Oracle Utilities Energy Information Platform

Rules Language Reference Guide

Release 1.6.1.23 for Windows

E18203-24

December 2018

(Revised July 2019)

ORACLE®

NOTIFICATION OF THIRD-PARTY LICENSES

Oracle Utilities software contains third party, open source components as identified below. Third- party license terms and other third-party required notices are provided below.

License: Apache 1.1

Module: xercesImpl.jar, xalan.jar

Copyright © 1999-2000 The Apache Software Foundation. All rights reserved.

Use of xercesImpl and xalan within the product is governed by the following (Apache 1.1):

(1) Redistributions of source code must retain the above copyright notice, this list of conditions and the disclaimer below. (2) Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the disclaimer below in the documentation and/or other materials provided with the distribution. (3) The end-user documentation included with the redistribution, if any, must include the following acknowledgment: "This product includes software developed by the Apache Software Foundation (http://www.apache.org/) ." Alternately, this acknowledgment may appear in the software itself, if and wherever such third-party acknowledgments normally appear. (4) Neither the component name nor Apache Software Foundation may be used to endorse or promote products derived from the software without specific prior written permission. (5) Products derived from the software may not be called "Apache", nor may "Apache" appear in their name, without prior written permission.

THIS SOFTWARE IS PROVIDED "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

License: Paul Johnston

Modules: md5.js

Copyright (C) Paul Johnston 1999 - 2002

Use of these modules within the product is governed by the following:

(1) Redistributions of source code must retain the above copyright notice, this list of conditions and the disclaimer below. (2) Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the disclaimer below in the documentation and/or other materials provided with the distribution. (3) Neither the component name nor the names of the copyright holders and contributors may be used to endorse or promote products derived from the software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

License: Tom Wu

Module: jsbn library

Copyright © 2003-2005 Tom Wu. All rights reserved

Use of this module within the product is governed by the following:

Contents

Chapter 4

Chapter 5

Chapter 6

Chapter 7

Chapter 8

Chapter 9
Interval Data Function Descriptions **9-1**

Appendix A

Appendix B

Index

What's New

New Features in the Oracle Utilities Rules Language Reference Guide

This chapter outlines the new features of the 1.6.0.0 release of the Oracle Utilities Rules Language that are documented in this guide.

New Features for Release 1.6.0.0

Feature	Description	For more information, refer to...
Term-Based Rules Language Functions	This release includes new Rules Language functions to retrieve and save terms and term details to and from the Oracle Utilities Data Repository.	**Term Functions** on page 13-74
Query Lists	Query lists are structured query language (SQL) queries that can be used by Oracle Utilities Rules Language to access records stored in the Oracle Utilities Data Repository. Query lists are created using the **Lists** function available through the Energy Information Platform user interface.	**Query Lists** on page 7-66 in the *Oracle Utilities Energy Information Platform User's Guide* **For Each x In List Statement** on page 3-10 **LISTVALUE Function** on page 13-15
Support for Oracle Business Intelligence Publisher	This release includes support for publishing reports using Oracle Business Intelligence Publisher 10.1.3.4. The CREATEREPORT Rules Language function has been enhanced to support initiation of Oracle BI Publisher reports.	**CREATEREPORT Function** on page 13-106

Chapter 1

Overview

This chapter provides a brief overview of the *Oracle Utilities Rules Language Reference Guide* and descriptions of the format used in the statement and function descriptions found in later chapters, including:

- **Statement Format**
- **Function Format**

Statement Format

Each statement type has its own format and rules for use. Most statements begin with a keyword, such as ALL or BLOCK. Each keyword is followed by one or more parameters that provide additional information as to how the instruction is to be processed. A parameter can be a constant, an identifier (variable), an arithmetic expression, or a function, depending upon the particular type of statement. See **Chapter 4**: **Identifiers, Constants, and Expressions** in the *Oracle Utilities Rules Language User's Guide* for more information. Some statement types consist of several keywords and parameters, organized on several lines.

It is possible to include statements inside other statements. This is called "nesting," and is used with a number of statements.

All statements end in a semicolon (;).

Conventions Used to Represent the Syntax of Statements

Throughout this manual, the format of statements is represented with the following conventions:

Convention	Meaning
CAPS	Words in all capital letters are reserved words, such as keywords. You must use them exactly as they appear in the manual. A complete list of reserved words is provided in **Appendix A**: **Reserved Words**.
< >	Pointed brackets are used to indicate a required parameter. Do not use the pointed brackets in your statements. In an actual statement, you substitute a real value for the placeholder that appears between the brackets.
[ ]	Square brackets are used to indicate an optional parameter. Do not use the square brackets in your statements.
|	Vertical bars are used to indicate a choice. Do not use the vertical bar in your statements.

Format Example:

```
ALL <determinant> CHARGE <price> INTO <$rev_id>;
```

Sample Example:

```
ALL KWH CHARGE $0.05 INTO $ENERGY_CHARGE;
```

Description Format

Each of the following statement descriptions uses the same format. If there are multiple types of the same statement (such as FOR EACH x IN statements), each is described separately.

Purpose

The function and purpose of the statement. If there are multiple formats of a given statement, each format is described.

Format

The specific format of the statement (see **Statement Format** on page 1-2).

Example

An example of the statement as it might be used in a rate form. If a given statement is used within the context of another statement (i.e., as a nested statement), the statement appears in **bold** type.

Notes

Comments and notes concerning how the statement is used in specific circumstances, and any special information you might need to use the statement.

To Create

A step-by-step explanation of how to create the statement in a rate form. If there are multiple formats of a statement, the steps to create a statement in each format are included.

.

Function Format

Each of the function descriptions that follow employ the same format:

Purpose

The first section of each description outlines the purpose of the function, as well as specific information to help you understand how the function is used in a rate form.

Format

The specific format of the function, including the proper syntax to be used with the function, and the function's parameters.

Where

The function parameters are described.

Example

An example of the function used in a rate form. If a function is used within the context of a statement (i.e., as part of a nested statement) the function will appear in **bold** type.

Chapter 2

General Statements

This chapter provides detailed descriptions of the General statements available in the Oracle Utilities Rules Language. General statements are used in a number of ways throughout rate forms.

General Statements

- **Assignment Statement**
- **Comment Statement**

Assignment Statement

Purpose

An ASSIGNMENT Statement assigns a value to an identifier. Identifiers are the equivalent of variables in programming languages and algebraic formulas. See **Chapter 4**: **Identifiers, Constants, and Expressions** in the *Oracle Utilities Rules Language User's Guide* for more information concerning identifiers. Assignment statements are the most basic type of statement; they are used for many purposes. They are often used in a rate form to do intermediate calculations.

You can set an identifier equal to a **constant**, to the **result of an arithmetic expression**, or to the **result of a function**.

Format

The formats for Assignment statements are:

```
<Identifier> = <constant|expression|function>;
```

or

```
<Identifier> =+ <constant|expression|function>;
(=+ sets a negative result to zero)
```

Examples

Statement Format	Sample Statement/Explanation
\<Identifier\> = \<constant\>	`$CUST_CHARGE = $5.00;` Set customer charge equal to $5.00.
\<Identifier\> = \<expression\>	`$DEMAND_RATE1 = $4.68 - VOLTAGE_DISCOUNT;` Calculate demand rate by subtracting customer's voltage discount from $4.68
\<Identifier\> = \<function\>	`BILL_KW = MAX(5, KW);` Set identifier "BILL_KW" equal to whichever is greater, 5 or the customer's value for KW during the billing period.

To Create Assignment Statements

1. Select **Statements-›Assignment**.

   The Assignment Statement template appears.

2. Specify the identifier.

   The identifier on the left side of the equal sign is the name to which you are assigning the value specified or computed on the right side of the equal sign. You can then use this identifier on the right side of an another ASSIGNMENT Statement, or, depending upon the type of identifier, in another type of statement.

   It's always best to use a descriptive name that is easily recognized.

   Depending upon the value or values to be computed or assigned on the right side of the statement, you may want to use one of the special types of identifiers described in **Chapter 4**: **Identifiers, Constants, and Expressions** in the *Oracle Utilities Rules Language User's Guide.*

   You can type the identifier, or if it is a predefined identifier or one you've assigned elsewhere in the rate form (including other rate forms that you've included in this rate form using an INCLUDE Statement), you can use the Rules Language Elements Editor to pick it. To use that feature, position the mouse pointer in the "Identifier" field and click the *right* mouse button.

   The identifier can be any name you choose, with the following restrictions:

   - You **cannot** use any of the reserved words shown in **Appendix A**: **Reserved Words**.

   - With the exception of hyphens, you can use any combination of letters, digits, and the underscore character (_), as long as the first character is a letter. Revenue identifiers, however, must begin with a dollar sign ($).

   - It can be up to 64 characters.

   - If an identifier consists of multiple words, you must join the words with an underscore (blanks are not allowed within identifiers).
     Examples: VOLTAGE_DISCOUNT and KWH_0_TO_150.

   - Identifiers are case-insensitive—for example, the identifiers aaa and AAA are the same.

3. In the **Expression** field, specify the value that you want to assign to the identifier. Expressions can be up to 256 characters. **You can specify a constant, an expression (arithmetic operation), or a function.**

   Constant: A constant is a value that doesn't change. You can set the identifier equal to any of the following types of constants:

   Number can be either an integer or a decimal number (e.g., 7 or 7.5).

   Text strings: A string constant is any set of characters (except a double quote) enclosed in double quotes.

   Dates are represented as 'mm/dd/yyyy' or 'yyyy-mm-dd'. A date can include a time: 'mm/dd/yyyy hh:mm:ss' (for example, 12/01/1998 12:00:00). If you don't include a time, midnight (00:00) is assumed.

   Recorder, channel. A specific recorder and channel are indicated by 'recorder,channel' (enclosed in single quote marks, with no spaces before or after the comma). The recorder is any combination of uppercase letters and digits, and the channel is any integer from 0 through 9 (for example, KWH_HNDL = 'RCDR1234,1'). When you assign a recorder,channel to an identifier in this way, the program goes to the Interval Database and gets the interval data for that recorder-id,channel-number for the current bill period, and puts a reference to it in the identifier (this is called *interval data loading*). **Note:** This approach "hard codes" a particular recorder channel, which you usually should not do, except possibly in a contract for a particular account. The INTDLOADxxx functions are a more flexible approach, because they automatically load data for whatever account is

being billed, and allow to you to specify time periods other than just the current bill period.

Expression: You can combine constants and variables into expressions using any of the standard arithmetic operators: add (+), subtract (-), multiply (*), and divide (/). Specifically, you can use constants (including 'recorder,channel' constants), interval data handles, TOU handles, bill determinant identifiers defined in the database tables or in the rate form, and/or any other identifiers assigned elsewhere in the rate form. **Note:** You can use 'recorder,channel' in an expression in the same way you would use an identifier that has been assigned an interval data cut; for example, KWH_NEW_HNDL = 'R1234,1' - 'R1234,2'.

For a complete description of the Arithmetic Expressions, including Operator Rules, Precedence, etc., see **Chapter 4**: **Identifiers, Constants, and Expressions** in the *Oracle Utilities Rules Language User's Guide.*

Function: The Rules Language includes a diverse and powerful set of functions designed for revenue calculations and reporting: for loading historical data (bill determinants or interval data cuts), performing operations on that data, working with dates and text strings, etc. You apply a function by putting it on the right side of an ASSIGNMENT Statement.

To enter a function in the ASSIGNMENT template, do one of the following:

- Click the mouse pointer in the Expression field. Click **Functions...** at the bottom of the template. Select the desired function from the list that appears. You must add the necessary parameters.

- Apply the Rules Language Elements Editor. To use that feature, position the mouse pointer in the Expression field and click the *right* mouse button. Included in the list of "Element Types" on the left are four categories of functions: Interval Data/Meter Value Functions, Math Functions, String Functions, and Other Functions. Select the desired category, select the function from the list that appears on the right, and click **OK**. You can add the parameter using a similar technique.

4. *Optional.* If you wish to apply positive assignment (+), check the box for **Set negative values to zero**. Use this option when a negative result is unacceptable. If the result of the expression or function on the right side of the statement is negative, the identifier will be assigned a value of 0.

For example, you may want to subtract a minimum demand from the actual demand and bill the positive difference. To do this:

```
BILL_KW = KW - MIN_KW;
BILL_KW = MAX(BILL_KW, 0);
```

However, you could accomplish the same thing more simply using positive assignment:

```
BILL_KW =+ KW - MIN_KW;
```

which means assign the result, but set it to zero if it is less than zero. "=+" can be used in any Assignment Statement, though its special effect is ignored if a string is assigned.

5. Click **OK**. The statement appears in the rate form.

Comment Statement

Purpose

Comments are statements that allow you to annotate your rate forms to help explain the purpose of the rate form and its various parts. Unlike most statements, comments do not affect computations, but they are important nonetheless.

You might preface the rate form with a description of the schedule's availability, applicability, and included rate codes, for example, or you might add comments at the end to note any riders. In the middle of a rate schedule, you could use comments to explain individual charges and computations. You can also use the Comments template to add blank lines to your rate forms, to make them easier to read.

Format

A comment is a string of characters that starts with /* and ends with */, or for single-line comments, starts with //.

Example

```
/* Applicable rate codes associated with this schedule */
/* 301, 302, 303, 308, 309 */
/* Include Shoulder Peak Rider */
```

To Create Comment Statements

1. Select **Statements-›Comments**.

    The COMMENT Statement template appears.

2. In the Comment field, type the line of text you wish to appear (up to 79 characters). You can include any characters, including spaces and upper and lower case. If you want to include a blank line in your rate form, simply press the space bar once, and continue with the following instructions.

3. When you have completed a line, click **Insert**. The text appears in the upper list box.

4. Repeat steps 2 and 3 for up to five lines of comments. To modify a comment in the upper box, highlight it there; change the text in the field below, and click **Update**. Similarly, to delete a comment, highlight it in the upper box and click **Delete**.

5. When you are satisfied with the comments as they appear in the upper box, click **OK**. The comments appear in the rate form.

Chapter 3

Control Statements

This chapter describes the control statements available in the Oracle Utilities Rules Language. Control statements are used to control the order and method for rate form processing.

Control Statements

- **Abort Statement**

- **Call Statement**

- **Done Statement**

- **For Each Statements**

  - **For Each x in Channel Statement**

  - **For Each x in Factor Statement**

  - **For Each x In List Statement**

  - **For Each x In Number Statement**

  - **For Each x In Override Statement**

  - **For Each x In Recordlist Statement**

  - **For Each x In Set Statement**

  - **For Each x In Week Statement**

  - **For Each x In Distribution Node Statement**

  - **For Each x In CSV File Statement**

  - **For Each x In COM IENUM Statement**

- **If-Then-Else Statement**

- **Include Statement**

- **Leave For Statement**

- **Leave Rider Statement**

- **Next For Statement**

- **Novalue Statement**

- **Section Statement**

- **Select Bill_Period Statement**

- **Select Expression Statement**
- **Select Rate_Code Statement**
- **Warn Statement**

Abort Statement

Purpose

The ABORT Statement is used with the **If-Then-Else Statement** to stop processing an account's bill when a condition you specify is met (or not met), and to issue an explanatory message on page 1 of the bill report.

If an ABORT Statement is triggered for an account by a user-specified condition during billing, the bill calculations for that account stop. When the bill report is created, it displays the message you supplied in the ABORT Statement. At that point, the bill can only be issued using the Current/Final Bill module. The data that triggered the ABORT Statement must be corrected before the bill can be run successfully.

Format

Abort statements have this format:

```
ABORT <'character_string'>;
```

Example

If the account's value for KWH for the current bill period exceeds 999999, stop processing that bill and display the following message in the bill report: "KWH is too high, invalid data."

```
IF KWH > 999999
THEN
    ABORT "KWH is too high, invalid data.";
END IF;
```

Notes

The ABORT Statement in similar to the **Warn Statement** on page 3-35. Both ABORT and WARN statements are used with IF-THEN-ELSE, so that they are triggered by a user-defined condition. However, unlike ABORT, WARN stops processing only in the Automatic billing mode; in the Approval Required mode, the bill is still computed but a warning message is displayed on page 1 of the bill report. You could use the WARN and ABORT statements together for a two-step validation; that is, if condition x is met, calculate the bill and issue a WARN message for the billing analyst; if condition y is met, stop processing the bill and issue the ABORT message.

The billing and analysis programs can display up to 50 messages in one report.

To Create

1. Select **Statements-›Abort** from the Rules Language Editor menu bar.

   The ABORT Statement template appears.

2. Type the message (up to 256 characters) that will appear on reports. The message must be a string, so it must be enclosed within double-quotes (" ").

3. Click **OK**. The ABORT statement appears in the rate form.

Call Statement

Purpose

The CALL Statement is used to dynamically execute one rate form while in another. Within a rate schedule, you can call riders, contracts, and other rate schedulers. Within a rate schedule or rider, you can call other riders. Any statements can appear in a CALLed rider, including CALL and INCLUDE statements. Their names will be resolved in the same way as the original CALL name is resolved.

Note: Exercise care when using CALL statements inside CALLed riders and contracts. In particular, a CALLed rider (or contract) should NOT include a CALL statement that calls "itself". For example, "Rider A" (a rider CALLed within a rate schedule) should NOT include CALL statements that call "Rider A". Doing so can result in recursion and the rider calling itself in an "endless loop," and can lead to undesirable results.

Note: A single rate schedule can contain up to a maximum of 20 riders and/or contracts (including those included using both the **Call Statement** and the **Include Statement**).

Format

```
CALL <string_expression>;
```

Where:

- <string_expression> is a string that is the name of the rate form, or that evaluates to the name of the rate form. The name can contain the operating company, jurisdiction, rate form code, and version, separated by colons. The operating company and jurisdiction are optional, and default to those of the CALLing rate schedule. The version is also optional, and defaults to the version in effect on the effective date of the current run. Valid formats are:

String Expressions	Description
name	name is the rate form code. Add current operating company and jurisdiction, and find the version in effect on the effective date.
name:version	name is the rate form code. Add current operating company and jurisdiction, and find the specified version (version may be a number or a date).
opco:juris:name	opco is the operating company code, juris is the jurisdiction code, and name is the rate form code. Find the version in effect on the effective date.
opco:juris:name:version	opco is the operating company code, juris is the jurisdiction code, name is the rate form code. Find the specified version (version may be a number or a date).

When the statement is executed, the string is evaluated and the correct rider version determined. If it has already been included or called, the previously compiled code will be "called." If it was not previously compiled, it will be loaded, compiled, and the first statement line in the rider will be executed. After the last statement in the rider executes, control returns to the next statement after the CALL Statement.

In Single Step, when a CALL Statement is executed and its rate form has not been loaded, the rider will be loaded and compiled, and the text of the rate form will be appended at the bottom of

the displayed rate schedule. The single step next-line highlight will be set to the first line of the rate form (the new text or previously loaded text).

If an error occurs when executing a CALL Statement, loading or compiling the rate form execution of the current rate schedule will stop with an appropriate error message.

Note that the values for Billing Determinants are retrieved from the database before a rate schedule is run. Usually only the determinants seen in the rate schedule are retrieved. Because the CALL Statement loads a rate form dynamically, if you reference a determinant only through the CALL Statement the determinant would not be loaded. There are two ways to make sure referenced determinants in CALL statements are retrieved. The first is to reference that determinant outside of the CALL Statement's rate form. The second is to load all determinants for each account by checking **Retrieve all Account Determinants** on the **Billing Rules** tab in the **CIS Billing Options** dialog.

Note: Attributes of interval data handles loaded within the "calling" rate schedule cannot be accessed within the "called" rate form. If this type of operation is required, use the INCLUDE statement.

Example

Determine if the value of KWH is greater than the value of KWH_MAX and, if so, call the "RIDER_1" rate form based on the account's Operating Company and Jurisdiction.

```
OPCO=ACCOUNT.OPCOCODE;
JURIS=ACCOUNT.JURISCODE;

IF KWH > KWH_MAX THEN
    CALL OPCO + ":" + JURIS + ":" + "RIDER_1";
END IF;
```

To Create

1. Select **Statements->Call**.

    The CALL Statement template appears.

2. Complete the template:

    Rate Form Name: The name (including opcocode, juriscode, and version if appropriate) of the rate form you wish to call.

3. Click **OK**. The CALL Statement appears in the rate form.

Done Statement

Purpose

The DONE Statement stops processing of the rate schedule. It can be used with IF-THEN-ELSE to stop processing an account's bill when a specified condition occurs.

Format

The DONE Statement consists only of the keyword DONE.

Example

If the transaction type is Cancel, create the CIS transaction record and quit the rate schedule.

```
IF ((BILL_TYPE = "CANCEL") OR (BILL_TYPE = "CANCEL/REBILL")) THEN
    /* Force CISREC to be a stem identifier */
    CISREC.DUMMY = 0;
    SAVE CISREC TO CIS SECTION "CANCEL";
    CLEAR CISREC;
        IF (BILL_TYPE = "CANCEL") THEN
        $EFFECTIVE_REVENUE = $0.00
        DONE;
        END IF;
END IF;
```

To Create

1. Select **Statements-›Done**.

    The DONE Statement appears in your rate form.

For Each Statements

FOR EACH statements direct the Rules Language to repeat a set of statements for each item you specify. There are several types of FOR EACH statements, including:

- **For Each x in Channel Statement**
- **For Each x in Factor Statement**
- **For Each x In List Statement**
- **For Each x In Number Statement**
- **For Each x In Override Statement**
- **For Each x In Recordlist Statement**
- **For Each x In Set Statement**
- **For Each x In Week Statement**
- **For Each x In Distribution Node Statement**
- **For Each x In CSV File Statement**
- **For Each x In COM IENUM Statement**

For Each x in Channel Statement

Purpose

The FOR EACH X IN CHANNEL Statement repeats one or more nested statements for each cut in a channel. It is used to apply operations to a collection of like or unlike items that are related to a given channel. For example, you could use it to process a set of peak values for a given channel.

Format

```
FOR EACH <identifier> IN CHANNEL <recorder,channel>,[:<start_date>,
<stop_date>]
    <nested_statements>
END FOR;
```

Example

Load and total each interval data cut in channel 1700,1.

```
FOR EACH CUT IN CHANNEL "1700,1"
    TOTAL = TOTAL + CUT
END FOR;
```

The rate schedule would repeat the nested statement once for each cut in the channel:

To Create

1. Select **Statement-›For Each x In-›Channel**.

   The FOR EACH Statement template appears.

2. Complete the template:

   Identifier: Supply a temporary identifier of your choosing. The program will assign it to each expression during processing by the FOR EACH loop. This identifier has no requirement beyond serving as a temporary variable in this statement. **Do not use any of the reserved words listed in Appendix A, such as CHANNEL, or any of the database table or column names.** This would cause errors in the program. You can use ITEM instead.

   Recorder,Channel: Supply the recorder,channel. This can either be in the "recorder,channel" format, or the "recorder,channel:start_date_time,stop_date_time" format. For the "recorder,channel" format, the FOR EACH loop processes each cut that intersects the current bill period. For the "recorder,channel:start_date_time,stop_date_time" format, it processes each cut that intersects with the specified time period.

   This can also be an identifier that was previously assigned to a channel in the rate form.

   Date Range: *Optional.* Supply a start date and stop date that indicate the date range the FOR EACH loop applies to.

3. Click **OK**. The first and last clauses of the statement appear in the rate form. Add the nested statements to apply to each factor record, using any of the other statement types described in this chapter.

For Each x in Factor Statement

Purpose

The FOR EACH x IN FACTOR Statement repeats a set of statements for each factor value that was in effect for an account during the current bill period. The factor value is considered in effect if it overlaps the bill period in any way.

This statement type is typically used when the unit price (factor value) for a bill determinant changed one or more times during the bill period, and you want to calculate and display the revenue for each portion of the total period for which a different factor value was in effect (that is, prorate the energy rather than the factor).

You can also allow the software to calculate a prorated value for the factor and apply that value in the rate form. For example, for the factor 'ENERGY CHARGE', if its value changed three times during the month of September, once every ten days, from $.12, to $.14, to $.15, and the PRORATE? flag is set to Yes in the FACTORVALUE Table for this factor, the software would calculate a single prorated factor value to apply during the period:

```
( (.12 x 10) + (.14 x 10) + (.15 x 10))  / 30 = .136
```

You could also calculate charges by applying each individual factor value to the energy, based on the portion of the bill period for which each factor was in effect. In this case, each factor was in effect for 1/3 of the bill period, so each factor is applied to 1/3 of the energy. You can then add the individual charges to get a total charge associated with the factor. The FOR EACH x IN FACTOR makes this possible.

In the FOR EACH x IN FACTOR Statement, you specify the factor of interest. The program automatically retrieves all of the records from the Factor Value Table that apply to the current account for the current bill period. (A factor value is considered in effect if its start time is less than the bill period stop, and its stop time is NULL or after the bill period start.) For each retrieved record, the program repeats the statements that you supply immediately after the FOR EACH x IN FACTOR clause.

All of the values associated with each retrieved factor value are available for calculations and reporting by the statements nested in the FOR EACH Statement. These values consists of the following components:

Component	Description
FACTORNAME	Name of the factor, assigned in the FACTOR Lookup Table.
UOMCODE	Unit of Measure code, assigned to the factor in the FACTOR Lookup Table.
STARTTIME	Beginning of the period over which the factor value applies.
STOPTIME	End of the period. NULL signifies that the factor is still in effect. A value other than NULL signifies that the factor is no longer in effect.
VAL	Value of the factor during this period.
PRORATEMETHOD?	Will Oracle Utilities Billing Component prorate the factor? (On the Data Manager interface, this field is labelled "Prorate?".)

To apply the nested statements to the contents of a particular field, use the following convention to identify the field of interest: stem.component. The "stem" is the temporary identifier (x) you supply in the FOR EACH x FACTOR portion of the statement. The program assigns this identifier to each record while it is being processed in the FOR EACH loop. A "component" is the name that identifies an individual field in the record; it comes from the column name in the

FACTOR Table and the FACTORVALUE Table (e.g., STARTTIME, STOPTIME, VAL, or PRORATEMETHOD). Therefore, if the FOR EACH clause were FOR EACH **FCTR** IN FACTOR, you could refer to the stop time in the current record using the identifier **FCTR.STOPTIME**.

Format

```
FOR EACH <identifier> IN FACTOR <factor_code>
    <nested_statements>
END FOR;
```

Example

The following sample statement prorates energy for each factor in the bill period:

```
BILLDIFF = DAYDIFF(BILL_STOP, BILL_START); /*LENGTH OF BILL PERIOD */

FOR EACH FCTR IN FACTOR "ENERGY CHARGE"
    FACTDIFF = DAYDIFF(FCTR.STOPTIME, FCTR.STARTTIME);
    FCTKWH = KWH * (FCTDIFF/BILLDIFF);  /*ENERGY USE IN FACTOR PERIOD*/
    ALL FCTKWH CHARGE FCTR.VAL INTO $FCTCHRG;
    $TOTAL_KWH_CHRG = $TOTAL_KWH_CHRG + $FCTCHRG
END FOR;

CLEAR FCTKWH, $FCTCHRG;
```

Without the CLEAR Statement, the report would display the units and charges for FCTKWH and $FCTCHRG that the program computed in the last FOR EACH loop (the values from the earlier loops would have been overwritten by succeeding loops). Because those "residual" values have little meaning, you can use the CLEAR Statement to reset them to Null before the bill report is printed. See the CLEAR Statement description (page 5-2) for more information.

Notes

If the PRORATE? field in the FACTORVALUE Table is set to YES for the factor, Oracle Utilities Billing Component automatically prorates the factor. *In that case, do not apply the FOR EACH x IN FACTOR Statement to the factor value.* To use this statement to prorate the energy rather than the factor, using the FOR EACH x IN FACTOR Statement described in this section, you must be sure that the PRORATE? flag is set to NO for the factor value.

STARTTIME is always greater than or equal to the BILL_START; STOPTIME is always less than or equal to BILL_STOP.

To Create

1. Select **Statement-›For Each x In-›Factor**.

   The FOR EACH Statement template appears.

2. Complete the template:

   Identifier: Supply a temporary identifier of your choosing. The program will assign it to each retrieved factor value record during processing by the FOR EACH loop. This identifier has no requirement beyond serving as a temporary variable in this statement. **Do not use any of the reserved words listed in Appendix A of the Oracle Utilities Rules Language User's Guide, such as FACTOR, or any of the database table or column names.** This would cause errors in the program.

   Factor Code: To specify the lookup code that identifies the factor, position the pointer in the field and click the *right* mouse button. When the Rules Language Elements dialog appears, select **Factor Codes** from the top box. A list of available factors appears in the bottom box. Each is identified by its lookup code and name. Highlight the desired factor.

3. Click **OK**. The first and last clauses of the statement appear in the rate form. Add the nested statements that will apply to each factor record, using any of the other statement types described in this chapter.

For Each x In List Statement

Purpose

The FOR EACH x IN LIST statement repeats a set of nested statements for each item in a list. In other words, it repeats a set of nested statements for each item in the Oracle Utilities Data Repository that matches a set of user-defined criteria expressed in a Table.Column query, Customer/Account list, or query list.

