

# DBMS\_ALERT

DBMS\_ALERT supports asynchronous notification of database events (alerts). By appropriate use of this package and database triggers, an application can notify itself whenever values of interest in the database are changed.

This chapter contains the following topics:

- [Overview](#)
- [Security Model](#)
- [Constants](#)
- [Restrictions](#)
- [Exceptions](#)
- [Operational Notes](#)
- [Examples](#)
- [Summary of DBMS\\_ALERT Subprograms](#)

## DBMS\_ALERT Overview

This scenario describes a possible use of the DBMS\_ALERT package.

Suppose a graphics tool is displaying a graph of some data from a database table. The graphics tool can, after reading and graphing the data, wait on a database alert (WAITONE) covering the data just read. The tool automatically wakes up when the data is changed by any other user. All that is required is that a trigger be placed on the database table, which performs a signal (SIGNAL) whenever the trigger is fired.

## DBMS\_ALERT Security Model

Security on this package can be controlled by granting EXECUTE on this package to selected users or roles. You might want to write a cover package on top of this one that restricts the alert names used. EXECUTE privilege on this cover package can then be granted rather than on this package.

## DBMS\_ALERT Constants

The DBMS\_ALERT package uses the constants listed and described in this topic.

**Table 20-1 DBMS\_ALERT Constants**

| Name    | Type    | Value    | Description   |
|---------|---------|----------|---|
| MAXWAIT | INTEGER | 86400000 | The maximum time to wait for an alert (1000 days which is essentially forever). |

## DBMS\_ALERT Restrictions

Because database alerters issue commits, they cannot be used with Oracle Forms. For more information on restrictions on calling stored procedures while Oracle Forms is active, refer to your Oracle Forms documentation.

## DBMS\_ALERT Exceptions


DBMS\_ALERT raises the application error -20000 on error conditions.

Table 20-2 shows the messages and the procedures that can raise them.

## DBMS\_ALERT Operational Notes

This topic lists notes related to general and specific applications. Also, a list of DBMS\_ALERT error messages is provided.

- Alerts are transaction-based. This means that the waiting session is not alerted until the transaction signalling the alert commits. There can be any number of concurrent signalers of a given alert, and there can be any number of concurrent waiters on a given alert.
- A waiting application is blocked in the database and cannot do any other work.
- An application can register for multiple events and can then wait for any of them to occur using the `WAITANY` procedure.
- An application can also supply an optional `timeout` parameter to the `WAITONE` or `WAITANY` procedures. A `timeout` of 0 returns immediately if there is no pending alert.
- The signalling session can optionally pass a message that is received by the waiting session.
- Alerts can be signalled more often than the corresponding application wait calls. In such cases, the older alerts are discarded. The application always gets the latest alert (based on transaction commit times).
- If the application does not require transaction-based alerts, the `DBMS_PIPE` package may provide a useful alternative.

 **See Also:**  
[DBMS\\_PIPE](#)

- If the transaction is rolled back after the call to `SIGNAL`, no alert occurs.
- It is possible to receive an alert, read the data, and find that no data has changed. This is because the data changed after the *prior* alert, but before the data was read for that *prior* alert.
- Usually, Oracle is event-driven; this means that there are no polling loops. There are two cases where polling loops can occur:
  - Shared mode. If your database is running in shared mode, a polling loop is required to check for alerts from another instance. The polling loop defaults to five seconds and can be set by the `SET_DEFAULTS` procedure.

- **WAITANY** procedure. If you use the **WAITANY** procedure, and if a signalling session does a signal but does not commit within one second of the signal, a polling loop is required so that this uncommitted alert does not camouflage other alerts. The polling loop begins at a one second interval and exponentially backs off to 30-second intervals.

**Table 20-2 DBMS\_ALERT Error Messages**

| Error Message  | Procedure |
|--|-----------|
| ORU-10001 lock request error, status: N                              | SIGNAL    |
| ORU-10015 error: N waiting for pipe status                           | WAITANY   |
| ORU-10016 error: N sending on pipe 'X'                               | SIGNAL    |
| ORU-10017 error: N receiving on pipe 'X'                             | SIGNAL    |
| ORU-10019 error: N on lock request                                   | WAIT      |
| ORU-10020 error: N on lock request                                   | WAITANY   |
| ORU-10021 lock request error; status: N                              | REGISTER  |
| ORU-10022 lock request error, status: N                              | SIGNAL    |
| ORU-10023 lock request error; status N                               | WAITONE   |
| ORU-10024 there are no alerts registered                             | WAITANY   |
| ORU-10025 lock request error; status N                               | REGISTER  |
| ORU-10037 attempting to wait on uncommitted signal from same session | WAITONE   |

## DBMS\_ALERT Examples

In this example, suppose that you want to graph average salaries by department, for all employees. Your application needs to know whenever **EMP** is changed.

