2)*

Bit	Switch Name	Explanation
128	`ignore_priority`	If the bit is set to `true` for both transactions contending for a lock, their priorities are ignored by OpenVOS in deciding which transaction wins.
64	`ignore_time`	If the bit is set to `true` for both transactions contending for a lock, then the time that each transaction began is ignored by OpenVOS in deciding which transaction wins.
32	---	The bit is reserved for use by OpenVOS.
16	`younger_wins`	If the bit is set to `true` for both transactions contending for a lock, then the one that began last wins.

Table 3-5. Value of `switches` Argument for `s$get_tp_parameters` *(Page 2 of 2)*

Bit	Switch Name	Explanation
8	`allow_deadlocks`	If the bit is set to `true` for both of two transactions contending for a lock, then OpenVOS ignores any deadlock occurring between them. If a deadlock occurs, OpenVOS does not abort either transaction; instead, it returns the error `e$record_in_use` (`2408`). When this happens, you must abort the transaction. If you do not do so, the deadlock could continue indefinitely, until one of the transactions is aborted. For this reason, the use of this option is not recommended. If `allow_deadlocks` is `false` (the default) for either transaction involved in a deadlock, OpenVOS breaks the deadlock by choosing a winner based on the value of an internal transaction identifier. In most cases the older transaction is the winner. However, if both transactions have `younger_wins` set to `true`, then the younger transaction is the winner. In either case, if one of the transactions has lost a deadlock immediately before, it wins this time.

The following program fragment illustrates a method for decoding the flag bit settings in `switches`.

```
%replace IGNORE_PRIORITY_BIT by 128;
%replace YOUNGER_WINS_BIT by 16;

declare switches_rank binary(15);
declare ignore_priority binary(15);     /* one if bit is on, zero if 
off */
declare younger_wins binary(15);        /* one if bit is on, zero if 
off */

call s$get_tp_parameters(program_or_process, parameter_info, 
error_code);

/* Find the returned values of 'ignore_priority' and 'younger_wins' 
*/

switches_rank=rank(parameter_info.switches);

ignore_priority=divide (mod (switches_rank,2*IGNORE_PRIORITY_BIT),
            IGNORE_PRIORITY_BIT, 15, 0);
younger_wins=divide (mod (switches_rank,2*YOUNGER_WINS_BIT),
            YOUNGER_WINS_BIT, 15, 0);
```

Related Information

`s$get_tp_default_parameters`
Returns the default lock contention parameters for transaction locking on a specified module.

`s$get_tp_parameters`
Returns the lock contention parameters for transaction locking for the current program or process.

`s$set_tp_default_parameters`
Sets the default lock contention parameters for transaction locking for a specified module.

`s$set_tp_parameters`
Sets the lock contention parameters for transaction locking for a specified program or process.

s$init_task

Purpose

This subroutine initializes a specified `uninitialized` task.

Usage

```
declare task_id            binary(15);
declare terminal_name      char(256) varying;
declare error_code         binary(15);

declare s$init_task entry( binary(15),
                           char(256) varying,
                           binary(15));

         call s$init_task( task_id,
                           terminal_name,
                           error_code);
```

Arguments

- `task_id` (input)
  The identifier of a task. `s$init_task` puts the task into the `initialized` state.

- `terminal_name` (input)
  The full path name of a terminal. `s$init_task` attaches the initialized task to the terminal.

- `error_code` (output)
  A returned error code.

Explanation

The subroutine `s$init_task` initializes the task `task_id` and attaches the task to the terminal `terminal_name`. `s$init_task` puts the terminal into `wait` mode. It enables `break_char` mode but does not enable the CTRL BREAK key sequence. See the description of `s$control` in the OpenVOS Subroutines manuals for instructions on enabling the CTRL BREAK sequence.

If `task_id` is 0, then `s$init_task` initializes the current task. Otherwise, `task_id` must be the valid identifier of a task in the current process.

The subroutine puts the task in the initialized state. Unless they were given initial values, all variables in the task's static region are undefined.

The argument terminal_name must be either the full path name of a terminal or the empty string. If you do not name a terminal, the task will not be connected to a terminal and attempts to do I/O to ports 1 through 5 (default_input, terminal_output, command_input, default_output, and terminal) will result in an error.

The following is a method used in many current applications to move a task from one terminal to another. It is reproduced here for reference purposes:

1. Call s$control_task from the current task, with stop_task_return as the value for action, and 0 as the value for task_id. This stops the current task. The task manager returns control to the current task after setting its state to stopped.

2. Call s$cleanup_task from the current task to clean up the current task (use the value 0 for task_id).

3. Call s$init_task, with task_id equal to 0.

However, use of the control action stop_task_return in new applications is **not recommended**. It is incompatible with dynamic tasks, setting task priorities and other new tasking capabilities. The subroutine s$set_task_terminal has been provided to move a task from one terminal to another. You can replace the sequence of calls described above with a single call to s$set_task_terminal.

Error Codes

The following lists some error codes this subroutine might return.

e$invalid_task_id (2575)	Invalid task ID. The value specified for task_id does not correspond to a task that currently exists in the process. Task is in wrong state to perform operation.
e$task_wrong_state (2576)	The specified task is not uninitialized.

Related Information

s$control_task
Performs one of a variety of functions on a specified task, depending on the value of action_code given.

s$delete_task
Deletes a dynamic task.

s$enable_tasking
Enables or disables tasking in the calling process.

s$init_task
Initializes a specified uninitialized task.

`s$init_task_config`
Reads a task configuration file, initializes the set of tasks described in the configuration file, and puts the tasks into the `ready` state. If a primary task (a task with identifier 1) is defined in the configuration file, then that task is given control.

`s$monitor`
Reads and carries out requests for the administration of a tasking process.

`s$monitor_full`
Like `s$monitor`, reads and carries out requests for the administration of a tasking process. It provides additional control, including allowing the caller to specify an initial request, with an option to return from the subroutine immediately after executing the request.

`s$reschedule_task`
Tells the task manager to dispatch another ready task.

`s$set_process_terminal`
Attaches a task's five predefined ports (`default_input`, `default_output`, `command_input`, `terminal_output`, and `terminal`), either to the process terminal or to the task's terminal.

`s$set_task_priority`
Sets the scheduling priority for a task.

`s$set_task_terminal`
Changes the terminal port attachment for the running task.

`s$start_task`
Makes an initialized or stopped task ready to run.

`s$start_task_full`
Makes an initialized or stopped task ready to run, passing it one or more initial arguments.

`s$init_task_config`

Purpose

This subroutine reads a task configuration file, initializes the set of tasks described in the configuration file, and puts the tasks into the `ready` state. If a primary task (a task with identifier 1) is defined in the configuration file, that task is given control.

Usage

```
declare config_path_name           char(256) varying;
declare call_debug_switch          binary(15);
declare error_code                 binary(15);

declare s$init_task_config entry(  char(256) varying,
                                   binary(15),
                                   binary(15));

         call s$init_task_config(  config_path_name,
                                   call_debug_switch,
                                   error_code);
```

Arguments

- `config_path_name` (input)
  The path name of a task configuration file. `s$init_task_config` reads the configuration file to obtain information on the tasks it configures.

- `call_debug_switch` (input)
  A switch indicating whether the primary task is to run under the control of the debugger.

- `error_code` (output)
  A returned error code.

Explanation

The subroutine `s$init_task_config` initializes and starts (puts into the `ready` state) a set of tasks as defined in the configuration file `config_path_name`. If you include instructions in the configuration file for initializing a primary task, then the subroutine starts running that task.

You can call this subroutine only from the primary task. You can initialize a new task numbered 1, however, by specifying in the configuration file the name of a procedure for the task to execute. In this case, the subroutine does not return to its caller, the old primary task, but starts running the new primary task at the specified entry point of the program. If you omit the definition of a primary task in the configuration file, then `s$init_task_config` returns normally to the calling task.

See "Task Configuration Files" under `initialize_configuration` in Chapter 2 for information on constructing a task configuration file.

Related Information

`s$control_task`
: Performs one of a variety of functions on a specified task, depending on the value of `action_code` given.

`s$delete_task`
: Deletes a dynamic task.

`s$enable_tasking`
: Enables or disables tasking in the calling process.

`s$init_task`
: Initializes a specified `uninitialized` task.

`s$init_task_config`
: Reads a task configuration file, initializes the set of tasks described in the configuration file, and puts the tasks into the `ready` state. If a primary task (a task with identifier 1) is defined in the configuration file, then that task is given control.

`s$monitor`
: Reads and carries out requests for the administration of a tasking process.

`s$monitor_full`
: Like `s$monitor`, reads and carries out requests for the administration of a tasking process. It provides additional control, including allowing the caller to specify an initial request, with an option to return from the subroutine immediately after executing the request.

`s$reschedule_task`
: Tells the task manager to dispatch another ready task.

`s$set_process_terminal`
: Attaches a task's five predefined ports (`default_input`, `default_output`, `command_input`, `terminal_output`, and `terminal`), either to the process terminal or to the task's terminal.

`s$set_task_priority`
: Sets the scheduling priority for a task.

`s$set_task_terminal`
: Changes the terminal port attachment for the running task.

`s$start_task`
Makes an initialized or stopped task ready to run.

`s$start_task_full`
Makes an initialized or stopped task ready to run, passing it one or more initial arguments.

s$monitor

Purpose

This subroutine reads and carries out requests for the administration of a tasking process.

Usage

```
declare caller_name              char(32) varying;
declare prompt_string            char(32) varying;
declare use_abbreviations_switch binary(15);
declare error_code               binary(15);

declare s$monitor entry(         char(32) varying,
                                 char(32) varying,
                                 binary(15),
                                 binary(15));

         call s$monitor(         caller_name,
                                 prompt_string,
                                 use_abbreviations_switch,
                                 error_code);
```

Arguments

- ▶ caller_name (input)

  The name of the calling program. s$monitor displays the name in error messages.

- ▶ prompt_string (input)

  A request prompt. s$monitor displays the prompt on the caller's terminal whenever it is ready to accept a request. If the prompt is the empty string, s$monitor does not visibly prompt for a request.

- ▶ use_abbreviations_switch (input)

  A switch telling the monitor process whether to expand abbreviations in requests and commands.

- ▶ error_code (output)

  A returned error code.

Explanation

The subroutine `s$monitor` reads and carries out requests for the administration of a tasking process. The subroutine is called from a primary task, making that task the monitor task for the program. An operator submits the requests at the terminal to which the primary task is attached. Once `s$monitor` is called, the primary task continues to accept and process requests until it receives a `quit` request, described below.

The subroutine reads requests from the `default_input` port.

The subroutine `s$monitor` processes two kinds of lines: those that begin with two periods and those that do not. If a line begins with two periods, they must be followed by OpenVOS internal commands. For example, the line issues an OpenVOS command to display the names of the processing modules in the current system, and then to list the files in the current directory.

```
..list_modules; list
```

One of the operating system internal commands that you can submit is:

```
..login
```

This starts a subprocess for you. You can now do anything you would do from command level. When you are done, simply type `logout` to return to running as the monitor.

Lines **not** beginning with two periods consist of requests for the administration of a tasking process.

You decide which requests can be called from the monitor by binding them in the program that runs in the monitor process. See the explanation of monitor requests at the beginning of Chapter 2 for further information on specifying requests.

Two requests, `help` and `quit`, are always available, since they are entry points in `s$monitor`. Issuing the `quit` request tells `s$monitor` to stop accepting requests and to return to its caller; it returns with an error code of 0. The `help` request lists the available requests. To see the list of OpenVOS internal commands, issue the request `..help`.

If `use_abbreviations_switch` is `true` (has the value 1), and if the process is using abbreviations, then `s$monitor` expands abbreviations using the current abbreviations file. If the switch is `false` (has the value 0), abbreviations are not expanded.

The terminal port of a process that calls `s$monitor` must be in `wait` mode. OpenVOS normally puts the port in this mode when the process is created.

Related Information

`s$control_task`
: Performs one of a variety of functions on a specified task, depending on the value of `action_code` given.

`s$delete_task`
: Deletes a dynamic task.

`s$enable_tasking`
Enables or disables tasking in the calling process.

`s$init_task`
Initializes a specified `uninitialized` task.

`s$init_task_config`
Reads a task configuration file, initializes the set of tasks described in the configuration file, and puts the tasks into the `ready` state. If a primary task (a task with identifier 1) is defined in the configuration file, then that task is given control.

`s$monitor_full`
Like `s$monitor`, reads and carries out requests for the administration of a tasking process. It provides additional control, including allowing the caller to specify an initial request, with an option to return from the subroutine immediately after executing the request.

`s$reschedule_task`
Tells the task manager to dispatch another ready task.

`s$set_process_terminal`
Attaches a task's five predefined ports (`default_input`, `default_output`, `command_input`, `terminal_output`, and `terminal`), either to the process terminal or to the task's terminal.

`s$set_task_priority`
Sets the scheduling priority for a task.

`s$set_task_terminal`
Changes the terminal port attachment for the running task.

`s$start_task`
Makes an initialized or stopped task ready to run.

`s$start_task_full`
Makes an initialized or stopped task ready to run, passing it one or more initial arguments.

s$monitor_full

Purpose

This subroutine, like s$monitor, reads and carries out requests for administering a tasking process. It provides additional control, including allowing the caller to specify an initial request, with an option to return from the subroutine immediately after executing the request.

Usage

```
declare 1 monitor_info                shortmap,
          2 version                   binary(15),
          2 caller_name               char(32) varying,
          2 prompt_string             char(32) varying,
          2 switches                  binary(15),
          2 initial_request_line      char(300) varying,
          2 num_req_restrictions      binary(15),
          2 request_restrictions      (N) char(32) varying,
          2 cleanup_entry_var         entry,
          2 unclaimed_input_entry_var entry(char(300) varying, binary(15));

declare error_code                    binary(15);

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

         call s$monitor_full(         monitor_info.version,
                                      error_code);
```

Arguments

▶ monitor_info (input)

The name of the structure containing information specifying the behavior of the subroutine.

▶ version (input)

The version number of the monitor_info structure. You must assign the value 3 to it.

▶ caller_name (input)

The name of the caller of the subroutine. The name is used in error messages.

▶ prompt_string (input)

The prompt that you want displayed when the monitor is waiting for further instructions.

- `switches` (input)
  A group of on-off switches allowing certain decisions: see the Explanation section for details.

- `initial_request_line` (input)
  May be any request line usable by the monitor task. `s$monitor_full` executes this line before turning control over to the user.

- `num_req_restrictions` (input)
  The number of request restrictions to be entered in the list in the argument `request_restrictions`.

- `request_restrictions` (input)
  A list of *N* entry point names which also appear in the `retain:` directive of the binder control file for the program. While retained entry points are normally available to the monitor as requests, you can make those that should not be treated as requests unavailable by listing them in `request_restrictions`.

- `cleanup_entry_var` (input)
  A pointer to an entry point in the executing program module. If the `cleanup_entry` switch is set to `true`, then `s$monitor_full` executes the routine that begins at the specified entry point after each call to a request.

- `unclaimed_input_entry_var` (input)
  A pointer to an entry point in the executing program module. If the `unclaimed_input_entry` switch is set to `true`, then `s$monitor_full` executes the routine that begins at the specified entry point whenever a monitor request line fails. `s$monitor_full` passes this routine two arguments:

  - the request line that failed (`char(300) varying`)

  - the error code that describes the reason for failure (`comp-4binary(15)`).

- `error_code` (output)
  A returned error code.

Explanation

The subroutine `s$monitor_full` works just like `s$monitor` with the following additions:

- the caller can specify an initial request line (`initial_request_line`)

- the subroutine can return immediately after executing the specified initial request (the switch `quit`)

- the caller can list entry points which may not be used by the monitor process (`request_restrictions`)

- the caller can specify a routine to be called after each request line completes (`cleanup_entry_var`)

- the caller can specify a routine to be called when a request line fails (`unclaimed_input_entry_var`).

You can obtain entry values from several sources:

- entry statements
- external entry constants
- external entry variables
- entry parameters
- calls to `s$find_entry`.

The subroutine `s$find_entry`, which returns the entry value corresponding to an entry point name, is documented in the *OpenVOS PL/I Subroutines Manual* (R005).

Entry values cannot be shared among tasks. An entry value consists of three components: a display pointer, a code pointer, and a static pointer. The static pointer points to the static region, which is different for each task. Consequently, each task must obtain separately the entry value for a particular routine.

Currently, if `unclaimed_input_entry_var` signals a condition or returns via a non-local goto, the command processor is not called to clean up, for example, the PDR heap. Consequently, `unclaimed_input_entry_var` itself must take care of cleaning up.

The value of `switches` is a binary coding of 6 logical variables. The variables are coded in the “1”, “2”, “4”, “8”, “16”, and “32” bits of the `switches` argument as shown in Table 3-6 In addition, there are 10 bits reserved for future use.

Table 3-6. Value of `switches` Argument for `s$monitor_full`

Bit	Switch Name	Explanation
32	`inhibit_cmd_funcs`	If this bit is `true`, then `s$monitor_full` does not expand command functions found in request lines.
16	`unclaimed_input_entry`	If this bit is `true`, then `s$monitor_full` calls the routine specified in `unclaimed_input_entry_var` (see above) whenever a monitor request line fails.
8	`allow_reenter`	If this bit is `true`, then the process returns to the beginning of the request input loop set up by `s$monitor_full` if `reenter` is signalled after a CTRL BREAK.
4	`cleanup_entry`	If this bit is `true`, then `s$monitor_full` goes through the cleanup routine pointed to by `cleanup_entry_var` after each call to a request.
2	`quit`	If this bit is `true`, then the execution returns from the subroutine immediately after executing `initial_request_line`.
1	`use_abbreviations`	If this bit is `true`, then OpenVOS expands abbreviations in requests and commands. For information about abbreviations files and their uses, see the *OpenVOS Commands User’s Guide* (R089).

Related Information

`s$control_task`

Performs one of a variety of functions on a specified task, depending on the value of `action_code` given.