Format

```
FOR EACH <identifier> IN LIST <list_name>
   <nested_statements>
END FOR;
```

Example

In the following example, the query for "LIST_CHAN" targets the UIDCHANNEL column in the CHANNEL Table. The query is WHERE Account.Accountid=ACCOUNT_ID.

```
FOR EACH CHANL IN LIST "LIST_CHAN"
   RC = "" + CHANL.RECORDERID + "," CHANL.CHANNELNUM;
   REPORT RC LABEL "Account Recorder, Channel";
END FOR;
```

This statement repeats the nested reporting statements for each channel that belongs to the account being billed. For example, suppose the account being billed has one recorder (1701) and three channel numbers 1, 2, and 3. The following lines would appear in the bill report:

```
Account Recorder,Channel
1701,1
1701,2
1701,3
```

Notes

Before you can apply a rate form containing a FOR EACH x IN LIST Statement, you must create a table.column or customer/account query that creates the list. You create these queries using Data Manager, and you save them in the Oracle Utilities Data Repository. See **Chapter 8**: **Working with Lists and Queries** in the *Data Manager User's Guide* for more information. You create query lists using the Lists function available through the Energy Information Platform user interface. See **Lists** on page 7-63 in the *Oracle Utilities Energy Information Platform User's Guide* for more information.

Queries enable you to specify the conditions that an item must meet in order for it to be included in the list, as well as the column in the Oracle Utilities Data Repository Table that you want to get the items from. For example, you can specify the Recorder ID column in the Recorder Table (RECORDER.RECORDERID) to get a list of Recorder IDs, or the group name column in the Channel Group Table (CHANNELGROUP.GROUPNAME) to get a list of channel groups. The query itself is often WHERE Account.Accountid=ACCOUNT_ID, which means retrieve the items belonging to the account currently being billed. The ACCOUNT_ID identifier has been set to the current account's ID in the Rules Language. Oracle Utilities Billing Component selects the Items from the table and column you specified, where the parent account is the one being billed.

About Stems and Components: Table.Column queries have another important feature you use. If the target table.column stores the unique identifier (UID) for the table, the program retrieves the UID for the item, as well as all other values in its database record. The program stores the

record in memory and assigns it the temporary identifier that you supply for x in the FOR EACH x in LIST Statement. If you used the clause FOR EACH MYITEM IN LIST CHANLIST, each record retrieved would be assigned the temporary identifier MYITEM for the purpose of processing by the following nested statements. Each field in the retrieved record would be identified in memory by the column name from the original target table. If the target table were the Channel Table, the following records would be available for processing with the nested statements: MYITEM.UIDCHANNEL, MYITEM.UIDRECORDER, MYITEM.CHANNELNUM (UIDCHANNEL, UIDRECORDER, and CHANNELNUM are the names of columns in the Channel Table). The temporary name of the record (in this example, MYITEM) is a "stem," and the column names are "components." See *Chapter Four* in the *Oracle Utilities Rules Language User's Guide* for more information about stem.component identifiers.

Appendix A: **Oracle Utilities Data Repository Database Schema** in the *Oracle Utilities Energy Information Platform Configuration Guide* contains a diagram of the database schema. It shows each table in the database, including its name (all caps), the unique identifiers (underlined), and the column names. This information is very helpful when constructing your queries and FOR EACH statements.

Displaying the List SQL

You can also include the SQL statements executed by the Rules Language in the Rules Language. The SQL statements are inserted as comments directly above the FOR EACH x IN LIST statement in the rate form.

How to include list SQL in a rate form:

1. Select the line in the rate form that contains the FOR EACH x IN LIST statement.

2. Select **Statement-›Include List SQL**.

   The SQL executed by the Rules Language appears in the rate form as comments directly above the FOR EACH x IN LIST statement.

Note: This feature can be used with both Account/Customer lists, as well as table-column lists.

To Create

1. Select **Statement-›For Each x In-›List**.

   The FOR EACH Statement template appears.

2. Complete the template:

   Identifier: Supply a temporary identifier of your choosing. The program will assign it to each member of the list during processing by the FOR EACH loop. This identifier has no requirement beyond serving as a temporary variable in this statement. **Do not use any of the reserved words listed in Appendix A, such as CHANNEL, or any of the database table or column names.** This would cause errors in the program. You can use CHANL instead.

   List Name Expression: To specify the table.column or customer/account list query that will retrieve the desired items, position the pointer in the field and click the *right* mouse button. When the Rules Language Elements dialog appears, select **Table-Column Lists** from the top box. A list of available list queries appears in the bottom box, each identified by its list name, target table, and target column. Highlight the desired list in the right box.

   > **Note:** If the list name you wish to use contains a percent sign (%), the list name MUST appear as a string in the rate form.

   > **Note**: Query lists created using the Lists function of the Energy Information Platform user interface cannot be selected using the Rules Language Elements Editor.

3. Click **OK**. The first and last clauses of the statement appear in the rate form. Add the nested statements using any of the other statement types described in this chapter.

For Each x In Number Statement

Purpose

The FOR EACH x IN NUMBER Statement is used with an expression that yields an integer. For each iteration of the nested statements, the identifier value is set first to 1, then to 2, and so on up through the value of the expression or number.

Format

```
FOR EACH <identifier> IN NUMBER <expression>
    <nested_statements>
END FOR;
```

Example

In the following example, the function HISTCOUNT is applied to count the number of values loaded by a previous statement in the rate form. For the account currently being processed, the temporary identifier NUMKW is assigned an integer. This will report every historical value in the Meter Value Table for the account. The "- 1" in the HISTCOUNT function is used to exclude the current bill period. "Recorder-id,channel" is the name stored in the Meter Value Table.

```
NUMKW = HISTCOUNT (OLDMVKW.VAL) - 1;

FOR EACH I IN NUMBER NUMKW
    RPTMV.READDATE = HISTVALUE(OLDMVKW.READDATE , I);
    RPTMV.CH = HISTVALUE(OLDMVKW.NAME, I);
    RPTMV.ENERGY = HISTVALUE(OLDMVKW.VAL , I);
    REPORT RPTMV LABEL "Historic Dates Channel Values";
    CLEAR RPTMV;
END FOR;
```

The values would be formatted in the report as shown:

READDATE	CH	ENERGY
03/31/1997	1701,1	150
02/28/1997	1701,1	200
01/31/1997	1701,1	100

To Create

1. Select **Statement-›For Each x In-›Number**.

   The FOR EACH Statement template appears.

2. Complete the template:

   Identifier: Supply a temporary identifier of your choosing. The program will assign it to each retrieved record during processing by the FOR EACH loop. This identifier has no requirement beyond serving as a temporary variable in this statement. **Do not use any of the reserved words listed in Appendix A, such as CHANNEL, or any of the database table or column names.** This would cause errors in the program.

   Integer Expression: Supply an expression whose value is an integer, or can be rounded to an integer. If the expression is an unassigned identifier or has a value of zero or less, the statement will not execute. Otherwise the identifier is set to 1 for the first iteration, then 2, ... and finally the value of the expression on the last iteration. After the last iteration the identifier will be set to the last value plus one. The identifier value may be different if a LEAVE FOR statement is executed. It is an error if the expression does not evaluate to a number.

3. Click **OK**. The first and last clauses of the statement appear in the rate form. Add the nested statements using any of the other statement types described in this chapter.

For Each x In Override Statement

Purpose

The FOR EACH x IN OVERRIDE Statement repeats a set of statements for each override that was in effect for an account during the current bill period. The override is considered in effect if it overlaps the bill period in any way.

In the FOR EACH x IN OVERRIDE Statement, you specify the override. The program automatically retrieves all of the records that apply the override to the current account for the current bill period. If the override applies to an account at the account level, the records are retrieved from the Override History Table (ACCOUNTOVERRIDEHIST). If the override applies to an account at the recorder channel, channel group, or CIS account level, the records are retrieved from the Name Override History Table (ACCOUNTNAMEOVERHIST). For each retrieved record, the program repeats the statements that you supply immediately after the FOR EACH x IN OVERRIDE clause.

All the values in each retrieved override history record are available for calculations and reporting by the statements nested in the FOR EACH Statement. An override history record consists of the following fields:

From **ACCOUNTOVERRIDEHIST**

Column Name	Description
STARTTIME	Beginning of the period over which the override applies
STOPTIME	End of the period (Null if still in effect)
VAL	(*Optional*) Value that express the magnitude of the override
STRVAL	(*Optional*) Value for a string variable (utility-defined; could be an informative note, or a qualifier for the override).

From **ACCOUNTNAMEOVERRIDEHIST**

Column Name	Description
NAME	Name of the channel group, CIS account, or 'recorder-id, channel' to which the override applies
STARTTIME	Beginning of the period over which the override applies
STOPTIME	End of the period (Null if still in effect)
VAL	(*Optional*) Value that express the magnitude of the name override
STRVAL	(*Optional*) Value for a string variable (utility-defined; could be an informative note, or a qualifier for the override).

You can identify these fields in the nested statements using the *stem.component* convention. In this case the stem is the temporary identifier (*x*) you supply in the FOR EACH x IN OVERRIDE portion of the statement. The program assigns this identifier to each record while it is being processed in the FOR EACH loop. A component is the name that identifies a single value in the record; it comes from the column name in the Override Table (e.g., HISTSTARTTIME, HISTSTOPTIME, VAL, or STRVAL). The following example uses the stem.components EVENT.HISTSTARTTIME and EVENT.HISTSTOPTIME.

Format

```
FOR EACH <identifier> IN OVERRIDE <override_code>
    <nested_statements>
END FOR;
```

Example

The following sample statements would count the number of times the account was interrupted:

```
COUNT_INTR = 0;
FOR EACH EVENT IN OVERRIDE "INTERRUPT"
    COUNT_INTR = COUNT_INTR +1;
END FOR;
```

Notes

STARTTIME is always greater than or equal to the BILL_START; STOPTIME is always less than or equal to BILL_STOP.

To Create

1. Select **Statement-›For Each x In-›Override**.

    The FOR EACH Statement template appears

2. Complete the template:

    Identifier: Supply a temporary identifier of your choosing. The program will assign it to each retrieved override record during processing by the FOR EACH loop. This identifier has no requirement beyond serving as a temporary variable in this statement. **Do not use any of the reserved words listed in Appendix A, such as OVERRIDE, or any of the database table or column names.** This would cause errors in the program. You could use EVENT instead.

    Override Code Expression: To specify the lookup code that identifies the override to process, position the pointer in the field and click the *right* mouse button. When the Rules Language Elements dialog appears, select **Override Codes** from the left box. A list of available overrides appears in the right box, each identified by its lookup code and name. Highlight the desired override.

3. Click **OK**. The first and last clauses of the statement appear in the rate form. Add the nested statements that you want to apply to each override history record, using any of the other statement types described in this chapter.

For Each x In Recordlist Statement

Purpose

The FOR EACH x IN RECORDLIST statement repeats a set of nested statements for each item in a table-column list. In other words, it repeats a set of nested statements for each item in the LODESTAR Data Repository that matches a set of user-defined criteria expressed in a Table.Column query.

The FOR EACH x IN RECORDLIST statement returns the entire database record for each record in the list query. Individual column values can be obtained or set using a STEM.TAIL syntax, where STEM is the identifier specified in the statement call, and TAIL is the name of the specific column you wish to get or set.

Format

```
FOR EACH <identifier> IN RECORDLIST <list_name>
    <nested_statements>
END FOR;
```

Example

The following example updates the Status Code of each Account record in the "JURIS_ACCT_LIST" list to "ACTIVE". In this example, the query for "JURIS_ACCT_LIST" is WHERE ACCOUNT.JURISCODE=JURIS_CODE (an identifier set elsewhere in the rate form).

```
FOR EACH ACCT IN RECORDLIST "JURIS_ACCT_LIST"
    ACCT.ACCOUNTSTATUSCODE = "ACTIVE";
    SAVE ACCT TO TABLE ACCOUNT;
END FOR;
```

Notes

Before you can apply a rate form containing a FOR EACH x IN RECORDLIST Statement, you must create a table.column query that creates the list. You create these queries using Data Manager, and you save them in the LODESTAR Data Repository. See **Chapter 8**: **Working with Lists and Queries** in the *Data Manager User's Guide* for more information.

> **Note:** If the list name you wish to use contains a percent sign (%), the list name MUST appear as a string in the rate form.

Appendix A: **Oracle Utilities Data Repository Database Schema** in the *Oracle Utilities Energy Information Platform Configuration Guide* contains a diagram of the database schema. It shows each table in the database, including its name (all caps), the unique identifiers (underlined), and the column names. This information is very helpful when constructing your queries and FOR EACH statements.

To Create

1. Select **Statement-›For Each x In-›RecordList**.

   The FOR EACH Statement template appears.

2. Complete the template:

   Identifier: Supply a temporary identifier of your choosing. The program will assign it to each member of the list during processing by the FOR EACH loop. This identifier has no requirement beyond serving as a temporary variable in this statement. **Do not use any of the reserved words listed in Appendix A, such as CHANNEL, or any of the database table or column names.**

   List Name: To specify the table.column list query that will retrieve the desired items, position the pointer in the field and click the *right* mouse button. When the Rules Language Elements dialog appears, select **Table-Column Lists** from the top box. A list of available list

queries appears in the bottom box, each identified by its list name, target table, and target column.

> **Note**: This statement works only with lists that return the full database record. List queries that return only a specific column cannot be used with FOR EACH x in RECORDLIST.

3. Highlight the desired list in the right box and click **OK**. The first and last clauses of the statement appear in the rate form.

For Each x In Set Statement

Purpose

The FOR EACH x IN SET Statement repeats one or more nested statements for each key value in a list of expressions or in an array identifier. It is used to apply operations to a collection of like or unlike items. For example, you could use it to process a set of charges for an account.

Format

```
FOR EACH <identifier> IN SET <expression1>[,<expression2>...]
    <nested_statements>
END FOR;
```

Examples

For the purposes of this example, assume that the value for the factor FUEL_CHARGE is $.10; for DISTRIBUTION_CHARGE, $.07. The account used 100 kWh during the bill period.

```
ENERGY_CHARGE = $.05
FUEL_CHARGE = FACTOR["FUEL_CHARGE"].VALUE
DISTRIBUTION_CHARGE = FACTOR["DISTRIBUTION_CHARGE"].VAL
FOR EACH ITEM IN SET ENERGY_CHARGE, FUEL_CHARGE, DISTRIBUTION_CHARGE
    $TOTAL_CHARGE = $TOTAL_CHARGE + (KWH * ITEM)
END FOR;

REVENUE  $TOTAL_CHARGE "Total charge"
```

To perform the same logic using an array identifier:

```
#ARR_CHRG[1] = $.05
#ARR_CHRG[2] = FACTOR["FUEL_CHARGE"].VALUE
#ARR_CHRG[3] = FACTOR["DISTRIBUTION_CHARGE"].VAL
FOR EACH ITEM IN SET #ARR_CHRG[]
    $TOTAL_CHARGE = $TOTAL_CHARGE + (KWH * ITEM)
END FOR;
```

In both examples, the rate schedule would repeat the nested statement three times, once for each item, as follows:

0 + (100 * .05) (energy charge)
5.00 + (100 * .10) (fuel charge)
15.00 + (100 * .07) (distribution charge)

The following line would be displayed in the revenue section of the bill report.

```
Total charge            $22.00
```

To Create

1. Select **Statement-›For Each x In-›Set**.

   The FOR EACH Statement template appears.

2. Complete the template:

   Identifier: Supply a temporary identifier of your choosing. The program will assign it to each expression during processing by the FOR EACH loop. This identifier has no requirement beyond serving a temporary variable in this statement. **Do not use any of the reserved words listed in Appendix A, such as CHARGE, or any of the database table or column names.** This would cause errors in the program. You could use ITEM instead.

   Expressions: Supply one or more expression. Each "expression" may be virtually any type of item that can be processed by the following nested statements; for example, identifiers, determinants, mathematical operations, recorder-id,channel-numbers, and so on.

   You can also supply an array identifier, in which case the statement will execute the nested statements for each record in the array.

   Note: The array should NOT contain an identifier in the index position (between the brackets [X])

3. Click **OK**. The first and last clauses of the statement appear in the rate form. Add the nested statements that you want to apply to each factor record, using any of the other statement types described in this chapter.

For Each x In Week Statement

Purpose

The FOR EACH x IN WEEK Statement repeats a set of nested statements for each week in a bill period. If a week overlaps the start of the bill period, it is included. If it overlaps the end of the bill period, it is excluded.

For each week, the programs retrieves two values: the week's start and stop times. You can refer to these values using the convention stem.STARTTIME and stem.STOPTIME, where *stem* is the identifier you supply after FOR EACH. In the example below, the stem is WK.

Format

```
FOR EACH <identifier> IN WEEK <week_day>
   <nested_statements>
END FOR;
```

Example

The following sample statements find the account's peak for each week in the bill period.

```
FOR EACH WK IN WEEK "SUNDAY"
   WK_HNDL = INTDLOADDATES ('1701,1',WK.STARTTIME,WK.STOPTIME);
   WK_PEAK = WK_HNDL.MAX;
END FOR;
```

To Create

1. Select **Statement-›For Each x In-›Week**.

   The FOR EACH Statement template appears.

2. Complete the template:

   Identifier: Supply a temporary identifier of your choosing. The program will assign it to each week record while it is processed by the FOR EACH loop. This identifier has no requirement beyond serving as a temporary variable in this statement. **Do not use any of the reserved**

words listed in Appendix A, such as WEEK, or any of the database table or column names. This would cause errors in the program.

Start Day of Week: Select the day to use as the first day of the week.

3. Click **OK**. The first and last clauses of the statement appear in the rate form. Add the nested statements using any of the other statement types described in this chapter.

For Each x In Distribution Node Statement

Purpose

The FOR EACH x IN DISTRIBUTIONNODE Statement repeats a set of nested statements for each record in the Distribution Node or Distribution Node History Table whose STARTTIME / STOPTIME overlaps the current bill period. This date range can be changed by setting the identifiers DISTNODE_HIST_START and DISTNODE_HIST_STOP to cover a different date range.

> **Note:** This statement is only available if you have a DISTRIBUTIONNODE table in your database.

Values assigned to the DISTNODE_HIST_START and DISTNODE_HIST_STOP identifiers must be in single quotes (' ').

Format

```
FOR EACH <identifier> IN DISTRIBUTIONNODE <node-id>
    <nested_statements>
END FOR;
```

Where:

• <node-id> is any expression whose value is a string that is a node ID in a DISTRIBUTIONNODE record.

Example

The following sample statements obtain the values of specified attributes of each distribution node for inclusion in a report.

```
FOR EACH X IN DISTRIBUTIONNODE "0152"
    NODE.ID = X.NODEID;
    NODE.NAME = X.NAME;
    NODE.NOTE = X.NOTE;
    NODE.KWREC = X.KWRECORDERID;
    NODE.KWCHAN = X.KWCHANNELNUM;
    REPORT NODE LABEL "DISTRIBUTION_NODE INFORMATION";
    CLEAR NODE;
END FOR;
```

To Create

1. Select **Statement-›For Each x In-›Distribution Node**.

   The FOR EACH Statement template appears.

2. Complete the template:

   Identifier: Supply a temporary identifier of your choosing. The program will assign it to each record while it is processed by the FOR EACH loop. This identifier has no requirement beyond serving as a temporary variable in this statement. **Do not use any of the reserved words listed in Appendix A, such as WEEK, or any of the database table or column names.** This would cause errors in the program.

   Node ID Expression: Select the Node ID.

3. Click **OK**. The first and last clauses of the statement appear in the rate form. Add the nested statements using any of the other statement types described in this chapter.

For Each x In CSV File Statement

Purpose

The FOR EACH x IN CSV FILE Statement repeats a set of nested statements for each line or record in a specified CSV (comma separated values) file.

Format

```
FOR EACH <identifier> IN CSV FILE <csv-filename>
    <nested_statements>
END FOR;
```

Where:

• <csv-filename> is any expression whose value is a string that is the name of a file.

Example

Display and label the values in a csv file.

```
FILENAME = "d:\lodestar\user\d377.lse";
LABEL X.COLUMN1 "Column 1";
LABEL X.COLUMN2 "Column 2";
LABEL X.COLUMN3 "Column 3";
LABEL X.COLUMN4 "Column 4";
LABEL X.COLUMN5 "Column 5";
FOR EACH X IN CSV_FILE FILENAME
    REPORT X LABEL "File: " + FILENAME;
END FOR;
```

To Create

1. Select **Statement->For Each x In->CSV File**.

    The FOR EACH Statement template appears.

2. Complete the template:

    Identifier: Supply a temporary identifier of your choosing. The program will assign it to each record while it is processed by the FOR EACH loop. This identifier has no requirement beyond serving as a temporary variable in this statement. **Do not use any of the reserved words listed in Appendix A, such as WEEK, or any of the database table or column names.** This would cause errors in the program.

    File Name Expression: Enter the file name of the CSV file.

3. Click **OK**. The first and last clauses of the statement appear in the rate form. Add the nested statements, using any of the other statement types described in this chapter.

Notes

The CSV file must be an ASCII text file. Each line or record must be less than 4095 characters. Records are read one at a time, and the nested statements are processed after all columns in a record are assigned. The column values are retrieved and assigned to stem.tail identifiers, where the stem is the FOR EACH statement identifier and the tails are COLUMN1, COLUMN2, ... Leading and trailing spaces around commas are ignored. If a field starts with a double quote ("), then it must end with another double quote. All characters within the quotes are used, and a string value is assigned for this column. Otherwise, blanks are removed from the beginning and end of the field, and the characters in the field are used to determine if the field is an integer, bill month, float, or date. The maximum lengths for conversions are 10 numbers to an integer, exactly seven

characters for a bill month, 25 characters to a float, and 19 characters to a date; longer values are assumed to be strings, even if all characters are numbers. If the value contains characters that are not part of an integer, float, or date, the field is loaded as a string.

For Each x In COM IENUM Statement

Purpose

The FOR EACH x IN COM IENUM Statement repeats a set of nested statements for each variant in a COM object.

See **COM Object Functions** in **Chapter 8**: **Working with COM Components** in the *Oracle Utilities Rules Language User's Guide* for more information about using this statement.

Format

```
FOR EACH <identifier> IN IENUM <expression>
   <nested_statements>
END FOR;
```

Where:

• <expression> is any expression whose value is a reference to a COM object.

Example

The following sample statements set the values of the "ACCOUNTID" nodes in an XML document to the value of the "TEXT" property.

```
OBJECT = CREATEOBJECT ("MSXML.DOMDocument");
XMLNODES = OBJECT->SELECTNODES ("//ACCOUNTID");
FOR EACH X IN IENUM XMLNODES
   ACCOUNTID = X->TEXT;
END FOR;
```

To Create

1. Select **Statement-›For Each x In-›COM ENUM**.

   The FOR EACH Statement template appears.

2. Complete the template:

   Identifier: Supply a temporary identifier of your choosing. The program will assign it to each record while it is processed by the FOR EACH loop. This identifier has no requirement beyond serving as a temporary variable in this statement. **Do not use any of the reserved words listed in Appendix A, such as WEEK, or any of the database table or column names.** This would cause errors in the program.

   COM Enumeration: Select the COM Object.

3. Click **OK**. The first and last clauses of the statement appear in the rate form. Add the nested statements using any of the other statement types described in this chapter.

Notes

The COM object file must have been previously created within the rate form using the **CREATEOBJECT Function**.

If-Then-Else Statement

Purpose

IF-THEN-ELSE statements direct the program to evaluate a condition and take action based on that condition. If the condition is met, the program takes the action specified in the THEN clause; if the condition is not met, the program takes the action specified in the ELSE clause. You define the actions to be taken using other statement types, such as ASSIGNMENT or ALL statements, nested within the IF-THEN-ELSE Statement.

Format

If-Then-Else statements are constructed according to the following format:

```
IF <logical_expression> THEN
    <nested_statements>
[ELSE
    <nested_statements>]
END IF;
```

Examples

Here are three sample IF-THEN-ELSE statements:

Example 1: Compare a minimum charge to the energy charge; always use the larger of the two.

```
IF ($MIN_CHARGE > $ENERGY_CHARGE)
    THEN $EFFECTIVE_REVENUE = $CUST_CHARGE + $MIN_CHARGE;
    ELSE
    $EFFECTIVE_REVENUE = $CUST_CHARGE + $ENERGY_CHARGE;
END IF;
```

Example 2: If the demand exceeds 300 kW, add in an RKVA charge.

```
IF KW > 300 THEN
    ALL RKVA CHARGE $0.35 INTO $RKVA_CHARGE;
    /* ELSE is default $RKVA_CHARGE = $0.00 */
END IF;
```

Example 3: If the currently computed billing kW is zero, use the recorded kW.

```
IF ((BILL_KW = 0) AND (KW > 0)) THEN
    BILL_KW = KW;
END IF;
```

To Create

1. Select **Statements-›If Then Else**.

    The IF THEN ELSE Statement template appears.

2. In the **IF** field, construct a logical expression. The expression can compare two identifiers, or an identifier and a constant, using any of the standard relational operators:

    - < (less than)

    - <= (less than or equal to)

    - = (equals)

    - > (greater than)

    - >= (greater than or equal to)

    - <> (not equal to)

    Note: When comparing strings, the programs use the ACSII values of the characters to evaluate the clauses.

The expression can also contain the logical connectors AND and OR to join two or more simple expressions, each of which must be enclosed in parentheses. The entire logical expression may also be enclosed in parentheses, but that's optional. (See example 3 above.)

3. *Optional.* If you intend to specify an ELSE subclause (that is, the action the program takes if the result of logical expression is false), check the **ELSE** box. (If you do not supply an ELSE subclause, the programs assume the default if the logical expression is false: take no action.)

4. Click **OK**. The IF portion of the statement appears in the rate form.

   The final clause of the statement, "END IF;" is automatically supplied.

5. Using any of the other statement types (ALL, ASSIGNMENT, etc.), specify the action you want to occur if the condition specified in the logical expression is true. Select **Statements-›[statement type]**, and complete the statement template as desired. Click **OK**. Notice that the statement appears in the rate form, indented under "THEN". This is a nested statement. You can have any number of nested statements.

6. If you checked the ELSE box, you must specify the action you want the program to take if the logical expression is false for the account. Highlight "ELSE" in the rate form, and use any statement type to specify the action. Notice that this time, the completed statement appears indented under "ELSE". You can add any number of nested statements.

Include Statement

Purpose

The INCLUDE Statement is used to insert the contents of one rate form into another. When the Oracle Utilities software processes a rate form, it automatically replaces any INCLUDE statements with the included rate form.

This feature enables you to save a set of standardized statements in several other rate schedules, riders, or contracts without tedious retyping. This approach is especially time-saving when dealing with calculations or other operations that are used widely and are expected to change over time. You need only modify the included rate form; and all rate forms that include it are in effect updated automatically.

You can create a whole library of useful routines, such as saving data to the databases or writing to the CIS transaction records, and INCLUDE them in your rate schedules as needed. You can also include account-specific contracts in a rate schedule.

The riders or contracts must already exist before you can INCLUDE them in the current rate form.

There are two versions of the INCLUDE Statement:

- **INCLUDE RIDER:** A "rider" refers to any "subform" that you might want to include; a tariff rider, or a set of statements that executes a frequently-used routine.

- **INCLUDE CONTRACT:** A "contract" is a set of statements used to tailor a rate schedule to a particular account. You can think of a contract as a rider that makes the rate schedule account-specific. A contract can consist of different "sections," such as an Energy section and a Discount section. You can include selected sections at different points in the rate schedule.

Note: All Oracle Utilities Rate Management analyses, except Single Step, ignore INCLUDEd contracts.

Format

The format for Include statements is:

```
INCLUDE <rate_form_name_constant>;
```

Examples

```
INCLUDE "R1";
/* Including company and jurisdiction, rate R1, */
/* current or historical version */

INCLUDE "UMS:MA:1-GL:1";
/* Company UMS, Jurisdiction MA, rate 1-GL, */
/* Trial version 1 */
```

Notes

There are some rules to keep in mind when applying the INCLUDE Statement:

- A rate schedule can include one or more riders or contracts.

- A contract can include a rider.

- Riders can include other riders or contracts.

- A single rate schedule can contain up to a maximum of 20 riders and/or contracts (including those included using both the **Include Statement** and the **Call Statement**).

There are additional limitations for trial versus current versions. The table below indicate what can be included in each type of rate form:

Rate Form Type	Trial version	Current version
Rate Schedule	Trial rider or contract Current rider or contract	Current rider or contract
Contract	Trial rider Current rider	Current rider or contract
Rider	Trial Rider contract Current rider or contract	Current rider or contract

Another important point about the INCLUDE Statement is that the included rate form is literally included in the other when the rate form is processed. The two rate forms share all identifiers. (For that reason, if you have identifiers that are intended for use only within a particular rate form, you should identify them with an appropriate prefix.)

When you include one rate form in another, the statements that make up the included rate form are not visible. To view the included statements, select **Statements-›View-›Expand Includes**. You cannot edit this view, but you can print it.

To Create — Include Contract

1. Select **Statements-›Include-›Contract**.

   The INCLUDE CONTRACT Statement template appears.

2. *Optional. I*f you only want to include a section or piece of the contract at this point in your rate form, input the identifier of that section.

3. Click **OK**. The INCLUDE Statement appears in your rate form. To view the INCLUDEd statements, select **Statements-›View-›Expands Includes**. You cannot edit the expanded view, but you can print it.

To Create — Include Rider

1. Select **Statements-›Include-›Rider**.

   The INCLUDE Statement dialog appears.

2. Complete the dialog to select the rider you wish to include. The dialog allows you to examine different versions of the rider before making your selection:

   Select the **Operating Company** and **Jurisdiction** that the rider belongs to.

   Select the version(s) of the rider that you're interested in by placing a checkmark in either or both boxes: **Current** (the version currently in effect) or **Trial**. A list of all riders that match your criteria appears in the middle list box.

   To view the statements that make up a particular rider, highlight it in the middle list box. The statements appear in the lower box.

   To make your selection, highlight the rider in the middle list box

3. Click **OK**. The INCLUDE Statement appears in your rate form. To view the INCLUDEd statements, select **Statements-›View-›Expands Includes**. You cannot edit the expanded view, but you can print it.