Your application would look similar to this code:

```
DBMS_ALERT.REGISTER('emp_table_alert');
<<readagain>>:
/* ... read the emp table and graph it */
  DBMS_ALERT.WAITONE('emp_table_alert', :message, :status);
  if status = 0 then goto <<readagain>>; else
  /* ... error condition */
```

The **EMP** table would have a trigger similar to this:

```
CREATE TRIGGER emptrig AFTER INSERT OR UPDATE OR DELETE ON emp
BEGIN
  DBMS_ALERT.SIGNAL('emp_table_alert', 'message_text');
END;
```

When the application is no longer interested in the alert, it makes this request:

```
DBMS_ALERT.REMOVE('emp_table_alert');
```

This reduces the amount of work required by the alert signaller. If a session exits (or dies) while registered alerts exist, the alerts are eventually cleaned up by future users of this package.

The example guarantees that the application always sees the latest data, although it may not see every intermediate value.

## Summary of DBMS\_ALERT Subprograms

This table lists the DBMS\_ALERT subprograms and briefly describes them.

**Table 20-3 DBMS\_ALERT Package Subprograms**

| Subprogram                             | Description  |
|--|--|
| <a href="#">REGISTER Procedure</a>     | Receives messages from an alert  |
| <a href="#">REMOVE Procedure</a>       | Disables notification from an alert  |
| <a href="#">REMOVEALL Procedure</a>    | Removes all alerts for this session from the registration list                                   |
| <a href="#">SET_DEFAULTS Procedure</a> | Sets the polling interval  |
| <a href="#">SIGNAL Procedure</a>       | Signals an alert (send message to registered sessions)   |
| <a href="#">WAITANY Procedure</a>      | Waits <code>timeout</code> seconds to receive alert message from an alert registered for session |
| <a href="#">WAITONE Procedure</a>      | Waits <code>timeout</code> seconds to receive message from named alert                           |

### REGISTER Procedure

This procedure lets a session register interest in an alert.

#### Syntax

```
DBMS_ALERT.REGISTER (  
    name      IN  VARCHAR2,  
    cleanup   IN  BOOLEAN DEFAULT TRUE);
```

#### Parameters

**Table 20-4 REGISTER Procedure Parameters**

| Parameter            | Description   |
|----------------------|---|
| <code>name</code>    | Name of the alert in which this session is interested   |
| <code>cleanup</code> | Specifies whether to perform cleanup of any extant orphaned pipes used by the DBMS_ALERT package. This cleanup is only performed on the first call to REGISTER for each package instantiation. The default for the parameter is TRUE. |

#### WARNING:

Alert names beginning with 'ORA\$' are reserved for use for products provided by Oracle. Names must be 30 bytes or less. The name is case insensitive.

#### Usage Notes

A session can register interest in an unlimited number of alerts. Alerts should be deregistered when the session no longer has any interest, by calling `REMOVE`.

## REMOVE Procedure

This procedure enables a session that is no longer interested in an alert to remove that alert from its registration list. Removing an alert reduces the amount of work done by signalers of the alert.

### Syntax

```
DBMS_ALERT.REMOVE (  
    name IN VARCHAR2);
```

### Parameters

**Table 20-5 REMOVE Procedure Parameters**

| Parameter | Description  |
|-----------|--|
| name      | Name of the alert (case-insensitive) to be removed from registration list. |

### Usage Notes

Removing alerts is important because it reduces the amount of work done by signalers of the alert. If a session dies without removing the alert, that alert is eventually (but not immediately) cleaned up.

## REMOVEALL Procedure

This procedure removes all alerts for this session from the registration list. You should do this when the session is no longer interested in any alerts.

This procedure is called automatically upon first reference to this package during a session. Therefore, no alerts from prior sessions which may have terminated unusually can affect this session.

This procedure always performs a commit.

### Syntax

```
DBMS_ALERT.REMOVEALL;
```

## SET\_DEFAULTS Procedure

In case a polling loop is required, use the SET\_DEFAULTS procedure to set the polling interval.