`s$delete_task`

Deletes a dynamic task.

`s$enable_tasking`

Enables or disables tasking in the calling process.

`s$init_task`

Initializes a specified `uninitialized` task.

`s$init_task_config`

Reads a task configuration file, initializes the set of tasks described in the configuration file, and puts the tasks into the `ready` state. If a primary task (a task with identifier 1) is defined in the configuration file, then that task is given control.

`s$monitor`

Reads and carries out requests for the administration of a tasking process.

`s$reschedule_task`

Tells the task manager to dispatch another ready task.

`s$set_process_terminal`

Attaches a task's five predefined ports (`default_input`, `default_output`,`command_input`, `terminal_output`, and `terminal`), either to the process terminal or to the task's terminal.

`s$set_task_priority`

Sets the scheduling priority for a task.

`s$set_task_terminal`

Changes the terminal port attachment for the running task.

`s$start_task`

Makes an initialized or stopped task ready to run.

`s$start_task_full`

Makes an initialized or stopped task ready to run, passing it one or more initial arguments.

s$msg_cancel_receive

Purpose

This subroutine can be used by a server to cancel the receipt of a message contained in a server queue, or by a server or requester to cancel the receipt of a message contained in a message queue.

Usage

```
declare port_id                     binary(15);
declare msg_id                      binary(31);
declare error_code                  binary(15);

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

         call s$msg_cancel_receive( port_id,
                                    msg_id,
                                    error_code);
```

Arguments

▶ `port_id` (input)
The identifier of a port attached to the desired queue. `s$msg_cancel_receive` cancels the receipt of a message contained in the queue.

▶ `msg_id` (input)
The identifier of the message to be canceled.

▶ `error_code` (output)
A returned error code.

Explanation

The subroutine `s$msg_cancel_receive` changes the state of the `busy` message `msg_id` in the queue connected to the port `port_id` to `non_busy`. The subroutine marks the message `non_busy` and `been_busy`.

Either a requester or a server can call this subroutine when the queue is a message queue, but only a server can call it when the queue is a server queue.

Error Codes

The following lists some error codes this subroutine might return.

`e$invalid_io_operation`
`(1040)`
	Invalid I/O operation for current port state or attachment. The queue is a one-way server queue — not a two-way server or message queue as required.

`e$message_not_yours`
`(2753)`
	The specified message has not been received on this port.

Related Information

`s$msg_cancel_receive`
: Can be used by a server to cancel the receipt of a message contained in a two-way server queue, or by a server or a requester to cancel the receipt of a message contained in a message queue.

`s$msg_delete`
: Can be used by a requester or a server to delete a message from a message queue.

`s$msg_open`
: Is used by a process to open (to connect to) a server queue, a message queue, or a one-way server queue.

`s$msg_open_direct`
: Is used by a process to open (to connect to) a direct queue or a one-way direct queue.

`s$msg_read`
: Can be used by a requester or a server to read a message contained in a queue.

`s$msg_receive`
: Can be used by a requester or a server to receive a message contained in a queue.

`s$msg_rewrite`
: Can be used by a requester or a server to rewrite a message contained in a message queue.

`s$msg_send_reply`
: Can be used by a server to put a reply into a two-way server queue or a two-way direct queue.

`s$set_max_queue_depth`
: Sets the maximum queue depth (or maximum number of messages) of a server queue or one-way server queue.

`s$truncate_queue`
: Can be used by a server to truncate an empty message queue.

s$msg_delete

Purpose

This subroutine can be used by a requester or a server to delete a message from a message queue.

Usage

```
declare port_id             binary(15);
declare msg_id              binary(31);
declare error_code          binary(15);

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

         call s$msg_delete( port_id,
                            msg_id,
                            error_code);
```

Arguments

- `port_id` (input)
  The identifier of a port attached to the desired queue. `s$msg_delete` deletes a message from the queue.

- `msg_id` (input)
  The identifier of the message to be deleted.

- `error_code` (output)
  A returned error code.

Explanation

The subroutine `s$msg_delete` deletes the message `msg_id` from the message queue connected to the port `port_id` and notifies the event associated with the queue.

Either a requester or a server can call this subroutine. The queue must be a message queue.

A requester can call this subroutine to cancel a request it submitted earlier. However, before doing so, the requester must first mark the message `busy` by calling `s$msg_receive` to receive it.

When the requester attempts to delete a message that a server has started to service, the subroutine sets the message's `requester_aborted` switch in the message header to true. When the subroutine notifies the event associated with the queue, on which the server normally waits, this informs the server that it can stop servicing the request.

A server can call this subroutine to remove a queue entry after the entry has been serviced.

Error Codes

The following lists some error codes this subroutine might return.

`e$record_not_found`
`(1112)`
	The given key does not locate a record. Either a server or a requester tried to delete a message that was no longer in the queue.

`e$message_marked_aborted`
`(2825)`
	The specified message has not been received on this port. A server tried to delete a message that had not been received or had been received by another server.

`e$message_marked_aborted`
`(2825)`
	Message has been marked as aborted by requester. A requester attempted to delete a message that had already been received by a server. This value of `error_code` informs the requester that the message has been marked for the server as aborted.

`e$message_not_busy`
`(2862)`
	Message is not being processed by a server. A requester attempted to delete a message that had not yet been received by a server or by a requester. It must be received before it can be deleted.

Related Information

`s$msg_cancel_receive`

Can be used by a server to cancel the receipt of a message contained in a two-way server queue, or by a server or a requester to cancel the receipt of a message contained in a message queue.

`s$msg_open`

Is used by a process to open (to connect to) a server queue, a message queue, or a one-way server queue.

`s$msg_open_direct`

Is used by a process to open (to connect to) a direct queue or a one-way direct queue.

`s$msg_read`

Can be used by a requester or a server to read a message contained in a queue.

`s$msg_receive`

Can be used by a requester or a server to receive a message contained in a queue.

`s$msg_rewrite`
Can be used by a requester or a server to rewrite a message contained in a message queue.

`s$msg_send_reply`
Can be used by a server to put a reply into a two-way server queue or a two-way direct queue.

`s$set_max_queue_depth`
Sets the maximum queue depth (or maximum number of messages) of a server queue or one-way server queue.

`s$truncate_queue`
Can be used by a server to truncate an empty message queue.

s$msg_open

Purpose

This subroutine is used by a process to open (to connect to) a server queue, a one-way server queue, or a message queue.

Usage

```
declare port_id           binary(15);
declare io_type           binary(15);
declare error_code        binary(15);

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

         call s$msg_open( port_id,
                          io_type,
                          error_code);
```

Arguments

▶ `port_id` (input)

The identifier of a port attached to the desired queue.

▶ `io_type` (input)

The I/O type of the opening. `s$msg_open` opens the queue for the given I/O type. The possible values are are shown in Table 3-7.

Table 3-7. Queue I/O Types

Value	I/O Type	Queue Type
5	requester	message or two_way_server
6	server	message or two_way_server
7	send-only requester	one_way_server
8	receive-only server	one_way_server
12	server notify_one	two_way_server
13	receive-only server notify_one	one_way_server

- ▶ error_code (output)
  A returned error code.

Explanation

The subroutine s$msg_open opens the queue connected to the port port_id.

You must attach the port to the queue before you open it, and the queue must exist before you call s$msg_open.

The value of io_type determines the relation between the calling process and the queue. To open a queue, a process must have execute, read, or write access to the queue (to the file that implements the queue).

Normally you create a queue with the command create_file. You can also create a queue by calling s$create_file. See the description of the subroutine s$create_file in the OpenVOS Subroutines manuals. You cannot create indexes for queue files. Table 3-8 shows the permitted combinations of io_type (used in s$msg_open) and file_type (used in s$create_file) and the resulting relationships:

Table 3-8. I/O Types and File Types

I/O Type	File Type		
	Server	**Message**	**one_way_server**
`requester (5)`	Process will be a requester for a server queue.	Process will be a requester for a message queue.	Not allowed.
`server (6)`	Process will be a server for a server queue.	Process will be a server for a message queue.	Not allowed.
`send-only requester (7)`	Not allowed.	Not allowed.	Process will be a requester for a one-way server queue.
`receive-only server (8)`	Not allowed.	Not allowed.	Process will be a server for a one-way server queue.
`server notify_one (12)`	Process will be a server for a server queue.	Not allowed.	Not allowed.
`receive-only server notify_one (13)`	Not allowed.	Not allowed.	Process will be a server for a one-way server queue.

The operating system opens these queue files in implicit_locking mode.

Single-event Notify

Use `io_type` values 12 (`server_notify_one`) and 13 (`receive_only_server_notify_one`) to request single-event notify for servers of the queue specified by `port_id`.

Single-event notify is relevant only when more than one server process is waiting on a given queue. With single-event notify in effect, only one server process is notified when, for example, an `s$msg_send` operation is completed. Otherwise, **all** processes waiting on the queue are notified. Since only one server actually processes the new message, use of single-event notify saves the resources that would have been used to resume the server processes that do **not** handle the message but are immediately suspended. This saving is significant only when the ratio of the number of servers waiting on the queue to the number of unprocessed messages in the queue is large. So, when the number of unprocessed messages is large, single-event notify is temporarily suspended.

Different processes can have the same queue open as `server_notify_one` (`io_type = 12`) and `server` (`io_type = 6`) or as `receive_only_server_notify_one` (`io_type = 13`) and `receive_only_server` (`io_type = 8`). However, single-event notify can be in effect only when **all** server processes have the queue open as `server_notify_one` or all as `receive_only_server_notify_one`. If this is the case, and a new process opens the queue as another compatible type (`server` or `receive_only_server`, respectively), single-event notify is suspended. Also, if all servers but one have the queue open as `server_notify_one` or `receive_only_server_notify_one`, and the one that does not closes the queue, single-event notify is enabled.

Related Information

`s$call_server`
: Is used by a requester to put a message into a two-way queue and to receive a reply from the queue.

`s$msg_cancel_receive`
: Can be used by a server to cancel the receipt of a message contained in a two-way server queue, or by a server or a requester to cancel the receipt of a message contained in a message queue.

`s$msg_delete`
: Can be used by a requester or a server to delete a message from a message queue.

`s$msg_open_direct`
: Is used by a process to open (to connect to) a direct queue or a one-way direct queue.

`s$msg_read`
: Can be used by a requester or a server to read a message contained in a queue.

`s$msg_receive`
: Can be used by a requester or a server to receive a message contained in a queue.

`s$msg_receive_reply`
: Can be used by a requester to receive a reply from a two-way server queue or a two-way direct queue.

`s$msg_rewrite`

Can be used by a requester or a server to rewrite a message contained in a message queue.

`s$msg_send`

Can be used by a `send_only_requester` to put a message into any one-way queue (message, server, or direct) or by a two-way requester to put a message into any two-way queue.

`s$msg_send_reply`

Can be used by a server to put a reply into a two-way server queue or a two-way direct queue.

`s$truncate_queue`

Can be used by a server to truncate an empty message queue.

s$msg_open_direct

Purpose

This subroutine is used by a process to open (to connect to) a direct queue or a one-way direct queue.

Usage

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

declare 1 direct_info            shortmap,
          2 version              binary(15),
          2 maximum_msg_size     binary(15),
          2 msg_header_version   binary(15),
          2 reserved             (13) binary(15);

declare error_code               binary(15);

declare s$msg_open_direct entry( binary(15),
                                 binary(15),
                                 1 like direct_info,
                                 binary(15));

         call s$msg_open_direct( port_id,
                                 io_type,
                                 direct_info,
                                 error_code);
```

Arguments

▶ `port_id` (input)
The identifier of a port attached to a direct queue.

▶ `io_type` (input)
The I/O type of the opening. The possible values are shown in Table 3-9.

Table 3-9. Direct Queue I/O Types

I/O Type	Value
requester	5
server	6
send-only requester	7
receive-only server	8

`s$msg_open_direct` opens the queue for the given I/O type.

▶ `direct_info` (input)
Information about the queue to be opened.

▶ `version` (input)
The version number of the `direct_info` structure. You must assign the value 1 to it.

▶ `maximum_msg_size` (input)
The maximum number of bytes in a request or reply message. This must be a value between 0 and 3072. All openers of a given direct queue must specify the same value. Since each call to `s$msg_open_direct` allocates a buffer of the specified size in wired memory, it is better not to specify a value of `maximum_msg_size` larger than necessary.

▶ `msg_header_version` (input)
The version number of the message header structure expected by a server process calling `s$msg_receive`. This value can be either 0 for no message header, or 1 for the standard message header. All servers must specify the same value for a given direct queue. Requesters ignore this argument.

▶ `reserved` (output)
Reserved for future use.

▶ `error_code` (output)
A returned error code.

Explanation

To create a direct queue, use the command `create_file` or the subroutine `s$create_file` to create a sequential file. That file will become a direct queue when you use `s$msg_open_direct` to open it.

A direct queue must be on the same module as the server(s), and a server must open it before any requester can do so.

Direct queues can be one- or two-way. The type of queue is determined by the server process which first opens it. If the server first opening the queue specifies itself as a `receive_only_server` (`io_type = 8`), then the queue will be one-way, and can be subsequently opened only by `receive_only_servers` and `send_only_requesters` (`io_type = 7`). If the server first opening the queue specifies itself as a `server` (`io_type = 6`), then the queue will be two-way, and can subsequently be opened only by `servers` and `requesters` (`io_type = 5`).

A direct queue can contain only one message at a time for each port attached to the queue.

A direct queue cannot be transaction-protected. That is, it cannot be used to communicate between requesters and servers after a transaction has been started by a call to `s$start_transaction` but before the transaction has been committed or aborted.

Precautions When Closing Direct Queues

When a requester calls `s$msg_send` to place a message in a direct queue, the message initially is written into the requester's memory buffer. When a server's buffer becomes available, OpenVOS copies the message into that buffer. If the server closes the queue before the copy has taken place, the last message sent may be lost. Consequently, the requester should use `s$msg_send` to send a final dummy "closing" message before closing the queue. A requester can have only one message in its buffer at a time; so `s$msg_send` will not return with a zero error code until the last "real" message has been copied into a server's buffer. At that time the requester can safely close the queue.

Related Information

`s$call_server`
Is used by a requester to put a message into a two-way queue and to receive a reply from the queue.

`s$msg_cancel_receive`
Can be used by a server to cancel the receipt of a message contained in a two-way server queue, or by a server or a requester to cancel the receipt of a message contained in a message queue.

`s$msg_delete`
Can be used by a requester or a server to delete a message from a message queue.

`s$msg_open`
Is used by a process to open (to connect to) a server queue, a message queue, or a one-way server queue.

`s$msg_open_virtual`
Is used by a process to open (to connect to) a virtual queue.

`s$msg_read`
Can be used by a requester or a server to read a message contained in a queue.

`s$msg_receive`
Can be used by a requester or a server to receive a message contained in a queue.

`s$msg_receive_reply`

Can be used by a requester to receive a reply from a two-way server queue or a two-way direct queue.

`s$msg_rewrite`

Can be used by a requester or a server to rewrite a message contained in a message queue.

`s$msg_send`

Can be used by a `send_only_requester` to put a message into any one-way queue (message, server, or direct) or by a two-way requester to put a message into any two-way queue.

`s$msg_send_reply`

Can be used by a server to put a reply into a two-way server queue or a two-way direct queue.

`s$truncate_queue`

Can be used by a server to truncate an empty message queue.

s$msg_open_virtual

Purpose

The `s$msg_open_virtual` subroutine is used by a process to open (to connect to) a virtual queue.

Usage