Leave For Statement

Purpose

The LEAVE FOR Statement exits the nearest enclosing FOR EACH Statement. LEAVE FOR has no effect if it is not inside a FOR EACH Statement.

Format

The LEAVE FOR Statement consists only of the keywords LEAVE FOR.

To Create

1. Select **Statements-›Leave For**.

   The LEAVE FOR statement appears in your rate form.

Leave Rider Statement

Purpose

The LEAVE RIDER Statement can be used within an INCLUDE'd or CALL'ed rider or contract to terminate it. The LEAVE RIDER statement exits to the statement after the INCLUDE or CALL statement. The LEAVE RIDER has no effect if it is not inside a rider or contract.

Format

The LEAVE RIDER Statement consists only of the keywords LEAVE RIDER.

To Create

1. Select **Statements-›Leave Rider**.

   The LEAVE RIDER statement appears in your rate form.

   Note: The Leave Rider statement only appears on the Statements menu when a rider or contract is the open document in the Rules Language Editor.

Next For Statement

Purpose

The NEXT FOR Statement is used within a FOR EACH loop to skip the remaining statements in the FOR LOOP. NEXT FOR has no effect if it is not inside a FOR EACH Statement.

Format

The NEXT FOR Statement consists only of the keywords NEXT FOR.

To Create

1. Select **Statements-›Next For**.

   The NEXT FOR statement appears in your rate form.

Novalue Statement

Purpose

The NOVALUE Statement assigns a value to an identifier whose value would otherwise be Null at the time the statement is executed. This statement is used to supply a value for an empty customer record field. That is, if the database field referenced by the identifier is empty, the program computes a value for it using the statements nested in the NOVALUE Statement.

The NOVALUE Statement is not limited to database or determinant identifiers; it can also refer to other identifiers that are not yet defined.

Format

The format for NOVALUE statements is:

```
NOVALUE <identifier>
   [<nested_statements>]
   <identifier> = <constant|expression|function>;
END NOVALUE;
```

Example

If the account's current bill history record has no value for KW, compute one by dividing the account's value for KWH by 200. Use the computed value for KW in the ALL Statement

```
NOVALUE KW
    KW = KWH /200;
END NOVALUE;
ALL KW CHARGE $1.00 INTO DEMAND_CHARGE;
```

To Create

1. Select **Statements-›Novalue**.

    The NOVALUE Statement template appears.

    In the **Identifier** field, specify the desired identifier.

2. Click **OK**. The statement appears in the rate form.

    Apply any of the statement types to assign or compute a value for the identifier. Select **Statements-›[statement type]**. Complete the statement template as desired and click **OK**. You can supply as many nested statements here as necessary. The Rules Language Editor automatically supplies the required "END NOVALUE;".

Section Statement

Purpose

The SECTION Statement separates lines in a contract that should take effect at different times. The SECTION Statement is only available in a contract. The INCLUDE CONTRACT SECTION ...; Statement can be used to include those lines in the contract that belong within a section.

Oracle recommends that either all lines in a contract reside in sections, or none do. The Section name should indicate the determinant or other value that is computed in the section.

Format

```
SECTION <name>
...
END SECTION;
```

Note: SECTION and END SECTION are required keywords. The variable <name> is a used to designate a section in the contract.

Example

```
SECTION KWH;
KWH_HNDL = INTDLOAD(KWH);
KWH = INTDVALUE(KWH_HNDL, "ENERGY");
END SECTION;
```

To Create

1. Select **Statements-›Section**.

   A Section entry selection dialog box is displayed.

2. Enter a Section name to create a SECTION. You can then add lines within the section.

3. Click **OK** to insert the Section Statement after the last line highlighted in the rate form.

Select Bill_Period Statement

Purpose

The SELECT BILL_PERIOD Statement makes it possible to apply different rates during different seasons. For example, a rate class might receive a voltage discount during the winter months, but not in the summer. The seasons are defined in the season schedules, which you create with Data Manager and store in the Oracle Utilities Data Repository. The rates are defined by other statement types, such as Block statements or ASSIGNMENT statements, nested within the SELECT BILL_PERIOD Statement.

Format

The SELECT BILL_PERIOD Statement format is:

```
SELECT BILL_PERIOD
   WHEN <season_name> [, <season_name>]
   <nested_statements>
   WHEN <season_name> [, <season_name>]
   <nested_statements>
   ...
   OTHERWISE
   <nested_statements>
END SELECT;
```

Examples

This example is a Select Statement used to define seasonal Block Rates with different limits and charges.

```
SELECT BILL_PERIOD
   WHEN "WINTER"
      BLOCK KWH
      FROM 0 TO 400 CHARGE $0.03709
      FROM 400 CHARGE $0.03000
      TOTAL $ENERGY_CHARGE_WIN;
   WHEN "SUMMER"
      BLOCK KWH
      FROM 0 TO 700 CHARGE $0.06542
      FROM 700 TO 1100 CHARGE $0.05339
      FROM 1100 CHARGE $0.04238
      TOTAL $ENERGY_CHARGE_SUM;
END SELECT;
$ENERGY_CHARGE = $ENERGY_CHARGE_WIN + $ENERGY_CHARGE_SUM;
```

Here's another example of a Select Statement, used to define seasonal rates.

```
SELECT BILL_PERIOD
   WHEN "SUMMER"
      $VOLTAGE_DISCOUNT_SUM = $0.00
   WHEN "WINTER"
      $VOLTAGE_DISCOUNT_WIN = $200.00
   OTHERWISE
      $VOLTAGE_DISCOUNT_OTHER = $100.00
END SELECT;
$VOLTAGE_DISCOUNT = $VOLTAGE_DISCOUNT_SUM + $VOLTAGE_DISCOUNT_WIN +
$VOLTAGE_DISCOUNT_OTHER;
```

Notes

Season schedules are stored in the Oracle Utilities Data Repository. You can view, create, and modify a Season Schedule by selecting **File-›Setup-› Season Schedules** (see the *Data Manager's User Guide* for details). Each season schedule consists of two or more periods, such as "Summer" and "Winter." Each period covers a specific date range, such as 04/01/1998 through 09/30/1998.

When you run a billing or analysis program using a rate form that includes a season reference, you must specify which of the Season schedules in the database you wish to apply. There are two ways to do this:

• You can specify the Season Schedule by selecting **Tools-›Options-› Rate Analysis**, then making your selection under **Default Season Schedule**.

• You can specify it in the rate form by assigning the Season Schedule name to the special identifier SEASON_SCHEDULE_NAME. For example, if the database contained a Season Schedule called SEASON1, you could include the following Assignment Statement near the beginning of your rate form:

```
SEASON_SCHEDULE_NAME = SEASON1
```

The Season Schedule specified via Options is the default. A Season Schedule specified in a rate form overrides the Options value

How the programs determine which Season applies to the bill period: The SELECT BILL_PERIOD Statement always applies to the *current bill period*. Oracle Utilities Billing Component and Oracle Utilities Rate Management determine which season the bill period belongs in by comparing one date in the current bill period to the seasons. You define *which date* is to be compared by assigning a value to the BILL_PERIOD_SELECT identifier: 0 for the bill stop date, 1 for the bill start date, 2 for the bill month date, 3 for the Schedule Read Date, and 4 for the Governing Date. For example, if you wanted to use the first day of the bill period to determine which season it belongs in, you would include the following Assignment Statement near the beginning of your rate form: BILL_PERIOD_SELECT = 1.

To Create

1. Select **Statements-›Select-›Billing Period.**

   The SELECT BILL_PERIOD Statement template appears.

2. The WHEN clause sets the condition under which a set of nested statements should be applied; in this case, when the bill period falls within the specified season period(s). (You'll supply the nested statements later.) To create the WHEN clause, highlight one or more season periods in the lower box and click **Insert**.

   Note: Although the list box displays all season periods defined for all season schedules in the database, you can select only the season periods that belong to the current season schedule. See **Notes** on page 3-29 for more information.)

3. Repeat for each additional WHEN clause. You can have any number of WHEN clauses, but you must have at least one.

4. If your statement includes an OTHERWISE clause, check the **OTHERWISE** box. The OTHERWISE clause directs the program to apply the set of nested statements when the current bill period does not match any of the season periods in the preceding WHEN clauses.

5. Click **OK**. The WHEN and OTHERWISE clauses appear in the rate form.

6. Immediately following each WHEN and (if supplied) OTHERWISE clause, you must define the charge formulas you wish to apply during the season period. You can use one or more of the rate definition statements described in this chapter (BLOCK, ASSIGNMENT, and so on). To do this, highlight the clause and add the desired statement according to the instructions for that statement type.

Select Expression Statement

Purpose

The SELECT EXPRESSION Statement makes it possible to specify different actions depending on the value of an identifier or expression.

Format

The format for SELECT EXPRESSION statements is:

```
SELECT <expression>
   WHEN <expression, expression, ....>
   <nested statements>
   WHEN <expression, expression,...>
   <nested statements>
   ....
   OTHERWISE
   <nested statements>
END SELECT;
```

Example

Apply tariffs based on the JURISCODE (Jurisdiction Code) associated with the account being processed.

```
SELECT ACCOUNT.JURISCODE
   WHEN "MA", "RI"
   /* Apply special MA and RI tariff */
   WHEN "CA", "OR"
   /* Apply special CA and OR tariff */
END SELECT;
```

Notes

Any identifier or expression **except** BILL_PERIOD and RATE_CODE can be used in a SELECT EXPRESSION Statement.

To Create

1. Select **Statements-›Select-›Expression.**

    The SELECT EXPRESSION Statement template appears.

2. The SELECT clause sets the identifier or expression for the statement. To enter a previously defined identifier, position the cursor in the SELECT field and click the *right* mouse button to open the Rules Language Elements Editor.

3. The WHEN clause sets the condition under which a set of nested statements should be applied; in the preceding example, when an account's Jurisdiction code matches one of the supplied codes. (You'll supply the nested statements later.) To create the WHEN clause, highlight each applicable rate code in the lower box and click **Insert**. The WHEN clause appears in the upper list box; for example, WHEN "MA" or "RI".

4. Repeat for each additional WHEN clause. You can have any number of WHEN clauses. Each WHEN clause can have any number of values; however, a value cannot be in more than one WHEN clause.

5. If your statement includes an OTHERWISE clause, check the **OTHERWISE** box. The OTHERWISE clause directs the program to apply the set of nested statements when the expression does not match any of the values in the preceding WHEN clauses.

6. Click **OK**. The SELECT, WHEN and OTHERWISE clauses appear in the rate form.

7. Immediately following each WHEN and (if supplied) OTHERWISE clause, you must define the charge formulas you wish to apply to accounts subject to the expression. You can use one or more of the rate definition statements described in this chapter (BLOCK, ASSIGNMENT, and so on). To do this, highlight the clause and add the desired statement according to the instructions for that statement type.

Select Rate_Code Statement

Purpose

The SELECT RATE_CODE Statement allows you to specify different pricing options for a number of rate codes within a single rate form. The pricing options are defined by other statement types, such as Block statements or All statements, nested within the Select Statement (see the *Data Manager User's Guide* for details).

Format

The format for SELECT RATE_CODE statements is:

```
SELECT RATE_CODE
    WHEN <rate_code>[, <rate_code>...]
    <nested_statements>
    WHEN <rate_code>[, <rate_code>...]
    <nested_statements>
    ...
    OTHERWISE
    <nested_statements>
END SELECT;
```

Example

Here is an example of a Select Rate Code Statement used to define charges for five rates codes: 222, 223, 226, 221, and 227.

```
SELECT RATE_CODE
    WHEN "222", "223"
        /* Space heating */
        BLOCK KWH
        FROM   0 TO 300 CHARGE $0.09646
        FROM 300         CHARGE $0.05039
        TOTAL $ENERGY_CHARGE_223;
    WHEN "226"
        /* Separately metered space heating */
        $MIN_CHARGE = $4.75;
        ALL KWH CHARGE $0.05347 INTO $ENERGY_CHARGE_226;
    OTHERWISE
        /* Non heating - 221 OR 227 */
        BLOCK KWH
        FROM     0 TO  300 CHARGE $0.09646
        FROM   300 TO 1200 CHARGE $0.07920
        FROM  1200         CHARGE $0.06761
        TOTAL $ENERGY_CHARGE_OTH;
END SELECT;
```

Notes

Before you can specify a rate code in a SELECT RATE_CODE Statement, you must have already entered the rate code in the Oracle Utilities Data Repository.

To Create

1. Select **Statements-›Select-›Rate Code.**

   The SELECT RATE CODE Statement template appears.

2. The WHEN clause sets the condition under which a set of nested statements should be applied; in this case, when an account's rate code matches one of the supplied rate codes. (You'll supply the nested statements later.) To create the WHEN clause, highlight each applicable rate code in the lower box and click **Insert**. The WHEN clause appears in the upper list box; for example, WHEN "010" or "020".

3. Repeat for each additional WHEN clause. You can have any number of WHEN clauses. Each WHEN clause can have any number of rate codes; however, a rate code cannot be in more than one WHEN clause.

4. If your statement includes an OTHERWISE clause, check the **OTHERWISE** box. The OTHERWISE clause directs the program to apply the set of nested statements when an account's rate code does not match any of the rate codes in the preceding WHEN clauses.

5. Click **OK**. The WHEN and OTHERWISE clauses appear in the rate form.

6. Immediately following each WHEN and (if supplied) OTHERWISE clause, you must define the charge formulas you wish to apply to accounts subject to the rate code(s). You can use one or more of the rate definition statements described in this chapter (BLOCK, ASSIGNMENT, and so on). To do this, highlight the clause and add the desired statement according to the instructions for that statement type.

Warn Statement

Purpose

The WARN Statement is used with IF-THEN-ELSE to issue a warning message in the bill report when a condition you specify is met (or not met). The message you supply in the WARN Statement appears on page 1 of the bill report. The WARN Statement is typically used to validate data.

Note: In the Approval Required mode, the bill is still processed. The warning simply alerts the Billing Analyst to a condition that should be investigated before the bill is approved. In the Automatic mode, the bill for the account that triggered the warning is **not** processed. In that case, the bill can only be issued using the Current/Final Bill module.

Format

WARN statements have this format:

```
WARN <'character_string'>;
```

Example

The following example uses the WARN Statement for data validation.

```
/* Do + or - 25% validation, comparing current KWH to past 11 months */
MAX_KWH = MAXRANGE(KWH, 1, 11)
IF KWH > (MAX_KWH * 1.25)
    THEN
        WARN "KWH exceeds 125% of the maximum reading in the past 11
months.";
END IF;

MIN_KWH = MINRANGE(KWH, 1, 11);
IF KWH < (MIN_KWH * .75)
    THEN
        WARN "KWH is less than 75% of the minimum reading in the past 11
months.";
END IF;
```

Notes

The ABORT and WARN statements are similar. Like ABORT (see **Abort Statement** on page 3-2), the WARN Statement is used with IF-THEN-ELSE, so that it is triggered by a user-defined condition. However, unlike ABORT, WARN stops processing only in the Automatic billing mode; in the Approval Required mode, the bill is still computed, but a warning message is displayed on page 1 of the bill report. You could use the WARN and ABORT statements together for a two-step validation; that is, if condition x is met, calculate the bill and issue a WARNING message for the billing analyst; if condition y is met, stop processing the bill and issue the ABORT message.

The billing and analysis programs can display up to 50 messages in one report.

To Create

1. Select **Statements-›Warn**.

   The WARN Statement template appears.

2. Type the message (up to 256 characters) you wish to appear on reports. The message must be a string, so must be enclosed within double-quotes (" ").

3. Click **OK**. The statement appears in the rate form.

Chapter 4

Revenue Computation Statements

This chapter describes the revenue computation statements available in the Oracle Utilities Rules Language. Revenue computation statements are used to compute revenue based on bill determinants, unit charges, and other factors.

Revenue Computation Statements

- **All Statement**
- **Block Statements**
- **Unbilled and Ignore Statements**

All Statement

Purpose

ALL statements are used to calculate charges. ALL statements assign a unit price to a billing determinant, and also trace the number of units that the account consumed during the bill period, and the total charge for that determinant. If the Oracle Utilities Billing Component Print Detail Option in effect for the account is "Normal" or "All", the information is automatically printed in reports (see **Chapter 3**: **Working with Reports** in the *Oracle Utilities Billing Component User's Guide*).

Format

ALL statements use the following format:

```
ALL <determinant> CHARGE <price> INTO <$revenue_identifier>;
```

Example

Bill all KWH used during the billing period at a rate of $0.05094/KWH, assign the results to the revenue identifier $ENERGY_CHARGE, and report the usage and resulting revenue on the line labelled "$ENERGY_CHARGE".

```
ALL KWH CHARGE $0.05094 INTO $ENERGY_CHARGE;
```

For example, if the customer used 120 KWH during the bill period, the line shown below would automatically appear on the report under "Bill Calculation Results."

	Billing Units	Distribution	Charge Rate	Revenue
ENERGY_CHARGE	120.0		0.05094	$6.11

Notes

About the Difference Between All Statements and Assignment Statements: An ALL Statement is equivalent to a simple Assignment Statement as far as calculating revenue, but it also gathers information for reports (specifically, number of billing units and billing charge rate). Compare the following Assignment Statement with the preceding All Statement example:

```
$ENERGY_CHARGE = KWH * 0.05094;
```

Both statements calculate the customer's energy charge based on KWH consumption. Both label and report the resulting revenue as "$ENERGY_CHARGE" in reports. However, the All Statement also reports the number of billing units (KWH) consumed and the billing charge rate ($0.05094), as illustrated above.

To Create

1. Select **Statements-›All** from the Rules Language Editor menu bar.

   The ALL Statement template appears.

2. Complete the template:

   Determinant: Supply a usage variable that the charge will be based on; for example, KW or KWH. You can use either a billing determinant stored in the database, or a derived determinant calculated elsewhere in the rate form.

   - To use a billing determinant from the database, click the down arrow to the right of the field. A list of identifiers currently defined in the Bill Determinants table appears. Highlight your selection.

   - To use a variable calculated elsewhere in the rate form, type its identifier.

   Price per determinant unit: Unit price; may be a constant, a factor, or an identifier that points to the results of a function or expression elsewhere in the schedule.

   - To supply a constant, type the value in the field. The dollar sign is optional. It will not affect calculations, and may make your rate form easier to read.

   - To use an identifier defined elsewhere, you can type it or use the advanced features of the editor. To use the editor, position the cursor in the field, click the *right* mouse button, and select from the options that appear.

   Revenue identifier: Enter the name you wish to assign to the results of the charge calculation. The results (number of billing units, charge rate, and revenue) will appear in a row labelled with your input here (see example on previous page).

   You can type a revenue identifier or select from a set of previously defined identifiers:

   - If you type a revenue identifier of your own choosing, it must begin with a dollar sign. You can use any combination of letters, digits, and the underscore character (_), but you cannot use spaces. $ENERGY_CHARGE is acceptable; $ENERGY CHARGE is not. Also, **do not use any of the reserved words listed in Appendix A.** The identifier can be up to 32 characters.

   - To select from revenue identifiers that have been previously defined in other rate forms, position the cursor in the "Revenue Identifier" field and click the *right* mouse button. Select from the options that appear.

3. Click **OK**. Your new statement appears in the rate form.

Block Statements

Purpose

BLOCK statements define the block limits and unit charges for any type of block rate, including declining, increasing, and hours of use. The optional INTO clause tells the program to keep track of the account's usage, price, distribution, and revenue for each block for the bill period. If the report option in effect for the account is "Normal" or "All", that information is printed in reports.

The Rules Language provides two formats for blocked rates: "From/To" and "First/Next/Additional". They produce the same result, but the "First/Next/Additional" format is easier to input.

Formats

The Rules Language provides two formats for Blocked Rates:

First/Next/Additional Format

```
BLOCK <determinant>
    FIRST <constant|identifier> CHARGE <price> [INTO <$revenue_id>]
    NEXT <constant|identifier> CHARGE <price> [INTO <$revenue_id>]
    ADDITIONAL CHARGE <price> [INTO <$revenue_id>]
    TOTAL <$revenue_id>;
```

A "First/Next/Additional" BLOCK Statement can have any number of NEXT clauses.

From/To Format

```
BLOCK <determinant>
    FROM <constant|identifier> TO <constant|identifier> CHARGE <price>
[INTO <$revenue_id>]
    FROM <constant|identifier> CHARGE <price> [INTO <$revenue_id>]
    TOTAL <$revenue_id>;
```

A "From/To" BLOCK Statement can have any number of FROM/TO clauses.

Examples

In both of the following examples, if a customer used 500 kWh during the billing period, the first 150 kWh would be billed at $0.06, the second 150 at $0.05, and the remaining 200 at $0.04. The total energy charge for the customer would be $24.50.

In the first pair of examples (1a and 1b), the limits of the block are defined by constants; in the second pair (2a and 2b), by identifiers.

Example 1a: Declining Block (First/Next/Additional Format)

```
BLOCK KWH
    FIRST  150 CHARGE $0.06 INTO $KWH_0_150
    NEXT   150 CHARGE $0.05 INTO $NEXT_150
    ADDITIONAL CHARGE $0.04 INTO $KWH_ADDITIONAL
    TOTAL $ENERGY_CHARGE;
```

Example 1b: Declining Block (From/To Format)

```
BLOCK KWH
   FROM   0 TO 150 CHARGE $0.06 INTO $KWH_0_150
   FROM 150 TO 300 CHARGE $0.05 INTO $NEXT_150
   FROM 300 CHARGE $0.04 INTO $KWH_ADDITIONAL
   TOTAL $ENERGY_CHARGE;
```

Because the optional INTO clauses were used, the following lines would appear in the bill report under "Bill Calculation Results":

KWH	Billing Units	Distribution	Charge Rate	Revenue
KWH_0_150	150.0	30%	0.06	$9.00
NEXT_150	150.0	30%	0.05	$7.50
KWH_ADDITIONAL	200.0	40%	0.04	$8.00
ENERGY_CHARGE	500.0	100%		$24.50

Example 2a: Hours Use (First/Next/Additional Format)

```
HIGH1 = 20;/* First 20 hours use */
HIGH2 = 40;/* Next 40 hours use */

BLOCK KWH
   FIRST HIGH1 CHARGE $0.13037 INTO $BLK_1
   NEXT  HIGH2 CHARGE $0.09025 INTO $BLK_2
   ADDITIONAL  CHARGE $0.04764 INTO $BLK_3
   TOTAL $ENERGY_CHARGE;
```

Example 2b: Hours Use (From/To Format)

```
HIGH1 = 20;/* First 20 hours use */
HIGH2 = HIGH1 + (40);/* Next 40 hours use */

BLOCK KWH
   FROM    0 TO HIGH1 CHARGE $0.13037 INTO $BLK_1
   FROM HIGH1 TO HIGH2 CHARGE $0.09025 INTO $BLK_2
   FROM HIGH2 CHARGE $0.04764 INTO $BLK_3
   TOTAL $ENERGY_CHARGE;
```

To Create — Block, First/Next/Additional

1. Select **Statements->Block - First/Next/Addtl**.

   The Block (First/Next/Additional) Statement template appears.

   To complete the statement, specify a BLOCK clause, a FIRST clause, any number of NEXT clauses (including none), an ADDITIONAL clause, and a TOTAL clause. Each is explained below. All are required except NEXT.

2. In the **BLOCK** field, supply the usage variable the block rate will be based on; for example, KW or KWH. You can use either a billing determinant stored in the database, or a derived determinant calculated elsewhere in the rate form:

   - To use a billing determinant from the database, click the down arrow to the right of the BLOCK field. A list of identifiers currently defined in the Bill Determinants Lookup Code Table appears. Highlight your selection.

   - To use a variable calculated elsewhere in the rate form, type its identifier.

3. The **FIRST** subclause defines the lowest block and its unit price. (You can think of the FIRST, NEXT, and ADDITIONAL subclauses as directing the program, "Bill the portion of the usage that falls within this block as the price specified here.") To specify the FIRST clause, you'll enter its parameters in the 1st/Next line in the middle of the template, which is pointed out in the illustration above. This line consists of the following three text fields:

   1st/Next. Specify the upper limit of the block (the lower limit of the first block is automatically zero). You have two options.

   - If you want to use a constant, such as 150, simply type it. Use the Tab key to advance to the next field.

   - You can specify an identifier defined earlier in the rate form. (This approach enables you to apply block expanders.) To do this, either type the identifier or position the cursor in the field and click the *right* mouse button. A list of identifiers appears. For instance, if you were creating Example 2a shown earlier, the identifiers HIGH1 and HIGH2 would be available for selection.

   CHARGE. Unit price for the first block; may be a constant, a factor, or an identifier that points to the results of a function or expression elsewhere in the schedule:

   - To supply a constant, type the value in the field. The dollar sign is optional. It will not affect calculations, and it may make your rate form easier to read.

   - To use an identifier defined elsewhere, type it or use the advanced features of the editor. To use the editor, position the cursor in the field and click the *right* mouse button. Select from the options that appear.

   INTO. *Optional.* Enter a revenue identifier that you wish to assign to the results of the intermediate block calculation. If you supply a revenue identifier, the number of billing units, the charge rate, distribution, and revenue for the block will appear in a row on the bill report, labelled with the identifier. If you do not supply an INTO clause, only the total revenue from all blocks is reported.

   If one clause in your block rate includes an INTO component, all others must as well. Also, within a rate form, the revenue identifier for each block must be unique, and you cannot use the same revenue identifier in another block or ALL Statement in the rate form.

   You can use the default ($BLOCK1), type your own revenue identifier, or select from a set of previously defined identifiers:

   - If you type a revenue identifier of your own choosing, it **must** begin with a dollar sign. You can use any combination of letters, digits, and the underscore character (_), but you cannot use spaces. $KWH_0_150 is acceptable; _$KWH 0 TO 150 is not. **Do not use any of the reserved words listed in Appendix A.** The identifier can be up to 32 characters.

- To select from revenue identifiers that have been previously defined, position the cursor in the "Revenue Identifier" field and click the *right* mouse button. Select from the options that appear.

4. When you have completed the three text fields that make up the FIRST clause, click **Insert**. Your input appears in the upper list box.

   If you change your mind about a portion of the clause, highlight it in the list box; change the components in the same text fields in which you originally entered them, and click **Update**. To delete a clause, highlight it in the list box and click **Delete**.

5. If desired, define intermediate blocks using the **NEXT** clause. You can have any number of NEXT clauses, including none. Use the same technique as described in steps 2 and 3.

6. You must supply an **ADDITIONAL** clause that defines the highest block in the rate. Its upper limit is automatically infinity. Specify the desired values in the CHARGE and INTO fields.

7. Finally, specify a revenue identifier for the **TOTAL** clause (required). You can type the identifier, or right-click to open the **Rules Language Elements Editor** and select from the list that appears. Your input here identifies the total revenue for the sum of the blocks; it will appear on the bill reports, and you can use it as a variable in other statements.

   Note: If you type the revenue identifier, it must begin with a dollar sign ($) so the programs recognize it as a revenue identifier.

8. When you have completed the template, click **OK** to insert the BLOCK Statement into the rate form.

To Create — Block, From/To

1. Select **Statements-›Block - From/To**.

   The BLOCK (From/To) Statement template appears.

   To complete the statement, you'll specify a BLOCK clause, one or more FROM clauses, and a TOTAL clause. Each is explained below. All are required.

2. In the **BLOCK** field, supply the usage variable the block rate is to be based on; for example, KW or KWH. You can use either a billing determinant stored in the database or a derived determinant calculated elsewhere in the rate form:

   - To use a billing determinant from the database, click the down arrow to the right of the BLOCK field. A list of identifiers currently defined in the Bill Determinants Lookup Code Table appears. Highlight your selection.

   - To use a variable calculated elsewhere in the rate form, type its identifier.

3. The **FROM** clauses define the blocks and their unit prices. You can think of each FROM clause as directing the program, "Bill the portion of the usage that falls within this block at the price specified here." You can have any number of FROM clauses in a BLOCK Statement.

   To specify a FROM clause, enter its parameters in the box in the middle of the template, shown in the preceding illustration. This box consists of the following four fields:

   FROM and **TO**. Specify the lower limit and upper limit of the block. The lower limit of the first block must be zero. The lower limit of successive blocks must be the upper limit of the previous block. The upper limit of the last block must be blank (meaning infinity.) Your options for specifying inputs here are:

   - If you want to use a constant, such as 150, type it. Use the Tab key to advance to the next field.

   - Specify an identifier defined earlier in the rate form. (This approach enables you to apply block expanders.) To do this, type the identifier, or position the cursor in the field and

click the *right* mouse button. A list of identifiers appears. For instance, if you were creating Example 2b shown earlier, the identifiers HIGH1 and HIGH2 would be available for selection.

CHARGE. Unit price for the block; may be a constant, or an identifier that points to the results of a function or expression elsewhere in the schedule:

- To supply a constant, type the value in the field. The dollar sign is optional. It will not affect calculations, but it may make your rate form easier to read.

- To use an identifier defined elsewhere, you can type it or use the advanced features of the editor. To use the editor, position the cursor in the field and click the *right* mouse button. Select from the options that appear.

INTO. *Optional.* Enter a revenue identifier to assign to the results of the intermediate block calculations. If you use the INTO sub-clause, the number of billing units, the charge rate, distribution, and revenue for the block will appear in a row on the bill report, labeled with the identifier. If you do not supply an INTO clause, only the total revenue from all blocks is reported.

Note: If one clause in your block rate includes an INTO component, all others must as well. Also, within a rate form, the revenue identifier for each block must be unique, and you cannot use the same revenue identifier in another BLOCK or ALL Statement.