### Syntax

```
DBMS_ALERT.SET_DEFAULTS (  
    sensitivity IN NUMBER);
```

## Parameters

**Table 20-6 SET\_DEFAULTS Procedure Parameters**

| Parameter   | Description   |
|-------------|---|
| sensitivity | Polling interval, in seconds, to sleep between polls. The default interval is five seconds. |

## SIGNAL Procedure

This procedure signals an alert. The effect of the `SIGNAL` call only occurs when the transaction in which it is made commits. If the transaction rolls back, `SIGNAL` has no effect.

All sessions that have registered interest in this alert are notified. If the interested sessions are currently waiting, they are awakened. If the interested sessions are not currently waiting, they are notified the next time they do a wait call.

Multiple sessions can concurrently perform signals on the same alert. Each session, as it signals the alert, blocks all other concurrent sessions until it commits. This has the effect of serializing the transactions.

### Syntax

```
DBMS_ALERT.SIGNAL (  
    name      IN  VARCHAR2,  
    message   IN  VARCHAR2);
```

## Parameters

**Table 20-7 SIGNAL Procedure Parameters**

| Parameter | Description   |
|-----------|---|
| name      | Name of the alert to signal.  |
| message   | Message, of 1800 bytes or less, to associate with this alert.<br><br>This message is passed to the waiting session. The waiting session might be able to avoid reading the database after the alert occurs by using the information in the message. |

## WAITANY Procedure

Call this procedure to wait for an alert to occur for any of the alerts for which the current session is registered.

### Syntax

```
DBMS_ALERT.WAITANY (  
    name      OUT  VARCHAR2,  
    message   OUT  VARCHAR2,  
    status     OUT  INTEGER,  
    timeout    IN   NUMBER DEFAULT MAXWAIT);
```

## Parameters

**Table 20-8 WAITANY Procedure Parameters**

| Parameter | Description  |
|-----------|--|
| name      | Returns the name of the alert that occurred.   |
| message   | Returns the message associated with the alert.<br>This is the message provided by the <code>SIGNAL</code> call. If multiple signals on this alert occurred before <code>WAITANY</code> , the message corresponds to the most recent <code>SIGNAL</code> call. Messages from prior <code>SIGNAL</code> calls are discarded. |
| status    | Values returned:<br>0 - alert occurred<br>1 - timeout occurred   |
| timeout   | Maximum time to wait for an alert.<br>If no alert occurs before <code>timeout</code> seconds, this returns a status of 1.  |

## Usage Notes

An implicit `COMMIT` is issued before this procedure is executed. The same session that waits for the alert may also first signal the alert. In this case remember to commit after the signal and before the wait; otherwise, `DBMS_LOCK.REQUEST` (which is called by `DBMS_ALERT`) returns status 4.

## Exceptions

-20000, ORU-10024: there are no alerts registered.

# WAITONE Procedure

This procedure waits for a specific alert to occur.

An implicit `COMMIT` is issued before this procedure is executed. A session that is the first to signal an alert can also wait for the alert in a subsequent transaction. In this case, remember to commit after the signal and before the wait; otherwise, `DBMS_LOCK.REQUEST` (which is called by `DBMS_ALERT`) returns status 4.

## Syntax

```
DBMS_ALERT.WAITONE (  
    name      IN   VARCHAR2,  
    message   OUT  VARCHAR2,  
    status    OUT  INTEGER,  
    timeout   IN   NUMBER DEFAULT MAXWAIT);
```

## Parameters

**Table 20-9 WAITONE Procedure Parameters**

| Parameter | Description                    |
|-----------|--------------------------------|
| name      | Name of the alert to wait for. |

**Table 20-9 (Cont.) WAITONE Procedure Parameters**

| Parameter | Description  |
|-----------|--|
| message   | Returns the message associated with the alert.<br>This is the message provided by the <code>SIGNAL</code> call. If multiple signals on this alert occurred before <code>WAITONE</code> , the message corresponds to the most recent <code>SIGNAL</code> call. Messages from prior <code>SIGNAL</code> calls are discarded. |
| status    | Values returned:<br>0 - alert occurred<br>1 - timeout occurred   |
| timeout   | Maximum time to wait for an alert.<br>If the named alert does not occurs before <code>timeout</code> seconds, this returns a status of 1.  |