```
declare   port_id                     fixed bin (15);
declare   io_type                     fixed bin (15);

declare 1 virtual_info                shortmap,
          2 version                   fixed bin (15),
          2 maximum_msg_size          fixed bin (15),
          2 msg_header_version        fixed bin (15),
          2 reserved                  (13) fixed bin (15);

declare   error_code                  fixed bin (15);

declare   s$msg_open_virtual entry (  fixed bin (15),
                                      fixed bin (15),
                                      1 like virtual_info,
                                      fixed bin (15));

          call s$msg_open_virtual (   port_id,
                                      io_type,
                                      virtual_info,
                                      error_code);
```

Arguments

▶ `port_id (input)`

The identifier of a port attached to a virtual queue.

▶ `io_type (input)`

The type of I/O for which you are opening the queue. The allowed values for `io_type` are listed in the following table.

I/O Value	**I/O Type**
5	requester
6	server

- ▶ virtual_info (input)

  Information about the queue to be opened.

- ▶ version (input)

  The version number of the virtual_info structure. Requesters and servers use the same version number; the value must be 1 or 2.

- ▶ maximum_msg_size (input)

  The maximum number of bytes in a message or reply. The value can be between 0 and 32,767, inclusive. All openers of the queue must specify the same value. Since each call to s$msg_open_virtual allocates a buffer of the specified size in wired memory, you should specify the smallest value possible for maximum_msg_size.

- ▶ msg_header_version (input)

  The version of the message header structure expected by a server process. The value is 0 (no header) for servers; requesters ignore this argument.

- ▶ reserved

  Reserved for use by the operating system.

- ▶ error_code (output)

  A returned error code.

Explanation

A virtual queue is a special type of two-way direct queue that is allocated in wired memory and can improve the performance of queue-related operations by up to 50%. The virtual queue and all processes using the virtual queue must exist on the same module. A virtual queue is compatible with existing direct queue operations, which were implemented with s$msg_open_direct.

To create a virtual queue, use the command create_file or the subroutine s$create_file to create a sequential file. You specify the virtual queue type by using the s$msg_open_virtual subroutine to open the file. That file will become a virtual queue when you use s$msg_open_virtual to open it.

Servers must have write access to the file; requesters must have non-null access. The virtual queue must be on the smae module as the servers. A server must open a virtual queue before any requester can do so.

As with a direct queue, messages in a virtual queue are handled on a first-in, first-out (FIFO) basis. Message priority is ignored. Since the maximum queue depth for a virtual queue is 1, a virtual queue can contain only one message at a time for each port attached and opened to the queue. The requester must get a reply before sending another message to the queue. The server cannot send a message unless it has sent a reply to the previous message.

A virtual queue does not support tasking and is not associated with operating system events. A program cannot use s$set_no_wait_mode to enable no-wait mode and to get an event ID for the port associated with a virtual queue. A program cannot wait for the operating system to notify a system event when a queue-related subroutine call returns. For example, a server cannot wait for the operating system to notify a system event when a requester sends a message to the queue. If your application requires tasking or the use of system events for event notification, use a queue type other than a virtual queue.

With a virtual queue, the server program waits for messages, and the requester program waits for a reply. Both the server and requester can use the `s$set_io_time_limit` subroutine with a virtual queue to limit the amount of time they wait for the queue-related operation to occur. For example, you can use `s$set_io_time_limit` to set a limit on how long the operating system waits for a reply from a server when a requester calls `s$msg_receive_reply` to get a reply from a virtual queue.

A virtual queue cannot be transaction-protected. That is, a virtual queue, unlike a two-way server queue, does not support the transaction-protection capability that allows a requester to transmit transaction protection to a server program.

When a requester calls `s$msg_send` to place a message in the virtual queue, the message is initially written into the requester's memory buffer. When a server's buffer becomes available, the operating system copies the message into that buffer. If the server closes the queue before the copy has occurred, the last message sent may be lost. Consequently, the requester should use `s$msg_send` to send a final dummy "closing" message before closing the queue. A requester can have only one message in its buffer at a time, so `s$msg_send` will not return with a zero error code until the last "real" message has been copied into a server's buffer. At that time, the requester can safely close the queue.

For more information on queues, see the *VOS Transaction Processing Facility Guide* (R215).

Error Codes

The following lists some error codes this subroutine might return.

`e$invalid_io_operation`
`(1040)`
	The file attached to the port specified by `port_id` is not a sequential file.

`e$invalid_record_size`
`(1053)`
	The value given in `maximum_msg_size` is outside inclusive).

`e$invalid_io_type`
`(1070)`
	The value given in `io_type` must be `5` (requester) or `6` (server).

`e$wrong_version`
`(1083)`
	The version number for the `virtual_info` structure is incorrect.

Related Information

`s$call_server`
: Is used by a requester to put a message into a two-way queue and to receive a reply from the queue.

`s$msg_cancel_receive`
: Can be used by a server to cancel the receipt of a message contained in a two-way server queue, or by a server or a requester to cancel the receipt of a message contained in a message queue.

`s$msg_delete`
Can be used by a requester or a server to delete a message from a message queue.

`s$msg_open`
Is used by a process to open (to connect to) a server queue, a message queue, or a one-way server queue.

`s$msg_open_direct`
Is used by a process to open (to connect to) a direct queue or a one-way direct queue.

`s$msg_read`
Can be used by a requester or a server to read a message contained in a queue.

`s$msg_receive`
Can be used by a requester or a server to receive a message contained in a queue.

`s$msg_receive_reply`
Can be used by a requester to receive a reply from a two-way server queue or a two-way direct queue.

`s$msg_rewrite`
Can be used by a requester or a server to rewrite a message contained in a message queue.

`s$msg_send`
Can be used by a `send_only_requester` to put a message into any one-way queue (message, server, or direct) or by a two-way requester to put a message into any two-way queue.

`s$msg_send_reply`
Can be used by a server to put a reply into a two-way server queue or a two-way direct queue.

`s$truncate_queue`
Can be used by a server to truncate an empty message queue.

s$msg_read

Purpose

This subroutine can be used by a requester or a server to read a message contained in a queue.

Table 3-10 describes which io_type may use s$msg_read for which kind of queue.

Table 3-10. Permitted Use of s$msg_read by Various I/O Types

I/O Type	Queue Type			
	Message	Server	One-way Server	Direct
requester	Yes	Yes	—	No
server	Yes	Yes	—	No
send_only_requester	—	—	Yes	No
receive_only_server	—	—	Yes	No

Note: "—" means that the io_type is invalid for that kind of queue.

Usage

```
declare port_id                        binary(15);
declare msg_selector                   binary(15);
declare msg_id                         binary(31);

declare 1 msg_header                   shortmap,
          2 version                    binary(15),
          2 header_msg_id              binary(31),
          2 date_time_queued           binary(31),
          2 request_priority           binary(15),
          2 msg_subject                char(32) varying,
          2 requester_process_id       binary(31),
          2 requester_person_name      char(32) varying,
          2 requester_group_name       char(32) varying,
          2 server_process_id          binary(31),
          2 switches                   binary(15),
          2 requester_maximum_priority  binary(15),
          2 requester_maximum_processes binary(15),
          2 jiffies_queued             binary(15),
          2 reserved1                  (10) binary(15),
          2 language                   char(32) varying,
          2 character_set              binary(15),
          2 reserved2                  (3) binary(15);

declare msg_length_in                  binary(31);
declare msg_length_out                 binary(31);
declare msg                            data_type;

declare error_code                     binary(15);
declare s$msg_read entry(              binary(15),
                                       binary(15),
                                       binary(31),
                                       1 like msg_header,
                                       binary(31),
                                       binary(31),
                                       data_type,
                                       binary(15));
```

(Continued on next page)

```
        call s$msg_read(                port_id,
                                        msg_selector,
                                        msg_id,
                                        msg_header,
                                        msg_length_in,
                                        msg_length_out,
                                        msg,
                                        error_code);
```

Arguments

- ▶ `port_id` (input)

  The identifier of a port attached to the queue. `s$msg_read` reads a message contained in the queue.

- ▶ `msg_selector` (input)

  A value determining the message that `s$msg_read` is to read. Table 3-11 shows the possible values.

Table 3-11. The Values of `msg_selector` for `s$msg_read`

Message to be Read	Selector
`first_message_non_busy`	1
`this_message_non_busy`	2
`next_message_non_busy`	3
`first_message`	4
`this_message`	5
`next_message`	6

- ▶ `msg_id` (input-output)

  The identifier of the message to be read or that is read.

- ▶ `msg_header` (input-output)

  Information about the message the subroutine reads.

- ▶ `version` (input)

  The version number of the `msg_header` structure. You must assign the value 1 to it.

- ▶ `header_msg_id` (output)

  Same as `msg_id`.

- ▶ `date_time_queued` (output)

  The date and time that the message was placed in the queue, as determined by the clock on the module where the queue resides.

- ▶ `request_priority` (output)

  The priority assigned to the message when it was sent.

- ▶ `msg_subject` (output)
  The subject argument supplied when the message was sent.

- ▶ `requester_process_id` (output)
  The requester's process identifier.

- ▶ `requester_person_name` (output)
  The requester's person name.

- ▶ `requester_group_name` (output)
  The requester's group name.

- ▶ `server_process_id` (output)
  If a server has received the message, this tells the process identifier of the server.

- ▶ `switches` (output)
  Information about the status of the message. See Explanation section for details.

- ▶ `requester_maximum_priority` (output)
  The priority of the requester process.

- ▶ `requester_maximum_processes` (output)
  The maximum number of processes that the requester may have going at any one time.

- ▶ `jiffies_queued` (output)
  Tells the number of jiffies (1/65536 of a second) to be added to the value of `date_time_queued` to get the exact time that the message was placed in the queue. The `date_time_queued` and `jiffies_queued` values are determined by the clock on the module where the queue resides. Clocks on different modules are not guaranteed to be set to exactly the same time. Therefore, if the caller and the queue reside on different modules, the `date_time_queued` and `jiffies_queued` values may not be meaningful when they are compared to the time on the caller's module.

- ▶ `reserved1` (output)
  Reserved for use by OpenVOS.

- ▶ `language` (output)
  The natural language in which the message is written, for example, `us_english`.

- ▶ `character_set` (output)
  The character set in which the message is written. The following table shows the character sets currently recognized and their values.

Character Set	**Value**
`ascii_char_set`	`0`
`latin_1_char_set`	`1`
`kanji_char_set`	`2`
`katakana_char_set`	`3`

- ▶ `reserved2` (output)
  Reserved for use by OpenVOS.

- ▶ `msg_length_in` (input)
  The length of the output buffer `msg`, in bytes.

- ▶ `msg_length_out` (output)
  The actual length of the message read.

- ▶ `msg` (output)
  The output buffer containing the message read from the queue. The maximum length of a message is 2^{23} bytes for a queue on the same module as the calling process or 2^{20} bytes for a queue on a different module.

- ▶ `error_code` (output)
  A returned error code.

Explanation

For both MESSAGE and SERVER queues:

The subroutine `s$msg_read` reads a message contained in the queue connected to the port `port_id`. The caller does not receive the message, however, so the message is not marked as `busy` and can be serviced by another process. The caller can also read a message that is marked as `busy`.

The `msg_selector` argument determines which message in the queue the subroutine reads. The following list shows the values `msg_selector` can take, along with their meanings and some possible applications:

`first_message_non_busy`
The message is the closest one to the front of the queue that is `non_busy`. A program can use this selector to initiate a scan of messages in the queue that have not yet been processed. The subroutine returns the identifier of the message in `msg_id`.

`this_message_non_busy`
The message is the one with the identifier `msg_id`. A program can use this selector to determine whether a specific message has been received; for example, when two servers are exchanging information by receiving and rewriting a known message in a message queue.

`next_message_non_busy`
The message is the closest one after the message most recently read by `s$msg_read` or received by `s$msg_receive` that is `non_busy`. A program can use this selector to continue the scan of unprocessed messages begun by `first_message_non_busy`. The subroutine returns the identifier of the message in `msg_id`.

`first_message`
The message is the message at the front of the queue. A program can use this selector to initiate a scan of all messages in the queue, processed or not. For example, you might do this when cleaning up a message queue. The subroutine returns the identifier of the message in `msg_id`.

`this_message`
The message is the message with the identifier `msg_id`. A program can use this selector to get from a message queue the rest of a message received by `s$msg_receive` that

was truncated because `msg_length_in` was too small.

`next_message`

The message is the closest message after the message most recently read by `s$msg_read` or received by `s$msg_receive`. A program can use this selector to continue the scan of all messages begun by `first_message`. The subroutine returns the identifier of the message in `msg_id`.

The possible results of the call are given in the following tables. Beginning with Table 3-12, each table gives the results for a given value of `msg_selector`.

Table 3-12. The Results of Trying to Read `first_message_non_busy`

`first_message_non_busy`				
Message Status	**Port Mode**	**Access of Caller to Queue**	**Caller Type**	
			Requester	**Server**
Not Available	`wait`			Caller Waits
	`no_wait`		Subroutine returns immediately. Error code is `e$end_of_file` (1025).	Subroutine returns immediately. Error code is `e$caller_must_wait` (1277).
	I/O time limit set			Caller waits for the time limit.
Available		`execute`	Subroutine returns the first message submitted by a process with same person name as caller. If no message fits the description, subroutine returns the error code `e$record_not_found` (1112).	
		`read` or `write`	Subroutine returns the first non-busy message submitted by any process.	

Note that in Table 3-13 the port mode is not relevant for the `this_message_non_busy` argument value.

Table 3-13. The Results of Trying to Read `this_message_non_busy` *(Page 1 of 2)*

`this_message_non_busy`		
		Caller Type
Message Status	**Access of Caller to Queue**	**Requester or Server**
Not in queue		Subroutine returns the error code `e$record_not_found` (1112).
Busy		Subroutine returns the error code `e$message_busy` (2750).

Table 3-13. The Results of Trying to Read `this_message_non_busy` *(Page 2 of 2)*

`this_message_non_busy`		
Message Status	**Access of Caller to Queue**	**Caller Type** **Requester or Server**
Available	`execute`	Subroutine returns the message only if it was sent to the queue by a process with the same person name as the caller. Otherwise, subroutine returns the error code `e$record_not_found` (1112).
	`read` or `write`	Subroutine returns the message.

Note that in Table 3-14 the port mode is not relevant for the **`next_message_non_busy`** argument value.

Table 3-14. The Results of Trying to Read `next_message_non_busy`

`next_message_non_busy`		
Message Status	**Access of Caller to Queue**	**Caller Type** **Requester or Server**
None available		Subroutine returns the error code `e$end_of_file` (1025).
Available	`execute`	Subroutine returns the message only if it was sent to the queue by a process with the same person name as the caller. Otherwise, subroutine returns the error code `e$end_of_file` (1025).??
	`read` or `write`	Subroutine returns the message.

Note that in Table 3-15 port mode is relevant for `first_message`.

Table 3-15. The Results of Trying to Read `first_message`

`first_message`				
Message Status	**Port Mode**	**Access of Caller to Queue**	**Caller Type**	
			Requester	**Server**
None in queue	`wait`			Caller Waits
	`no_wait`		Subroutine returns immediately. Error code is `e$end_of_file` (1025).	Subroutine returns immediately. Error code is `e$caller_must_wait` (1277).
	I/O time limit set			Caller waits for the time limit.
In queue		`execute`	Subroutine returns the first message submitted by a process with same person name as caller. If no message fits the description, subroutine returns the error code `e$record_not_found` (1112).	
		`read` or `write`	Subroutine returns the first non-busy message submitted by any process.	

Note that in Table 3-16 the port mode is not relevant for the **`this_message`** argument value.

Table 3-16. The Results of Trying to Read `this_message`

`this_message`		
Message Status	**Access of Caller to Queue**	**Caller Type** **Requester or Server**
Not in queue		Subroutine returns the error code `e$end_of_file` (1025).
In queue	`execute`	Subroutine returns the message only if it was sent to the queue by a process with the same person name as the caller. Otherwise, subroutine returns the error code `e$end_of_file` (1025).
	`read` or `write`	Subroutine returns the message.

Note that in Table 3-17 the port mode is not relevant for the **`next_message`** argument value.

Table 3-17. The Results of Trying to Read `next_message`

`next_message`		
Message Status	**Access of Caller to Queue**	**Caller Type**
		Requester or Server
None available		Subroutine returns the error code `e$end_of_file` (1025).
Available	`execute`	Subroutine returns the message only if it was sent to the queue by a process with the same person name as the caller. Otherwise, subroutine returns the error code `e$end_of_file` (1025).
	`read` or `write`	Subroutine returns the message.

The value of `switches` is a binary coding of 9 logical variables. The variables are coded in the 9 high order bits of `switches` as shown in Table 3-18. In addition, the 7 low order bits are reserved for future use.

Table 3-18. Value of `switches` Argument for `s$msg_read`

Bit	Switch Name	Explanation
-32768	`been_busy`	If this bit is `true`, then the message has been received but is not currently in use.
16384	`now_busy`	If this bit is `true`, then the message is currently in use.
8192	`reply_requested`	If this bit is `true`, then the requester asked for a reply when it sent the message.
4096	`requester_aborted`	If this bit is `true`, then the requester aborted the transaction of which the message was a part.
2048	`server_done`	If this bit is `true`, then a server has finished servicing the message.
1024	`server_aborted`	If this bit is `true`, then the server that received a message in a two-way server queue stopped running or was stopped while servicing the message.
512	`requester_privileged`	If this bit is `true`, then the requester is a privileged process.
256	`no_servers`	If this bit is `true`, then there are no servers currently active.
128	`language_attrs_valid`	If this bit is `true`, then `language` and `character_set` contain valid information. This switch is provided for compatibility with earlier message headers that did not contain International Character Set Support information.

For SERVER queues only:

For two-way queues, if the message is longer than msg_length_in and msg_length_in is nonzero, s$msg_read returns as much of the message as fits in the space allocated, returns the true length of the message in msg_length_out, and returns the error code e$long_record (1026). If msg_length_in is zero, s$msg_read returns the true length of the message in msg_length_out and a zero error code.

For one-way queues, if the message is longer than msg_length_in (including when msg_length_in is zero), s$msg_read returns the error code e$buffer_too_small (1133) and the contents of msg are unreliable.

Related Information

s$msg_cancel_receive
: Can be used by a server to cancel the receipt of a message contained in a two-way server queue, or by a server or a requester to cancel the receipt of a message contained in a message queue.

s$msg_delete
: Can be used by a requester or a server to delete a message from a message queue.

s$msg_open
: Is used by a process to open (to connect to) a server queue, a message queue, or a one-way server queue.

s$msg_open_direct
: Is used by a process to open (to connect to) a direct queue or a one-way direct queue.

s$msg_receive
: Can be used by a requester or a server to receive a message contained in a queue.

s$msg_rewrite
: Can be used by a requester or a server to rewrite a message contained in a message queue.

s$msg_send_reply
: Can be used by a server to put a reply into a two-way server queue or a two-way direct queue.

s$set_max_queue_depth
: Sets the maximum queue depth (or maximum number of messages) of a server queue or one-way server queue.

s$truncate_queue
: Can be used by a server to truncate an empty message queue.

s$msg_receive

Purpose

This subroutine can be used by a requester or a server to receive a message contained in a queue.

Table 3-19 describes which io_type may use s$msg_receive for which kind of queue.

Table 3-19. Permitted Use of s$msg_receive by Various I/O Types

I/O Type	**Queue Type**			
	Message	**Server**	**One-way Server**	**Direct**
requester	Yes	No	—	No
server	Yes	Yes	—	Yes
send_only_requester	—	—	No	No
receive_only_server	—	—	Yes	Yes

Note: "—" means that the io_type is invalid for that kind of queue.

Usage

```
declare port_id                         binary(15);
declare msg_selector                    binary(15);
declare msg_id                          binary(31);

declare 1 msg_header                    shortmap,
          2 version                     binary(15),
          2 header_msg_id               binary(31),
          2 date_time_queued            binary(31),
          2 requester_priority          binary(15),
          2 msg_subject                 char(32) varying,
          2 requester_process_id        binary(31),
          2 requester_person_name       char(32) varying,
          2 requester_group_name        char(32) varying,
          2 server_process_id           binary(31),
          2 switches                    binary(15),
          2 requester_maximum_priority  binary(15),
          2 requester_maximum_processes binary(15),
          2 jiffies_queued              binary(15),
          2 reserved1                   (10) binary(15),
          2 language                    char(32) varying,
          2 character_set               binary(15),
          2 reserved2                   (3) binary(15);

declare msg_length_in                   binary(31);
declare msg_length_out                  binary(31);
declare msg                             data_type;
declare error_code                      binary(15);

declare s$msg_receive entry(            binary(15),
                                        binary(15),
                                        binary(31),
                                        1 like msg_header,
                                        binary(31),
                                        binary(31),
                                        data_type,
                                        binary(15));
```

(Continued on next page)

(Continued)