You can use the default ($BLOCK1), type your own revenue identifier, or select from a set of previously defined identifiers:

- If you type a revenue identifier of your own choosing, it must begin with a dollar sign. You can use any combination of letters, digits, and the underscore character (_), but you cannot use spaces. $KWH_0_150 is acceptable; _$KWH 0 TO 150 is not. Also, **do not use any of the reserved words listed in Appendix A.** The identifier can be up to 32 characters.

- To select from revenue identifiers that were previously defined, position the cursor in the **Revenue Identifier** field and click the *right* mouse button. Select from the options that appear.

4. When you have completed the fields that make up a FROM clause, click **Insert**. Your input appears in the upper list box.

    If you change your mind about a portion of the clause, highlight it in the list box; change the components in the same text fields in which you originally entered them, and click **Update**. To delete a clause, highlight it in the list box and click **Delete**.

5. If desired, define additional blocks using the FROM clause, as described in steps 3 and 4. You can have any number of FROM clauses, but the upper limit of the last block must be blank (meaning infinity).

6. Specify a revenue identifier for the TOTAL clause (required). You can type the identifier, or *right*-click to open the **Rules Language Elements Editor** and select from the list that appears. Your input here identifies the total revenue for the sum of the blocks; it will appear on the bill reports, and you can use it as a variable in other statements.

    Note: If you type the revenue identifier, it must begin with a dollar sign ($) so the programs recognize it as a revenue identifier.

7. When you have completed the template, click **OK** to insert the BLOCK Statement into the rate form.

Unbilled and Ignore Statements

Purpose

UNBILLED and IGNORE are special statements used in rate schedules that include a minimum bill provision. They are typically nested in IF-THEN-ELSE statements, to exclude the usage-based determinants and revenue associated with customers who are billed the minimum amount from summary calculations.

When a minimum charge is specified in a rate schedule, it typically means that a customer is billed the greater of two amounts; either a usage-based charge or a minimum charge. (The minimum charge may be a flat fee or a usage charge based on a second determinant, such as KW instead of KWH.) If a customer is billed the minimum charge, the customer's value for the first determinant is not used to calculate the bill (because it would yield a smaller charge than the minimum) and, therefore, neither that determinant value nor the charge based on that determinant should be included in the customer's bill or the summaries for the group or class. The UNBILLED and IGNORE statements are used in these cases.

The **IGNORE Statement** directs the program to exclude a customer's determinant-based revenue from the bill (typically in favor of the minimum charge) or vice versa. For example, `IGNORE $ENERGY_CHARGE;`

The **UNBILLED Statement** directs the program to accumulate and report the unbilled usage as a separate total. Although the determinant is not used to calculate the bill, it is still useful to report. For example, `UNBILLED KWH;`

Format

IGNORE statements have this format:

`IGNORE <$revenue_identifier>;`

UNBILLED statements have this format:

`UNBILLED <determinant_identifier>;`

Examples

Ignore and Unbilled statements are typically nested in If-Then-Else statements. In the example, look at the If-Then-Else Statement near the bottom of the schedule. If the minimum charge for a customer exceeds that based on KWH, then the program will exclude the KWH-based revenue from the revenue accumulation and will report KWH usage as a separate item.

```
$MIN_CHARGE = $2.50;

BLOCK KWH
    FROM 0 TO  50 CHARGE $0.10 INTO $KWH_0_50
    FROM 50 TO 200 CHARGE $0.05 INTO $KWH_50_200
    FROM 200        CHARGE $0.02 INTO $KWH_200_OR_MORE
    TOTAL $ENERGY_CHARGE;

$EFFECTIVE_REVENUE = MAX($MIN_CHARGE, $ENERGY_CHARGE);
```

```
IF ($MIN_CHARGE > $ENERGY_CHARGE) THEN
    IGNORE $ENERGY_CHARGE;
    UNBILLED KWH;
ELSE
    IGNORE $MIN_CHARGE;
END IF;
```

Here is another example to illustrate the use of the UNBILLED and IGNORE statements. In this case, the minimum charge is computed from a second usage variable (determinant) rather than just a flat rate.

```
KVA = COMPKVA(RKVA, KW);  /* Function to compute kVA */
ALL KVA CHARGE $0.97 INTO $KVA_CHARGE;
MIN_KVA_CHARGE = $48.50;
$MIN_CHARGE = MAX(MIN_KVA_CHARGE, $KVA_CHARGE);

HIGH1 = 20 * KW;  /* First 20 hours use */

BLOCK KWH
    FROM     0 TO HIGH1 CHARGE $0.13037
    FROM HIGH1          CHARGE $0.04764
    TOTAL $ENERGY_CHARGE;

$EFFECTIVE_REVENUE = MAX($MIN_CHARGE, $ENERGY_CHARGE);

IF ($MIN_CHARGE > $ENERGY_CHARGE) THEN
    IGNORE $ENERGY_CHARGE;
    UNBILLED KWH;
ELSE
    IGNORE $KVA_CHARGE, $MIN_CHARGE;
END IF;
```

To Create — Ignore

1. Select **Statements-›Ignore**.

   The IGNORE Statement template appears.

2. In the **Revenue Identifier(s)** field, specify one or more revenue identifiers that will be excluded from the bill calculations. You can type the identifier, or use the **Rules Language Elements** selector to pick one. To use the **Rules Language Elements** feature, position the mouse pointer in the field and click the *right* mouse button.

3. Click **OK**. The statement appears in the rate form.

To Create — Unbilled

1. Select **Statements-›Unbilled**.

   The UNBILLED Statement template appears.

2. In the **Determinant Identifiers** list box, specify the determinant that you want to report, even though it will not be included in the revenue calculations. Click **Insert**. Repeat for each desired UNBILLED determinant.

3. Click **OK**. The statement appears in the rate form.

Chapter 5

Report Statements

This chapter describes the report statements available in the Oracle Utilities Rules Language. Report statements are used with the Print Detail options to identify the values that appear in reports, and how they are labeled.

Report Statements

- **Clear Statement**

- **Determinant Statement**

- **Label Statement**

- **Remove Statement**

- **Report Statement**

- **Revenue Statement**

Clear Statement

Purpose

The CLEAR Statement is used to reset the values of identifiers to null. You may need to apply the CLEAR Statement when your rate schedule has a FOR EACH Statement that involves stem.component identifiers (see **Record Identifiers (stem.component)** on page 4-14 in the *Oracle Utilities Rules Language User's Guide*), or when you have applied a REPORT Statement to a revenue identifier.

> **Note**: Oracle Utilities recommends using the CLEAR statement to clear stem identifiers after saving records using the SAVE TO TABLE statement.

Using CLEAR statements with FOR EACH Loops

By nesting a CLEAR Statement in a FOR EACH loop, you are assured that no "residual" values assigned to a stem.component in one loop of the FOR EACH Statement will appear in the next loop.

For example, suppose that your utility's CISREC format includes just three components: CISREC.BILLSTART, CISREC.BILLSTOP, and CISREC.KW. You've created a FOR EACH Statement that computes the account's values for these three components for each channel and saves them to the CISREC.TXT file (see example below).

You have an account with three channels, and Channel C is missing the last value, which is for KW:

Channel A - 1998/01/01 00:00:00, 1998/01/31 00:00:00 120
Channel B - 1998/01/02 00:00:00, 1998/02/01 00:00:00 110
Channel C - 1998/01/01 00:00:00, 1998/01/31 00:00:00

You create a CIS record for each channel using a FOR EACH loop, but you don't include a CLEAR CISREC Statement in the loop. There would be no problem between the loop for A and B, because B's values would simply overwrite A's. However, C is missing a value for KW. B's value for KW (110) would still be in memory, and would therefore be written to C's CIS record.

You can avoid this circumstance if you always nest a CLEAR Statement in a FOR EACH Statement when the FOR EACH Statement involves reporting or saving stem components.

About Revenue Identifiers and REPORT statements: If you apply a REPORT Statement to a revenue identifier, the bill report displays the identifier's value twice; once for the REPORT Statement that you supplied, and once in the revenue identifier section that's automatically a part of the default format for bill reports. However, if you apply the CLEAR Statement to the revenue identifier, the value appears only for the REPORT Statement. It's eliminated from the pre-formatted, default section for revenue identifiers.

Format

```
CLEAR <identifier1> [,identifier2...];
```

Example

Save the values associated with the stem identifier CISREC to CIS, then clear the values associated with the stem identifier CISREC.

```
FOR EACH CHNL IN LIST ("ACCT_CHAN")
   CISREC.BILLSTART = BILL_START;
   CISREC.BILLSTOP = BILL_STOP;
   CISREC.KW = KW;
   SAVE CISREC TO CIS;
   CLEAR CISREC;
END FOR;
```

To Create

1. Select **Statements-›Clear**.

   The CLEAR Statement template appears.

2. In the **Identifier(s)** field, enter one or more identifiers whose values you wish to clear. You can specify a revenue or determinant identifier, or any other Rules Language identifier. If you specify a stem variable, all components are cleared. You can specify any number of identifiers in one CLEAR Statement. If you supply multiple identifiers, be sure to separate them with commas.

   You can type the identifiers or use the **Rules Language Elements Editor** to select from categories of identifiers. To use the **Rules Language Elements Editor**, position the mouse pointer in the field and click the *right* mouse button. Make your selections as desired.

3. When you have entered all of the desired identifiers, click **Insert**. Your input appears in the upper list box.

   If you change your mind about your entry, highlight the line in the upper box; make the desired changes in the lower box, and click **Update**. To delete a clause, highlight it in the upper box and click **Delete**.

   If you are supplying multiple identifiers in the CLEAR Statement, you can alternatively specify the first identifier in the **Identifier(s)** field, click **Insert**, specify the second identifier in the field, click **Insert**, and so on. In that way, the template automatically inserts the comma between the identifiers for you.

4. Click **OK**. The statement appears in the rate form.

Determinant Statement

Purpose

The DETERMINANT Statement has three applications:

- Replaces a determinant identifier with an easier-to-read or more meaningful label of your choosing in reports.

- Assigns the status of "determinant identifier" to a locally-defined identifier. Then, the rules and features that apply to determinant identifiers apply to the new identifier. You can apply any of the statements or functions that work with bill determinants to it. In addition, it will be listed under "Bill Determinant Identifiers" in the **Rules Language Elements** dialog box.

- Loads historical values for determinants stored in the Bill History Table or the Bill History Values Table. (See Notes below.)

Format

Determinant statements have this format:

```
DETERMINANT <determinant_identifier> <number_of_bill_periods> <"character_string">;
```

Example

Load the account's demand values for last 24 bill periods*, and replace the determinant identifier "KW" with the word "Demand" in the bill reports. (* See Notes below.)

```
DETERMINANT KW 24 "Demand";
```

Notes

If you request a number of bill periods greater than 1, the DETERMINANT Statement causes the program to load historical determinant values for at least the number of bill periods requested (if available), and may cause it to load all historical determinant values stored for the account. (The number loaded is dictated by performance considerations, depending upon what is most efficient for the table in which the values are stored.) This is an important consideration when supplying a function or operation in the same rate form that requires a specific number of historical values. *If a function requires a specific number of bill periods to get the value you need, be sure to specify the number of bill periods in the function.* **Do not use the DETERMINANT Statement for that purpose.**

Consider the following two sets of statements:

Example 1 - would provide possibly unreliable results:

```
DETERMINANT KW 12 "Demand";
X = KW;
RATCH_KW = MAXRANGE(X);
```

Example 2 - would provide reliable results:

```
DETERMINANT KW "Demand";
RATCH_KW = MAXRANGE(KW, 0, 11);
```

In Example 1, the variable X may contain all historical values for the account, so the maximum found by the MAXRANGE function may have occurred further back than the intended 12 periods. In Example 2, the DETERMINANT Statement labels the identifier, and the MAXRANGE function applies only to the desired number of bill periods.

To Create

1. Select **Statements-›Determinant**.

   The DETERMINANT Statement template appears.

2. Complete the template:

   Determinant: Click the down arrow to the right of the field. A list of determinant identifiers currently defined in the Bill Determinants Lookup Code Table appears. Highlight your selection.

   Number of months of historical bill periods to load: Type a value from 1 to 36, indicating the number of bill periods to load. For example, if you enter 6, the program loads at least the current bill period and the previous 5 periods of data for the determinant specified. *Remember:* The program loads at least the number of bill periods that you specify and, depending upon performance considerations, may load all historical values for a particular table. See Notes, above.

   Report Label: Enter the name you wish to appear in reports in place of the determinant identifier.

3. Click **OK**. Your new statement appears in the rate form.

Label Statement

Purpose

LABEL statements are similar to Determinant and Revenue statements, except that they are used to put labels on non-revenue and non-determinant identifiers. If the Oracle Utilities Billing Component Print Detail Option in effect for the account is "All," the value assigned to a labeled identifier is displayed in the bill reports. This is useful for reporting the results of intermediate calculations, for example.

> **Note:** You can use the LABEL Statement with any identifier; however, it is an error if the identifier already has a label.

Format

Label statements have this format:

```
LABEL <identifier> <"character_string">;
```

Example

Label the value computed for "METER_PEAK" with the title "Peak of Total Meters" and display in the bill report.

```
/* Display the Meters' Peak */
ME_HNDL = INTDLOAD('1700,1')
METER_PEAK = INTDVALUE(ME_HNDL , "MAXIMUM");
LABEL METER_PEAK "Peak of Total Meters";
```

For example, if the result of the INTDVALUE function for an account is 6.749279 and the Oracle Utilities Billing Component Print Detail Option in effect for the account is set to ALL, the following will appear in the bill report for the account:

```
Peak of Total Meters: 6.749279
```

To Create

1. Select **Statements->Label**.

   The LABEL Statement template appears.

2. Complete the template:

   Identifier: The identifier whose value you want to include in the bill reports.

   Report Label: The title you wish to appear in reports in place of the identifier.

3. Click **OK**. The statement appears in the rate form.

Remove Statement

Purpose

The REMOVE Statement is used to remove identifiers from the Shared Symbol table (the table where identifier values are stored in memory) in order to free up memory.

Using CLEAR statements with FOR EACH Loops

By nesting a REMOVE Statement in a FOR EACH loop, you are assured that no "residual" values assigned to a stem.component in one loop of the FOR EACH Statement will appear in the next loop.

Format

```
REMOVE <identifier1> [,identifier2...];
```

Example

Load and aggregate interval data associated with a transformer, then remove the values associated with the handle ADD_HNDL.

```
FOR EACH CHNL IN LIST ("TRANS_CHAN")
    RECORDERID = CHNL.RECORDERID;
    CHANNELNUM = CHNL.CHANNELNUM;
    REC_CHAN = RECORDERID + "," + CHANNELNUM;
    THIS_CHNL_HNDL = INTDLOADDATES (REC_CHAN, BILL_START, BILL_STOP);
    AGG_HNDL = AGG_HNDL + THIS_CHNL_HNDL;
END FOR;
AGG_TOTAL = AGG_HNDL.TOTAL
REMOVE AGG_HNDL;
```

To Create

1. Select **Statements-›Remove**.

   The REMOVE Statement template appears.

2. In the **Identifier(s)** field, enter one or more identifiers whose values you wish to remove. You can specify a revenue or determinant identifier, or any other Rules Language identifier, including array identifiers. If you specify a stem variable, all components are removed. You can specify any number of identifiers in one REMOVE Statement. If you supply multiple identifiers, separate them with commas.

   You can type the identifiers or use the **Rules Language Elements Editor** to select from categories of identifiers. To use the **Rules Language Elements Editor**, position the mouse pointer in the field and click the *right* mouse button. Make your selections as desired.

3. When you have entered all of the desired identifiers, click **Insert**. Your input appears in the upper list box.

   If you change your mind about your entry, highlight the line in the upper box; make the desired changes in the lower box, and click **Update**. To delete a clause, highlight it in the upper box and click **Delete**.

   If you are supplying multiple identifiers in the REMOVE Statement, you can alternatively specify the first identifier in the **Identifier(s)** field, click **Insert**, specify the second identifier in the field, click **Insert**, and so on. In that way, the template automatically inserts the comma between the identifiers for you.

4. Click **OK**. The statement appears in the rate form.

Report Statement

Purpose

The REPORT Statement is used to label and report values for identifiers (including stem.components) that may be assigned different values during the rate form's execution by the billing or analysis program. While the LABEL, DETERMINANT, and REVENUE statements report whatever value is assigned to their identifier when the rate form is done executing, a REPORT Statement writes out a value each time it is executed.

For example, if you nest a REPORT Statement inside a FOR EACH Statement, the statement will report its values for every pass of the FOR EACH loop. If you nest a Label inside a FOR EACH loop, only the value for the last pass of the loop appears in the report.

> **Note:** This information appears in an account's bill report only if the "All" or "Normal" Print Detail Option is in effect for the account.

Format

REPORT statements have this format:

```
REPORT <identifier> LABEL <[PAGE]"character_string"|label_identifier>;
```

Example

This example illustrates using the REPORT Statement to display the value of stem.components (here, the stem is PK):

```
/* Report top five peaks */
LABEL PK.NM "Peak Number";
LABEL PK.MX "Value";
LABEL PK.MD " Date/Time";
FOR EACH I IN NUMBER 5
    MX = "MAX" + I;
    MD = "MAXDATE" + I;
    PK.NM = "Peak " + I;
    PK.MX = INTDVALUE(KWH_HNDL , MX);
    PK.MD = INTDVALUE(KWH_HNDL , MD);
    REPORT PK LABEL "Top Five Peaks";
    CLEAR PK;
END FOR;
```

Here is how the results would be formatted in the bill report:

Top Five Peaks

Peak Number	Value	Date/Time
Peak 1	4.158719	01/27/1998 19:00:00
Peak 2	4.055039	01/27/1998 08:00:00
Peak 3	3.997439	01/03/1998 09:00:00
Peak 4	3.951359	01/02/1998 09:00:00
Peak 5	3.000372	01/01/1998 09:00:00

Notes

About reporting Stem.Component values: If the identifier you use in a REPORT Statement is a "stem," the values for its "components" will appear in the corresponding section of the bill report (as long as Oracle Utilities Billing Component's "All" or "Normal" Print Detail Option is in effect for the account). The billing program uses the following rules to format this section of the report, based on the parameters you supply in the REPORT Statement:

1. The title of the section is the value to the right of the word LABEL in the REPORT Statement. If you supply "" (two double quotes), the report will use the stem identifier.

2. All stem identifiers reported with the same title (e.g., whatever you supply after the word LABEL) appear together in the same section. They are ordered when the REPORT Statement executes. If you have multiple REPORT statements with different values for <identifier>, the sections are ordered when the REPORT Statement for the first row in the section executes.

3. The columns in the section are ordered by the appearance of the component in the rate form. The first component to appear in the rate form is the leftmost column in the report. The column header string for a column is the label of the corresponding component. The width of the column is based on the width of this string, so it must be wide enough to fully display the column values. The LABEL statements in the previous example set up the column headers.

4. If a column value is unassigned, spaces will fill in the column value.

    For example, you can insert a single-line report statement into the Bill Calculation Result section of a report by including "Bill Calculation Result" as the title of the section (the value to the right of the word LABEL in the REPORT Statement).

To Create

1. Select **Statements-›Report**.

    The REPORT Statement template appears.

2. Complete the template:

    Identifier: Specify the identifier whose value you want to report. It can be a simple identifier or a compound identifier (e.g., a "stem.").

    Report Label: Enter the text for the report line label, or the table of values, if the identifier was a stem. Note that the report label must be inside quotations ("kWh").

    To insert a page break in the report, include the PAGE keyword inside the report label ("PAGE kWh").

3. Click **OK**. The statement appears in the rate form.

Revenue Statement

Purpose

REVENUE statements label revenue identifiers in reports—that is, replace the identifier in the rate form with a more meaningful or easier to read label in bill reports. You can also use a Revenue Statement to establish an identifier as a revenue identifier, even if it has no leading $. (Revenue identifiers are a special class of identifiers used for charges; their values are automatically printed at the end of bill reports when Oracle Utilities Billing Component's "All" or "Normal" Print Detail Option is in effect.)

About the TOTAL clause: You can use the optional TOTAL clause in a REVENUE Statement to designate an identifier of your own choosing to represent the bill total, or you can use the predefined identifier $EFFECTIVE_REVENUE. Regardless of which Print Detail Option you specify, the bill report always includes the value for $EFFECTIVE_REVENUE, or the user-specified identifier you substitute using the TOTAL clause.

Format

Revenue statements have this format:

```
REVENUE <$revenue_identifier> TOTAL <"character_string">;
```

Examples

Label the value for minimum charge, "Minimum charge."

```
REVENUE $MIN_CHARGE "Minimum charge";
```

Revenue statements are especially useful for labeling the results accumulated by Into clauses in Block statements. For example:

```
BLOCK KWH
    FROM   0 TO 300 CHARGE $0.09646 INTO $KWH1
    FROM 300         CHARGE $0.05039 INTO $KWH2
    TOTAL $ENERGY_CHARGE_223;
```

Without a Revenue Statement, the report rows for the two blocks would be labeled $KWH1 and $KWH2, respectively. You could assign more meaningful labels using the following Revenue statements:

```
REVENUE $KWH1 "0 to 300 kWh"";
REVENUE $KWH2 "300+ kWh";
```

To Create

1. Select **Statements-›Revenue**.

   The REVENUE Statement template appears.

2. Complete the template:

   Revenue Identifier: Specify the identifier. You can type it or use the Rules Language Elements feature to select it. To use the Rules Language Elements feature, position the mouse pointer in the field and click the *right* mouse button.

   TOTAL: *Optional.* To designate the identifier as representing the total bill, place a check in the TOTAL checkbox.

   Note: If you apply the TOTAL clause, you cannot use the special identifier $EFFECTIVE_REVENUE in the same rate schedule. You can have only one REVENUE Statement with a TOTAL clause in a rate schedule.

   Report Label: Enter the name you wish to appear in reports in place of the identifier that you specified in the first field.

3. Click **OK**. Your new statement appears in the rate form.

Chapter 6

Miscellaneous Statements

This chapter describes the miscellaneous statements available in the Oracle Utilities Rules Language.

Miscellaneous Statements

- **Delete Statement**
- **Save Statements**

Delete Statement

Purpose

The DELETE statement is used to remove a record from the database. A record can be deleted immediately when SAVE COMMIT is used, or it can be deleted when a bill is approved.

Format

```
DELETE <identifier> FROM TABLE <"table-name">;
```

Where:

- <identifier> is the stem identifier whose record you wish to delete. All values in the record key (i.e. all identity columns) must be specified as corresponding tail identifiers.

- <"table-name"> is the name of the table in the Oracle Utilities Data Repository. The name must begin and end with double quotation marks, or must be the value of a string identifier or expression.

Example

Delete a record from the Meter Read table.

```
METER_READ.METERID = "METER_01";
METER_READ.MANUFACTURER = "METERSRUS";
METER_READ.SERIALNO = "12345";
METER_READ.UNINUMBER = "54321";
METER_READ.METERREADTIME = "01/31/2007 00:00:00";
METER_READ.BILLDETERMCODE = "1";
METER_READ.READMONTH = "01/2007";
DELETE METER_READ FROM TABLE "METERREAD";
```

To Create

1. Select **Statements-›Delete-›From Table.**

   The DELETE Statement template appears.

2. Complete the template:

   DELETE: The stem identifier of the record you wish to delete.

   FROM TABLE: The name of the table the record is to be deleted from.

3. Click **OK**. The DELETE statement appears in the rate form.

Save Statements

Purpose

The SAVE statements enable you to save bill determinant values, records, or interval data cuts. If a determinant value is already stored in the database for the current bill month, it will be overwritten with the SAVE'd value. If you save a computed cut with the same cut series key (recorder-id,channel-number) as an existing cut, it will overwrite any data that already exists for the current bill month. If desired, you can use the Default Options feature to specify that any overwritten cuts be exported to an archive file before deletion from the database. To apply this option, go to the Data Manager desktop. Select **Tools-›Options-›Interval Data Source**. Check the box next to **Export Records Overwritten by SAVE to File**.

There are seven versions of the SAVE Statement:

SAVE AS: Saves a single value for a bill determinant identifier to the Bill History Table or the Bill History Value Table, or saves a computed interval data cut to the Interval Database; for example, SAVE AS BILL_KW; or SAVE AS '1701,1'.

SAVE TO TABLE: Saves a user-created data record (or an array of records) to a row(s) in a user-specified table in the Oracle Utilities Data Repository. This is useful for saving data to a specific table; for example, SAVE MV TO TABLE 'METERVALUE"; (where MV is a stem identifier).

The SAVE TO TABLE statement first tries to update the record; if that fails, it tries to add the record.

> **Note**: If the **SAVE_TO_TABLE_ADDFIRST=1** parameter is specified in the LODESTAR.CFG configuration file, the SAVE TO TABLE statement first tries to ADD the record, and if that fails, it tries to UPDATE the record. See **LODESTAR.CFG** on page 2-2 in the *Oracle Utilities Energy Information Platform Configuratin Guide* for more information.

If the record is always added or updated, use the SAVE_ADD TO TABLE or the SAVE_UPDATE TO TABLE versions of this statement. If you use SAVE_ADD and a record with the same key exists in the database, it is an error. If you use SAVE_UPDATE and a record with the same key does not exist in the database, it is an error. You can select the Save mode in the SAVE TO TABLE dialog box. If you use SAVE_UPDATE, the value of the stem must be "" or the original key of the record. If it is the original key of the record (set automatically in a FOR EACH IN LIST Statement), you can then update one or more of the identity columns.

> **Note**: Oracle Utilities recommends using the CLEAR statement to clear stem identifiers after saving records using the SAVE TO TABLE statement. This is especially important when using the SAVE TO TABLE statement within a FOR EACH statement to create a series of records in the same table.

To set a column value to NULL, save the column as "" (double quotes). For example:

```
X.KWH=""
SAVE X TO TABLE BILL HISTORY
```

would set the KWH column on the Bill History record to NULL.

> **Note:** The SAVE TO statement should not be used with the Bill History and Bill History Value Tables when running a rate schedule in the context of an account. Use the SAVE AS statement above to save records to these tables. This recommendation is intended to prevent errors when saving bill determinants during billing calculations.
>
> Using the SAVE TO TABLE statement with these tables can result in an error in the Rules Language editor. To disable this error message, include the DONOT_CHECK_SAVETOTABLE_BH_AND BHV" parameter in the LODESTAR.CFG file.

Note: If one of the columns in the saved record is XML, you must have an Automatic Save mode turned on. You cannot delay the save until the user approves the report.

Note: When saving large volumes of records, using Array identifiers will improve performance and speed up processing.

SAVE TO CHANNEL: Saves a computed interval data cut (or an array of cuts) to the Interval Database: for example, SAVE INTD_HNDL1 to CHANNEL '12345,1'.

Note: When saving large volumes of cuts, using Array identifiers will improve performance and speed up processing.

SAVE to CIS: Saves a user-created data record to the CIS transaction record. The components of the record must match the names of fields defined in your utility's CISFORMT.TXT file (typically in the CFG directory).

SAVE to XML: Saves user-created data in XML format. The data will be part of the report in a section titled SAVE TO XML

SAVE to STAGING: Saves an interval data handle to either the Interval Data Staging (LSINTDSTAGING) or Interval Data Reporting (LSRFINTDHEADER and LSRFINTDVALUES) tables.

SAVE COMMIT: Commits database changes due to SAVE TO TABLE, SAVE TO CHANNEL statements, interval data delete from the relational database, and the LISTUPDATE function. It also writes out all save CIS and XML records. It will then start a new database transaction. It applies only when the Save mode is 'Automatically save / approve each page if it is OK.'

SAVE ROLLBACK: Removes (rolls back) database changes due to SAVE TO TABLE, SAVE TO CHANNEL (interval data in the relational database) statements, interval data delete from the relational database, and the LISTUPDATE function. It also deletes saved CIS and XML records (that were not directly written). No other SAVEs are affected. It will then start a new database transaction. It applies only when the Save mode is 'Automatically save / approve each page if it is OK'.

Format

Its format is:

```
SAVE <identifier_save_spec>;
```

An <identifier_save_spec> is one of:

```
<identifier_defined> or
```

```
<identifier_scalar> AS <determinant_identifier> or
```

```
<identifier_interval> AS <'recorder,channel'> or
```

```
<identifier_stem|array_stem> TO TABLE <table_identifier|literal> or
```

```
<identifier_interval|array_stem> TO CHANNEL <'recorder,channel'> or
```

```
<identifier_interval|array_stem> TO CHANNEL <'identifier,rcdr'> or
```

```
<identifier_stem> TO CIS or
```

```
<identifier_stem> TO XML or
```

```
<identifier_interval> TO STAGING [<TABLE_NAME>]<'recorder,channel'> or
```

```
<array_stem> TO STAGING <TABLE_NAME> or
```

```
SAVE COMMIT or

SAVE ROLLBACK
```

where:

- **<identifier_stem>** is a stem identifier previously referenced in the rate form.

- **<array_stem>** is an array stem identifier previously referenced in the rate form. The array stem supplied should not include the index. See **Array Identifiers** on page 4-20 in the *Oracle Utilities Rules Language User's Guide* for more information about using array identifiers.

- **<identifier_interval>** is an interval data reference.

Examples

Example 1: Perform calculations on all intervals and then save the cut with a new key, leaving the original cut intact:

```
SIMPLE_CUT = 'RCDR1234,1';
SAVE SIMPLE_CUT AS 'RCDR1234,2';
```

Example 2: Save $EFFECTIVE_REVENUE as a determinant called CURRENT_CHARGES:

```
SAVE $EFFECTIVE_REVENUE AS CURRENT_CHARGES;
```

In this case, CURRENT_CHARGES must have been defined in the database as a billing determinant.