```
        call s$msg_receive(           port_id,
                                      msg_selector,
                                      msg_id,
                                      msg_header,
                                      msg_length_in,
                                      msg_length_out,
                                      msg,
                                      error_code);
```

Arguments

- `port_id` (input)
  The identifier of a port attached to the desired queue. `s$msg_receive` receives a message contained in the queue.

- `msg_selector` (input)
  A value determining which message `s$msg_receive` is to receive. The possible values are shown in Table 3-20.

Table 3-20. The Values of `msg_selector` for `s$msg_receive`

Message to be Received	Selector
`first_message_non_busy`	1
`this_message_non_busy`	2
`next_message_non_busy`	3

To receive a message from a direct queue, you must set this argument to 1.

- `msg_id` (input-output)
  The identifier of the message to be received or that is received. To receive a message from a direct queue, you must set this argument to 0.

- `msg_header` (input-output)
  Information about the message the subroutine receives.

- `version` (input)
  The version number of the `msg_header` structure. If the queue is not a direct queue, then you must assign the value 1.

  If the queue is a direct queue, then you must assign the same value that was assigned when the queue was opened. A value of 0 means the `msg_header` structure will not be filled in on return from `s$msg_receive`. A value of 1 means the `msg_header` structure will be filled in.

- `header_msg_id` (output)
  The identifier of the message that is received.

- ▶ `date_time_queued` (output)
  The date and time that the message was placed in the queue, as determined by the clock on the module where the queue resides.

- ▶ `requester_priority` (output)
  The priority assigned to the message when it was sent.

- ▶ `msg_subject` (output)
  The subject argument supplied when the message was sent.

- ▶ `requester_process_id` (output)
  The requester's process identifier.

- ▶ `requester_person_name` (output)
  The requester's person name.

- ▶ `requester_group_name` (output)
  The requester's group name.

- ▶ `server_process_id` (output)
  If a server has received the message, this tells the process identifier of the server.

- ▶ `switches` (output)
  Information about the status of the message. See the Explanation for details.

- ▶ `requester_maximum_priority` (output)
  The priority of the requester process.

- ▶ `requester_maximum_processes` (output)
  The maximum number of processes that the requester may have going at any one time.

- ▶ `jiffies_queued` (output)
  Tells the number of jiffies (1/65536 of a second) to be added to the value of `date_time_queued` to get the exact time that the message was placed in the queue. The `date_time_queued` and `jiffies_queued` values are determined by the clock on the module where the queue resides. Clocks on different modules are not guaranteed to be set to exactly the same time. Therefore, if the caller and the queue reside on different modules, the `date_time_queued` and `jiffies_queued` values may not be meaningful when they are compared to the time on the caller's module.

- ▶ `reserved1` (output)
  Reserved for use by OpenVOS.

- ▶ `language` (output)
  The natural language in which the message is written, for example, `us_english`.

- `character_set` (output)
  The character set in which the message is written. The following table shows the character sets currently recognized and their values.

Character Set	Value
`ascii_char_set`	0
`latin_1_char_set`	1
`kanji_char_set`	2
`katakana_char_set`	3

- `reserved2` (output)
  Reserved for use by OpenVOS.

- `msg_length_in` (input)
  The length of the output buffer `msg`, in bytes.

- `msg_length_out` (output)
  The actual length of the message received.

- `msg` (output)
  The output buffer containing the message read from the queue. The maximum length of a message is 2^{23} bytes for a queue on the same module as the calling process, 2^{20} bytes for a queue on a different module, or the value of `maximum_msg_size` specified to `s$msg_open_direct` for a direct queue.

- `error_code` (output)
  A returned error code.

Explanation

For ALL queues:

The subroutine `s$msg_receive` receives a message contained in the queue connected to the port `port_id`.

The `msg_selector` argument determines which message in the queue the subroutine reads. The following list shows the values `msg_selector` can take, along with their meanings and some possible applications:

`first_message_non_busy`
The message is the closest one to the front of the queue that is `non_busy`. A program can use this selector to initiate a scan of messages in the queue that have not yet been processed. The subroutine returns the identifier of the message in `msg_id`.

`this_message_non_busy`
The message is the one with the identifier `msg_id`. A program can use this selector to determine whether a specific message has been received, for example, when two servers are exchanging information by receiving and rewriting a known message in a message queue.

`next_message_non_busy`
The message is the closest one after the message most recently read by

`s$msg_receive` or received by `s$msg_receive` that is `non_busy`. A program can use this selector to continue the scan of unprocessed messages begun by using `first_message_non_busy`. The subroutine returns the identifier of the message in `msg_id`.

The possible results of the call for direct queues are given in Table 3-21.

Table 3-21. The Results of Calling `s$msg_receive` for Direct Queues

Message Status	Port Mode	Result
None available	`wait`	Caller waits until a message becomes available.
	`no_wait`	Subroutine returns immediately. Error code is `e$caller_must_wait` (1277).
	`I/O time limit set`	Caller waits for time limit.
Available		Subroutine returns the first non-busy message in the queue.

The possible results of the call for message and server queues are given in the following tables. Beginning with Table 3-22, each table gives the results for a given value of `msg_selector`.

Table 3-22. The Results of Trying to Receive `first_message_non_busy`

`first_message_non_busy`				
Message Status	Port Mode	Access of Caller to Queue	Caller Type	
			Requester	Server
None available	`wait`			Caller Waits
	`no_wait`		Subroutine returns immediately. Error code is `e$end_of_file` (1025).	Subroutine returns immediately. Error code is `e$caller_must_wait` (1277).
	I/O time limit set			Caller waits for the time limit.
Available		`execute` or `read`	Subroutine returns the first message submitted by a process with same person name as caller. If no message fits the description, subroutine returns the error code `e$record_not_found` (1112).	
		`write`	Subroutine returns the first non-busy message submitted by any process.	

Note that in Table 3-23 the port mode is not relevant for the **this_message_non_busy** argument value.

Table 3-23. The Results of Trying to Receive this_message_non_busy

this_message_non_busy		
Message Status	**Access of Caller to Queue**	**Caller Type**
		Requester or Server
Not in queue		Subroutine returns the error code e$record_not_found (1112).
Busy		Subroutine returns the error code e$message_busy (2750).
In queue and not busy.	execute	Subroutine returns the message only if sender process had same person name as the caller. Otherwise, acts as if the message is not in the queue.
	read	Subroutine returns the message only if sender process had same person name as the caller. Otherwise, subroutine returns the error code e$msg_insufficient_access (2888).
	write	Subroutine returns the message.

Note that in Table 3-24 the port mode is not relevant for the **next_message_non_busy** argument value.

Table 3-24. The Results of Trying to Receive next_message_non_busy

next_message_non_busy		
Message Status	**Access of Caller to Queue**	**Caller Type**
		Requester or Server
Not available		Subroutine returns the error code e$end_of_file (1025).
Available	execute or read	Subroutine returns the message only if sender process had same person name as the caller. Otherwise, acts as if the message is not in the queue.
	write	Subroutine returns the next non-busy message.

The value of switches is a binary coding of 9 logical variables. The variables are coded in the 9 high order bits of switches as shown in Table 3-24. In addition, the 7 low order bits are reserved for future use.

Table 3-25. Value of `switches` Argument for `s$msg_receive`

Bit	Switch Name	Explanation
-32768	`been_busy`	If this bit is `true`, then the message has been received but is not currently in use.
16384	`now_busy`	If this bit is `true`, then the message is currently in use.
8192	`reply_requested`	If this bit is `true`, then the requester asked for a reply when it sent the message.
4096	`requester_aborted`	If this bit is `true`, then the requester aborted the transaction of which the message was a part.
2048	`server_done`	If this bit is `true`, then a server has finished servicing the message.
1024	`server_aborted`	If this bit is `true`, then the server that received a message in a two-way server queue stopped running or was stopped while servicing the message.
512	`requester_privileged`	If this bit is `true`, then the requester is a privileged process.
256	`no_servers`	If this bit is `true`, then there are no servers currently active.
128	`language_attrs_valid`	If this bit is `true`, then `language` and `character_set` contain valid information. This switch is provided for compatibility with earlier message headers that did not contain International Character Set Support information.

For MESSAGE queues only:

The message is marked `busy` and cannot be serviced by another server. The caller cannot receive a message that is marked `busy`.

If the message is longer than `msg_length_in` and `msg_length_in` is nonzero, `s$msg_receive` returns as much of the message that fits in the space allocated, returns the true length of the message in `msg_length_out`, and returns the error code e$`long_record` (1026). The message is marked busy. The caller can get the rest of the message, if needed, by using `s$msg_read` with the selector argument set to `this_message` (5) and a `msg_length_in` value large enough to accommodate the entire message. If `msg_length_in` is zero, `s$msg_receive` returns a zero error code and the length of the message in `msg_length_out`. The message is marked busy. To get the message, the caller must use `s$msg_read` with a `msg_length_in` value large enough to accommodate the message and the selector argument set to `this_message` (5).

For SERVER queues only:

If the queue is a one-way queue, then the message is marked `busy` and cannot be serviced by another server. The caller cannot receive a message marked `busy`. If the receipt of the message successfully completes then the message is removed from the queue.

If the queue is a two-way queue, and the message is part of a transaction (it was sent after a call to `s$start_transaction` or `s$start_priority_transaction` and before a call to `s$commit_transaction` or `s$abort_transaction`) then the message's transaction identifier is assigned to the server. The transaction identifier remains set until the server replies to the message.

For two-way queues, if the message is longer than `msg_length_in` and `msg_length_in` is nonzero, `s$msg_receive` returns as much of the message that fits in the space allocated, returns the true length of the message in `msg_length_out`, and returns the error code `e$long_record` (1026). When `msg_length_in` is zero for two-way queues, `s$msg_receive` returns a zero error code and the length of the message in `msg_length_out`. In either case, the message is marked busy. To read the entire message, the caller must use `s$msg_read` with a `msg_length_in` value large enough to accommodate the message and the selector argument set to `this_message` (5).

For one-way queues, if the message is longer than `msg_length_in` (including when `msg_length_in` is zero), `s$msg_receive` returns the error code `e$buffer_too_small` (1133), and sets `msg_length_out` to the true length of the message. The message is not received. The content of `msg` is unpredictable. The message remains in the queue with a nonbusy status. The caller can call `s$msg_receive` again, with a larger `msg_length_in` value, to receive the message.

For DIRECT queues only:

The message is removed from the queue when it is received.

If the message is longer than `msg_length_in`, the message is lost and the subroutine returns the error code `e$buffer_too_small` (`1133`). Since the maximum message length for a direct queue is specified when the queue is opened, and can never be more than 3072 bytes, you can avoid this problem by specifying the same length in `msg_length_in` as specified by the `maximum_msg_size` argument in `s$msg_open_direct`.

Error Codes

The following table explains an error code this subroutine might return.

`e$tp_in_progress` (2932)	A transaction is already in progress. A server program started a transaction and then called `s$msg_receive` on a two-way server queue. To correct the problem, either start the transaction after calling `s$msg_receive`, start the transaction in the requester program, or use a different type of queue. See the Explanation in `s$start_transaction` for more information.

Related Information

`s$msg_cancel_receive`
: Can be used by a server to cancel the receipt of a message contained in a two-way server queue, or by a server or a requester to cancel the receipt of a message contained in a message queue.

`s$msg_delete`
: Can be used by a requester or a server to delete a message from a message queue.

`s$msg_open`
: Is used by a process to open (to connect to) a server queue, a message queue, or a one-way server queue.

`s$msg_open_direct`
: Is used by a process to open (to connect to) a direct queue or a one-way direct queue.

`s$msg_read`
: Can be used by a requester or a server to read a message contained in a queue.

`s$msg_rewrite`
: Can be used by a requester or a server to rewrite a message contained in a message queue.

`s$msg_send_reply`
: Can be used by a server to put a reply into a two-way server queue or a two-way direct queue.

`s$set_max_queue_depth`
: Sets the maximum queue depth (or maximum number of messages) of a server queue or one-way server queue.

`s$truncate_queue`
: Can be used by a server to truncate an empty message queue.

s$msg_receive_reply

Purpose

This subroutine can be used by a requester to receive a reply from a two-way, server or direct queue.

Usage

```
declare port_id                      binary(15);
declare msg_id                       binary(31);
declare reply_length_in              binary(31);
declare reply_length_out             binary(31);
declare reply                        --several-types--;
declare error_code                   binary(15);

declare s$msg_receive_reply entry( binary(15),
                                   binary(31),
                                   binary(31),
                                   binary(31),
                                   --several-types--,
                                   binary(15));

         call s$msg_receive_reply( port_id,
                                   msg_id,
                                   reply_length_in,
                                   reply_length_out,
                                   reply,
                                   error_code);
```

Arguments

- `port_id` (input)

  The identifier of a port attached to the desired queue. `s$msg_receive_reply` receives a reply from the queue.

- `msg_id` (input-output)

  The identifier of the message whose reply is to be received. This argument is ignored if the queue is a direct queue, since there can only be one message connected to the specified port.

- `reply_length_in` (input)

  The length of the output buffer `reply`, in bytes.

- ▶ `reply_length_out` (output)

  The actual length of the reply received.

- ▶ `reply` (output)

  The output buffer containing the reply to the message. The maximum length of a reply is 2^{23} bytes for a queue on the same module as the requester process, 2^{20} bytes for a queue on a different module, or the value of `maximum_msg_size` specified to `s$msg_open_direct` for a direct queue.

- ▶ `error_code` (output)

  A returned error code.

Explanation

For both SERVER and DIRECT queues:

The subroutine `s$msg_receive_reply` returns the reply to a message previously sent to the queue connected to the port `port_id`.

Only a requester can call this subroutine. The queue connected to `port_id` must be a two-way, server or direct queue.

You use `s$msg_receive_reply` together with `s$msg_send` to make a request of a server and to receive the reply. Alternatively, you can use `s$call_server`, which combines the two calls into one.

If the port `port_id` is in `no_wait` mode, then the subroutine returns immediately, and if the reply had not yet been received at the queue, it returns the error code `e$caller_must_wait` (`1277`).

If the port is in `wait` mode, then `s$msg_receive_reply` does not return until the server has replied to the message. If the port has a time limit set on it, the subroutine waits for the time limit, tries again, and if it does not succeed, returns the error code `e$timeout` (`1081`).

If there is no server servicing the queue, the subroutine returns the error code `e$no_msg_server_for_queue` (`2817`).

For SERVER queues only:

The argument `msg_id` must be either -1 or the identifier of a message previously sent to the queue by the calling process. If `msg_id` is -1, then `s$msg_receive_reply` returns the first reply by any server to any message from the caller in `reply`, and it returns the identifier of the message in `msg_id`. Otherwise, the subroutine returns the reply to the message with the specified identifier.

When the subroutine returns a reply, it removes the message and `reply` from the queue.

If `reply_length_in` is zero, `s$msg_receive_reply` discards the reply, removes the message and reply from the queue, and returns an error code of zero to the caller. If `reply_length_in` is nonzero but smaller than the length of the reply, `s$msg_receive_reply` returns a partial reply to the caller, sets `reply_length_out` to the true length of reply, and returns the error code `e$long_record` (`1026`). To receive the entire reply, the server should expand the reply buffer to a size greater than or equal to `reply_length_out`, set `reply_length_in` equal to the new buffer size, and call `s$msg_receive_reply` again.

If a server receives a message and closes the queue without replying to it (`s$msg_send_reply`), the behavior of `s$msg_receive_reply` when the requestor is in `wait` mode depends on three factors:

- whether the server cancels receipt of the message by calling `s$msg_cancel_receive` before closing the queue
- whether additional servers have the queue open
- the priority of the message.

Table 3-26 shows the responses of `s$msg_receive_reply` to these factors.

Table 3-26. Response of `s$msg_receive_reply` to Aborted Replies

Server Cancelled Receipt	**Additional Servers**	**Message Priority**	`s$msg_receive_reply` **Action**
Yes	Yes	N/A	Waits
Yes	No	0-9	Waits
Yes	No	10-19	Returns `e$no_msg_server_for_queue` (`2817`)
No	N/A	N/A	Returns `e$server_aborted` (`2759`)

For DIRECT queues only:

The original message is removed when it is received by the server. No other message may be put into the queue until the reply has been received.

The argument `msg_id` is ignored if the queue is a direct queue.

When the subroutine returns a reply, it removes `reply` from the queue.

If `reply` is longer than `reply_length_in`, `s$msg_receive_reply` returns none of `reply`, returns the true length of `reply` in `reply_length_out`, and returns the error code `e$buffer_too_small` (`1133`). To receive the reply, the server should expand the reply buffer to a size greater than or equal to `reply_length_out`, set `reply_length_in` equal to the new buffer size, and call `s$msg_receive_reply` again. You can avoid this problem entirely if you make the buffer size equal to the `maximum_msg_size` specified in `s$msg_open_direct`. The absolute maximum size is 3072 bytes.

If a server at the queue receives a request but aborts or is stopped while servicing the request, `s$msg_receive_reply` returns the error code `e$server_aborted` (`2759`).

Error Codes

The following lists some error codes this subroutine might return.

`e$record_not_found`
`(1112)`
	The argument `msg_id` does not correspond to a message that is currently in the queue.

`e$invalid_io_operation`
`(1040)`
	There are no outstanding messages pending.

`e$long_record`
`(1026)`
	Record is too long. For a server queue, `reply` is longer than `reply_length_in`.

`e$timeout`
`(1081)`
	Timeout period has expired. `s$receive_reply` failed a second time to receive a reply, after waiting the length of port `port_id`'s time limit.

`e$buffer_too_small`
`(1133)`
	The specified buffer is too small. For a direct queue, `reply` is longer than `reply_length_in`.

`e$caller_must_wait`
`(1277)`
	Caller must wait for operation to complete. The port `port_id` is in `no_wait` mode and a reply has not yet been received at the queue.

`e$server_aborted`
`(2759)`
	Server closed queue file without replying to message. For a message of priority between 10 and 19, the server that received the message stopped or was stopped.

`e$no_msg_server_for_queue`
`2817`
	There is no message server for queue.

`e$drq_connection_broken`
`(3972)`
	The cross-module connection from this port to the queue has been broken. A remote requester port attached to a direct queue has lost connection. The requester should detach the port, close the queue, reattach the port, and reopen the queue. These actions create a new connection if a new connection is possible.

Related Information

`s$call_server`

Is used by a requester to put a message into a two-way queue and to receive a reply from the queue.

`s$msg_open`

Is used by a process to open (to connect to) a server queue, a message queue, or a one-way server queue.

`s$msg_open_direct`

Is used by a process to open (to connect to) a direct queue or a one-way direct queue.

`s$msg_rewrite`
: Can be used by a requester or a server to rewrite a message contained in a message queue.

`s$msg_send`
: Can be used by a `send_only_requester` to put a message into any one-way queue (message, server, or direct) or by a two-way requester to put a message into any two-way queue.

s$msg_rewrite

Purpose

This subroutine can be used by a requester or a server to rewrite a message contained in a message queue.

Usage

```
declare port_id              binary(15);
declare msg_id               binary(31);
declare msg_priority         binary(15);
declare msg_length_in        binary(31);
declare new_msg              --several-types--;
declare error_code           binary(15);

declare s$msg_rewrite entry( binary(15),
                             binary(31),
                             binary(15),
                             binary(31),
                             --several-types--,
                             binary(15));

         call s$msg_rewrite( port_id,
                             msg_id,
                             msg_priority,
                             msg_length_in,
                             new_msg,
                             error_code);
```

Arguments

- `port_id` (input)

  The identifier of a port attached to the desired queue. `s$msg_rewrite` rewrites a message contained in the queue.

- `msg_id` (input)

  The identifier of the message to be rewritten.

- `msg_priority` (input)

  The priority of the message. The value must be either -1 or a valid priority between 0 and 19 inclusive.

▶ msg_length_in (input)

The length of the new message, in bytes. It must be equal to the length of the message identified by msg_id.

▶ new_msg (input)

The input buffer containing the new text of the message that is to replace the message contained in the queue. The length of new_msg, in bytes, must be at least msg_length_in.

▶ error_code (output)

A returned error code.

Explanation

The subroutine s$msg_rewrite replaces the message msg_id in the queue connected to the port port_id with new_msg.

Either a requester or a server can call this subroutine. The queue must be a message queue.

The argument msg_priority must be either -1 or a valid priority. If it is -1 or the existing priority of the message, then s$msg_rewrite does not change the priority of the message or the position of the message in the queue. However, if msg_priority is different from the existing priority of the message, then the rewritten message is repositioned in the queue.

When an unprivileged process rewrites a message that was originally written by a privileged process, either as a requestor or a server, OpenVOS clears the privileged flag in the message header. This behavior applies even when the person name is the same, and is necessary to prevent a "Trojan Horse" program from using an unprivileged process to rewrite a message that was originally written by a privileged process.

Error Codes

The following is an error code this subroutine might return.

e$message_not_yours (2753)	The specified message has not been received on this port. The message was not marked busy or had not been received by the caller.
e$invalid_record_size (1053)	The length of the new message is not equal to the length of the existing message.
e$invalid_msg_priority (2816)	The specified priority is invalid.

Related Information

s$call_server

Is used by a requester to put a message into a two-way queue and to receive a reply from the queue.

s$msg_cancel_receive

Can be used by a server to cancel the receipt of a message contained in a two-way server queue; or can be used by a server or a requester to cancel the receipt of a message contained in a message queue.

`s$msg_delete`
Can be used by a requester or a server to delete a message from a message queue.

`s$msg_open`
Is used by a process to open (to connect to) a two-way server queue, a message queue, or a one-way server queue.

`s$msg_open_direct`
Is used by a process to open (to connect to) a direct queue.

`s$msg_read`
Can be used by a requester or a server to read a message contained in a queue.

`s$msg_receive`
Can be used by a requester or a server to receive a message contained in a queue.

`s$msg_receive_reply`
Can be used by a requester to receive a reply from a two-way server queue or a two-way direct queue.

`s$msg_send`
Can be used by a `send_only_requester` to put a message into any one-way queue (message, server, or direct) or by a two-way requester to put a message into any two-way queue.

`s$msg_send_reply`
Can be used by a server to put a reply into a two-way server queue or a two-way direct queue.

`s$truncate_queue`
Can be used by a server to truncate an empty message queue.

s$msg_send

Purpose

This subroutine can be used by a send-only requester to put a message into any one-way queue (message, server, or direct) or by a requester to put a message into any two-way queue.

Usage

```
declare port_id           binary(15);
declare msg_priority      binary(15);
declare msg_subject       char(32) varying;
declare msg_length_in     binary(31);
declare msg               --several-types--;
declare msg_id            binary(31);
declare error_code        binary(15);

declare s$msg_send entry( binary(15),
                          binary(15),
                          char(32) varying,
                          binary(31),
                          --several-types--,
                          binary(31),
                          binary(15));

         call s$msg_send( port_id,
                          msg_priority,
                          msg_subject,
                          msg_length_in,
                          msg,
                          msg_id,
                          error_code);
```

Arguments

- `port_id` (input)
  The identifier of a port attached to the desired queue. `s$msg_send` puts a message into the queue.

- `msg_priority` (input)
  The priority of the message. The value must be between 0 and 19, inclusive.

- `msg_subject` (input)
  The subject of the message. This value is put into the message header.

- `msg_length_in` (input-output)
  The length of the message, in bytes. This value cannot be greater than the length of the input buffer `msg`. The maximum length of a message is 2^{23} bytes for a queue on the same module as the calling process, 2^{20} bytes for a queue on a different module, or the value of `maximum_msg_size` specified to `s$msg_open_direct` for a direct queue. On input, the argument specifies the length of the message to send. On output, the argument returns the exact length of the message that was sent.

- `msg` (input)
  The input buffer containing the message to be put into the queue. The length of `msg`, in bytes, must be at least `msg_length_in`.

- `msg_id` (output)
  The identifier of the message that the subroutine sends. In the case of direct queues, this argument is always set to 0.

- `error_code` (output)
  A returned error code.

Explanation

For ALL queues:

The subroutine `s$msg_send` puts the message contained in `msg` into the queue connected to the port `port_id`.

Only a requester can call this subroutine. The queue connected to `port_id` can be any type.

A process can send messages to a queue if the process has read, write, or execute access to the queue.

The `s$msg_send` subroutine includes the `msg_length_in` argument, which is both an input and output argument. On input, the argument specifies the length of the message to send. When the `s$msg_send` subroutine returns, the kernel changes the value of this argument to show the exact length of the message that was sent. In most cases, the two values will be the same. However, if an error prevents the kernel from sending the message, the kernel changes the value of this argument to `0`. In this case, the calling routine must reestablish the correct message length value for the variable before repeating the call to the `s$msg_send` subroutine. For example, an error that prevents the kernel from sending the message occurs when the `s$msg_send` subroutine is called in `no_wait` mode for a queue that already has a number of pending messages equal to the established maximum queue depth. (The `s$set_max_queue_depth` subroutine sets the maximum queue depth for the maximum number of messages of a server or one-way server queue.)

For SERVER queues only:

If `msg_priority` is between 0 and 9, and if no server is active at the queue, `s$msg_send` places the message in the queue and returns an error code of 0.

If `msg_priority` is between 10 and 19, and if no server is active at a queue or if all servers at the queue are stopped before any can receive the request by taking it out of the queue, `s$msg_send` returns the error code `e$no_msg_server_for_queue` (`2817`).

Otherwise, s$msg_send immediately returns the identifier of the message in msg_id. The caller can use that value to receive a reply.

> **Caution:** On a one-way server queue, if the port is in wait mode and a requester is waiting, the requester will not be notified with e$no_msg_server_for_queue if the server is terminated after the requester begins waiting. You should design the application to handle this situation.

For MESSAGE queues only:

Regardless of priority or whether a server is serving the queue, the message is put into the queue and returns immediately.

For DIRECT queues only:

If no server is active at a queue or if all servers at the queue are stopped before any can receive the request by taking it out of the queue, s$msg_send returns the error code e$no_msg_server_for_queue (2817). However, if the queue is on a different module from the requester, s$msg_send returns an error code of 0 for the first message it tries to send after the server has stopped. A subsequent call to s$msg_send (for a one-way queue) or s$msg_receive_reply (for a two-way queue) will then return e$no_msg_server_for_queue.

To recover from the error code e$no_msg_server_for_queue (2817) when using a one-way direct queue, a server must be attached to the queue, and the requester port that received the error code must be closed and reopened. If the requester port is not closed and reopened, s$msg_send continues to return the e$no_msg_server_for_queue (2817) error code, regardless of whether a server is attached.

For one-way direct queues, if a previous message from the same requester has not yet been received and the port is in wait mode, s$msg_send waits to put the message in the queue. However, if there is a pending message and the port is in no-wait mode, s$msg_send returns immediately with the message e$caller_must_wait (1277).

Otherwise, s$msg_send places the message in the queue and returns immediately.

Error Codes

The following lists some error codes this subroutine might return.

`e$invalid_io_operation (1040)`	Invalid I/O operation for current port state or attachment. For a two-way direct queue, you attempted to send a message when the reply to the previous message had not yet been sent and received.
`$no_msg_server_for_queue (2817)`	There is no message server for queue. (See the text above for further explanation.)
`e$drq_connection_broken (3972)`	The cross-module connection from this port to the queue has been broken. A remote requester port attached to a direct queue has lost connection. The requester should detach the port, close the queue, reattach the port, and reopen the queue. These actions create a new connection if a new connection is possible.

Related Information

`s$call_server`
Is used by a requester to put a message into a two-way queue and to receive a reply from the queue.

`s$msg_open`
Is used by a process to open (to connect to) a server queue, a message queue, or a one-way server queue.

`s$msg_open_direct`
Is used by a process to open (to connect to) a direct queue or a one-way direct queue.

`s$msg_receive_reply`
Can be used by a requester to receive a reply from a two-way, server or direct queue.

`s$msg_rewrite`
Can be used by a requester or a server to rewrite a message contained in a message queue.

s$msg_send_reply

Purpose

`s$msg_send_reply` can be used by a server to put a reply into a two-way, server or direct queue.

Usage

```
declare port_id                 binary(15);
declare msg_id                  binary(31);
declare reply_length_in         binary(31);
declare reply                   --several-types--;
declare error_code              binary(15);

declare s$msg_send_reply entry( binary(15),
                                binary(31),
                                binary(31),
                                --several-types--,
                                binary(15));

         call s$msg_send_reply( port_id,
                                msg_id,
                                reply_length_in,
                                reply,
                                error_code);
```

Arguments

▶ `port_id` (input)

The identifier of a port attached to the desired queue. `s$msg_send_reply` puts a reply into the queue.

▶ `msg_id` (input)

The identifier of the message to which the contents of `reply` is the reply. This argument is ignored if the queue is a direct queue.

▶ `reply_length_in` (input)

The length of the reply, in bytes. This value cannot be greater than the length of the input buffer `reply`. The maximum length of a message is 2^{23} bytes for a queue on the same module as the calling process, 2^{20} bytes for a queue on a different module, or the value of `maximum_msg_size` specified to `s$msg_open_direct` for a direct queue.

- ▶ reply (input)
  The input buffer containing the reply to the message. The length of reply, in bytes, must be at least reply_length_in.

- ▶ error_code (output)
  A returned error code.

Explanation

The subroutine s$msg_send_reply puts the specified reply into the queue connected to the port port_id. The reply must be sent by the same server and on the same port on which the original message was received.

A server can call this subroutine. The queue connected to port_id must be a two-way server or a two-way direct queue.

For server queues, msg_id must be the identifier of a message already received on port_id.

For direct queues, msg_id is ignored, since only one message can exist on port_id at a time.

Related Information

s$msg_cancel_receive
Can be used by a server to cancel the receipt of a message contained in a two-way server queue, or by a server or a requester to cancel the receipt of a message contained in a message queue.

s$msg_delete
Can be used by a requester or a server to delete a message from a message queue.

s$msg_open
Is used by a process to open (to connect to) a server queue, a message queue, or a one-way server queue.

s$msg_open_direct
Is used by a process to open (to connect to) a direct queue or a one-way direct queue.

s$msg_read
Can be used by a requester or a server to read a message contained in a queue.

s$msg_receive
Can be used by a requester or a server to receive a message contained in a queue.

s$msg_rewrite
Can be used by a requester or a server to rewrite a message contained in a message queue.

s$set_max_queue_depth
Sets the maximum queue depth (or maximum number of messages) of a server queue or one-way server queue.

`s$truncate_queue`
Can be used by a server to truncate an empty message queue.

s$reschedule_task

Purpose

This subroutine tells the task manager to dispatch another ready task.

Usage

```
declare s$reschedule_task entry;

         call s$reschedule_task;
```

Arguments

None

Explanation

The subroutine s$reschedule_task switches tasks, if there is another task ready. The state of the calling task changes from running to ready.

If no other task is in the ready state when s$reschedule_task is called, then the task manager dispatches the calling task again. In any case, the caller will eventually be scheduled again.

This subroutine is useful for breaking up large tasks so other tasks can run.

Related Information

s$control_task
: Performs one of a variety of functions on a specified task, depending on the value of action_code given.

s$delete_task
: Deletes a dynamic task.

s$enable_tasking
: Enables or disables tasking in the calling process.

s$init_task
: Initializes a specified uninitialized task.

`s$init_task_config`
Reads a task configuration file, initializes the set of tasks described in the configuration file, and puts the tasks into the `ready` state. If a primary task (a task with identifier 1) is defined in the configuration file, then that task is given control.

`s$monitor`
Reads and carries out requests for the administration of a tasking process.

`s$monitor_full`
Like `s$monitor`, reads and carries out requests for the administration of a tasking process. It provides additional control, including allowing the caller to specify an initial request, with an option to return from the subroutine immediately after executing the request.

`s$reschedule_task`
Tells the task manager to dispatch another ready task.

`s$set_process_terminal`
Attaches a task's five predefined ports (`default_input`, `default_output`, `command_input`, `terminal_output`, and `terminal`), either to the process terminal or to the task's terminal.

`s$set_task_priority`
Sets the scheduling priority for a task.

`s$set_task_terminal`
Changes the terminal port attachment for the running task.

`s$start_task`
Makes an initialized or stopped task ready to run.

`s$start_task_full`
Makes an initialized or stopped task ready to run, passing it one or more initial arguments.

s$set_default_lock_wait_time

Privileged

Purpose

This subroutine sets the default maximum time (in units of 1/1024 of a second) that a task or process will wait to acquire an implicit lock during any I/O operation. The default time applies to all tasks and processes running on the specified module (unless the lock wait time has been specifically changed from the default) and to all types of I/O.

Usage

```
declare module_name                         char(66) varying;
declare lock_wait_time                      binary(31);
declare error_code                          binary(15);

declare s$set_default_lock_wait_time entry( char(66) varying,
                                            binary(31),
                                            binary(15));

         call s$set_default_lock_wait_time( module_name,
                                            lock_wait_time,
                                            error_code);
```

Arguments

▶ `module_name` (input)
The default is the name of the current module. Otherwise, specify the path name of the desired module.

▶ `lock_wait_time` (input)
The default lock wait time for the specified module, expressed in units of 1/1024 of a second.

▶ `error_code` (output)
A returned error code.

Explanation

Lock wait time is the amount of time a process or task will wait for an implicit lock on a file, record, or key. If the process or task has to wait longer, then it gives up and returns one of the following error codes depending on the type of lock sought:

```
e$file_in_use     (1084)
e$record_in_use   (2408)
e$key_in_use      (2918)
```

The default module lock wait time set by OpenVOS is 0 seconds. To change it, call `s$set_lock_wait_time` (which sets the lock wait time for the current program or process) or `s$set_default_lock_wait_time` (which sets the default lock wait time for a module). 10 seconds is a typical lock wait time. A program uses the program lock wait time if one has been set. Otherwise it looks for a process lock wait time, and if that has not been set it uses the module lock wait time.

This is a privileged subroutine. Therefore, it can be called only by a privileged process.

Related Information

`s$get_default_lock_wait_time`

Returns the default maximum time (in units of 1/1024 of a second) that a task or process will wait to acquire an implicit lock during any I/O operation. The default time applies to all tasks and processes running on the specified module (unless the lock wait time has been specifically changed from the default) and to all types of I/O.

`s$get_lock_wait_time`

Returns the maximum time (in units of 1/1024 of a second) that the current program or process will wait to acquire an implicit lock during any I/O operation.

`s$set_default_lock_wait_time`

Sets the default maximum time (in units of 1/1024 of a second) that a task or process will wait to acquire an implicit lock during any I/O operation. The default time applies to all tasks and processes running on the specified module (unless the lock wait time has been specifically changed from the default) and to all types of I/O.

`s$set_lock_wait_time`

Sets the maximum time (in units of 1/1024 of a second) that the current program or process will wait to acquire an implicit lock during any I/O operation.

s$set_lock_wait_time

Purpose

This subroutine sets the maximum time (in units of 1/1024 of a second) that the current program or process will wait to acquire an implicit lock during any I/O operation.

Usage

```
declare program_or_process          binary(15);
declare lock_wait_time              binary(31);
declare error_code                  binary(15);