Example 3: Create and save bought and sold cuts:

```
 /* Load the generator cut */
INTD_GEN_CUT = 'RCDR1234,1';

/* Load the use cut */
INTD_USE_CUT = 'RCDR1234,2';

/* Compute the bought cut */
INTD_BOUGHT_CUT =+ INTD_GEN_CUT - INTD_USE_CUT;

/* Compute the sold cut */
INTD_SOLD_CUT =+ INTD_USE_CUT - INTD_GEN_CUT;

/* Save them */
SAVE INTD_BOUGHT_CUT AS 'RCDR1234,3';
SAVE INTD_SOLD_CUT AS 'RCDR1234,4';
```

Example 4: Create account IDs for customer "CUSTOMER_1".

```
FOR EACH Y IN NUMBER 25
   FOR EACH X IN NUMBER 100
      #ARR[X].ACCOUNTID = "ACCOUNT_" + Y + "_" + X;
      #ARR[X].STARTTIME = '01/01/2004';
      #ARR[X].CUSTOMERID = "CUSTOMER_1";
   END FOR;
   SAVE #ARR[] TO TABLE ACCOUNT;
END FOR;
```

Example 5: Save five years of interval data for recorder,channel "A20991,1" as monthly cuts.

```
#ARR[1] = INTDLOADDATES ('A20991,1', '01/01/1999 00:00:00', '01/31/
1999 23:59:59');
#ARR[2] = INTDLOADDATES ('A20991,1', '02/01/1999 00:00:00', '02/28/
1999 23:59:59');
#ARR[3] = INTDLOADDATES ('A20991,1', '03/01/1999 00:00:00', '03/31/
1999 23:59:59');
...
```

```
#ARR[60] = INTDLOADDATES ('A20991,1', '12/01/2004 00:00:00', '12/31/
2004 23:59:59');
```
SAVE #ARR[] TO CHANNEL "A20991_MONTHLY,1"

Example 6: Save interval data for recorder,channel "METER_AA,1" to numbered recorder IDs (1-100) with corresponding channel numbers (1-100) to the LSCHCLDB interval data table.

```
INTD_HNDL = INTDLOADDATES("METER_AA,1", START, STOP);
REC = "RECORDER";
FOR EACH X IN NUMBER 100
   #ARR[X] = INTD_HNDL;
   RST = INTDSETATTRIBUTE(#ARR[X] , "CHANNEL" , X);
   RST = INTDSETATTRIBUTE(#ARR[X] , "RECORDERID" , REC + X);
END FOR;
```
SAVE #ARR [ ] TO CHANNEL "RDB/LSCHLDB";

Notes

All versions of the SAVE statements (SAVE AS, SAVE TO TABLE, etc.) are ignored by Oracle Utilities Rate Management.

To Create - Save As

1. Select **Statements-›Save-›As**.

    The SAVE Statement template appears.

2. Complete the template:

    Identifier: Specify a bill determinant identifier, an interval data handle, or a cut series key (the cut series key uses the format 'recorder,channel'; e.g., '1234,1').

    Optional Save As Name: This is the new name for the determinant, record, or interval data cut.

3. Click **OK**. The statement appears in the rate form.

To Create — Save To Table

1. Select **Statements-›Save-›To Table**.

    The SAVE Statement template appears.

2. Complete the template:

    Identifier: The stem identifier or stem array identifier referenced earlier in the rate form.

    Add/Update: Select Add/Update, Add, or Update, as appropriate.

    Table Name: The name of the table you wish to save the record to.

3. Click **OK**. The statement appears in the rate form.

To Create - Save To Channel

1. Select **Statements-›Save-›To Channel**.

    The SAVE Statement template appears.

2. Complete the template:

    Identifier: Specify a bill determinant identifier, an interval data handle, a stem array identifier, or a cut series key (the cut series key uses the format 'recorder,channel'; e.g., '1234,1').

    Recorder,Channel: A cut key. If the key already exists in the database and it has interval data for the current period, the old data will be overwritten. If you use a new key, the new data is saved and the old data remains intact.

3. Click **OK**. The statement appears in the rate form.

To Create - Save To CIS

1. Select **Statements-›Save-›To CIS**.

   The SAVE Statement template appears.

2. Complete the template:

   Identifier: The stem identifier referenced earlier in the rate form.

   Optional Section Name: An optional SECTION name; for example, SAVE x TO CIS SECTION *name*, where x is a stem identifier and *name* is a section in the CISFORMT.TXT file. If you supply a name, it overrides the record type (10, 20, etc.) and formats the record based on the section with the same name. If there is a HEADER section, its fields are always put first. This feature is useful for processing cancelled bills (see **Cancel/Rebill Rider** in **Chapter 1**: **Introducing the Oracle Utilities Rules Language** of the *Oracle Utilities Rules Language User's Guide* for more information).

   Optional File Name: An optional path and file name. The file name may be an identifier or expression whose value is a string, or a literal string. The path and file name can be no longer than 65 characters. When approved or committed, the record is written to this file instead of the CISFILE set in the LODESTAR.CFG file. The default is the CISFILENAME configuration parameter value, or the user-entered file name.

   Note: Detail record types (31,41,51, and 61) can only be saved to the default CIS file, and **cannot** be saved to specific CIS files. See **Appendix A: Creating a CIS Transaction Record Output File** in the *Oracle Utilities Billing Component Installation and Configuration Guide, Volume 1* for more information.

3. Click **OK**. The statement appears in the rate form.

To Create - Save To XML

1. Select **Statements-›Save-›To XML**.

   The SAVE Statement template appears.

2. Complete the template:

   Identifier: The stem identifier referenced earlier in the rate form.

   Optional File Name: An optional file name. The file name may be an identifier or expression whose value is a string, or a literal string. When approved or committed, the record will be written to this file. The data will be part of the report in a section titled SAVE TO XML. If a File Name is not specified, the XML data must be created and used by a COM object; otherwise, it is ignored.

   If a file name is specified, the rate schedule that contains the SAVE TO XML statement must create a correctly formatted XML file. A correctly formatted XML file has one root element, with any number of sub-elements. This means that the first line to the file must be the begin tag, and the last line must be the end tag.

   Two special identifier names can be used to create these lines.

   ```
   SAVE BEGIN_ELEMENT TO XML FILE x;
   ```

   will write the opening XML tag, where the element name is the string value of the identifier BEGIN_ELEMENT.

   ```
   SAVE END_ELEMENT TO XML FILE x;
   ```

   will write the closing XML tag, where the element name is the string value of the identifier END_ELEMENT.

   Note that these statements always create opening and closing tags, so they can be used several times to create nested XML. However, a SAVE BEGIN_ELEMENT must be the first XML save to a file, and a SAVE END_ELEMENT must be the last XML save to a file. (This is first

and last period, not just first and last for the rate schedule.) It is possible to create the opening line in one rate schedule, and the closing line in another.

3. Click **OK**. The statement appears in the rate form.

To Create - Save To Staging

1. Select **Statements-›Save-›To Staging**.

    The SAVE Statement template appears.

2. Complete the template:

    Identifier: Specify an interval data handle, or a cut series key (the cut series key uses the format 'recorder,channel'; e.g., '1234,1').

    Recorder,Channel: A cut key preceded by an optional table name, in the following format:

    [QUAL/<alternate_qualifier>;][RDB/<alternate_table>;]<'rec,chan'>

    where:

    - <alternate_qualifier> is a string containing the name of a alternate database qualifier containing the interval data to be loaded.

        - The meta-data of the alternate qualifier **must** be the same as the original qualifier.

        - When using an alternate qualifier and processing in the context of an Account (such as when running billing via Oracle Utilities Billing Component), the account **must** be present in both the qualifiers.

    - <alternate_table> is a string containing the either LSINTDSTAGING or an equivalent (optional), or LSRFINTDHEADER. The table name is optional for saving data to the LSINTDSTAGING table, but is required when saving data to the LSRFINTDHEADER table (used when reporting on interval data).

    - <'rec,chan'> is a cut key. If the key already exists in the database and it has interval data for the current period, the old data will be overwritten. If you use a new key, the new data is saved and the old data remains intact. This is an optional parameter.

3. Click **OK**. The statement appears in the rate form.

To Create - Save Commit

1. Select **Statements-›Save-›COMMIT**.

2. The SAVE COMMIT statement appears in your rate form.

    Note: SAVE COMMIT does not work on the Bill History Table.

To Create - Save Rollback

1. Select **Statements-›Save-›ROLLBACK**.

2. The SAVE ROLLBACK statement appears in your rate form.

Chapter 7

Financial Management Statements

Financial Management statements are used to post transactions to the Oracle Utilities Receivables Component (a component of Oracle Utilities Billing Component). This chapter provides detailed explanations of the Financial Management statements available in the Oracle Utilities Rules Language, including:

- **Using the Financial Management Statements**

- **Deprecated Statements**

Using the Financial Management Statements

The Financial Management statements are used to post charges or credits to the Oracle Utilities Receivables Component's Financial Engine. Each statement takes a transaction identifier as a single argument. The transaction identifier is a stem that should contain several tail attributes, as described below. Attributes marked with an asterisk (*) are required.

Attribute	Description
ACCOUNTID	An account ID that identifies the account for posting or cancelling a transaction. If not provided, the rate schedule account context is used. It is an error if no account ID is provided and the rate schedule is not run within the context of an account.
UID	Unique ID of a posted transaction. Used with the CANCEL_TRAN statement.
TRANSACTIONID	A transaction ID for the transaction. If not provided, the default transaction ID for the transaction type is used.
REVENUEMONTH	The revenue month for the transaction. If not provided, the rate schedule bill month is used.
NOTE	A note for the transaction.
CANCELREVENUEMONTH	The revenue month for a cancelled transaction. Optional attribute used with the CANCEL_TRAN statement.
CANCELREASONCODE	The reason for cancelling a transaction. Optional attribute used with the CANCEL_TRAN statement.
CANCELNOTE	A note for a cancelled transaction. Optional attribute used with the CANCEL_TRAN statement.
CHARGEORCREDIT	Indicates whether transaction is a charge (CH) or a credit (CR). The default is charge ("CH") unless otherwise indicated.
DEFERBALANCE	Indicates whether transaction balance is deferred ("TRUE") or not ("FALSE"). The default is "FALSE" unless otherwise indicated
AMOUNT*	The amount of the transaction.
CURRENCY	The currency code for the currency associated with the account for the transaction. This is required if the LS Currency column is populated in the Account table.
BILLEDDATE*	The billed date for the charge transaction. This is only required for charge transactions.
DUEDATE*	The due date for the charge transaction. This is only required for charge transactions.
RECEIVABLETYPENAME	The receivable type name for transactions (required for all charge type transactions, except for POST BILL, if CHARGETYPEID is not provided).

Attribute	Description
CHARGETYPEID	The charge type identifier for transactions (required for all charge type transactions, except for POST BILL, if RECEIVABLETYPENAME is not provided).
OPCOCODE	The operating company code associated with the transaction.
JURISCODE	The jurisdiction code associated with the transaction.
STATEMENTDATE	This attribute is used only with the POST STATEMENT statement. The statement date associated with the transaction.
INVOICEID	The invoice ID associated with the transaction.
INVOICEDATE	The invoice date associated with the transaction.
BILLCYCLEDATE	The bill cycle date for the transaction. If not provided, the rate schedule read date is used. It is an error if no bill cycle date is provided and the rate schedule does not have an associated read date.
BILLSTARTTIME	This attribute is used only with the POST BILL statement. The bill start time for the transaction. If not provided, the transaction time of the previous BILL transaction with the same transaction ID is used.
BILLSTOPTIME	This attribute is used only with the POST BILL statement. The bill stop time for the transaction. If not provided, the transaction time is used.
SUSPENDAUTOPAYMENT	This attribute is used only with the POST BILL statement. Indicates that automatic payments for the bill transaction should be suspended.
SERVICEPLAN*	This attribute is required for the POST SERVICE CHARGE and POST BUDGET SERVICE CHARGE statements. The service plan attribute is a stem itself that requires both STARTDATE and SERVICETYPECODE attributes; optional attributes are ADDRESS1, ADDRESS2, ADDRESS3, CITY, COUNTY, STATE, ZIP (to identify the associated premise), and LDCACCOUNTNO.
BUDGETPLAN*	This attribute is used only with the POST BUDGET SERVICE CHARGE, POST BUDGET BILL CHARGE, and POST BUDGET BILL TRUEUP statements. The budget plan attribute is a stem itself that requires STARTDATE and BUDGETTYPECODE attributes; the SERVICEPLAN attribute (to identify any associated service plan) is optional.

Attribute	Description
TAXRATE	The tax rate associated with either a TAX transaction or one or more individual TAXEDTRANSACTIONS.
TAXEDTRANSACTION<ID>	One of the taxed transactions associated with a POST TAX statement. The taxed transaction attribute is a stem itself that may contain the following attributes: UIDTRANSACTION or TRANSACTIONNO (at least one of which is required), AMOUNT, TAXAMOUNT, TAXRATE, TAXEXEMPT ("TRUE" or "FALSE").
UIDINSTALLMENTPLAN	Unique ID of associated installment plan. Either this or INSTALLMENTPLANNO below is required for the POST INSTALLMENT statement.
INSTALLMENTPLANNO	The transaction number of deferred charge transaction associated with installment plan. Either this or UIDINSTALLMENTPLAN above is required for the POST INSTALLMENT statement.
UIDDEPOSIT	Unique ID of associated deposit. Either this or DEPOSITTIME below is required for the POST DEPOSIT INTEREST and POST DEPOSIT APPLICATION statements.
DEPOSITTIME	Time of associated deposit. Either this or UIDDEPOSIT above is required for the POST DEPOSIT INTEREST and POST DEPOSIT APPLICATION statements.
DEPINTRATE	Optional interest rate for deposit. This is used by the POST DEPOSIT statement.
APPLICATIONMETHOD	Indicates how to apply the transaction against outstanding charges or credits. Valid values are "DEFERRED", "IMMEDIATE", and "INVOICEID". Default is "DEFERRED" unless otherwise indicated.
DEFACCOUNTID	Required default account id used by the POST PAYMENT statement.
SOURCECODE	Required payment source code used by the POST PAYMENT statement.
PAYMENTID	Optional payment id used by the POST PAYMENT statement.
METHODCODE	Optional payment method code used by the POST PAYMENT statement.
INSTITUTION	Optional institution name from which payment is drawn; used by the POST PAYMENT statement.
ACCOUNTNO	Optional account number from which payment is drawn; used by the POST PAYMENT statement.

Attribute	Description
CHECKNO	Optional payment check number used by the POST PAYMENT statement.
RELATEDTRANSACTIONn	Optional related transaction(s) to which credits are applied when using the POST CHARGEORCREDIT statement. If multiple related transactions are specified, credits are applied in the order specified in the Rules Language. For example, RELATEDTRANSACTION1 first, RELATEDTRANSACTION2 second, etc.
MISC1	Optional user-defined miscellaneous attribute used by the POST PAYMENT statement.
MISC2	Optional user-defined miscellaneous attribute used by the POST PAYMENT statement.
MISC3	Optional user-defined miscellaneous attribute used by the POST PAYMENT statement.

Example:

```
SERV_PLAN.STARTDATE = "01/01/2000";
SERV_PLAN.SERVICETYPECODE = "ELECTRIC";

BUDGET_PLAN.STARTDATE = "01/01/2000";
BUDGET_PLAN.BUDGETTYPECODE = "SIMPLE";
BUDGET_PLAN.SERVICEPLAN = "SERV_PLAN";

SERV_CHG_1.TRANSACTIONID = "350";
SERV_CHG_1.AMOUNT = 59.95;
SERV_CHG_1.CURRENCY = "USD";
SERV_CHG_1.BILLEDDATE = "07/15/2000";
SERV_CHG_1.DUEDATE = "07/30/2000";
SERV_CHG_1.CHARGETYPEID = "ELECTRIC_USAGE_CHARGE";
SERV_CHG_1.SERVICEPLAN = "SERV_PLAN";
SERV_CHG_1.BUDGETPLAN = "BUDGET_PLAN";

POST CHARGEORCREDIT SERV_CHG_1;
```

Using User-Defined Attributes

If the Transaction Table in your database contains columns that are not included in the base schema (and therefore not among the data elements listed), you can post values to those columns by assigning values to corresponding STEM.COLUMN_NAME identifiers in the rate schedule. In this case, the column name specified in the rate schedule must be the exact name of the column in the database. For example, if your Transaction Table contains a column called ZONE, you could post data to that column by including the following line in your rate schedule:

`USAGE_SERV_CHG.ZONE = "ZONE_1"`

```
/* Post Service Charge Statement */
POST STATEMENT USAGE_SERV_CHG;
```

This would post the value assigned to the USAGE_SERV_CHG.ZONE identifier ("ZONE_1") to the ZONE column in the Transaction Table.

Post Charge Or Credit Statement

Purpose

The POST CHARGEORCREDIT Statement posts a charge or credit as a single transaction. The transaction may be either deferred or not deferred. An optional service plan or budget plan may be associated with the transaction. If a budget plan is provided, the plan's variance will be updated accordingly.

Format

POST CHARGEORCREDIT statements have this format:

```
POST CHARGEORCREDIT <stem_identifier>;
```

Where:

• <stem_identifier> is a stem that contains the appropriate attributes, as described under **Using the Financial Management Statements**.

To Create

1. Select **Statements-›Financials-›Post Charge or Credit** from the Rules Language Editor menu bar.

    The POST CHARGEORCREDIT Statement template appears.

2. Enter the appropriate stem identifier, or click the *right* mouse button and choose the stem identifier from the **Other Identifiers** list in the **Rules Language Elements Editor**.

3. Click **OK**. The statement appears in the rate form.

Example

Post a service charge transaction based on energy usage.

```
/* Set Service Charge Attributes */
USAGE_SERV_CHG.ACCOUNTID = ACCOUNT.ACCOUNTID;
USAGE_SERV_CHG.TRANSACTIONID = "310";
USAGE_SERV_CHG.REVENUEMONTH = BILLMONTH;
USAGE_SERV_CHG.NOTE = "Electric Energy Charge - Energy Service
Provider";

USAGE_SERV_CHG.AMOUNT = $ENERGY_CHARGE;
USAGE_SERV_CHG.CURRENCY = "USD";
USAGE_SERV_CHG.BILLEDDATE = "07/15/2000";
USAGE_SERV_CHG.DUEDATE = "08/15/2000";

USAGE_SERV_CHG.RECEIVABLETYPENAME = "ESCO ELECTRIC";
USAGE_SERV_CHG.CHARGETYPEID = "ESCO ENERGY";
USAGE_SERV_CHG.OPCOCODE = OPCOCODE;
USAGE_SERV_CHG.JURISCODE = JURISCODE;
USAGE_SERV_CHG.BILLCYCLEDATE = BILLCYCLEDATE.READDATE;

/* Post Usage Service Charge */
POST CHARGEORCREDIT USAGE_SERV_CHG;
```

Notes

In the above example, several of the USAGE_SERV_CHG attributes are 'hard-coded' into the rate schedule. In actual practice, this data would probably come directly from records in the Oracle Utilities Data Repository, or through list queries. Also, the above example includes values for all the optional USAGE_SERV_CHG attributes. These are included to illustrate how those attributes might be supplied in a rate schedule.

Post Tax Statement

Purpose

The POST TAX Statement posts a tax charge or credit transaction for a specified account. The transaction may be either deferred or not deferred. An optional service plan or budget plan may be associated with the transaction. If a budget plan is provided, the plan's variance will be updated accordingly. Additionally, the tax transaction may be associated with one or more previously posted transactions.

Format

POST TAX statements have this format:

```
POST TAX <stem_identifier>;
```

Where:

• <stem_identifier> is a stem that contains the appropriate attributes, as described under **Using the Financial Management Statements**.

To Create

1. Select **Statements-›Financials-›Post Tax** from the Rules Language Editor menu bar.

   The POST TAX Statement template appears.

2. Enter the appropriate stem identifier, or click the *right* mouse button and choose the stem identifier from the **Other Identifiers** list in the **Rules Language Elements Editor**.

3. Click **OK**. The statement appears in the rate form.

Example

Post a tax transaction based on energy usage.

```
/* Set Energy Tax Charge Attributes */
USAGE_TAX.ACCOUNTID = ACCOUNT.ACCOUNTID;
USAGE_TAX.REVENUEMONTH = BILLMONTH;
USAGE_TAX.NOTE = "Electric Energy Tax Charge - Energy Service
Provider";

USAGE_TAX.AMOUNT = $TAX_CHARGE;
USAGE_TAX.CURRENCY = "USD";
USAGE_TAX.BILLEDDATE = "07/15/2000";
USAGE_TAX.DUEDATE = "08/15/2000";

USAGE_TAX.RECEIVABLETYPENAME = "ESCO ELECTRIC";
USAGE_TAX.CHARGETYPEID = "ESCO ENERGY";
USAGE_TAX.OPCOCODE = OPCOCODE;
USAGE_TAX.JURISCODE = JURISCODE;
USAGE_TAX.BILLCYCLEDATE = BILLCYCLEDATE.READDATE;

/* Post Tax Transaction */
POST TAX USAGE_TAX;
```

Notes

In the above example, several of the USAGE_TAXG attributes are 'hard-coded' into the rate schedule. In actual practice, this data would probably come directly from records in the Oracle Utilities Data Repository, or through list queries. Also, the above example includes values for all the optional USAGE_TAX attributes. These are included to illustrate how those attributes might be supplied in a rate schedule.

Post Installment Statement

Purpose

The POST INSTALLMENT Statement posts a non-deferred charge transaction related to a previously created installment plan against a specified account.

Format

POST INSTALLMENT statements have this format:

```
POST INSTALLMENT <stem_identifier>;
```

Where:

- <stem_identifier> is a stem that contains the appropriate attributes, as described under **Using the Financial Management Statements**.

To Create

1. Select **Statements-›Financials-›Post Installment** from the Rules Language Editor menu bar.

   The POST INSTALLMENT Statement template appears.

2. Enter the appropriate stem identifier, or click the *right* mouse button and choose the stem identifier from the **Other Identifiers** list in the **Rules Language Elements Editor**.

3. Click **OK**. The statement appears in the rate form.

Example

Post an installment transaction based on energy usage.

```
/* Set installment Attributes */
USAGE_INST.ACCOUNTID = ACCOUNT.ACCOUNTID;
USAGE_INST.TRANSACTIONID = "310";
USAGE_INST.REVENUEMONTH = BILLMONTH;
USAGE_INST.NOTE = "Electric Energy Charge - Energy Service Provider";

USAGE_INST.AMOUNT = $ENERGY_CHARGE;
USAGE_INST.CURRENCY = "USD";
USAGE_INST.BILLEDDATE = "07/15/2000";
USAGE_INST.DUEDATE = "08/15/2000";

USAGE_INST.RECEIVABLETYPENAME = "ESCO ELECTRIC";
USAGE_INST.CHARGETYPEID = "ESCO ENERGY";
USAGE_INST.OPCOCODE = OPCOCODE;
USAGE_INST.JURISCODE = JURISCODE;
USAGE_INST.BILLCYCLEDATE = BILLCYCLEDATE.READDATE;

/* Post Installment */
POST INSTALLMENT USAGE_INST;
```

Notes

In the above example, several of the USAGE_INST attributes are 'hard-coded' into the rate schedule. In actual practice, this data would probably come directly from records in the Oracle Utilities Data Repository or through list queries. Also, the above example includes values for all the optional USAGE_INST attributes. These are included to illustrate how those attributes might be supplied in a rate schedule.

Post Statement Statement

Purpose

The POST STATEMENT Statement posts a single statement transaction against an account. The transaction indicates the current balance for the account. The account's current balance will not change.

Format

POST STATEMENT statements have this format:

```
POST STATEMENT <stem_identifier>;
```

Where:

• <stem_identifier> is a stem that contains the appropriate attributes, as described under **Using the Financial Management Statements**.

To Create

1. Select **Statements-›Financials-›Post Statement** from the Rules Language Editor menu bar.

   The POST STATEMENT Statement template appears.

2. Enter the appropriate stem identifier, or click the *right* mouse button and choose the stem identifier from the **Other Identifiers** list in the **Rules Language Elements Editor**.

3. Click **OK**. The statement appears in the rate form.

Example

Post a statement transaction based on energy usage.

```
/* Set Service Charge Attributes */
USAGE_SERV_CHG.ACCOUNTID = ACCOUNT.ACCOUNTID;
USAGE_SERV_CHG.TRANSACTIONID = "310";
USAGE_SERV_CHG.REVENUEMONTH = BILLMONTH;
USAGE_SERV_CHG.NOTE = "Electric Energy Charge - Energy Service
Provider";

USAGE_SERV_CHG.AMOUNT = $ENERGY_CHARGE;
USAGE_SERV.CURRENCY = "USD";
USAGE_SERV_CHG.BILLEDDATE = "07/15/2000";
USAGE_SERV_CHG.DUEDATE = "08/15/2000";

USAGE_SERV_CHG.RECEIVABLETYPENAME = "ESCO ELECTRIC";
USAGE_SERV_CHG.CHARGETYPEID = "ESCO ENERGY";
USAGE_SERV_CHG.OPCOCODE = OPCOCODE;
USAGE_SERV_CHG.JURISCODE = JURISCODE;
USAGE_SERV_CHG.STATEMENTDATE = "08/01/2000;
USAGE_SERV_CHG.BILLCYCLEDATE = BILLCYCLEDATE.READDATE;

/* Post Usage Statement */
POST STATEMENT USAGE_SERV_CHG;
```

Notes

In the above example, several of the USAGE_SERV_CHG attributes are 'hard-coded' into the rate schedule. In actual practice, this data would probably come directly from records in the Oracle Utilities Data Repository, or through list queries. Also, the above example includes values for all the optional USAGE_SERV_CHG attributes. These are included to illustrate how those attributes might be supplied in a rate schedule.

Post Bill Statement

Purpose

The POST BILL Statement posts a bill transaction against an account. This will trigger the IMMEDIATE credit application process, unless the APPLICATIONMETHOD is set to "DEFERRED". It may also initiate an autopayment for the account, if set up to do so. The account's current balance will not change, unless DEFERBALANCE is set to "FALSE".

Format

POST BILL statements have this format:

```
POST BILL <stem_identifier>;
```

Where:

* <stem_identifier> is a stem that contains the appropriate attributes, as described under **Using the Financial Management Statements**.

To Create

1. Select **Statements-›Financials-›Post Bill** from the Rules Language Editor menu bar.

   The POST BILL Statement template appears.

2. Enter the appropriate stem identifier, or click the *right* mouse button and choose the stem identifier from the **Other Identifiers** list in the **Rules Language Elements Editor**.

3. Click **OK**. The statement appears in the rate form.

Example

Post a bill for the total charge to the customer.

```
/* Set Bill Attributes */
TOTAL_BILL.ACCOUNTID = ACCOUNT.ACCOUNTID;
TOTAL_BILL.TRANSACTIONID = "3000";
TOTAL_BILL.REVENUEMONTH = BILLMONTH;
TOTAL_BILL.NOTE = "Total Bill, including customer and energy charges";

TOTAL_BILL.AMOUNT = $EFFECTIVE_REVENUE;
TOTAL_BILL.CURRENCY = "USD";
TOTAL_BILL.BILLEDDATE = "07/15/2000";
TOTAL_BILL.DUEDATE = "08/15/2000";

TOTAL_BILL.RECEIVABLETYPENAME = "ESCO ELECTRIC";
TOTAL_BILL.CHARGETYPEID = "ESCO ENERGY";
TOTAL_BILL.OPCOCODE = OPCOCODE;
TOTAL_BILL.JURISCODE = JURISCODE;
TOTAL_BILL.BILLCYCLEDATE = BILLCYCLEDATE.READDATE;

/* Post Bill */
POST BILL TOTAL_BILL;
```

Notes

In the above example, several of the TOTAL_BILL attributes are 'hard-coded' into the rate schedule. In actual practice, this data would probably come directly from records in the Oracle Utilities Data Repository, or through list queries. Also, the above example includes values for all the optional TOTAL_BILL attributes. These are included to illustrate how those attributes might be supplied in a rate schedule.

The POST BILL statements returns the following tail identifiers to allow utilizing a created transaction in subsequent processing:

- **UIDTRANSACTION**: The UID of the transation

- **TRANSACTIONTIME**: The time of the transaction

- **TRANSACTIONO**: The transaction number

Post Payment Statement

Purpose

The POST PAYMENT Statement posts a payment transaction against an account. This will trigger the IMMEDIATE credit application process, unless the APPLICATIONMETHOD is set to "DEFERRED" or "INVOICEID". The account's current balance will change.

Format

POST PAYMENT statements have this format:

```
POST PAYMENT <stem_identifier>;
```

Where:

• <stem_identifier> is a stem that contains the appropriate attributes, as described under **Using the Financial Management Statements**.

To Create

1. Select **Statements-›Financials-›Post Payment** from the Rules Language Editor menu bar.

   The POST PAYMENT Statement template appears.

2. Enter the appropriate stem identifier, or click the *right* mouse button and choose the stem identifier from the **Other Identifiers** list in the **Rules Language Elements Editor**.

3. Click **OK**. The statement appears in the rate form.

Example

Post a payment.

```
/* Set Payment Attributes */
PAYMENT.ACCOUNTID = ACCOUNT.ACCOUNTID;
PAYMENT.DEFACCOUNTID = "99999";
PAYMENT.SOURCECODE = "LOCKBOX";

PAYMENT.AMOUNT = "$90.00";
PAYMENT.CURRENCY = "USD";

/* Post Payment */
POST PAYMENT PAYMENT;
```

Notes

In the preceding example, several of the PAYMENT attributes are 'hard-coded' into the rate schedule. In actual practice, this data could also come directly from records in the Oracle Utilities Data Repository or through list queries.