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

         call s$set_lock_wait_time( program_or_process,
                                    lock_wait_time,
                                    error_code);
```

Arguments

▶ `program_or_process` (input)

There are two permitted values:

`0` set lock wait time for program
`1` set lock wait time for process.

▶ `lock_wait_time` (input)

The time, in units of 1/1024 of a second, that the program or process will wait for a lock before timeout occurs.

▶ `error_code` (output)

A returned error code.

Explanation

Lock wait time is the amount of time a process or task will wait for an implicit lock on a file, record, or key. If the process or task has to wait longer, then it gives up and returns one of the following error codes, depending on the type of lock sought:

```
e$file_in_use     (1084)
e$record_in_use   (2408)
e$key_in_use      (2918).
```

The default module lock wait time set by OpenVOS is 0 seconds. To change it call `s$set_lock_wait_time` (which sets the lock wait time for the current program or process) or `s$set_default_lock_wait_time` (which sets the default lock wait time for a module). 10 seconds is a typical lock wait time. A program uses the program lock wait time if one has been set. Otherwise it looks for a process lock wait time, and if that has not been set it uses the module lock wait time.

Related Information

`s$get_default_lock_wait_time`
Returns the default maximum time (in units of 1/1024 of a second) that a task or process will wait to acquire an implicit lock during any I/O operation. The default time applies to all tasks and processes running on the specified module (unless the lock wait time has been specifically changed from the default) and to all types of I/O.

`s$get_lock_wait_time`
Returns the maximum time (in units of 1/1024 of a second) that the current program or process will wait to acquire an implicit lock during any I/O operation.

`s$set_default_lock_wait_time`
Sets the default maximum time (in units of 1/1024 of a second) that a task or process will wait to acquire an implicit lock during any I/O operation. The default time applies to all tasks and processes running on the specified module (unless the lock wait time has been specifically changed from the default) and to all types of I/O.

`s$set_lock_wait_time`
Sets the maximum time (in units of 1/1024 of a second) that the current program or process will wait to acquire an implicit lock during any I/O operation.

`s$set_max_queue_depth`

Purpose

This subroutine sets the maximum queue depth (or maximum number of messages) of a server or one-way server queue.

Usage

```
declare path_name                    char(256) varying;
declare max_queue_depth              binary(31);
declare error_code                   binary(15);

declare s$set_max_queue_depth entry( char(256) varying,
                                     binary(31),
                                     binary(15));

         call s$set_max_queue_depth( path_name,
                                     max_queue_depth,
                                     error_code);
```

Arguments

- `path_name` (input)
  The name of a server queue or one-way server queue. Only server queue and one-way server queue files are valid; otherwise, `e$invalid_file_type` (`1052`) is returned.

- `max_queue_depth` (input)
  The maximum queue depth for the server queue or one-way server queue. Valid values range from 1 to 32767 inclusive; any other value results in the error `e$out_of_range` (`1038`).

- `error_code` (output)
  A returned status code.

Explanation

The default `max_queue_depth` for any server queue or one-way server queue is 256.

The maximum queue depth is reached when the total number of messages equals `max_queue_depth`. At this point, the distribution of messages in the queue determines whether a message may be added.

A priority level is full if the number of messages at that priority is greater than or equal to:

```
max (number_of_servers, 4).
```

Once the `max_queue_depth` has been reached, new messages may be added to the server queue only at priority levels higher than the highest filled priority. If priority level 19 (the highest level) is filled, no more messages can be added to the queue.

Note: The only case in which the number of messages at any given priority is restricted is when the number of messages in the server queue is greater than or equal to `max_queue_depth`. Thus, if the queue has fewer than `max_queue_depth` messages in it, the number of messages at a given priority level is unlimited.

The action taken for a message that cannot be added to the queue depends on whether the caller is in wait or no-wait mode, or if an I/O time limit is set:

- If the caller is in wait mode, the application waits until the message can be added to the queue.
- If the caller is in no-wait mode, the error `e$caller_must_wait` (`1277`) is returned to the application.
- If an I/O time limit is set, the error `e$timeout` (`1081`) is returned to the application and all messages sent on the port with the I/O time limit are removed from the queue.

For more information about I/O time limits, see the description of `s$set_io_time_limit` in the OpenVOS Subroutines manuals.

You can get the current `max_queue_depth` for a server queue or one-way server queue using `s$get_file_status` or `s$get_open_file_info` with `version` set to `4`. See the OpenVOS Subroutines manuals.

Related Information

`s$msg_cancel_receive`
Can be used by a server to cancel the receipt of a message contained in a two-way server queue, or by a server or a requester to cancel the receipt of a message contained in a message queue.

`s$msg_delete`
Can be used by a requester or a server to delete a message from a message queue.

`s$msg_open`
Is used by a process to open (to connect to) a server queue, a message queue, or a one-way server queue.

`s$msg_open_direct`
Is used by a process to open (to connect to) a direct queue or a one-way direct queue.

`s$msg_read`
Can be used by a requester or a server to read a message contained in a queue.

`s$msg_receive`
Can be used by a requester or a server to receive a message contained in a queue.

`s$msg_rewrite`
Can be used by a requester or a server to rewrite a message contained in a message queue.

`s$msg_send_reply`
Can be used by a server to put a reply into a two-way server queue or a two-way direct queue.

`s$set_max_queue_depth`
Sets the maximum queue depth (or maximum number of messages) of a server queue or one-way server queue.

`s$truncate_queue`
Can be used by a server to truncate an empty message queue.

s$set_process_terminal

Purpose

This subroutine attaches a task's five predefined ports (`default_input`, `terminal_output`, `command_input`, `default_output`, and `terminal`) either to the task's process terminal or to the task's terminal.

Usage

```
declare terminal_switch                binary(15);

declare s$set_process_terminal entry( binary(15));

         call s$set_process_terminal( terminal_switch);
```

Arguments

- `terminal_switch` (input-output)

  On input this switch indicates the terminal to which `s$set_process_terminal` is to attach the predefined ports of the calling task. On output it indicates to which terminal the ports were attached immediately before this call was made.

Explanation

The subroutine `s$set_process_terminal` attaches the calling task's five predefined ports (`default_input`, `terminal_output`, `command_input`, `default_output`, and `terminal`) either to the task's process terminal or to the task's terminal.

A task's process terminal is the terminal of the process executing the task.

When a task is initialized, its predefined ports are attached to its terminal. If the task must read from or write to the process terminal, you can use this call to attach to the process terminal.

Before calling `s$set_process_terminal` for the first time, set `terminal_switch` equal to `1`. If `terminal_switch` is `1` on input, the subroutine attaches the ports to the process terminal of the task and disables tasking in the process. From this point on, always call `s$set_process_terminal` with `terminal_switch` set to the value returned by the previous call, thus returning the task to the previous state (either ports attached to the task's terminal and tasking enabled or ports attached to the process terminal and tasking disabled).

A task that calls `s$set_process_terminal(1)` and subsequently calls either `s$sleep` or `s$wait_event` will cause the process to suspend execution rather than the calling task. For information about `s$sleep` and `s$wait_event`, see the OpenVOS Subroutines manuals.

Related Information

`s$control_task`
Performs one of a variety of functions on a specified task, depending on the value of `action_code` given.

`s$delete_task`
Deletes a dynamic task.

`s$enable_tasking`
Enables or disables tasking in the calling process.

`s$init_task`
Initializes a specified `uninitialized` task.

`s$init_task_config`
Reads a task configuration file, initializes the set of tasks described in the configuration file, and puts the tasks into the `ready` state. If a primary task (a task with identifier 1) is defined in the configuration file, then that task is given control.

`s$monitor`
Reads and carries out requests for the administration of a tasking process.

`s$monitor_full`
Like `s$monitor`, reads and carries out requests for the administration of a tasking process. It provides additional control, including allowing the caller to specify an initial request, with an option to return from the subroutine immediately after executing the request.

`s$reschedule_task`
Tells the task manager to dispatch another ready task.

`s$set_process_terminal`
Attaches a task's five predefined ports (`default_input`, `default_output`, `command_input`, `terminal_output`, and `terminal`), either to the process terminal or to the task's terminal.

`s$set_task_priority`
Sets the scheduling priority for a task.

`s$set_task_terminal`
Changes the terminal port attachment for the running task.

`s$start_task`
Makes an initialized or stopped task ready to run.

`s$start_task_full`
Makes an initialized or stopped task ready to run, passing it one or more initial arguments.

s$set_task_priority

Purpose

This subroutine sets the scheduling priority for a task.

Usage

```
declare task_id                    binary(15);
declare priority                   binary(15);
declare error_code                 binary(15);

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

         call s$set_task_priority( task_id,
                                   priority,
                                   error_code);
```

Arguments

- `task_id` (input)

  The task identifier of the task whose priority is to be set.

- `priority` (input)

  A priority value in the range 0 to 255 inclusive.

- `error_code` (output)

  A returned error code.

Explanation

The subroutine `s$set_task_priority` sets the scheduling priority of a task. You can set the priority of both static and dynamic tasks.

A task with a higher-numbered priority is scheduled before a task with a lower-numbered priority. Also, events associated with a task having a higher-numbered priority are notified before those associated with a task having a lower-numbered priority. For example, suppose that two tasks with unequal priorities are waiting for terminal input. If the terminals attached to each task receive input simultaneously, the event associated with the terminal attached to the task having higher priority is notified first.

A priority change on a running task may cause a task switch. The running task will be preempted if a task in the ready state has a higher priority. A task in any other state will not preempt the running task. To avoid the possibility of a task switch, call `s$set_task_priority` before starting the task.

Error Codes

The following lists some error codes this subroutine might return.

`e$invalid_task_id` (2575)	Invalid task ID. The value specified for `task_id` does not correspond to a task that currently exists in the process.
`e$invalid_priority` (4009)	Task priorities must be in the range of 0 to 255 inclusive.
`e$invalid_task_data_region` (4011)	The task data region has been corrupted. This code is not returned in `error_code` but an error handler can access it via the `oncode` built-in function.

Related Information

`s$control_task`
Performs one of a variety of functions on a specified task, depending on the value of `action_code` given.

`s$delete_task`
Deletes a dynamic task.

`s$enable_tasking`
Enables or disables tasking in the calling process.

`s$init_task`
Initializes a specified `uninitialized` task.

`s$init_task_config`
Reads a task configuration file, initializes the set of tasks described in the configuration file, and puts the tasks into the `ready` state. If a primary task (a task with identifier 1) is defined in the configuration file, then that task is given control.

`s$monitor`
Reads and carries out requests for the administration of a tasking process.

`s$monitor_full`
Like `s$monitor`, reads and carries out requests for the administration of a tasking process. It provides additional control, including allowing the caller to specify an initial request, with an option to return from the subroutine immediately after executing the request.

`s$reschedule_task`
Tells the task manager to dispatch another ready task.

`s$set_process_terminal`
: Attaches a task's five predefined ports (`default_input`, `default_output`, `command_input`, `terminal_output`, and `terminal`), either to the process terminal or to the task's terminal.

`s$set_task_terminal`
: Changes the terminal port attachment for the running task.

`s$start_task`
: Makes an initialized or stopped task ready to run.

`s$start_task_full`
: Makes an initialized or stopped task ready to run, passing it one or more initial arguments.

s$set_task_terminal

Purpose

This subroutine changes the terminal port attachment for the running task.

Usage

```
declare terminal_name               char(256) varying;
declare error_code                  binary(15);

declare s$set_task_terminal entry(  char(256) varying,
                                    binary(15));

        call s$set_task_terminal(   terminal_name,
                                    error_code);
```

Arguments

- ▶ `terminal_name` (input)
  The device name of the new task terminal. If you specify a null value, the task will have no default port attachments.

- ▶ `error_code` (output)
  A returned error code.

Explanation

The subroutine `s$set_task_terminal` changes the terminal port attachment for the running task. It closes the current terminal and detaches its port, then attaches the specified terminal and opens it as the task terminal.

If an error occurs during the attempt to open the new terminal, the task is left with no terminal attachment. At this point you can call `s$set_task_terminal` again to reattach the task to the original terminal or to another valid terminal.

> **Caution:** Input or output may be lost if you use `s$set_task_terminal` in conjunction with PL/I I/O. You can avoid this problem by using OpenVOS I/O subroutine calls (see OpenVOS Subroutines manuals) instead of PL/I I/O statements.

Error Codes

The following is an error code this subroutine might return.

`e$invalid_task_data_region (4011)`	The task data region has been corrupted. This code is not returned in `error_code` but an error handler can access it via the `oncode` built-in function.

Related Information

`s$control_task`
Performs one of a variety of functions on a specified task, depending on the value of `action_code` given.

`s$delete_task`
Deletes a dynamic task.

`s$enable_tasking`
Enables or disables tasking in the calling process.

`s$init_task`
Initializes a specified `uninitialized` task.

`s$init_task_config`
Reads a task configuration file, initializes the set of tasks described in the configuration file, and puts the tasks into the `ready` state. If a primary task (a task with identifier 1) is defined in the configuration file, then that task is given control.

`s$monitor`
Reads and carries out requests for the administration of a tasking process.

`s$monitor_full`
Like `s$monitor`, reads and carries out requests for the administration of a tasking process. It provides additional control, including allowing the caller to specify an initial request, with an option to return from the subroutine immediately after executing the request.

`s$reschedule_task`
Tells the task manager to dispatch another ready task.

`s$set_process_terminal`
Attaches a task's five predefined ports (`default_input`, `default_output`, `command_input`, `terminal_output`, and `terminal`), either to the process terminal or to the task's terminal.

`s$set_task_priority`
Sets the scheduling priority for a task.

`s$set_task_terminal`
Changes the terminal port attachment for the running task.

`s$start_task`
Makes an initialized or stopped task ready to run.

`s$start_task_full`
Makes an initialized or stopped task ready to run, passing it one or more initial arguments.

s$set_task_wait_info

Purpose

This subroutine defines a number of groups of events to be used by `s$task_setup_wait` and `s$task_wait_event`.

Usage

```
declare macro_event_id              binary(31);
declare number_tasks                binary(15);
declare switches                    binary(15);
declare error_code                  binary(15);

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

         call s$set_task_wait_info( macro_event_id,
                                    number_tasks,
                                    switches,
                                    error_code);
```

Arguments

- `macro_event_id` (input)

  The `event_id` of an unnamed event, previously obtained from `s$attach_event` (see the OpenVOS Subroutines manuals).

- `number_tasks` (input)

  The number of “tasks” for which events are to be created. The maximum number is 4096. However, if the number of tasks waiting on an event exceeds 512, the next task to wait for that event will receive an `e$too_many_events` in response to the subroutine call that causes the wait to occur. (Since there is an OpenVOS limit of 512 subevents on a macro event, only 512 tasks can be waiting on an event.) In general, if tasks will be waiting on events in your program, it is best to use 512 as the maximum number of tasks for this argument.

- `switches` (input)
  A group of switches allowing certain decisions: see the Explanation section for details.

- `error_code` (output)
  A returned error code.

Explanation

The subroutine `s$set_task_wait_info` defines a number of groups of events to be used by `s$task_setup_wait` and `s$task_wait_event`. Each group is associated with a logical "task." These differ from true tasks, created by the binder or by calling `s$create_task`. Notification of an event in a given group might lead to the execution of a section of code associated with the corresponding "task."

The value of `switches` is a binary coding of two logical variables. The variables are coded in the "1" and "2" bits of `switches`. The other 14 bits are reserved for future use.

Bit	Switch Name	Explanation
1	`update_event_count`	If this bit is `true`, then the count for each event in the array of events is updated each time one of those events is notified.
2	`keep_time_out`	If this bit is `true`, then the timeout remains the same for any particular "task," no matter how many times events associated with that "task" are notified.

The following example illustrates the effect of the `keep_time_out` switch.

1. You call `s$set_task_wait_info` to define one "task" and set `keep_time_out` to `true`.

2. You call `s$task_setup_wait` to associate some events with the "task" and set its timeout to 5 milliseconds from now.

3. You call `s$task_wait_event` to wait on the events.

4. 2 milliseconds after you called `s$task_setup_wait`, one of the events is notified; so `s$task_wait_event` returns.

5. You immediately call `s$task_wait_event` to wait on the events again.

6. No event associated with the "task" is notified before another 3 milliseconds has elapsed; so the "task" times out 5 milliseconds after `s$task_setup_wait` was called and `s$task_wait_event` returns.

   If the example is changed to set `keep_time_out` to `false`, then you must redefine the timeout after each notify, using `s$task_setup_wait`.

Then steps 1, 5, and 6 change, and an additional step is required.

1. You call `s$set_task_wait_info` to define one "task" and set `keep_time_out` to `false`.

   .
   .
   .

2. You call `s$task_setup_wait` again to set the timeout for the "task" to 5 milliseconds from **now**.

3. You immediately call `s$task_wait_event` to wait on the events again.

4. No event associated with the "task" is notified before another 5 milliseconds has elapsed; so the "task" times out 5 milliseconds after `s$task_setup_wait` was called the **second** time and `s$task_wait_event` returns.

Related Information

`s$task_setup_wait`
: Specifies which events are associated with one of the "tasks" defined by `s$set_task_wait_info` and specifies a timeout for that "task."

`s$task_wait_event`
: Causes a process to wait until any one of the events (specified by `s$task_setup_wait`) associated with any one of the "tasks" defined by `s$set_task_wait_info` is notified.

s$set_tp_default_parameters *Privileged*

Purpose

This subroutine sets the default lock contention parameters for transaction locking for a specified module.

Usage