Post Adjustment Statement

Purpose

The POST ADJUSTMENT Statement posts an adjustment transaction against an account. This will, by default, trigger the credit application process if the adjustment is a credit, unless the APPLICATIONMETHOD is overridden. The account's current balance will change by default unless the DEFERBALANCE is set to TRUE.

Format

POST ADJUSTMENT statements have this format:

```
POST ADJUSTMENT <stem_identifier>;
```

Where:

• <stem_identifier> is a stem that contains the appropriate attributes, as described under **Using the Financial Management Statements**.

To Create

1. Select **Statements-›Financials-›Post Adjustment** from the Rules Language Editor menu bar.

    The POST ADJUSTMENT Statement template appears.

2. Enter the appropriate stem identifier, or click the *right* mouse button and choose the stem identifier from the **Other Identifiers** list in the **Rules Language Elements Editor**.

3. Click **OK**. The statement appears in the rate form.

Example

Post a credit adjustment.

```
/* Set Payment Attributes */
CHG_ADJUST.ACCOUNTID = ACCOUNT.ACCOUNTID;
CHG_ADJUST.CHARGEORCREDIT = "CR"

CHG_ADJUST.AMOUNT = "$90.00";
CHG_ADJUST.CURRENCY = "USD";

/* Post Adjustment */
POST ADJUSTMENT CHG_ADJUST;
```

Notes

In the preceding example, several of the ADJUSTMENT attributes are 'hard-coded' into the rate schedule. In actual practice, this data could also come directly from records in the Oracle Utilities Data Repository or through list queries.

Post Refund Statement

Purpose

The POST Refund Statement posts a refund transaction against an account. This will trigger the IMMEDIATE credit application process, unless the APPLICATIONMETHOD is set to "DEFERRED" or "INVOICEID". The account's current balance will change.

Format

POST REFUND statements have this format:

```
POST REFUND <stem_identifier>;
```

Where:

• <stem_identifier> is a stem that contains the appropriate attributes, as described under **Using the Financial Management Statements**.

To Create

1. Select **Statements-›Financials-›Post Refund** from the Rules Language Editor menu bar.

   The POST REFUND Statement template appears.

2. Enter the appropriate stem identifier, or click the *right* mouse button and choose the stem identifier from the **Other Identifiers** list in the **Rules Language Elements Editor**.

3. Click **OK**. The statement appears in the rate form.

Example

Post a refund for the total charge to the customer.

```
/* Set Bill Attributes */
BILL_REFUND.ACCOUNTID = ACCOUNT.ACCOUNTID;
BILL_REFUND.TRANSACTIONID = "3000";
BILL_REFUND.REVENUEMONTH = BILLMONTH;
BILL_REFUND.NOTE = "Refund for Total Bill, including customer and
energy charges";

BILL_REFUND.AMOUNT = $EFFECTIVE_REVENUE;
BILL_REFUND.CURRENCY = "USD";

BILL_REFUND.RECEIVABLETYPENAME = "ESCO ELECTRIC";
BILL_REFUND.CHARGETYPEID = "ESCO ENERGY";
BILL_REFUND.OPCOCODE = OPCOCODE;
BILL_REFUND.JURISCODE = JURISCODE;
BILL_REFUND.BILLCYCLEDATE = BILLCYCLEDATE.READDATE;

BILL_REFUND.APPLICATIONMETHOD = "IMMEDIATE";
BILL_REFUND.DEFERBALANCE = "FALSE";

/* Post Refund */
POST REFUND BILL_REFUND;
```

Notes

In the above example, several of the BILL_REFUND attributes are 'hard-coded' into the rate schedule. In actual practice, this data would probably come directly from records in the Oracle Utilities Data Repository, or through list queries. Also, the above example includes values for all the optional BILL_REFUND attributes. These are included to illustrate how those attributes might be supplied in a rate schedule.

Post Writeoff Statement

Purpose

The POST WRITEOFF Statement is used to write off an account. All transactions for the account with an outstanding balance will be written off. This will trigger the IMMEDIATE credit application process. The account's current balance will change.

Format

POST WRITEOFF statements have this format:

```
POST WRITEOFF <stem_identifier>;
```

Where:

- <stem_identifier> is a stem that contains the appropriate attributes, as described under **Using the Financial Management Statements**.

To Create

1. Select **Statements-›Financials-›Post Writeoff** from the Rules Language Editor menu bar.

   The POST WRITEOFF Statement template appears.

2. Enter the appropriate stem identifier, or click the *right* mouse button and choose the stem identifier from the **Other Identifiers** list in the **Rules Language Elements Editor**.

3. Click **OK**. The statement appears in the rate form.

Example

Write off an account.

```
/* Set Write Off Attributes */
ACCT_WRITEOFF.ACCOUNTID = ACCOUNT.ACCOUNTID;

/* Post Write Off */
POST WRITEOFF ACCT_WRITEOFF;
```

Post Deposit Statement

Purpose

The POST DEPOSIT Statement posts a deposit charge transaction against an account. The account's current balance will change by default unless DEFERBALANCE is set to TRUE.

Format

POST DEPOSIT statements have this format:

```
POST DEPOSIT <stem_identifier>;
```

Where:

- <stem_identifier> is a stem that contains the appropriate attributes, as described under **Using the Financial Management Statements**.

To Create

1. Select **Statements-›Financials-›Post Deposit** from the Rules Language Editor menu bar.

   The POST DEPOSIT Statement template appears.

2. Enter the appropriate stem identifier, or click the *right* mouse button and choose the stem identifier from the **Other Identifiers** list in the **Rules Language Elements Editor**.

3. Click **OK**. The statement appears in the rate form.

Example

Post a deposit.

```
/* Set Deposit Attributes */
ACCT_DEP.ACCOUNTID = ACCOUNT.ACCOUNTID;
ACCT_DEP.DEPINTRATE = FACTOR[DEPOSIT_INT_RATE].VALUE

ACCT_DEP.AMOUNT = "$90.00";
ACCT_DEP.CURRENCY = "USD";

/* Post Deposit */
POST DEPOSIT ACCT_DEP;
```

Post Deposit Interest Statement

Purpose

The POST DEPOSIT INTEREST Statement posts deposit interest as a single transaction.

Format

POST DEPOSIT INTEREST statements have this format:

```
POST DEPOSIT INTEREST <stem_identifier>;
```

Where:

- <stem_identifier> is a stem that contains the appropriate attributes, as described under **Using the Financial Management Statements**.

To Create

1. Select **Statements-›Financials-›Post Deposit Interest** from the Rules Language Editor menu bar.

   The POST DEPOSIT INTEREST Statement template appears.

2. Enter the appropriate stem identifier, or click the *right* mouse button and choose the stem identifier from the **Other Identifiers** list in the **Rules Language Elements Editor**.

3. Click **OK**. The statement appears in the rate form.

Example

Post deposit interest.

```
/* Set Deposit Interest Attributes */
DEP_INT.ACCOUNTID = ACCOUNT.ACCOUNTID;

DEP_INT.AMOUNT = $DEPOSIT_INTEREST;
DEP_INT.CURRENCY = "USD";

DEP_INT.OPCOCODE = OPCOCODE;
DEP_INT.JURISCODE = JURISCODE;

/* Post Deposit Interest */
POST DEPOSIT INTEREST USAGE_SERV_CHG;
```

Notes

In the preceding example, several of the DEP_INT attributes are 'hard-coded' into the rate schedule. In actual practice, this data would probably come directly from records in the Oracle Utilities Data Repository, or through list queries. Also, the above example includes values for all the optional DEP_INT attributes. These are included to illustrate how those attributes might be supplied in a rate schedule.

Post Deposit Application Statement

Purpose

The POST DEPOSIT APPLICATION Statement is used to apply a deposit as a single transaction.

Format

POST DEPOSIT APPLICATION statements have this format:

```
POST DEPOSIT APPLICATION <stem_identifier>;
```

Where:

- <stem_identifier> is a stem that contains the appropriate attributes, as described under **Using the Financial Management Statements**.

To Create

1. Select **Statements-›Financials-›Post Deposit Application** from the Rules Language Editor menu bar.

   The POST DEPOSIT APPLICATION Statement template appears.

2. Enter the appropriate stem identifier, or click the *right* mouse button and choose the stem identifier from the **Other Identifiers** list in the **Rules Language Elements Editor**.

3. Click **OK**. The statement appears in the rate form.

Example

Post a deposit application.

```
/* Set deposit application attributes */
DEP_APP.ACCOUNTID = ACCOUNT.ACCOUNTID;

DEP_APP.AMOUNT = $ENERGY_CHARGE;
DEP_APP.CURRENCY = "USD";

DEP_APP.OPCOCODE = OPCOCODE;
DEP_APP.JURISCODE = JURISCODE;

/* Post Deposit Application */
POST DEPOSIT APPLICATION DEP_APP;
```

Notes

In the above example, several of the DEP_APP attributes are 'hard-coded' into the rate schedule. In actual practice, this data would probably come directly from records in the Oracle Utilities Data Repository, or through list queries. Also, the above example includes values for all the optional DEP_APP attributes. These are included to illustrate how those attributes might be supplied in a rate schedule.

Cancel Transaction Statement

Purpose

The CANCEL_TRAN statement cancels a single transaction.

Format

CANCEL_TRAN statements have this format:

```
CANCEL_TRANS <stem_identifier>;
```

Where:

- <stem_identifier> is a stem that contains the following:

    - UID

    - ACCOUNTID

    - TRANSACTIONID

    - CANCELREVENUEMONTH (optional)

    - CANCELREASONCODE (optional)

    - CANCELNOTE (optional)

    Note: Either the UIDTRANSACTION or the ACCOUNTID and TRANSACTIONID are required to identify the specific transaction to be cancelled. See **Using the Financial Management Statements** on page 7-2 for more information about attributes used with Oracle Utilities Receivables Component Rules Language statements.

To Create

1. Select **Statements-›Financials-›Cancel Transaction** from the Rules Language Editor menu bar.

    The CANCEL_TRAN statement template appears.

2. Enter the appropriate stem identifier, or click the *right* mouse button and choose the stem identifier from the **Other Identifiers** list in the **Rules Language Elements Editor**.

3. Click **OK**. The statement appears in the rate form.

Example

Cancel a service charge transaction based on energy usage.

```
/* Set Cancel Attributes */
CANCEL_SERV_CHG.UID = 24579;
CANCEL_SERV_CHG.CANCELNOTE = "Cancelled";
CANCEL_SERV_CHG.CANCELREASONCODE = "ERROR";
CANCEL_SERV_CHG.CANCELREVENUEMONTH = BILLMONTH;

/* Cancel Service Charge */
CANCEL_TRAN CANCEL_SERV_CHG;
```

or

Cancel a service charge transaction based on energy usage.

```
/* Set Cancel Attributes */
CANCEL_SERV_CHG.ACCOUNTID = ACCOUNT.ACCOUNTUID;
CANCEL_SERV_CHG.TRANSACTIONID = 110;
CANCEL_SERV_CHG.CANCELNOTE = "Cancelled";
CANCEL_SERV_CHG.CANCELREASONCODE = "ERROR";
CANCEL_SERV_CHG.CANCELREVENUEMONTH = BILLMONTH;

/* Cancel Service Charge */
CANCEL_TRAN CANCEL_SERV_CHG;
```

Notes

In the above example, several of the CANCEL_SERV_CHG attributes are 'hard-coded' into the rate schedule. In actual practice, this data would probably come directly from records in the Oracle Utilities Data Repository, or through list queries. Also, the above example includes values for all the optional CANCEL_SERV_CHG attributes. These are included to illustrate how those attributes might be supplied in a rate schedule.

CALCULATE_LATEPAYMENT Function

Purpose

The CALCULATE_LATEPAYMENT function is used to calculate a late payment fee based on an account's outstanding balance and current collections status.

Format

Statements using the CALCULATE_LATEPAYMENT function have this format:

```
<identifier> = CALCULATE_LATEPAYMENT (<ACCOUNTID> , <DATETYPE> ,
<COLLECTIONSTATUS) ;
```

Where:

- <ACCOUNTID> The Account ID of the account.

- <DATETYPE> The date type used for the calculation. Valid values include:

    - TRANSACTIONTIME

    - STATEMENTDATE

    - INVOICEDATE

    - DUEDATE

- <COLLECTIONSTATUS> The account's current collection status, from the Account Oracle Utilities Receivables Component table.

The function will return a structure that includes:

- The calculated late payment fee

- The maximum amount the late payment should be for the account.

Example

Calculate latepayment fees for each account in the BACK_OFFICE list based on INVOICEDATE.

```
FOR EACH ACCT IN LIST BACK_OFFICE
   ACCT_ID = ACCOUNTS.ACCOUNTID
   STATUS = ACCOUNTFME.COLLECTIONSTATUS
   ACCOUNT_LATE_FEE = CALCULATE_LATEPAYMENT(ACCT_ID, "INVOICEDATE",
STATUS)
   ...
END FOR;
```

FMGETBILLINFO Function

Purpose

The FMGETBILLINFO function is used to gather bill information for an account.

Format

Statements using the FMGETBILLINFO function have this format:

```
<identifier> = FMGETBILLINFO [(<ID> , <DATE>)];
```

Where:

• <ID> <DATE> *Optional.* Account ID and date. If the account ID is not provided, the account processed in the rate form will be used. If the date is not provided, the current date will be used.

The function will return a structure (stem) that includes:

• The Account's Receivable Status

• The Account's Current Balance

• The Account's Past Due Balance.

Example

Get the account bill info for account ID BACK-OFFICE-1 for August 15, 2000.

```
ACCOUNT_BILL_INFO = FMGETBILLINFO(BACK-OFFICE-1, '08/15/2000')
```

PROCESSAUTOPAYMENT Function

Purpose

The PROCESSAUTOPAYMENT function is used to process an automatic payment for an account. This function is typically used if normal automatic payments for an account have been suspended.

Format

Statements using the PROCESSAUTOPAYMENT function have this format:

```
<identifier> = PROCESSAUTOPAYMENT (<STEM>);
```

Where:

•   <stem_identifier> is a stem that contains the appropriate attributes, as described under **Using the Financial Management Statements**.

The function return zero (0) if successful.

Example

Process an automatic payment after posting a bill.

```
/* Set Bill Attributes */
TOTAL_BILL.ACCOUNTID = ACCOUNT.ACCOUNTID;
TOTAL_BILL.TRANSACTIONID = "3000";
TOTAL_BILL.REVENUEMONTH = BILLMONTH;
TOTAL_BILL.NOTE = "Total Bill, including customer and energy charges";

TOTAL_BILL.AMOUNT = $EFFECTIVE_REVENUE;
TOTAL_BILL.CURRENCY = "USD";
TOTAL_BILL.BILLEDDATE = "07/15/2000";
TOTAL_BILL.DUEDATE = "08/15/2000";

TOTAL_BILL.RECEIVABLETYPENAME = "ESCO ELECTRIC";
TOTAL_BILL.CHARGETYPEID = "ESCO ENERGY";
TOTAL_BILL.OPCOCODE = OPCOCODE;
TOTAL_BILL.JURISCODE = JURISCODE;
TOTAL_BILL.SUSPENDAUTOPAYMENT = "TRUE";
TOTAL_BILL.BILLCYCLEDATE = BILLCYCLEDATE.READDATE;

/* Post Bill */
POST BILL TOTAL_BILL;
...
...
/* Process autopayment */
PAP = PROCESSAUTOPAYMENT (TOTAL_BILL);
```

Deprecated Statements

The following statements have been replaced or fallen into disuse. Use the POST CHARGEORCREDIT statement instead of the following statements where possible.

Post Service Charge Statement

Purpose

The POST SERVICE CHARGE Statement posts a service charge against an account associated with a service plan. The account's current balance should increase by the amount of the charge, unless the DEFERBALANCE is set to "TRUE".

Format

POST SERVICE CHARGE statements have this format:

```
POST SERVICE CHARGE <stem_identifier>;
```

Where:

- <stem_identifier> is a stem that contains the appropriate attributes, as described under **Using the Financial Management Statements**.

To Create

1. Select **Statements-›Financials-›Post Service Charge** from the Rules Language Editor menu bar.

   The POST SERVICE CHARGE Statement template appears.

2. Enter the appropriate stem identifier, or click the *right* mouse button and choose the stem identifier from the **Other Identifiers** list in the **Rules Language Elements Editor**.

3. Click **OK**. The statement appears in the rate form.

Example

Post a service charge for energy usage.

```
/* Set Service Charge Attributes */
USAGE_SERV_CHG.ACCOUNTID = ACCOUNT.ACCOUNTID;
USAGE_SERV_CHG.TRANSACTIONID = "310";
USAGE_SERV_CHG.REVENUEMONTH = BILLMONTH;
USAGE_SERV_CHG.NOTE = "Electric Energy Charge - Energy Service
Provider";

USAGE_SERV_CHG.AMOUNT = $ENERGY_CHARGE;
USAGE_SERV_CHG.BILLEDDATE = "07/15/2000";
USAGE_SERV_CHG.DUEDATE = "08/15/2000";

USAGE_SERV_CHG.RECEIVABLETYPENAME = "ESCO ELECTRIC";
USAGE_SERV_CHG.CHARGETYPEID = "ESCO ENERGY";
USAGE_SERV_CHG.OPCOCODE = OPCOCODE;
USAGE_SERV_CHG.JURISCODE = JURISCODE;
USAGE_SERV_CHG.BILLCYCLEDATE = BILLCYCLEDATE.READDATE;

/* Post Usage Service Charge */
POST SERVICE CHARGE USAGE_SERV_CHG;
```

Notes

In the above example, several of the USAGE_SERV_CHG attributes are 'hard-coded' into the rate schedule. In actual practice, this data would probably come directly from records in the Oracle Utilities Data Repository, or through list queries. Also, the above example includes values for all the optional USAGE_SERV_CHG attributes. These are included to illustrate how those attributes might be supplied in a rate schedule.

If the Transaction Table in your database contains columns that are not included in the base schema (and therefore not among the data elements listed under **Using the Financial Management Statements**), you can post values to those columns by assigning values to corresponding STEM.COLUMN_NAME identifiers in the rate schedule. In this case, the column name specified in the rate schedule must be the exact name of the column in the database. For example, if your Transaction Table contains a column called ZONE, you could post data to that column by including the following line in your rate schedule:

USAGE_SERV_CHG.ZONE = "ZONE_1"

```
/* Post Usage Service Charge */
POST SERVICE CHARGE USAGE_SERV_CHG;
```

This would post the value assigned to the USAGE_SERV_CHG.ZONE identifier ("ZONE_1") to the ZONE column in the Transaction Table.

Post Deferred Service Charge Statement

Purpose

The POST DEFERRED SERVICE CHARGE Statement posts a deferred service charge against an account associated with a service plan. The account's current balance will not change.

Format

POST DEFERRED SERVICE CHARGE statements have this format:

```
POST DEFERRED SERVICE CHARGE <stem_identifier>;
```

Where:

- <stem_identifier> is a stem that contains the appropriate attributes, as described under **Using the Financial Management Statements**.

To Create

1. Select **Statements-›Financials-›Post Deferred Service Charge** from the Rules Language Editor menu bar.

   The POST DEFERRED SERVICE CHARGE Statement template appears.

2. Enter the appropriate stem identifier, or click the *right* mouse button and choose the stem identifier from the **Other Identifiers** list in the **Rules Language Elements Editor**.

3. Click **OK**. The statement appears in the rate form.

Example

Post a deferred service charge for energy usage.

```
/* Set Deferred Service Charge Attributes */
USAGE_SERV_CHG.ACCOUNTID = ACCOUNT.ACCOUNTID;
USAGE_SERV_CHG.TRANSACTIONID = "310";
USAGE_SERV_CHG.REVENUEMONTH = BILLMONTH;
USAGE_SERV_CHG.NOTE = "Electric Energy Charge - Energy Service
Provider";

USAGE_SERV_CHG.AMOUNT = $ENERGY_CHARGE;
USAGE_SERV_CHG.BILLEDDATE = "07/15/2000";
USAGE_SERV_CHG.DUEDATE = "08/15/2000";

USAGE_SERV_CHG.RECEIVABLETYPENAME = "ESCO ELECTRIC";
USAGE_SERV_CHG.CHARGETYPEID = "ESCO ENERGY";
USAGE_SERV_CHG.OPCOCODE = OPCOCODE;
USAGE_SERV_CHG.JURISCODE = JURISCODE;
USAGE_SERV_CHG.BILLCYCLEDATE = BILLCYCLEDATE.READDATE;

/* Post Usage Service Charge */
```
POST DEFERRED SERVICE CHARGE USAGE_SERV_CHG;

Notes

In the above example, several of the USAGE_SERV_CHG attributes are 'hard-coded' into the rate schedule. In actual practice, this data would probably come directly from records in the Oracle Utilities Data Repository, or through list queries. Also, the above example includes values for all the optional USAGE_SERV_CHG attributes. These are included to illustrate how those attributes might be supplied in a rate schedule.

If the Transaction Table in your database contains columns that are not included in the base schema (and therefore not among the data elements listed under **Using the Financial Management Statements**), you can post values to those columns by assigning values to corresponding STEM.COLUMN_NAME identifiers in the rate schedule. In this case, the column name specified in the rate schedule must be the exact name of the column in the database. For example, if your Transaction Table contains a column called ZONE, you could post data to that column by including the following line in your rate schedule:

USAGE_SERV_CHG.ZONE = "ZONE_1"

```
/* Post Usage Service Charge */
POST DEFERRED SERVICE CHARGE USAGE_SERV_CHG;
```

This would post the value assigned to the USAGE_SERV_CHG.ZONE identifier ("ZONE_1") to the ZONE column in the Transaction Table.

Post Budget Service Charge Statement

Purpose

The POST BUDGET SERVICE CHARGE Statement posts a service charge against an account associated with a service plan and a budget plan. The account's current balance will not change; however, the budget plan variance will increase by the amount of the charge.

Format

POST BUDGET SERVICE CHARGE statements have this format:

```
POST BUDGET SERVICE CHARGE <stem_identifier>;
```

Where:

• <stem_identifier> is a stem that contains the appropriate attributes, as described under **Using the Financial Management Statements**.

To Create

1. Select **Statements-›Financials-›Post Budget Service Charge** from the Rules Language Editor menu bar.

   The POST BUDGET SERVICE CHARGE Statement template appears.

2. Enter the appropriate stem identifier, or click the *right* mouse button and choose the stem identifier from the **Other Identifiers** list in the **Rules Language Elements Editor**.

3. Click **OK**. The statement appears in the rate form.

Example

Post a budget service charge for energy usage.

```
/* Set Service Plan Attributes */
SERV_PLAN.STARTDATE = "01/01/1998";
SERV_PLAN.SERVICETYPECODE = "ELECTRIC";

/* Set Budget Plan Attributes */
BUDGET_PLAN.STARTDATE = "01/01/1998";
BUDGET_PLAN.BUDGETTYPECODE = "BUDGETELECTRIC";
BUDGET_PLAN.SERVICEPLAN = "SERV_PLAN";

/* Set Budget Service Charge Attributes */
USAGE_SERV_CHG.ACCOUNTID = ACCOUNT.ACCOUNTID;
USAGE_SERV_CHG.TRANSACTIONID = "1105";
USAGE_SERV_CHG.REVENUEMONTH = BILLMONTH;
USAGE_SERV_CHG.NOTE = "Electric Energy Charge - Energy Service
Provider";
USAGE_SERV_CHG.AMOUNT = $ENERGY_CHARGE;
USAGE_SERV_CHG.BILLEDDATE = "07/15/2000";
USAGE_SERV_CHG.DUEDATE = "08/15/2000";
USAGE_SERV_CHG.RECEIVABLETYPENAME = "ESCO ELECTRIC";
USAGE_SERV_CHG.CHARGETYPEID = "ESCO ENERGY";
USAGE_SERV_CHG.OPCOCODE = OPCOCODE;
USAGE_SERV_CHG.JURISCODE = JURISCODE;
USAGE_SERV_CHG.BILLCYCLEDATE = BILLCYCLEDATE.READDATE;

USAGE_SERV_CHG.SERVICEPLAN = "SERV_PLAN";
USAGE_SERV_CHG.BUDGETPLAN = "BUDGET_PLAN";

/* Post Usage Budget Service Charge */
POST BUDGET SERVICE CHARGE USAGE_SERV_CHG;
```

Notes

In the above example, several of the USAGE_SERV_CHG attributes are 'hard-coded' into the rate schedule. In actual practice, this data would probably come directly from records in the Oracle Utilities Data Repository, or through list queries. Also, the above example includes values for all the optional USAGE_SERV_CHG attributes. These are included to illustrate how those attributes might be supplied in a rate schedule.

If the Transaction Table in your database contains columns that are not included in the base schema (and therefore not among the data elements listed under **Using the Financial Management Statements**), you can post values to those columns by assigning values to corresponding STEM.COLUMN_NAME identifiers in the rate schedule. In this case, the column name specified in the rate schedule must be the exact name of the column in the database. For example, if your Transaction Table contains a column called ZONE, you could post data to that column by including the following line in your rate schedule:

USAGE_SERV_CHG.ZONE = "ZONE_1"

```
/* Post Usage Service Charge */
POST BUDGET SERVICE CHARGE USAGE_SERV_CHG;
```

This would post the value assigned to the USAGE_SERV_CHG.ZONE identifier ("ZONE_1") to the ZONE column in the Transaction Table.

Post Budget Bill Charge Statement

Purpose

The POST BUDGET BILL CHARGE Statement posts a budget bill charge against an account associated with a budget plan. The account's current balance should increase by the amount of the charge, and the budget plan variance should decrease by the amount of the charge.

Format

POST BUDGET BILL CHARGE statements have this format:

```
POST BUDGET BILL CHARGE <stem_identifier>;
```

Where:

• <stem_identifier> is a stem that contains the appropriate attributes, as described under **Using the Financial Management Statements**.

To Create

1. Select **Statements-›Financials-›Post Budget Bill Charge** from the Rules Language Editor menu bar.

   The POST BUDGET BILL CHARGE Statement template appears.

2. Enter the appropriate stem identifier, or click the *right* mouse button and choose the stem identifier from the **Other Identifiers** list in the **Rules Language Elements Editor**.

3. Click **OK**. The statement appears in the rate form.

Example

Post a budget bill charge for energy usage.

```
/* Set Service Plan Attributes */
SERV_PLAN.STARTDATE = "01/01/1998";
SERV_PLAN.SERVICETYPECODE = "ELECTRIC";

/* Set Budget Plan Attributes */
BUDGET_PLAN.STARTDATE = "01/01/1998";
BUDGET_PLAN.BUDGETTYPECODE = "BUDGETELECTRIC";
BUDGET_PLAN.SERVICEPLAN = "SERV_PLAN";

/* Set Budget Bill Attributes */
USAGE_BUDGET_CHG.ACCOUNTID = ACCOUNT.ACCOUNTID;
USAGE_BUDGET_CHG.TRANSACTIONID = "1100";
USAGE_BUDGET_CHG.REVENUEMONTH = BILLMONTH;
USAGE_BUDGET_CHG.NOTE = "Electric Energy Charge - Energy Service
Provider";
USAGE_BUDGET_CHG.AMOUNT = $ENERGY_CHARGE;
USAGE_BUDGET_CHG.BILLEDDATE = "07/15/2000";
USAGE_BUDGET_CHG.DUEDATE = "08/15/2000";
USAGE_BUDGET_CHG.RECEIVABLETYPENAME = "ESCO ELECTRIC";
USAGE_BUDGET_CHG.CHARGETYPEID = "ESCO ENERGY";
USAGE_BUDGET_CHG.OPCOCODE = OPCOCODE;
USAGE_BUDGET_CHG.JURISCODE = JURISCODE;
USAGE_BUDGET_CHG.BILLCYCLEDATE = BILLCYCLEDATE.READDATE;

USAGE_BUDGET_CHG.SERVICEPLAN = "SERV_PLAN";
USAGE_BUDGET_CHG.BUDGETPLAN = "BUDGET_PLAN";

/* Post Budget Bill Usage Charge */
```
POST BUDGET BILL CHARGE USAGE_BUDGET_CHG;

Notes

In the above example, several of the USAGE_BUDGET_CHG attributes are 'hard-coded' into the rate schedule. In actual practice, this data would probably come directly from records in the Oracle Utilities Data Repository, or through list queries. Also, the above example includes values for all the optional USAGE_BUDGET_CHG attributes. These are included to illustrate how those attributes might be supplied in a rate schedule.

If the Transaction Table in your database contains columns that are not included in the base schema (and therefore not among the data elements listed under **Using the Financial Management Statements**), you can post values to those columns by assigning values to corresponding STEM.COLUMN_NAME identifiers in the rate schedule. In this case, the column name specified in the rate schedule must be the exact name of the column in the database. For example, if your Transaction Table contains a column called ZONE, you could post data to that column by including the following line in your rate schedule:

USAGE_BUDGET_CHG.ZONE = "ZONE_1"

```
/* Post Budget Bill Usage Charge */
POST BUDGET BILL CHARGE USAGE_BUDGET_CHG;
```

This would post the value assigned to the USAGE_BUDGET_CHG.ZONE identifier ("ZONE_1") to the ZONE column in the Transaction Table.

Post Budget Bill Trueup Statement

Purpose

The POST BUDGET BILL TRUEUP Statement posts a budget bill true-up charge or credit against an account associated with a budget plan. The account's current balance should increase (if a charge) or decrease (if a credit) by the amount of the transaction. The budget plan variance should decrease (if a charge) or increase (if a credit) by the amount of the transaction.

Format

POST BUDGET BILL TRUEUP statements have this format:

```
POST BUDGET SERVICE CHARGE <stem_identifier>;
```

Where:

- <stem_identifier> is a stem that contains the appropriate attributes, as described under **Using the Financial Management Statements**.

To Create

1. Select **Statements-›Financials-›Post Budget Bill Trueup** from the Rules Language Editor menu bar.

   The POST BUDGET BILL TRUEUP Statement template appears.

2. Enter the appropriate stem identifier, or click the *right* mouse button and choose the stem identifier from the **Other Identifiers** list in the **Rules Language Elements Editor**.

3. Click **OK**. The statement appears in the rate form.

Example

Post a budget bill true-up charge based on energy usage.

```
/* Set Service Plan Attributes */
SERV_PLAN.STARTDATE = "01/01/1998";
SERV_PLAN.SERVICETYPECODE = "ELECTRIC";

/* Set Budget Plan Attributes */
BUDGET_PLAN.STARTDATE = "01/01/1998";
BUDGET_PLAN.BUDGETTYPECODE = "BUDGETELECTRIC";
BUDGET_PLAN.SERVICEPLAN = "SERV_PLAN";

/* Set Budget Bill Trueup Attributes */
BUDGET_TRUEUP_USAGE_CHG.ACCOUNTID = ACCOUNT.ACCOUNTID;
BUDGET_TRUEUP_USAGE_CHG.TRANSACTIONID = "1130";
BUDGET_TRUEUP_USAGE_CHG.REVENUEMONTH = BILLMONTH;
BUDGET_TRUEUP_USAGE_CHG.NOTE = "Budget Bill Trueup - Electric Energy
Charge";
BUDGET_TRUEUP_USAGE_CHG.CHARGEORCREDIT = "CH"
BUDGET_TRUEUP_USAGE_CHG.AMOUNT = $ENERGY_CHARGE;
BUDGET_TRUEUP_USAGE_CHG.BILLEDDATE = "07/15/2000";
BUDGET_TRUEUP_USAGE_CHG.DUEDATE = "08/15/2000";
BUDGET_TRUEUP_USAGE_CHG.RECEIVABLETYPENAME = "ESCO ELECTRIC";
BUDGET_TRUEUP_USAGE_CHG.CHARGETYPEID = "ESCO ENERGY";
BUDGET_TRUEUP_USAGE_CHG.OPCOCODE = OPCOCODE;
BUDGET_TRUEUP_USAGE_CHG.JURISCODE = JURISCODE;
BUDGET_TRUEUP_USAGE_CHG.BILLCYCLEDATE = BILLCYCLEDATE.READDATE;
BUDGET_TRUEUP_USAGE_CHG.SERVICEPLAN = "SERV_PLAN";
BUDGET_TRUEUP_USAGE_CHG.BUDGETPLAN = "BUDGET_PLAN";

/* Post Budget Bill Trueup Usage Charge */
POST BUDGET BILL TRUEUP BUDGET_TRUEUP_USAGE_CHG;
```

Notes

In the above example, several of the BUDGET_TRUEUP_USAGE_CHG attributes are 'hard-coded' into the rate schedule. In actual practice, this data would probably come directly from records in the Oracle Utilities Data Repository, or through list queries. Also, the above example includes values for all the optional BUDGET_TRUEUP_USAGE_CHG attributes. These are included to illustrate how those attributes might be supplied in a rate schedule.

If the Transaction Table in your database contains columns that are not included in the base schema (and therefore not among the data elements listed under **Using the Financial Management Statements**), you can post values to those columns by assigning values to corresponding STEM.COLUMN_NAME identifiers in the rate schedule. In this case, the column name specified in the rate schedule must be the exact name of the column in the database. For example, if your Transaction Table contains a column called ZONE, you could post data to that column by including the following line in your rate schedule:

```
BUDGET_TRUEUP_USAGE_CHG.ZONE = "ZONE_1"
/* Post Budget Bill Trueup Usage Charge */
POST BUDGET BILL TRUEUP BUDGET_TRUEUP_USAGE_CHG;
```

This would post the value assigned to the BUDGET_TRUEUP_USAGE_CHG.ZONE identifier ("ZONE_1") to the ZONE column in the Transaction Table.

Post Installment Charge Statement

Purpose

The POST INSTALLMENT CHARGE Statement posts an installment charge against an account. The account's current balance should increase by the amount of the charge, unless the DEFERBALANCE is set to "TRUE".

Format

POST INSTALLMENT CHARGE statements have this format:

```
POST INSTALLMENT CHARGE <stem_identifier>;
```

Where:

- <stem_identifier> is a stem that contains the appropriate attributes, as described under **Using the Financial Management Statements**.

To Create

1. Select **Statements-›Financials-›Post Installment Charge** from the Rules Language Editor menu bar.

   The POST INSTALLMENT CHARGE Statement template appears.

2. Enter the appropriate stem identifier, or click the *right* mouse button and choose the stem identifier from the **Other Identifiers** list in the **Rules Language Elements Editor**.

3. Click **OK**. The statement appears in the rate form.

Example

Post an installment charge.

```
/* Set Installment Charge Attributes */
BILL_INST_CHG.ACCOUNTID = ACCOUNT.ACCOUNTID;
BILL_INST_CHG.TRANSACTIONID = "1510";
BILL_INST_CHG.REVENUEMONTH = BILLMONTH;
BILL_INST_CHG.NOTE = "Bill Installment";

BILL_INST_CHG.AMOUNT = $INSTALL_CHG;
BILL_INST_CHG.BILLEDDATE = "07/15/2000";
BILL_INST_CHG.DUEDATE = "08/15/2000";

BILL_INST_CHG.RECEIVABLETYPENAME = "ESCO ELECTRIC";
BILL_INST_CHG.CHARGETYPEID = "ESCO ENERGY";
BILL_INST_CHG.OPCOCODE = OPCOCODE;
BILL_INST_CHG.JURISCODE = JURISCODE;
BILL_INST_CHG.BILLCYCLEDATE = BILLCYCLEDATE.READDATE;

/* Post Installment Charge */
POST INSTALLMENT CHARGE BILL_INST_CHG;
```

Notes

In the above example, several of the BILL_INST_CHG attributes are 'hard-coded' into the rate schedule. In actual practice, this data would probably come directly from records in the Oracle Utilities Data Repository, or through list queries. Also, the above example includes values for all the optional BILL_INST_CHG attributes. These are included to illustrate how those attributes might be supplied in a rate schedule.

If the Transaction Table in your database contains columns that are not included in the base schema (and therefore not among the data elements listed under **Using the Financial Management Statements**), you can post values to those columns by assigning values to corresponding STEM.COLUMN_NAME identifiers in the rate schedule. In this case, the column name specified in the rate schedule must be the exact name of the column in the database. For example, if your Transaction Table contains a column called ZONE, you could post data to that column by including the following line in your rate schedule:

```
BILL_INST_CHG.ZONE = "ZONE_1"

/* Post Installment Charge */
POST INSTALLMENT CHARGE BILL_INST_CHG;
```

This would post the value assigned to the BILL_INST_CHG.ZONE identifier ("ZONE_1") to the ZONE column in the Transaction Table.

Chapter 8

Workflow Management Statements

This chapter provides detailed explanations of the Workflow Management statements available in the Oracle Utilities Rules Language. These statements are used to work with processes and events using the workflow management functions of Oracle Utilities Billing Component.

Workflow Management Statements

- **Process Start Statement**
- **Process Suspend Statement**
- **Process Resume Statement**
- **Process Terminate Statement**
- **Process Event Statement**

Using the Workflow Management Statements

The Workflow Management statements are used with processes in run via the workflow management functionality of Oracle Utilities Billing Component. Each statement takes as a single argument an identifier. The identifier is a stem that should contain several tail attributes, as described below.

Attribute	Description
UID	The process instance UID. Required input for the Process Suspend, Process Resume, and Process Terminate statements, and is the output of the Process Start Statement.
NOTE	A process instance note. Required input for the Process Terminate Statement; optional for all others.
UIDACCOUNT	Optional account UID, used as input for the Process Start or Process Event statements. If an account is to be associated with the process, either this or the ACCOUNTID (below) must be provided.
ACCOUNTID	Optional account ID, used as for input for the Process Start or Process Event statements. If an account is to be associated with the process, either this or the UIDACCOUNT (above) must be provided.
PROCESSNAME	Name of the process model to be started by the Process Start Statement. Required input for the Process Start Statement.
OPCOCODE	Optional operating company code of the process model to be started by the Process Start Statement.
JURISCODE	Optional jurisdiction code of the process model to be started by the Process Start Statement.
EVENTCODE	Required event type code for the Process Event Statement.
CONTEXT	Optional context for the Process Start or Process Event statements. Should be set to an identifier that is either an XML DOM node or XML DOM document.

Example:

```
COLLECT_PROC.PROCESSNAME = "Collections";
COLLECT_PROC.OPCOCODE = "AGL";
COLLECT_PROC.ACCOUNTID = ACCOUNT.ID;

PROCESS START COLLECT_PROC;
```

Process Start Statement

Purpose

The PROCESS START Statement is used to start a new process instance.

Format

PROCESS START statements have this format:

```
PROCESS START <stem_identifier>;
```

Where:

- <stem_identifier> is a stem that contains the appropriate attributes, as described under **Using the Workflow Management Statements**.

To Create

1. Select **Statements-›Workflow-›Process Start** from the Rules Language Editor menu bar.

   The PROCESS START Statement template appears.
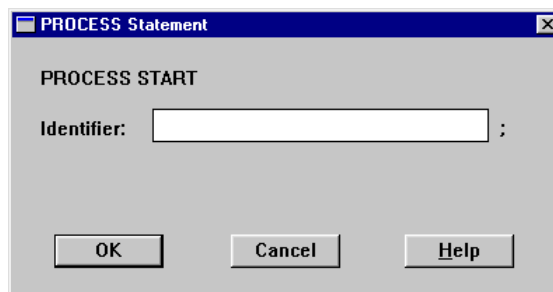
2. Enter the appropriate stem identifier, or click the *right* mouse button and choose the stem identifier from the **Other Identifiers** list in the **Rules Language Elements Editor**.

3. Click **OK**. The statement appears in the rate form.

Example

Start a process instance of the COLLECT_PROC process.

```
/* Set the process context */
PROC_CONTEXT = DOMDOCLOADFILE ("COLLECT_CONTEXT.XMl")

/* Set Process Instance Attributes */
COLLECT_PROC.ACCOUNTID = ACCOUNT.ID;
COLLECT_PROC.PROCESSNAME = "Collections";
COLLECT_PROC.OPCOCODE = "AGL";
COLLECT_PROC.JURISCODE = "GA";
COLLECT_PROC.CONTEXT = PROC_CONTEXT;

/* Start the process */
PROCESS START COLLECT_PROC;
```

Notes

In the preceding example, several of the COLLECT_PROC attributes are 'hard-coded' into the rate schedule. In actual practice, this data could also come directly from records in the Oracle Utilities Data Repository, or through list queries. Also, the above example includes values for several of the optional COLLECT_PROC attributes. These are included to illustrate how those attributes might be supplied in a rate schedule.

Process Suspend Statement

Purpose

The PROCESS SUSPEND Statement is used to suspend an existing running process instance.

Format

PROCESS SUSPEND statements have this format:

```
PROCESS SUSPEND <stem_identifier>;
```
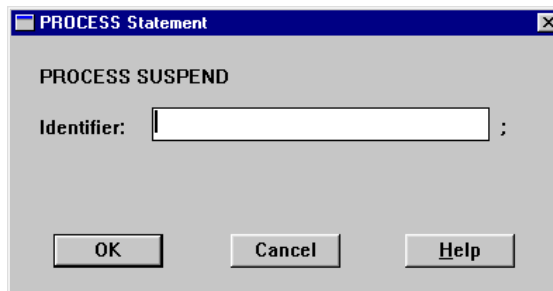
Where:

• <stem_identifier> is a stem that contains the appropriate attributes, as described under **Using the Workflow Management Statements**.

To Create

1. Select **Statements-›Workflow-›Process Suspend** from the Rules Language Editor menu bar.

   The PROCESS SUSPEND Statement template appears.

2. Enter the appropriate stem identifier, or click the *right* mouse button and choose the stem identifier from the **Other Identifiers** list in the **Rules Language Elements Editor**.

3. Click **OK**. The statement appears in the rate form.

Example

Suspend a process instance of the COLLECT_PROC process.

```
/* Set Process Instance Attributes */
COLLECT_PROC.UID = "123";
COLLECT_PROC.NOTE = "Verify customer status";

/* Suspend the process */
PROCESS SUSPEND COLLECT_PROC;
```

Notes

In the above example, several of the COLLECT_PROC attributes are 'hard-coded' into the rate schedule. In actual practice, this data could also come directly from records in the Oracle Utilities Data Repository, or through list queries.

Process Resume Statement

Purpose

The PROCESS RESUME Statement is used to resume an existing suspended process instance.

Format

PROCESS RESUME statements have this format:

```
PROCESS RESUME <stem_identifier>;
```
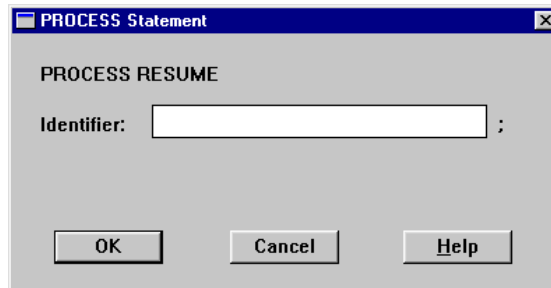
Where:

- <stem_identifier> is a stem that contains the appropriate attributes, as described under **Using the Workflow Management Statements**.

To Create

1. Select **Statements-›Workflow-›Process Resume** from the Rules Language Editor menu bar.

   The PROCESS RESUME Statement template appears.

2. Enter the appropriate stem identifier, or click the *right* mouse button and choose the stem identifier from the **Other Identifiers** list in the **Rules Language Elements Editor**.

3. Click **OK**. The statement appears in the rate form.

Example

Resume a suspended process instance of the COLLECT_PROC process.

```
/* Set Process Instance Attributes */
COLLECT_PROC.UID = "123";
COLLECT_PROC.NOTE = "Verify customer status";

/* Resume the process */
PROCESS RESUME COLLECT_PROC;
```

Notes

In the above example, several of the COLLECT_PROC attributes are 'hard-coded' into the rate schedule. In actual practice, this data could also come directly from records in the Oracle Utilities Data Repository, or through list queries.

Process Terminate Statement

Purpose

The PROCESS TERMINATE Statement is used to terminate an existing process instance.

Format

PROCESS TERMINATE statements have this format:

```
PROCESS TERMINATE <stem_identifier>;
```
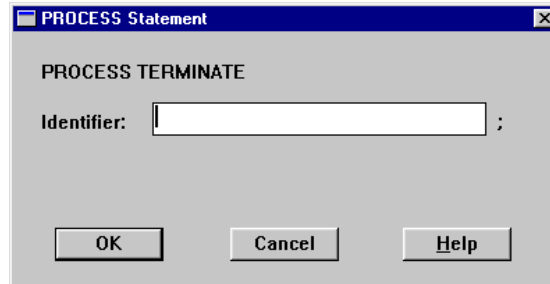
Where:

- <stem_identifier> is a stem that contains the appropriate attributes, as described under **Using the Workflow Management Statements**.

To Create

1. Select **Statements-›Workflow-›Process Terminate** from the Rules Language Editor menu bar.

   The PROCESS TERMINATE Statement template appears.

2. Enter the appropriate stem identifier, or click the *right* mouse button and choose the stem identifier from the **Other Identifiers** list in the **Rules Language Elements Editor**.

3. Click **OK**. The statement appears in the rate form.

Example

Terminate a process instance of the COLLECT_PROC process.

```
/* Set Process Instance Attributes */
COLLECT_PROC.UID = "123";
COLLECT_PROC.NOTE = "Verify customer status";

/* Terminate the process */
PROCESS TERMINATE COLLECT_PROC;
```

Notes

In the above example, several of the COLLECT_PROC attributes are 'hard-coded' into the rate schedule. In actual practice, this data could also come directly from records in the Oracle Utilities Data Repository, or through list queries.

Process Event Statement

Purpose

The PROCESS EVENT Statement posts an activity event.

Format

PROCESS EVENT statements have this format:

```
PROCESS EVENT <stem_identifier>;
```

Where:

- <stem_identifier> is a stem that contains the appropriate attributes, as described under **Using the Workflow Management Statements**.

To Create

1. Select **Statements-›Workflow-›Process Event** from the Rules Language Editor menu bar.

   The PROCESS EVENT Statement template appears.
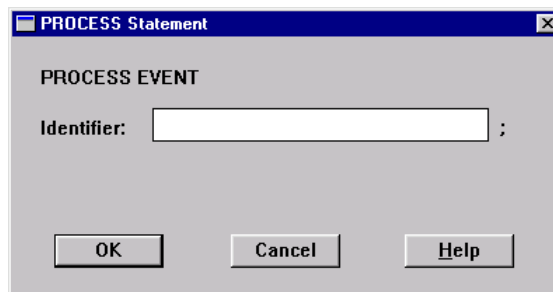
2. Enter the appropriate stem identifier, or click the *right* mouse button and choose the stem identifier from the **Other Identifiers** list in the **Rules Language Elements Editor**.

3. Click **OK**. The statement appears in the rate form.

Example

Post an activity event in the COLLECT_PROC process.

```
/* Set Event Attributes */
COLLECT_PROC.UID = "123";
COLLECT_PROC.NOTE = "Verify customer status";

/* Post the Event */
PROCESS EVENT COLLECT_PROC;
```

Notes

In the above example, several of the COLLECT_PROC attributes are 'hard-coded' into the rate schedule. In actual practice, this data could also come directly from records in the Oracle Utilities Data Repository, or through list queries.

Chapter 9

Interval Data Function Descriptions

This chapter describes all of the interval data functions available with the Oracle Utilities Rules Language, including:

- **Interval Data Functions**

- **Enhanced Interval Data Functions**

  - **INTDDELETEEX Function**

  - **INTDGETATTREXALL Function**

  - **INTDLOADEXACTUAL Function**

  - **INTDLOADEXCUT Function**

  - **INTDLOADEXDATES Function**

  - **INTDLOADEX Function**

  - **INTDLOADEXLIST Function**

  - **INTDLOADEXLISTDATES Function**

  - **INTDLOADEXRELATEDCHANNEL Function**

  - **INTDSAVEEX Function**

  - **INTDSAVEEXP Function**

  - **INTDSETATTREX Function**

  - **INTDSETATTREXALL Function**

  - **INTDVALUEEX Function**

  - **Enhanced Interval Data Functional Differences**

  - **Interval Data Functions and Enhanced Interval Data Handles**

Interval Data Functions

INTDADDATTRIBUTE Function

Purpose

The INTDADDATTRIBUTE Function adds a user-defined attribute to an interval data handle and returns an integer; 0 if successful, not 0 if an error.

This function can be used to add attributes related to EDI transactions. The maximum number of user-defined attributes is 20. These attributes are only visible if the handle is exported as an LSE file, in which case the attributes are listed in records with sort codes 00000030 through 00000039.

Format

```
<identifier> = INTDADDATTRIBUTE(<interval_data_reference>,
<attribute>, <identifier|expression>);
```

Where

- <interval_data_reference> is a reference to a loaded interval data handle. It may be a 'recorder, channel' constant or an interval data handle.

- <attribute> is a user-defined attribute.

- <identifier|expression> is either an identifier, or an expression that sets the values of the attribute. If an identifier, it must have been assigned earlier in the rate form.

Example

Add the "EDI_TRANSACTION" attribute with a value of "997" to the "HNDL_1' interval data handle.

```
HNDL_1 = INTDLOADUOM('01');
EDI_ID = "997"
HNDL_1_ADD_EDI = INTDADDATTRIBUTE(HNDL_1, "EDI_TRANSACTION", EDI_ID);
```

INTDADDVMSG Function

Purpose

The INTDADDVMSG Function adds a validation message to an interval data handle. Up to 10 messages can be added to the handle using this function. If successful, the function returns the index of the array where the message has been stored. If the array already has ten messages, the function fails, and returns an integer (99).

Format

```
<identifier> = INTDADDVMSG(<interval_data_reference>,
<validation_message_text>);
```

Where

- <interval_data_reference> is a reference to a loaded interval data handle. It may be a 'recorder, channel' constant or an interval data handle.

- <validation_message_text> is a string (or an identifier that contains a string) that is text for the validation message to be added to the handle.

Examples

Add the "Missing (Status Code 9) values found" message to "HNDL_1".

```
HNDL_1 = INTDLOADUOM('01');
HNDL_1_ADD_VAL_MSG = INTDADDVMSG(HNDL_1,"Missing (Status Code 9)
values found");
```

or

```
HNDL_1 = INTDLOADUOM('01');
VAL_MSG = "Missing (Status Code 9) values found";
HNDL_1_ADD_VAL_MSG = INTDADDVMSG(HNDL_1, VAL_MSG);
```

INTDBLOCKOP Function

Purpose

The INTDBLOCKOP Function enables you to perform operations on one interval data reference (add, subtract, etc.) using the values in another. The intervals in the first handle are operated on by the corresponding (same date and time) intervals in the second handle. If the first reference is zero it is an error. If the second reference is zero, the INTDSCALAROP function is called using the same operation and a value of 0.0. If the second handle contains intervals beyond the stop time of the first handle, those intervals are appended to the end of the resulting handle. Returns an interval data reference.

Format

```
<interval_data_reference> = INTDBLOCKOP(<interval_data_reference>,
<operation>, <interval_data_reference>);
```

Where

- <interval_data_reference> is a reference to a loaded interval data handle. It may be a 'recorder, channel' constant or an interval data handle.

- <operation> is one of the following:

    - **ATAN2**: Performs the **ATAN2 Function** using values in the first handle and corresponding values in the second handle.

    - **DIVQUOT**: Performs the **DIVQUOT Function** using values in the first handle and corresponding values in the second handle.

    - **DIVREM**: Performs the **DIVREM Function** using values in the first handle and corresponding values in the second handle.

    - **FMOD**: Performs the **FMOD Function** using values in the first handle and corresponding values in the second handle.

    - **POW**: Performs the **POW Function** using values in the first handle and corresponding values in the second handle.

    - **TOTAL**: Adds the value in the first handle to the corresponding value in the second handle.

    - **ADD**: Adds the value in the first handle to the corresponding value in the second handle.

    - **SUBTRACT**: Subtracts the value in the second handle from the value in the first.

    - **MULTIPLY**: Multiplies the value in the first handle by the value in the second.

    - **DIVIDE_BY**: Divides the value in the first handle by the corresponding value in the second.

    - **MAXIMUM**: Finds the maximum of the two corresponding interval values.

    - **MINIMUM**: Finds the minimum of the two corresponding interval values.

    - **MINNZ**: Finds the nonzero minimum of the two corresponding interval values.

    - **KVA**: Calculates KVA; one interval data reference must have a UOM of KW, the other of KVAR.

    - **IKVA**: Calculates IKVA (inverse KVA); the first handle must have a UOM of KVA and the second a UOM of either KVAR or KW. The resulting handle has the appropriate UOM.

    - **KVAH**: Calculates KVAH; one interval data reference must have a UOM of KW, the other of KVAR.

- **IKVAH**: Calculates IKVAH (inverse KVAH); the first handle must have a UOM of KVAH and the second a UOM of either RKVA or KW. The resulting handle has the appropriate UOM.

- **COMPVARHFROMKWKQ**: Performs the **COMPKVARHFROMKQKW Function** using values in the first handle and corresponding values in the second handle.

- **POWERFACTOR**: Performs the **POWERFACTOR Function** using values in the first handle and corresponding values in the second handle.

Example

Subtract the interval data values in a handle that measures energy use from those in a handle that measures energy generation.

```
KWH_HNDL = INTDLOADDATES ('ENERGY,1', BILL_START, BILL_STOP);
GEN_HNDL = INTDLOADDATES ('GEN,1", BILL_START, BILL_STOP);
BOUGHT_HNDL = INTDBLOCKOP(GEN_HNDL, "SUBTRACT", KWH_HNDL);
```

A more direct way to perform some of the block operations (specifically add, subtract, multiply, and divide) is to use an arithmetic expression on the right side of the equal sign in an ASSIGNMENT Statement. For example:

```
BOUGHT_HNDL = GEN_HNDL - KWH_HNDL;
```

INTDBLOCKOPNA Function

Purpose

The INTDBLOCKOPNA Function is similar to the **INTDBLOCKOP Function**, but does not require the intervals be aligned (i.e. have the same date and time).

This function enables you to perform operations on one interval data reference (add, subtract, etc.) using the values in another. The intervals in the first handle are operated on by the corresponding (same index, not time-aligned) intervals in the second handle. For example, the first interval in the first handle is operated on by the first interval in the second handle, the second interval in the first handle is operated on by the second interval in the second handle, etc.

Format

```
<interval_data_reference> = INTDBLOCKOPNA(<interval_data_reference>,
<operation>, <interval_data_reference>);
```

Where

- <interval_data_reference> is a reference to a loaded interval data handle. It may be a 'recorder, channel' constant or an interval data handle.

- <operation> is one of the following:

    - **ATAN2**: Performs the **ATAN2 Function** using values in the first handle and corresponding values in the second handle.

    - **DIVQUOT**: Performs the **DIVQUOT Function** using values in the first handle and corresponding values in the second handle.

    - **DIVREM**: Performs the **DIVREM Function** using values in the first handle and corresponding values in the second handle.

    - **FMOD**: Performs the **FMOD Function** using values in the first handle and corresponding values in the second handle.

    - **POW**: Performs the **POW Function** using values in the first handle and corresponding values in the second handle.

    - **TOTAL**: Adds the value in the first handle to the corresponding value in the second handle.

    - **ADD**: Adds the value in the first handle to the corresponding value in the second handle.

    - **SUBTRACT**: Subtracts the value in the second handle from the value in the first.

    - **MULTIPLY**: Multiplies the value in the first handle by the value in the second.

    - **DIVIDE_BY**: Divides the value in the first handle by the corresponding value in the second.

    - **MAXIMUM**: Finds the maximum of the two corresponding interval values.

    - **MINIMUM**: Finds the minimum of the two corresponding interval values.

    - **MINNZ**: Finds the nonzero minimum of the two corresponding interval values.

    - **KVA**: Calculates KVA; one interval data reference must have a UOM of KW, the other of RKVA.

    - **IKVA**: Calculates IKVA (inverse KVA); the first handle must have a UOM of KVA and the second a UOM of either RKVA or KW. The resulting handle has the appropriate UOM.

    - **KVAH**: Calculates KVAH; one interval data reference must have a UOM of KW, the other of RKVA.

- **IKVAH**: Calculates IKVAH (inverse KVAH); the first handle must have a UOM of KVAH and the second a UOM of either RKVA or KW. The resulting handle has the appropriate UOM.

- **COMPVARHFROMKWKQ**: Performs the **COMPKVARHFROMKQKW Function** using values in the first handle and corresponding values in the second handle.

- **POWERFACTOR**: Performs the **POWERFACTOR Function** using values in the first handle and corresponding values in the second handle.

Example

Add the interval data values from a baseline handle (channel 'BASELINE,1') to the values in a handle that measures energy for the current bill period.

```
BASE_HNDL = INTDLOAD ('BASELINE,1');
KWH_HNDL = INTDLOADDATES (KWH, BILL_START, BILL_STOP)
TOTAL_HNDL = INTDBLOCKOPNA (BASE_HNDL "ADD," KWH_HNDL);
```

INTDCLOSE Function

Purpose

The INTDCLOSE Function closes an Interval Data Database that was previously opened using the **INTDOPEN Function** on page 9-51. Returns 0.

Format

```
<identifier> = INTDCLOSE(<interval_data_source_index>);
```

Where

- <interval_data_source_index> is an index from a previously loaded interval data file. The parameter must be the result of an INTDOPEN function call.

Example

Close a previously opened interval data file (INTD_FILE).

```
INTD_FILE = INTDOPEN("C:\LODESTAR\USER\MYDATA.LSE");
INTD_COUNT = INTDRECCOUNT(INTD_FILE);
INTD_FILE = INTDCLOSE(INTD_FILE);
```

INTDCOUNT Function

Purpose

The INTDCOUNT Function returns the count of intervals that are not missing, are missing, or the combined number. Also counts the number of hours or days in the handle. This function enables you to get a count of interval data values in a handle (or mask) that you've previously loaded or calculated in the rate form.

Format

```
<identifier> = INTDCOUNT(<interval_data_reference>[, <type>]);
```

Where

- <interval_data_reference> is a handle that refers to a loaded interval data handle, or a 'recorder,channel' constant.

- <type> *(Optional)* is one of the following:

  - **INCLUDE**: Counts the number of intervals with any status code other than '9' (missing).

  - **EXCLUDE**: Counts the number of intervals with missing values (status code '9').

  - **NON_ZERO**: Counts the number of intervals with a nonzero value.

  - **ALL**: Counts all intervals, regardless of status code or value. This is the default.

  - **HOURS**: Counts the number of hours in which in the interval data handle (or mask) is not 0. To do this, the program counts the number of nonzero intervals, then divides that result by the IPH (intervals-per-hour) to get the number of hours. The number of hours is incremented by 1 if the remainder is 2 or more. **Note**: This parameter should not be used with handles of 1 day intervals.

  - **DAYS**: Same as HOURS, except the number of hours is divided by 24 to get the number of days. The number of days is incremented if the remainder is 12 or more. **Note**: This parameter should not be used with handles of 1 day intervals.

Example

Count the number of missing intervals in the handle KW_HNDL. If the result is greater than 10, include a warning message, "TOO MANY MISSING INTERVALS", on the bill report.

```
NUM_MISS = INTDCOUNT(KW_HNDL, "EXCLUDE");
IF NUM_MISS > 10 THEN
    WARN "TOO MANY MISSING INTERVALS";
END IF;
```

INTDCOUNTSTATUSCODE Function

Purpose

The INTDCOUNTSTATUSCODE Function enables you to get a count of interval data values in a handle (or mask) that you've previously loaded or calculated in the rate form based on a comparison of the intervals' status codes to a supplied status code.