```
declare module_name                        char(66) varying;

declare 1 parameter_info                   shortmap,
          2 version                        binary(15),
          2 transaction_priority           char(1),
          2 switches                       char(1),
          2 time_value                     binary(15);

declare error_code                         binary(15);

declare s$set_tp_default_parameters entry( char(66) varying,
                                           1 like parameter_info,
                                           binary(15));

         call s$set_tp_default_parameters( module_name,
                                           parameter_info,
                                           error_code);
```

Arguments

▶ `module_name` (input)

The name of the module for which the default lock contention parameters are to be set.

▶ `parameter_info` (input)

Information about how the parameters should be set.

▶ `version` (input)

The version number of the `parameter_info` structure. This must be set to 1.

▶ `transaction_priority` (input)

The default priority given to a transaction by `s$start_transaction`. `s$set_tp_default_parameters` accepts `transaction_priority` as a hexadecimal value. The value can be from 0 to 9.

- `switches` (input)
  Information about how the parameters are to be set. See the Explanation section for details.

- `time_value` (input)
  The number of seconds OpenVOS uses to calculate precedence between two transactions. If the amount of time between the beginning of two transactions differs by less than `time_value`, then the transactions are considered to have started at the same time, and neither is older than the other.

- `error_code` (output)
  A returned error code.

Explanation

This `s$set_tp_default_parameters` subroutine sets the default lock contention parameters for transaction locking for a specified module. You select the contention parameters by setting value for the `transaction_priority` argument and the bits for the switch names in the `switches` argument.

`transaction_priority`	0
`ignore_priority`	false
`ignore_time`	false
`younger_wins`	false
`allow_deadlocks`	false
`time_value`	10 seconds.

The `ignore_priority`, `ignore_time`, `younger_wins`, and `allow_deadlocks` parameters are contained in the argument `switches`. The value of switches is a binary coding of four variables. The variables are coded in the “128” bit, the “64” bit, the “16” bit, and the “8” bit switches. In addition, three bits are reserved for future use.

(Page 1 of 2)

Bit	Switch Name	Explanation
128	`ignore_priority`	If the bit is set to `true` for both transactions contending for a lock, their priorities are ignored by OpenVOS in deciding which transaction wins.
64	`ignore_time`	If the bit is set to `true` for both transactions contending for a lock, then the time that each transaction began is ignored by OpenVOS in deciding which transaction wins.
32		The bit is reserved for use by OpenVOS.
16	`younger_wins`	If the bit is set to `true` for both transactions contending for a lock, then the one which began last wins.

(Page 2 of 2)

Bit	Switch Name	Explanation
8	`allow_deadlocks`	If the bit is set to `true` for both of two transactions contending for a lock, then OpenVOS ignores any deadlock occurring between them. If a deadlock occurs, OpenVOS does not abort either transaction; instead, it returns the error `e$record_in_use` (`2408`). When this happens, you must abort the transaction. If you do not do so, the deadlock could continue indefinitely, until one of the transactions is aborted. For this reason, the use of this option is not recommended. If `allow_deadlocks` is `false` (the default) for either transaction involved in a deadlock, OpenVOS breaks the deadlock by choosing a winner based on the value of an internal transaction identifier. In most cases the older transaction is the winner. However, if both transactions have `younger_wins` set to `true`, then the younger transaction is the winner. In either case, if one of the transactions has lost a deadlock immediately before, it wins this time.

See `s$set_tp_parameters` for information on how to encode the bit values in `switches`.

The last parameter listed above (`time_value`) is contained in the `parameter_info` structure.

Related Information

`s$get_tp_default_parameters`
Returns the default lock contention parameters for transaction locking on a specified module.

`s$get_tp_parameters`
Returns the lock contention parameters for transaction locking for the current program or process.

`s$set_tp_default_parameters`
Sets the default lock contention parameters for transaction locking for a specified module.

`s$set_tp_parameters`
Sets the lock contention parameters for transaction locking for a specified program or process.

s$set_tp_parameters

Purpose

This subroutine sets the lock contention parameters for transaction locking for a specified program or process.

Usage

```
declare program_or_process         binary(15);

declare 1 parameter_info           shortmap,
          2 version                binary(15),
          2 transaction_priority   char(1),
          2 switches               char(1),
          2 time_value             binary(15);

declare error_code                 binary(15);

declare s$set_tp_parameters entry( binary(15),
                                   1 like parameter_info,
                                   binary(15));

         call s$set_tp_parameters( program_or_process,
                                   parameter_info,
                                   error_code);
```

Arguments

- `program_or_process` (input)

  If set to 0, then the subroutine sets the lock contention parameters for the current program. If set to anything but 0, then the subroutine sets the parameters for the current process.

- `parameter_info` (input)

  Information about how the parameters should be set.

- `version` (input)

  The version number of the `parameter_info` structure. This must be set to 1.

- `transaction_priority` (input)

  The default priority given to a transaction by `s$start_transaction`. `s$set_tp_parameters` accepts `transaction_priority` as a hexadecimal value. The value can be from 0 to 9.

- ▶ `switches` (input)

  Information about how the parameters are to be set. See the Explanation section for details.

- ▶ `time_value` (input)

  The number of seconds OpenVOS uses to calculate precedence between two transactions. If the amount of time between the beginning of two transactions differs by less than `time_value`, then the transactions are considered to have started at the same time, and neither is older than the other.

- ▶ `error_code` (output)

  A returned error code.

Explanation

This `s$set_tp_parameters` subroutine sets the default lock contention parameters for transaction locking for a specified program or process. You select the contention parameters by setting value for the `transaction_priority` argument and the bits for the switch names in the `switches` argument.

`transaction_priority`	0
`ignore_priority`	false
`ignore_time`	false
`younger_wins`	false
`allow_deadlocks`	false
`time_value`	10 seconds.

The `ignore_priority`, `ignore_time`, `younger_wins`, and `allow_deadlocks` parameters are contained in the argument `switches`. The value of switches is a binary coding of four variables. The variables are coded in the “128” bit, the “64” bit, the “16” bit, and the “8” bit switches. In addition, three bits are reserved for future use.

(Page 1 of 2)

Bit	Switch Name	Explanation
128	`ignore_priority`	If the bit is set to `true` for both transactions contending for a lock, their priorities are ignored by OpenVOS in deciding which transaction wins.
64	`ignore_time`	If the bit is set to `true` for both transactions contending for a lock, then the time that each transaction began is ignored by OpenVOS in deciding which transaction wins.
32		The bit is reserved for use by OpenVOS.
16	`younger_wins`	If the bit is set to `true` for both transactions contending for a lock, then the one which began last wins.

(Page 2 of 2)

Bit	Switch Name	Explanation
8	allow_deadlocks	If the bit is set to true for both of two transactions contending for a lock, then OpenVOS ignores any deadlock occurring between them. If a deadlock occurs, OpenVOS does not abort either transaction; instead, it returns the error e$record_in_use (2408). When this happens, you must abort the transaction. If you do not do so, the deadlock could continue indefinitely, until one of the transactions is aborted. For this reason, the use of this option is not recommended. If allow_deadlocks is false (the default) for either transaction involved in a deadlock, OpenVOS breaks the deadlock by choosing a winner based on the value of an internal transaction identifier. In most cases the older transaction is the winner. However, if both transactions have younger_wins set to true, then the younger transaction is the winner. In either case, if one of the transactions has lost a deadlock immediately before, it wins this time.

Example

The following program fragment illustrates a method of encoding the flag bit settings in switches.

```
%replace IGNORE_PRIORITY_BIT by 128;
%replace YOUNGER_WINS_BIT by 16;

declare ignore_priority binary(15);     /* one if bit is on, zero if
off */
declare younger_wins binary(15);        /* one if bit is on, zero if
off */


/* Set the values of 'ignore_priority' and 'younger_wins' */

parameter_info.switches = byte (ignore_priority*IGNORE_PRIORITY_BIT
+
     younger_wins*YOUNGER_WINS_BIT);

call s$set_tp_parameters(program_or_process, parameter_info,
error_code);
```

The last parameter listed above (time_value) is contained in the parameter_info structure.

Related Information

`s$get_tp_default_parameters`
Returns the default lock contention parameters for transaction locking on a specified module.

`s$get_tp_parameters`
Returns the lock contention parameters for transaction locking for the current program or process.

`s$set_tp_default_parameters`
Sets the default lock contention parameters for transaction locking for a specified module.

`s$set_tp_parameters`
Sets the lock contention parameters for transaction locking for a specified program or process.

s$set_transaction_file *Privileged*

Purpose

This subroutine turns transaction protection on or off for a specified file.

Usage

```
declare path_name                     char(256) varying;
declare protection_switch             binary(15);
declare error_code                    binary(15);

declare s$set_transaction_file entry( char(256) varying,
                                      binary(15),
                                      binary(15));

         call s$set_transaction_file( path_name,
                                      protection_switch,
                                      error_code);
```

Arguments

- `path_name` (input)
  The path name of a file for which transaction protection is to be turned on or off.

- `protection_switch` (input)
  If set to 0 then transaction protection is turned off. If set to 1, then transaction protection is turned on.

- `error_code` (output)
  A returned error code.

Explanation

This is a privileged subroutine. Therefore, it can be called only by a privileged process.

The subroutine `s$set_transaction_file` converts a file or set of files into transaction files or into ordinary files.

When you open a transaction file, the locking mode is set to implicit locking, regardless of the locking mode you specify in the open statement.

The operating system can back out the effects of an uncommitted transaction on a file only if the file is a transaction file.

You cannot rename a transaction file. You cannot create or delete an index to a transaction file. You cannot truncate a transaction file, including the truncation that normally occurs when a file is opened in output mode. If you open an existing transaction file in output mode, the file status is changed to a nontransaction file and the file is truncated. Output to the file is **not** transaction-protected.

You must have modify access to the directory containing a file to change it from a transaction file to an ordinary file or back.

You cannot convert a stream file to a transaction file. You must first convert it to a sequential file. The following sequence of commands accomplishes the conversion:

```
create_file new_file_name
copy_file stream_file new_file_name -truncate
```

You can then rename *`new_file_name`* to the old *`stream_file`* name.

This subroutine does not support extended sequential files.

Error Codes

The following is an error code this subroutine might return..

`e$invalid_log_protected_op (7149)`	Invalid operation on a log protected object. The file is a log-protected file. It cannot also be a transaction-protected file. This message supports the implementation of log-protected files. Log protection and transaction protection are mutually exclusive file attribute.

Related Information

`s$abort_transaction`
Aborts the current transaction of the calling task or process.

`s$commit_transaction`
Commits the current transaction of the calling task or process.

`s$start_priority_transaction`
Is used to start a transaction with priority different from the default priority given by OpenVOS when `s$start_transaction` is used.

`s$start_transaction`
Starts a transaction for the calling task or process.

s$start_priority_transaction

Purpose

This subroutine is used to start a transaction with priority different from the default priority given by OpenVOS when `s$start_transaction` is used.

Usage

```
declare transaction_priority                binary(15);
declare error_code                          binary(15);

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

         call s$start_priority_transaction( transaction_priority,
                                            error_code);
```

Arguments

- `transaction_priority` (input)

  A number between -1 and 9, inclusive.

- `error_code` (output)

  A returned error code.

Explanation

The subroutine `s$start_priority_transaction` starts a transaction for the calling task or process, giving it the specified priority.

Once a task or process starts a transaction, OpenVOS records all I/O calls that the task or process makes to access transaction files in order to undo the operations if the transaction is aborted.

The default priority set by OpenVOS is 0, unless it has been reset by `s$set_tp_default_parameters` or `s$set_tp_parameters`. You can use this subroutine to start a transaction with any priority in the allowed range of -1 to 9.

A priority of -1 means `always_lose`. A transaction with priority -1 will always lose in lock conflicts with transactions of any other priority regardless of how the `ignore_priority` switch is set (see the Explanation for any `tp_parameters` subroutine).

Error Codes

The following is an error code this subroutine might return.

`e$tp_in_progress`
`(2932)`
	A transaction is already in progress. The calling task or process already has a current transaction which is still running.

Related Information

`s$abort_transaction`
Aborts the current transaction of the calling task or process.

`s$commit_transaction`
Commits the current transaction of the calling task or process.

`s$set_transaction_file`
Turns transaction protection on or off for a specified file.

`s$start_transaction`
Starts a transaction for the calling task or process.

s$start_task

Purpose

This subroutine makes an `initialized` or `stopped` task ready to run.

Usage

```
declare task_id             binary(15);
declare entry_name          char(32) varying;
declare call_debug_switch   binary(15);
declare error_code          binary(15);

declare s$start_task entry( binary(15),
                            char(32) varying,
                            binary(15),
                            binary(15));

         call s$start_task( task_id,
                            entry_name,
                            call_debug_switch,
                            error_code);
```

Arguments

- `task_id` (input)
  The identifier of a task in the current process. `s$start_task` starts the task.

- `entry_name` (input)
  The name of an entry point in the executing program. `s$start_task` starts the execution of the task at that entry point.

- `call_debug_switch` (input)
  A switch indicating whether the task is to run under the control of the debugger.

- `error_code` (output)
  A returned error code.

Explanation

The `s$start_task` subroutine sets up for execution the procedure `entry_name` in the task `task_id`.

task_id must be the identifier of a task in the current process, and the specified task must be in the initialized or stopped state.

Unless you call the subroutine with task_id equal to the identifier of the calling task, s$start_task puts the task into the ready state and returns normally to the calling task.

The name of the program that the task is to execute is entry_name. The entry name must be in the entry map created for the program by the binder. To get the entry name into the entry map for the program, list it in the retain: directive of the binder control file for the program. See the beginning of Chapter 2 for an example of a binder control file.

You can call s$start_task for the current task to restart it, possibly with a different procedure for execution. To do so, you first must call s$control_task with the action stop_task_return to put the current task into the stopped state. Because s$control_task returns to the caller after performing this action, the current task retains control.

> **Note:** Use of stop_task_return is not compatible with dynamic tasks, task priorities and other forthcoming tasking facilities and is, therefore, not recommended.

For the special case of stopping a task and restarting it on a different terminal, use the subroutine s$set_task_terminal.

If call_debug_switch is nonzero, then the task runs under the control of the debugger.

Error Codes

The following lists some error codes this subroutine might return.

e$invalid_task_id (2575)	Invalid task ID. The value specified for task_id does not correspond to a task that currently exists in the process.
e$task_wrong_state (2576)	Task is in wrong state to perform operation. The specified task is not initialized or stopped.

Related Information

s$control_task
  Performs one of a variety of functions on a specified task, depending on the value of action_code given.

s$delete_task
  Deletes a dynamic task.

s$enable_tasking
  Enables or disables tasking in the calling process.

s$init_task
  Initializes a specified uninitialized task.

`s$init_task_config`

Reads a task configuration file, initializes the set of tasks described in the configuration file, and puts the tasks into the `ready` state. If a primary task (a task with identifier 1) is defined in the configuration file, then that task is given control.

`s$monitor`

Reads and carries out requests for the administration of a tasking process.

`s$monitor_full`

Like `s$monitor`, reads and carries out requests for the administration of a tasking process. It provides additional control, including allowing the caller to specify an initial request, with an option to return from the subroutine immediately after executing the request.

`s$reschedule_task`

Tells the task manager to dispatch another ready task.

`s$set_process_terminal`

Attaches a task's five predefined ports (`default_input`, `default_output`, `command_input`, `terminal_output`, and `terminal`), either to the process terminal or to the task's terminal.

`s$set_task_priority`

Sets the scheduling priority for a task.

`s$set_task_terminal`

Changes the terminal port attachment for the running task.

`s$start_task_full`

Makes an initialized or stopped task ready to run, passing it one or more initial arguments.

s$start_task_full

Purpose

This subroutine starts operation of the task, specified by `task_id`, beginning at the entry point specified by `entry_value`.

Usage

```
declare task_id                 binary(15);
declare entry_value             entry;
declare number_arguments        binary(15);
declare argument_pointers       (N) pointer;
declare call_debug_switch       binary(15);
declare error_code              binary(15);

declare s$start_task_full entry( binary(15),
                                 entry,
                                 binary(15),
                                 (N) pointer,
                                 binary(15),
                                 binary(15));

         call s$start_task_full( task_id,
                                 entry_value,
                                 number_arguments,
                                 argument_pointers,
                                 call_debug_switch,
                                 error_code);
```

Arguments

▶ `task_id` (input)

The identifier of a task in the current process. `s$start_task_full` starts the task.

▶ `entry_value` (input)

An entry point in the executing program module. `s$start_task_full` causes the new task to begin executing the procedure that begins at that entry point.

▶ `number_arguments` (input)

The number of arguments pointed to by the `argument_pointers` array. This number is automatically incremented by 1 before being passed to the invoked task. The maximum value is 256.

- ▶ `argument_pointers` (input)
  An array of *`N`* pointers to arguments you wish to pass when starting the task. *`N`* must be equal to or larger than the value of `number_arguments`.

- ▶ `call_debug_switch` (input)
  A switch indicating whether the task is to run under the control of the debugger.

- ▶ `error_code` (output)
  A returned error code.

Explanation

Unless you need to pass one or more arguments when starting the task, use `s$start_task` instead of `s$start_task_full`.

You can obtain entry values from several sources:

- entry statements
- external entry constants
- external entry variables
- entry parameters
- calls to `s$find_entry`.

The subroutine `s$find_entry`, which returns the entry value corresponding to an entry point name, is documented in the *OpenVOS PL/I Subroutines Manual* (R005).

Entry values cannot be shared among tasks. An entry value consists of three components: a display pointer, a code pointer, and a static pointer. The static pointer points to the static region, which is different for each task. Consequently, each task must obtain separately the entry value for a particular routine.

The invoked task must accept one more parameter than is passed in `argument_pointers`. This first parameter is the `number_arguments` parameter that was incremented when passed by `s$start_task_full`. Note that the value of `number_arguments` when it is received by the started task will be the total number of arguments passed to it, including `number_arguments`.

If the invoked task contains varying-length parameters, those parameters must be declared using an explicit maximum length, such as `char (256) varying`. Do not use the `char(*)` varying declaration for parameters in the invoked task. To do so might cause unpredictable parameter values. In the following example, the declaration for `p_arg2` is correct.

```
task_2: procedure (p_num_args, p_arg2);

declare p_num_args       binary(15);
declare p_arg2           char(256) varying;
```

To ensure that the pointers specified in `argument_pointers` remain valid, avoid giving the `dynamic` attribute to the arguments to which they point.declare the arguments to which they point with the `static` attribute.

For example, assume `task_2` is the task to be started by `s$start_task_full`. Also assume that `task_2` requires two arguments, one `fixed binary(15)` and the other `char (256) varying`. You must declare it as follows:

```
task_2:  procedure(p_num_args, p_arg1, p_arg2);

declare p_num_args       binary(15);
declare p_arg1           binary(15);
declare p_arg2           char(256) varying;
```

To start this procedure, you can call `s$start_task_full` as follows:

```
declare arg1             binary(15) internal static;
declare arg2             char(256) varying internal static;

.
. /* assign values to arg1 and arg2 if they are input args */
.

task_id = 2;
number_arguments = 2;
argument_pointers (1) = addr (arg1);
argument_pointers (2) = addr (arg2);
call s$start_task_full (task_id,
                   task_2,
                   number_arguments,
                   argument_pointers,
                   call_debug_switch,
                   error_code);
```

The value of `number_arguments` when it is passed to `s$start_task_full` is `2`. `s$start_task_full` increments this argument, so that the value of `p_num_args` received by `task_2` is `3`.

Related Information

`s$control_task`
: Performs one of a variety of functions on a specified task, depending on the value of `action_code` given.

`s$delete_task`
: Deletes a dynamic task.

`s$enable_tasking`
: Enables or disables tasking in the calling process.

`s$init_task`
: Initializes a specified `uninitialized` task.

`s$init_task_config`
: Reads a task configuration file, initializes the set of tasks described in the configuration file, and puts the tasks into the `ready` state. If a primary task (a task with identifier 1) is defined in the configuration file, then that task is given control.

`s$monitor`
Reads and carries out requests for the administration of a tasking process.

`s$monitor_full`
Like `s$monitor`, reads and carries out requests for the administration of a tasking process. It provides additional control, including allowing the caller to specify an initial request, with an option to return from the subroutine immediately after executing the request. Some capabilities of this subroutine are available only when it is called from a program written in C, COBOL, or PL/I.

`s$reschedule_task`
Tells the task manager to dispatch another ready task.

`s$set_process_terminal`
Attaches a task's five predefined ports (`default_input`, `default_output`, `command_input`, `terminal_output`, and `terminal`), either to the process terminal or to the task's terminal.

`s$set_task_priority`
Sets the scheduling priority for a task.

`s$set_task_terminal`
Changes the terminal port attachment for the running task.

`s$start_task`
Makes an initialized or stopped task ready to run.

s$start_transaction

Purpose

This subroutine starts a transaction for the calling task or process.

Usage

```
declare error_code                 binary(15);

declare s$start_transaction entry( binary(15));

         call s$start_transaction( error_code);
```

Arguments

▶ `error_code` (output)

A returned error code.

Explanation

The subroutine `s$start_transaction` starts a transaction for the calling task or process giving it the default priority set by OpenVOS or by a call to `s$set_tp_default_parameters` or `s$set_tp_parameters`.

Once a task or process starts a transaction, OpenVOS records all I/O calls that the task or process makes to access transaction files in order to undo the operations if the transaction is aborted.

A server program cannot start a transaction and then call `s$msg_receive` on a two-way server queue. The following paragraphs explain how to combine transaction protection and two-way server queues.

To protect a queue message and all work performed by a server program on behalf of the message, the requester program must start and commit the transaction. Either the requester or the server may abort the transaction. You cannot transmit protection from server to requester. The following programming scenario shows the correct way to transmit transaction protection from a requester to a server program.

Requester Program	**Server Program**
s$start_transaction	
s$msg_send	
	s$msg_receive
	service message
	(possibly s$abort_transaction)
	s$msg_send_reply
s$msg_receive_reply	
s$commit_transaction or s$abort_transaction	

The server program can start, commit, and abort transactions if it calls s$msg_receive outside of the transaction. This method protects the work performed by the server but not the message itself. The following programming scenario shows the correct way for a server program to start a transaction using a message received from a two-way server queue.

Requester Program	**Server Program**
s$msg_send	
	s$msg_receive
	s$start_transaction
	service message
	s$commit_transaction or s$abort_transaction
	s$msg_send_reply
s$msg_receive_reply	

Error Codes

The following is an error code this subroutine might return.

e$tp_in_progress (2932)	A transaction is already in progress. The calling task or process already has a current transaction which is still running.

Related Information

`s$abort_transaction`
Aborts the current transaction of the calling task or process.

`s$commit_transaction`
Commits the current transaction of the calling task or process.

`s$set_transaction_file`
Turns transaction protection on or off for a specified file.

`s$start_priority_transaction`
Is used to start a transaction with priority different from the default priority given by OpenVOS if `s$start_transaction` were used.

s$task_setup_wait

Purpose

This subroutine specifies which events are associated with one of the “tasks” defined by `s$set_task_wait_info` and specifies a timeout for that “task.”

> **Note:** This subroutine has been superseded by `s$task_setup_wait2`. Use `s$task_setup_wait2` for any new program when this type of functionality is needed. The subroutine `s$task_setup_wait` continues to be available for historical reasons.

Usage

```
declare macro_event_id           binary(31);
declare task_number              binary(15);
declare number_events            binary(15);
declare event_ids                (N) binary(31);
declare event_counts             (N) binary(31);
declare time_out                 binary(31);
declare event_index              binary(15);
declare error_code               binary(15);

declare s$task_setup_wait entry( binary(31),
                                 binary(15),
                                 binary(15),
                                 (N) binary(31),
                                 (N) binary(31),
                                 binary(31),
                                 binary(15),
                                 binary(15));

         call s$task_setup_wait( macro_event_id,
                                 task_number,
                                 number_events,
                                 event_ids,
                                 event_counts,
                                 time_out,
                                 event_index,
                                 error_code);
```

Arguments

- ▶ `macro_event_id` (input)

  This is the same `event_id` passed to `s$set_task_wait_info`.

- ▶ `task_number` (input)
  The number of the "task" to which the events relate. This value must be between `1` and `number_tasks`, the argument passed to `s$set_task_wait_info`, inclusive.

- ▶ `number_events` (input)
  The number of events in the `event_id` and `event_count` arrays. This value must be between `0` and `64`, inclusive.

- ▶ `event_ids` (input)
  An array of event identifiers to be waited on by the particular "task." Each event identifier must be obtained by calling one of the subroutines, like `s$read_file_lock_event` (see the OpenVOS Subroutines manuals), which return event identifiers. *`N`* must be equal to or greater than `number_events`.

- ▶ `event_counts` (input)
  An array of event counts associated with the events in `event_ids`. Each event count must be obtained by the same call used to obtain the corresponding event identifier (see `event_ids` above). *`N`* must be equal to or greater than `number_events`.

- ▶ `time_out` (input)
  The time limit beyond which the "task" will not wait, expressed in units of 1/1024 of a second. A `time_out` of -1 indicates no time limit. A `time_out` of -2 indicates that all other notifies currently waiting to be processed should be acted on before the notify for this "task". `time_out` is associated with the "task" and, therefore, with **all** the events associated with the "task."

- ▶ `event_index` (output)
  If one of the events is in error, this is the index of that event.

- ▶ `error_code` (output)
  A returned error code.

Explanation

The subroutine `s$task_setup_wait` specifies which events are associated with one of the "tasks" defined by `s$set_task_wait_info`.

Each "task" may have an arbitrary set of events, including none, and its own `time_out`.

The same event identifier may be associated with more than one "task."

If `number_events` is `0` and `time_out` is `-1`, this "task" is no longer active. If `number_events` is `0` and `time_out` is positive, this "task" will sleep until the time limit is reached.

If the set of events that a task waits on is static, and the `update_event_count` mode is enabled in `s$set_task_wait_info`, this call need not be repeated after the first initialization call.

Related Information

`s$set_task_wait_info`
: Defines a number of groups of events to be used by `s$task_setup_wait` and `s$task_wait_event`.

`s$task_setup_wait`
: Specifies which events are associated with one of the “tasks” defined by `s$set_task_wait_info` and specifies a timeout for that “task.”

`s$task_wait_event`
: Causes a process to wait until any one of the events (specified by `s$task_setup_wait`) associated with any one of the “tasks” defined by `s$set_task_wait_info` is notified.

s$task_setup_wait2

Purpose

This subroutine specifies which events are associated with one of the “tasks” defined by `s$set_task_wait_info` and specifies a time-out for that “task.”

Usage

```
declare macro_event_id            binary(31);
declare task_number               binary(15);
declare number_events             binary(15);
declare event_ids                 (N) binary(31);
declare event_counts              (N) binary(31);
declare time_out                  decimal(18);
declare event_index               binary(15);
declare error_code                binary(15);

declare s$task_setup_wait2 entry( binary(31),
                                  binary(15),
                                  binary(15),
                                  (N) binary(31),
                                  (N) binary(31),
                                  decimal(18),
                                  binary(15),
                                  binary(15));

         call s$task_setup_wait2( macro_event_id,
                                  task_number,
                                  number_events,
                                  event_ids,
                                  event_counts,
                                  time_out,
                                  event_index,
                                  error_code);
```

Arguments

- `macro_event_id` (input)

  This is the same `event_id` passed to `s$set_task_wait_info`.

- `task_number` (input)

  The number of the “task” to which the events relate. This value must be between `1` and `number_tasks`, the argument passed to `s$set_task_wait_info`, inclusive.

- `number_events` (input)
  The number of events in the `event_id` and `event_count` arrays. This value must be between `0` and `64`, inclusive.

- `event_ids` (input)
  An array of event identifiers to be waited on by the particular "task." Each event identifier must be obtained by calling one of the subroutines, like `s$read_file_lock_event` (see the OpenVOS Subroutines manuals), which return event identifiers. *`N`* must be equal to or greater than `number_events`.

- `event_counts` (input)
  An array of event counts associated with the events in `event_ids`. Each event count must be obtained by the same call used to obtain the corresponding event identifier (see `event_ids` above). *`N`* must be equal to or greater than `number_events`.

- `time_out` (input)
  The time limit beyond which the "task" will not wait, expressed in units of 1/65,536 of a second. A `time_out` of `-1` indicates no time limit. A `time_out` of `-2` indicates that all other notifies currently waiting to be processed should be acted on before the notify for this "task". `time_out` is associated with the "task" and, therefore, with **all** of the events associated with the "task."

- `event_index` (output)
  If one of the events is in error, this is the index of that event.

- `error_code` (output)
  A returned error code.

Explanation

The subroutine `s$task_setup_wait2` specifies which events are associated with one of the "tasks" defined by `s$set_task_wait_info`.

Each "task" may have an arbitrary set of events, including none, and its own `time_out`.

The same event identifier may be associated with more than one "task."

If `number_events` is `0` and `time_out` is `-1`, this "task" is no longer active. If `number_events` is `0` and `time_out` is positive, this "task" will sleep until the time limit is reached.

If the set of events that a task waits on is static, and the `update_event_count` mode is enabled in `s$set_task_wait_info`, a call to `s$task_setup_wait2` need not be repeated after the first initialization call.

Related Information

`s$set_task_wait_info`, `s$task_wait_event2`

s$task_wait_event

Purpose

This subroutine causes a process to wait until any one of the events specified by `s$task_setup_wait` associated with any one of the “tasks” defined by `s$set_task_wait_info` is notified.

> **Note:** This subroutine has been superseded by `s$task_wait_event2`. Use `s$task_wait_event2` for any new program when this type of functionality is needed. The subroutine `s$task_wait_event` continues to be available for historical reasons.

Usage

```
declare macro_event_id           binary(31);
declare time_out                 binary(31);
declare task_id                  binary(15);
declare event_id                 binary(31);
declare event_index              binary(15);
declare event_count              binary(31);
declare event_status             binary(31);
declare error_code               binary(15);

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

         call s$task_wait_event( macro_event_id,
                                 time_out,
                                 task_id,
                                 event_id,
                                 event_index,
                                 event_count,
                                 event_status,
                                 error_code);
```

Arguments

- ▶ macro_event_id (input)

  The event_id used in s$set_task_wait_info.

- ▶ time_out (input)

  This is a global time limit for this wait. This wait will end when one of the following occurs:

  — an event is notified
  — the time_out period for any one of the “tasks” elapses
  — this global time_out elapses.

  A value of -1 indicates no global time_out.

- ▶ task_id (output)

  The number of a “task” that has received a notify.

- ▶ event_id (output)

  The event_id of the event that has been notified.

- ▶ event_index (output)

  The index of the event notified in the event_id array for the “task” specified by s$task_setup_wait.

- ▶ event_count (output)

  The new event_count of the event.

- ▶ event_status (output)

  The current status of the event.

- ▶ error_code (output)

  A returned error code.

Explanation

The subroutine s$task_wait_event causes a process to wait until any one of the events specified by s$task_setup_wait associated with any one of the “tasks” defined by s$set_task_wait_info is notified. If one has already been notified, this call returns immediately. The task_id, event_id, and event_index variables indicate which event has been notified.

Error Codes

Table 3-27 explains some of the error codes returned by `s$task_wait_event`.

Table 3-27. `s$task_wait_event` Error Codes

Code	task_id	event_index	**Meaning**
`e$timeout`	`0`		Global timeout
`e$timeout`	`>0`		Task timeout
non-`0`	`0`		Error in the call (such as an invalid parameter)
non-`0`	`>0`	`>0`	Error in a specific event in the task specified (such as remote module going offline).

Related Information

`s$set_task_wait_info`
: Defines a number of groups of events to be used by `s$task_setup_wait` and `s$task_wait_event`.

`s$task_setup_wait`
: Specifies which events are associated with one of the “tasks” defined by `s$set_task_wait_info` and specifies a timeout for that “task.”

`s$task_wait_event`
: Causes a process to wait until any one of the events (specified by `s$task_setup_wait`) associated with any one of the “tasks” defined by `s$set_task_wait_info` is notified.

s$task_wait_event2

Purpose

This subroutine causes a process to wait until any one of the events specified by s$task_setup_wait2 associated with any one of the “tasks” defined by s$set_task_wait_info is notified.

Usage

```
declare macro_event_id           binary(31);
declare time_out                 decimal(18);
declare task_id                  binary(15);
declare event_id                 binary(31);
declare event_index              binary(15);
declare event_count              binary(31);
declare event_status             binary(31);
declare error_code               binary(15);

declare s$task_wait_event2 entry( binary(31),
                                 decimal(18),
                                 binary(15),
                                 binary(31),
                                 binary(15),
                                 binary(31),
                                 binary(31),
                                 binary(15));

         call s$task_wait_event2( macro_event_id,
                                 time_out,
                                 task_id,
                                 event_id,
                                 event_index,
                                 event_count,
                                 event_status,
                                 error_code);
```

Arguments

▶ macro_event_id (input)

The event_id used in s$set_task_wait_info.

▶ time_out (input)

The global time limit for this wait, expressed in units of 1/65,536 of a second. This wait ends when one of the following occurs:

- An event is notified.
- The time_out period for any one of the “tasks” elapses.
- This global time_out elapses.

A value of -1 indicates no global time limit.

▶ task_id (output)

The number of a “task” that has received a notify.

▶ event_id (output)

The event_id of the event that has been notified.

▶ event_index (output)

The index of the event notified in the event_id array for the “task” specified by s$task_setup_wait2.

▶ event_count (output)

The new event_count of the event.

▶ event_status (output)

The current status of the event.

▶ error_code (output)

A returned error code.

Explanation

The subroutine s$task_wait_event2 causes a process to wait until any one of the events specified by s$task_setup_wait2 associated with any one of the “tasks” defined by s$set_task_wait_info is notified. If one has already been notified, this call returns immediately. The task_id, event_id, and event_index variables indicate which event has been notified.

Error Codes

Table 3-28 explains some of the error codes returned by `s$task_wait_event2`.

Table 3-28. `s$task_wait_event2` Error Codes

Code	`task_id`	`event_index`	Meaning
`e$timeout`	0		Global time-out
`e$timeout`	>0		Task time-out
non-0	0		Error in the call (such as an invalid parameter)
non-0	>0	>0	Error in a specific event in the task specified (such as remote module going offline)
`0`	>0	>0	The indicated event for the indicated task has been notified.

Related Information

`s$set_task_wait_info`, `s$task_setup_wait2`

s$truncate_queue

Purpose

This subroutine can be used by a server to truncate an empty message queue.

Usage

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

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

         call s$truncate_queue( port_id,
                                error_code);
```

Arguments

- ▶ `port_id` (input)
  The identifier of a port attached to the desired queue. `s$truncate_queue` truncates the queue.

- ▶ `error_code` (output)
  A returned error code.

Explanation

The subroutine `s$truncate_queue` truncates an empty message queue; it resets the next available message identifier value for the queue to the initial value, and returns allocated disk space to the system.

Error Codes

The following is an error code this subroutine might return.

`e$no_truncate_queue`
`(2826)`
	Truncate queue cannot be done while queue in use. You called `s$truncate_queue` when either a requester had the queue open or there were one or more messages still in the queue.

Related Information

`s$msg_cancel_receive`
: Can be used by a server to cancel the receipt of a message contained in a two-way server queue, or by a server or a requester to cancel the receipt of a message contained in a message queue.

`s$msg_delete`
: Can be used by a requester or a server to delete a message from a message queue.

`s$msg_open`
: Is used by a process to open (to connect to) a server queue, a message queue, or a one-way server queue.

`s$msg_open_direct`
: Is used by a process to open (to connect to) a direct queue or a one-way direct queue.

`s$msg_read`
: Can be used by a requester or a server to read a message contained in a queue.

`s$msg_receive`
: Can be used by a requester or a server to receive a message contained in a queue.

`s$msg_rewrite`
: Can be used by a requester or a server to rewrite a message contained in a message queue.

`s$msg_send_reply`
: Can be used by a server to put a reply into a two-way server queue or a two-way direct queue.

`s$set_max_queue_depth`
: Sets the maximum queue depth (or maximum number of messages) of a server queue or one-way server queue.

Index