Format

```
<identifier> = INTDCOUNTSTATUSCODE(<interval_data_reference>,
<comparison>, <status_code>);
```

Where

- <interval_data_reference> is a handle that refers to a loaded interval data handle, or a 'recorder,channel' constant.

- <comparison> is one of the following.

  - "=" - Equal

  - "<>" - Not equal

  - "<" - Less than

  - ">" - Greater than

  - "<=" - Less than or equal

  - ">=" - Greater than or equal

  - "IN" - IN the status code string

  - "NOT IN" - NOT IN the status code string.

  The comparison order is (from highest to lowest): (space) A B C...Z 0 1 2...8 9.

- <status_code> is an identifier or string constant that specifies a valid status code, or, for "IN" and "NOT IN", a string of status codes (no separator: "ABC" for codes A, B, and C).

Example

Count the number of intervals in the handle KW_HNDL with a status code of 1.

NUM_STS_CODE_1 = INTDCOUNTSTATUSCODE(KW_HNDL, "=", "1");

INTDCREATEMASK Functions

An interval data mask is a handle whose values are all 0 or 1, as opposed to a data handle, which can have any value. A 0 value means the interval is excluded from the handle; a 1 means it is included. A mask can be used to remove values from a data handle, or it can be combined with another mask to include or exclude additional intervals. A mask can be created by the functions below, by an interval data function that operates on an existing mask, or by a divide operation that creates a handle with all 0s and 1s.

Interval Data Mask Functions

Interval data mask functions include:

* **INTDCREATEDAYMASK Function** on page 9-12

* **INTDCREATEFACTORMASK Function** on page 9-13

* **INTDCREATEMASK Function** on page 9-15

* **INTDCREATEOVERRIDEDAYMASK Function** on page 9-16

* **INTDCREATEOVERRIDEMASK Function** on page 9-17

* **INTDCREATESTATUSCODEMASK Function** on page 9-18

* **INTDCREATETOUPERIOD Function** on page 9-19

These functions return a mask if:

1. The operation is one of "MASK", "REVERSE_MASK", "ZERO", "NON_ZERO" or "MISSING", or

2. The input interval data reference is a mask and the operation is one of the above, plus "VALUE" and "REVERSE_VALUE".

Otherwise, an interval data handle is returned.

For more information about interval data mask operations, see **Interval Data Mask Operator Rules** on page 4-28 in the *Oracle Utilities Rules Language User's Guide* and **Computing Load Factor in Masked Cuts** on page 4-29 in the *Oracle Utilities Rules Language User's Guide*.

INTDCREATEDAYMASK Function

Purpose

The INTDCREATEDAYMASK Function sets all values within each 24-hour period (midnight to midnight) to either 1 or 0. It is the same as the **INTDCREATEMASK Function** on page 9-15, except that all values in each day are set to the same value.

Note: These masks can be used in any interval data function or expression.

Format

```
<interval_data_reference> =
INTDCREATEDAYMASK(<interval_data_reference>, <operation>);
```

Where

- <interval_data_reference> is a reference to a loaded interval data handle, or a recorder,channel constant.

- <operation> is one of "ZERO", "NON_ZERO", or "MISSING" (default is "ZERO").

  ZERO: If there are any zero interval values in a day, all values in that day's mask are set to 1. If there are no zero values in a day, all values in that day's mask are set to 0.

  NON_ZERO: If any interval values in a day are nonzero, all values in that day's mask are set to 1. If all values in a day are zero (0), all values in that day's mask are set to 0.

  MISSING: If any interval status code in a day is equal to 9, all values in that day's mask are set to 1. If no status code in a day are equal to 9, all values in that day's mask are set to 0.

Example

To create a mask of backup days, where a day is a backup day if at any time during the day the co-generation drops to 0:

```
BACKUPMASK_HNDL = INTDCREATEDAYMASK(GEN_HNDL, "ZERO");
```

If this is a representation of the values in the referenced handle in the example above:

GEN_HNDL	11	5	12	15	6	0	4	5	11	0	8	9
	Day 1			Day 2			Day 3			Day 4...		

Then this is a representation of the values in the mask that would be created:

BACKUPMASK_HNDL	0	0	0	1	1	1	0	0	0	1	1	1
	Day 1			Day 2			Day 3			Day 4...		

INTDCREATEFACTORMASK Function

Purpose

The INTDCREATEFACTORMASK Function sets the interval values to the factor value and the remaining values to 0 for all intervals in which a selected factor is applied.

This function creates an interval data handle that matches the referenced handle (same IPH, UOM, and number of intervals). It reads the FACTORVALUE Table for the specified factor code and determines the date ranges that have different factor values (there could be one date range with only one value, or none if no factor value is in effect over the handle). Intervals in the new handle that fall within these date ranges are set to the corresponding factor values.

Format

```
<interval_data_reference> =
INTDCREATEFACTORMASK(<interval_data_reference>,
<factor_code|factor_identifier>);
```

Where

- <interval_data_reference> is a handle that refers to a loaded interval data handle, or a 'recorder,channel' constant.

- <factor_code | factor_ identifier> is a string constant or identifier that specifies a factor. See the ACCOUNTFACTOR function description for information about specifying factors.

Example

Create a mask of factor values and multiply it by the KWH handle:

```
TEMP_MASK_HNDL = INTDCREATEFACTORMASK(KWH_HNDL, "ENERGY_CHG");

/* Multiply the Energy Charge factor times the KWH handle */
ENERGY_CHG_HNDL = TEMP_MASK_HNDL * KWH_HNDL;
```

INTDCREATEHANDLE Function

Purpose

The INTDCREATEHANDLE Function creates an interval data handle based on user-specified parameters. The interval values of the handle are either "missing values" (status code '9'), or all equal to a specified value and status code.

Once a handle has been created using this function, the handle can be manipulated via other interval data functions, such as setting of attributes for the handle, such as Recorder ID, Channel Number, etc. using the **INTDSETATTRIBUTE Function**.

Format

```
<identifier> = INTDCREATEHANDLE(<start>, <stop>, <SPI>, <DST>,
<VALUE>, <STATUS>);
```

Where

- <start> the start date and time for the handle. Can be either a datetime constant or an identifier that resolves to a datetime.

- <stop> the start date and time for the handle. Can be either a datetime constant or an identifier that resolves to a datetime.

- <SPI> is the Seconds-per-Interval value for the handle.

- <DST> is a string that is the DST Participant flag ("Y" or "N") for the handle.

- <VALUE> is an integer that is interval value for each interval in the handle.

- <STATUS> is a string that is the Status Code to assign to each interval value in the handle. To assign a blank status code, you must specify quote-space-quote (" ") for this parameter.

Examples

Create an empty interval data handle (values of 0) of hourly data for 5/1/2006, with a DST Participant flag of "Y".

```
EMPTY_HNDL = INTDCREATEHANDLE('05/01/2006 00:00:00', '05/01/2006
23:59:59', 3600, "Y", "0", " ");
```

Create an interval data handle of hourly data for 5/1/2006 with a DST Participant Flag of "Y" and interval values equal to 5.

```
EMPTY_HNDL = INTDCREATEHANDLE('05/01/2006 00:00:00', '05/01/2006
23:59:59', 3600, "Y", "5", " ");
```

Create an empty interval data handle (values of 0) of hourly data for 5/3/2006, and set the Recorder ID for the handle equal to "RECORDER_1."

```
EMPTY_HNDL = INTDCREATEHANDLE('05/03/2006 00:00:00', '05/03/2006
23:59:59', 3600, "Y", "0", " ");
```

```
SET_RECORDER_ID = INTDSETATTRIBUTE (EMPTY_HNDL, RECORDER,
"RECORDER_1");
```

INTDCREATEMASK Function

Purpose

The INTDCREATEMASK Function creates an interval data handle that matches the referenced handle (same IPH, UOM, and number of intervals). Its values are 0 or 1, depending on the zero or nonzero value in the referenced handle. The new handle can be used as a "mask" handle, as described under **Interval Data Mask Operator Rules** on page 4-28 in the *Oracle Utilities Rules Language User's Guide.* Returns an interval data reference.

> **Note:** Masks created using this function can be used in any interval data function or expression.

Format

```
<interval_data_reference> = INTDCREATEMASK(<interval_data_reference>,
<operation>);
```

Where

- <interval_data_reference> is a handle for a loaded interval data handle, or a 'recorder,channel' constant.

- <operation> is one of the following:

    - **ZERO**: Sets the interval value to 1 if matching input value is 0; sets all other values to 0. This is the default.

    - **NON_ZERO**: Sets the interval value to 0 if matching input value is 0; sets all other pulse values to 1.

    - **MISSING**: Sets the interval value to one if matching input status code is 9, sets all other interval values to zero. All status codes are set to '' (space).

Example

Create a mask for a handle where the interval value is 1 if the referenced handle value is 0:

```
MASK_HNDL = INTDCREATEMASK(KWH_HNDL, "ZERO");
```

If this is a representation of the values in the referenced handle in the example above:

```
KWH_HNDL1020    15 17 0  4  10 0  9 ...
```

Then this is a representation of the values in the mask that would be created:

```
MASK_HNDL00     0  0  1  0  0  1  0 ...
```

INTDCREATEOVERRIDEDAYMASK Function

Purpose

The INTDCREATEOVERRIDEDAYMASK Function creates an interval data mask for overrides expanded to a full day.

This is the same as the **INTDCREATEOVERRIDEMASK Function** on page 9-17, except that each date "range" is expanded to start and end at midnight (or the day start). It reads the ACCTOVERRIDEHIST Table or the ACCTNAMEOVERRIDEHIST Table for the current account and specified override code, and determines the date ranges where the override code is "On". Intervals in the new handle that fall within these date ranges are set to the values specified by the operation.

If the operation is "MASK" or "REVERSE_MASK", the new handle will be used as a "mask" handle, as described in the **Interval Data Mask Operator Rules** on page 4-28 in the *Oracle Utilities Rules Language User's Guide*.

Format

```
<interval_data_reference> =
INTDCREATEOVERRIDEDAYMASK(<interval_data_reference>, <override_code>,
<operation>);
```

Where

- <interval_data_reference> is a handle that refers to a loaded interval data handle, or a 'recorder,channel' constant.

- <override_code> is a string constant that specifies a code in the OVERRIDE database table. If the override is in the ACCTNAMEOVERRIDEHIST Table, this should be in the form of "OVERRIDE,NAME."

- <operation> is one of the following:

    - MASK: Set values in a date range to 1, values not in a date range to 0.

    - REVERSE_MASK: Set values *not* in a date range to 1, others to 0.

    - VALUE: Set values in a date range to the same as the corresponding values in the specified handle, others to 0.

    - REVERSE_VALUE: Set values *not* in a date range same as in the specified handle, others to 0.

    - OVERRIDE_VALUE: Set values in a date range to the first VAL value in the Override Table, others to 0. This is the default.

Example

Create a mask of backup days, where a day is a backup day if at any time during the day the co-generation drops to 0.

```
BACKUP_HNDL = INTDCREATEOVERRIDEDAYMASK(KWH_HNDL,"BACKUP", "VALUE");
```

INTDCREATEOVERRIDEMASK Function

Purpose

The INTDCREATEOVERRIDEMASK Function creates an interval data mask for overrides (special events).

This function creates an interval data handle that matches the referenced handle (same IPH, UOM, and number of intervals). It reads the ACCTOVERRIDEHIST or the ACCTNAMEOVERRIDEHIST Table for the current account and specified override code, and determines the date ranges where the override code is "On". Intervals in the new handle that fall within these date ranges are set to the values specified by the operation.

If the operation is "MASK" or "REVERSE_MASK", the new handle will be used as a "mask" handle, as described in the **Interval Data Mask Operator Rules** on page 4-28 in the *Oracle Utilities Rules Language User's Guide.*

Format

```
<interval_data_reference> =
INTDCREATEOVERRIDEMASK(<interval_data_reference>, <override_code>,
<operation>);
```

Where

- <interval_data_reference> is a handle that refers to a loaded interval data handle, or a 'recorder,channel' constant.

- <override_code> is a string constant that specifies a code in the OVERRIDE database table. If the override is in the ACCTNAMEOVERRIDEHIST Table, this should be in the form of "OVERRIDE,NAME."

- <operation> is one of the following:

  - MASK: Set values in a date range to 1, values not in a date range to 0.

  - REVERSE_MASK: Set values *not* in a date range to 1, others to 0.

  - VALUE: Set values in a date range to the same as the corresponding values in the specified handle, others to 0.

  - REVERSE_VALUE: Set values *not* in a date range same as in the specified handle, others to 0.

  - OVERRIDE_VALUE: Set values in a date range to the first VAL value in the Override Table, others to 0. This is the default.

Example

Create a handle of all curtailable override values:

```
OVR_MASK_HNDL = INTDCREATEOVERRIDEMASK(KWH_HNDL, "CURTIAL",
"OVERRIDE_VALUE");
```

INTDCREATESTATUSCODEMASK Function

Purpose

The INTDCREATESTATUSCODEMASK Function creates an interval data handle of 1 and 0 (true and false) as the result of status code comparisons.

The function creates an interval data handle that matches the referenced handle (same IPH, UOM, and number of intervals). The value of intervals in the new handle is determined by the success of a comparison of the input intervals status code to a supplied status code, and by the operation.

Format

```
<interval_data_reference> =
INTDCREATESTATUSCODEMASK(<interval_data_reference>, <comparison>,
<status_code>, <operation>, <result_status_code>);
```

Where

- <interval_data_reference> is a reference to a loaded interval data handle, or a recorder,channel constant.

- <comparison> is one of the following:

  - "=" - Equal

  - "<>" - Not equal

  - "<" - Less than

  - ">" - Greater than

  - "<=" - Less than or equal

  - ">=" - Greater than or equal

  - "IN" - IN the status code string

  - "NOT IN" - NOT IN the status code string.

  The comparison order is (from highest to lowest): (space) A B C...Z 0 1 2...8 9.

- <status_code> is an identifier or string constant that specifies a valid status code, or, for "IN" and "NOT IN", a string of status codes (no separator: "ABC" for codes A, B, and C).

- <operation> is one of the following:

  - MASK: Set true values to 1, false values to 0.

  - REVERSE_MASK: Set false values to 1, others to 0.

  - VALUE: Set true values same as in <interval_data_reference>, others to 0. This is the default.

  - REVERSE_VALUE: Set false values same as in <interval_data_reference>, others to 0.

- <result_status_code> is an optional identifier or string constant that specifies a valid status code, or "@". It is assigned as the status code of any value set to 0. If it is "@", the original status code is maintained. The default is a space.

Example

Create a mask that is 1 where status code is A or B, and 0 (zero) elsewhere.

```
HNDL3 = INTDCREATESTATUSCODEMASK (HNDL1, "IN," "AB," "MASK");
```

INTDCREATETOUPERIOD Function

Purpose

The INTDCREATETOUPERIOD Function creates a handle in which the interval values that fall within a user-specified Time-Of-Use period are distinguished from those that fall outside the period.

The interval values that are in the period may be set to 1 (and all others to 0) or to their actual value (and all others to 0). You can also specify the reverse. The resulting "mask" handle can be used in TOU computations.

To get average weekly values or other values for a Time-of-Use (TOU) period, the interval data must be masked first so only values in the period are used.

This function creates an interval data handle that matches the referenced handle (same IPH, UOM, and number of intervals). It uses the TOU schedule and period specified to determine the time and day ranges when the period is in effect. Intervals in the new handle that fall within these time and day ranges are set to the values specified by the operation.

> *Note:* TOU schedules and periods are set up in Data Manager. See the *Data Manager User's Guide* for more information.

If the operation is "MASK" or "REVERSE_MASK", the new handle will be used as a "mask" handle, as described in the **Interval Data Mask Operator Rules** on page 4-28 in the *Oracle Utilities Rules Language User's Guide.*

Format

```
<interval_data_reference> =
INTDCREATETOUPERIOD(<interval_data_reference>, <operation>,
<schedule_name>, <period>[, <holiday_list_name>]);
```

Where

- <interval_data_reference> is a reference to a loaded interval data handle, or a recorder,channel constant.

- <operation> is one of the following:

    - MASK: Set values in a TOU period to 1, values not in a date range to 0.

    - REVERSE_MASK: Set values not in a TOU period to 1, others to 0.

    - VALUE: Set values in a TOU period to the same as the corresponding values in the specified handle, others to 0.

    - REVERSE_VALUE: Set values *not* in a TOU period to the same as in the specified handle, others to 0.

- <schedule_name> is the name of a Time-of-Use schedule stored in the Data Repository.

- <period> is the name of a period in the selected Time-of-Use schedule.

- <holiday_list_name> *Optional.* A holiday list to be used with the TOU schedule. Default is the default holiday schedule for the rate schedule's operating company, jurisdiction, which is opcocode|juriscode (e.g., GECO|MA).

Examples

The following pairs of statements represent two ways of accomplishing the same result; for any operation supported by both functions, you would supply an operation for the placeholder operation:

```
TOU_MASK_HNDL = INTDCREATETOUPERIOD(KWH_HNDL, "VALUE", TOU_SCHED,
TOU_PERIOD);
TOU_MASK_VALUE = INTDVALUE(TOU_MASK_HNDL, operation);
```

and

```
TOU_HNDL = INTDTOU(KWH_HNDL, TOU_SCHED);
TOU_VALUE = INTDTOUVALUE(TOU_HNDL, TOU_PERIOD, operation);
```

The second approach is faster if you want several different period values.

INTDDELETE Function

Purpose

This INTDDELETE Function deletes one or more cuts from the database. It has two options. You can delete a cut from the database with a specific recorder,channel and start date/time, or delete all cuts from the database with the specified recorder and channel whose start date/time and stop date/time fall within a specified date range.

Formats

To delete the cut from the database with this recorder,channel and start date/time, use.

```
<identifier> = INTDDELETE(<recorder>, <channel>, <cut_start>);
```

To delete all cuts from the database with this recorder and channel whose start date/time and stop date/time fall within the date range, use the following. Cuts with a start or stop date outside the range are not deleted.

```
<identifier> = INTDDELETE(<recorder>, <channel>, <start_time>,
<stop_time>);
```

Where

- <recorder> is the recorder.
- <channel> is the channel.
- <cut_start> is the start time and date of the cut.
- <start_time> is the start time of the range of cuts to delete.
- <stop_time> is the stop time of the range of cuts to delete.

Example

Delete all cuts from channel '1700,1' that have a start date equal to the BILL_START date and a stop date equal to the BILL_STOP date.

```
DELETED_CUTS = INTDDELETE("1700", "1", BILL_START, BILL_STOP);
```

INTDDIPTEST Function

Purpose

This INTDDIPTEST Function examines the interval data for dips as defined by the two parameters N and P. A dip is defined as any interval that exceeds a percent (P) or greater than the rolling average of the N preceding intervals in the handle. Validation is NOT performed for all interval values preceding the Nth interval in the handle. Negative values for intervals are always treated as valid dips if they are in fact mathematical dips in the stream of data. This function returns a stem component variable that is the name of the variable in the Assignment statement (STEM in the example below). If no dips are found in the handle of interval data, the DIPCOUNT value will be set to zero. The stem will be set to "" if the function was successful, else it will be set to the integer zero. The component variables will include:

- STEM.DIPCOUNT contains an integer count of the number of dips found, up to 500.

- STEM.DIP1 contains the index of the first interval defined as a dip.

- STEM.DIP2 contains the index of the second interval defined as a dip.

- STEM.DIPn contains the index of the nth interval defined as a dip.

Format

```
<stem> = INTDDIPTEST(<interval_data_reference>, <N>, <P>,
<status_code>);
```

Where

- <interval_data_reference> is a reference to the loaded interval data handle to be tested

- <N> is number of preceding intervals used to calculate a rolling average

- <P> is the percent lower than the rolling average of the preceding intervals that any interval must be to be considered a dip

- <status_code> *Optional* is the status code that all intervals must be above or better than to be included in the validation. If not supplied, the status code will default to "9".

Example

Identify the intervals in CUT_HNDL that are more than 40 percent lower than the rolling average of the preceding 5 intervals, and that have a Status Code of "L" or better:

```
DIPTEST = INTDDIPTEST(CUT_HNDL, 5, 40, "L" );
```

Result :

```
DIPTEST.DIPCOUNT = 2
DIPTEST.DIP1 = 45
DIPTEST.DIP2 = 285
```

INTDEXPORT Function

Purpose

This INTDEXPORT Function outputs the data in the handle to a file. All file types are opened in "append" mode, so they are always added to (or created if they do not exist). If no path is specified in the file name, the file is created in the LODESTAR\USER directory. This function always returns the integer 0. This function is primarily for testing; it should not be used in actual billing rate schedules.

Format

```
<identifier> = INTDEXPORT(<interval_data_reference>, <file_name>
[, <export_flag>]);
```

Where

- <interval_data_reference> is a reference to a loaded interval data handle, or a recorder,channel constant.

- <file_name> is a string constant that specifies a file name, optionally including its path. The file name **must** include the file extension, and can be one of the following:

  - LSE - exports the loaded interval data in Oracle Utilities Enhanced Input/Output format.

  - CSV - exports the loaded interval data in Oracle Utilities CSV Interval Data format.

  - XML - exports the loaded interval data in Oracle Utilities XML Interval Data format.

- <export_flag> is a flag that specifies how to set the Validate_Record_Flag on the exported handle. Can be one of the following:

  - WV - Force Valid: Forces the Validate_Record_Flag on each exported handle to be set to "N".

  - WI - Force Invalid: Forces the Validate_Record_Flag on each exported handle to be set to "Y" (Default).

  - WU - Use Flag: Uses the Internal_Validation flag and the Merge flag in the stored handle to determine how to set the Validate_Record_Flag when exporting. If either the Merge flag or the Internal_Validation flag are "Y", then the Validate_Record_Flag in the exported handle is set to "N". If both are set to "N" then the Validate_Record_Flag is set to "Y".

Example

Export the handle to a file in the USER directory called MYFILE.LSE, and set the Validate_Record_Flag on the handle to "N":

```
MY_FILE = INTDEXPORT(KWH_HNDL, "MYFILE.LSE", WV);
```

Notes

This function executes in all modes, and is **not** disabled if saves are disabled. Also, if used with Oracle Utilities Billing Component, this function executes when the rate schedule is processed, and even if the bill report is rejected.

To override this default behavior, you can use an IF THEN statement and the LSRSENV.COMMIT Rate Schedule Environment identifier to make sure the rate schedule being processed is in "commit" mode (that is, saves are enabled), as follows:

Export interval data if in "commit" mode.

```
//Verify "commit" mode
IF LSRSENV.COMMIT = 1
   THEN
   //Export data
   MY_FILE = INTDEXPORT(KWH_HNDL, "MYFILE.LSE", WV);
END IF;
```

INTDGETERRORCODE Function

Purpose

The INTDGETERRORCODE Function returns the error code from the last interval data function call. Returns 0 if there was no error.

Format

```
<identifier> = INTDGETERRORCODE();
```

Example

Get the error code for the last function performed on a previously opened interval data file (INTD_FILE).

```
INTD_FILE = INTDOPEN("C:\LODESTAR\USER\MYDATA.LSE");
INTD_COUNT = INTDRECCOUNT(INTD_FILE);
INTD_FILE = INTDGETERRORCODE();
```

INTDGETERRORMESSAGE Function

Purpose

The INTDGETERRORMESSAGE Function returns an error message from the last function to use a specified interval data reference, or the last function that used the default Interval Data Database, if there is no parameter. Returns "" if there was no error.

Format

```
<identifier> = INTDGETERRORMESSAGE([<interval_data_source_index>]);
```

Where

- <interval_data_source_index> *(Optional)* is an index from a previously loaded interval data file. If present, this must be the result of an INTDOPEN function call.

Example

Get an error message for the last function performed on a previously opened interval data file (INTD_FILE).

```
INTD_FILE = INTDOPEN("C:\LODESTAR\USER\MYDATA.LSE");
INTD_COUNT = INTDRECCOUNT(INTD_FILE);
INTD_FILE = INTDGETERRORMESSAGE(INTD_FILE);
```

INTDISEQUAL Function

Purpose

Compares two handles.

The INTDISEQUAL Function compares a first handle with a second handle. The two handles will be considered equal if their start times, stop times, SPI, UOM, start and stop readings, pulse and meter multiplier and offsets, and corresponding intervals (values and status codes) match. Returns 0 if the two handles are not equal, nonzero if the two handles are equal.

Format

```
<identifier> = INTDISEQUAL(<interval_data_reference>,
<interval_data_reference>);
```

Where

- <interval_data_reference> is a reference to a loaded interval data handle, or a recorder,channel constant.

Example

Determine if two interval data handles (INT_HNDL_ONE and INT_HNDL_TWO) are equal.

```
INT_HNDL_ONE = INTDLOAD(KW);
INT_HNDL_TWO = INTDLOAD('1700,1');
EQUAL_HNDLS = INTDISEQUAL(INT_HNDL_ONE, INT_HNDL_TWO);
```

INTDJOIN Function

Purpose

The INTDJOIN Function merges two interval data handles into one based on user-specified criteria. The criteria by which the handles are merged can be selected from the **Tools‑›Options‑›Interval Data Merge Options** tab. See **Default Options** on page 2-9 in the *Data Manager User's Guide* for more information.

Format

```
<identifier> = INTDJOIN(<interval_data_reference>,
<interval_data_reference>, [<interval_data_merge criterion>], [<CHECK/
NOCHECK INTD VALIDATION>]);
```

Where

- <interval_data_reference> is a reference to a previously loaded interval data handle.

  Note: Both handles must have the same UOM and SPI in order to be joined using this function.

- <interval_data_merge_criterion> (*Optional)* specifies how overlapping interval handles are merged. Must be one of:

    - MOST RECENT START: Use the interval from the handle with the later start time (default). If the handles have the same start time, the interval values are averaged.

    - BEST STATUS: use the interval from the handle with a better status code. If the intervals have the same status code, the values are averaged.

    - AVERAGE VALUE: use the time-weighted average of the two overlapped intervals.

    - MAXIMUM VALUE: use the interval from the handle with a larger value.

- <CHECK/NOCHECK_INTD_VALIDATION> (*Optional)* is a flag ('Y' or 'N') that if set to 'Y' checks the handles' Merge validation flag. The default is 'Y'.

Example

Merge the HNDL_ONE and HNDL_TWO interval data handles.

```
HDNL_ONE = INTDLOAD ('1700,1')
HDNL_TWO = INTDLOAD ('1700,2')
HNDL_MERGE = INTDJOIN(HNDL_ONE, HNDL_TWO);
```

INTDLOAD Functions

The following general information applies to all INTDLOADxxxx functions.

Summary values loaded with the interval data (stem.component)

Along with the interval data, the program automatically computes a group of summary values about the handle. These are the result of adding, averaging, or taking the maximum of the interval values in the handle. This data is stored in memory until the program determines that the rate form no longer needs it, or until you explicitly release it using the **INTDRELEASE Function**. In the rate form, you can apply statements to this group of values by identifying them with the convention <stem.component>, where *stem* is the interval data handle that you assigned in the INDTLOAD Statement (which automatically refers to the entire group), and *component* is the name of a particular component in the group. For example, one of the computed summary values is AVERAGE, which is the average of all non-missing values in the interval data record. If you used the handle INT_MY_HNDL as shown above, you could print the average value in a bill report using a Label Statement:

```
LABEL INT_MY_HNDL.AVERAGE;
```

Following is a list of values that are automatically loaded whenever you load an interval data handle using one of the INTLOADxxx functions. They are the result of adding, averaging, or taking the maximum of the interval values.

Value	Description
TOTAL	The sum of all interval values in the interval data handle.
ENERGY	Total energy represented by the handle, computed properly for its UOM according to the TOTAL flag in the UOM Table. UOM for the interval values must be either KW or KWH. If not, result is 0.
AVERAGE	Average of all non-missing interval values.
AVERAGE_NZ	Average of all nonzero interval values.
MAXIMUM	Peak value (computed using actual values, not the absolute value of the values).
MAXIMUMn	Value of *n*th peak (e.g., MAXIMUM2 reports second highest peak. For *n*, you may supply any value from 2 through 10).
KW_MAXIMUM	The maximum KW value in the handle. If the UOM is KWH, the actual maximum is multiplied by the IPH (intervals per hour) to get this value.
ABS_MAXIMUM	Peak value (computed using absolute maximum). "Absolute" means the program converts negatives to positives and selects the largest.
MAXDATE	Date and time of the peak interval.
MAXDATEn	Date and time of *n*th peak interval (e.g., MAXDATE2 reports the date and time of second highest peak. For *n*, you may supply any value from 2 through 10).
ABS_MAXDATE	Date and time of peak interval (computed using ABS_MAXIMUM).
MINIMUM	Minimum interval value in the handle (computed using the actual values, not the absolute value of the values).

Value	Description
MINIMUM_NZ	Minimum of all nonzero values in the handle.
MINDATE	Date and time that the minimum occurred.
LF	Load factor (load factor=average interval/maximum interval value) (computed using the absolute maximum).
STARTTIME	Date and time of the start of the handle data.
STOPTIME	Date and time of the end of the handle data.
COUNT	Total number of intervals in the handle.
COUNT_NZ	Total number of nonzero intervals in the handle.
IPH	Intervals per hour.
SPI	Seconds per interval.
UOM	Unit of measure, denoted by a code
DSTTOTAL	Total of the interval values in the fall Daylight Saving Hour. If the handle does not include this hour, the value is 0.
DSTENERGY	Total of the interval values in the fall Daylight Saving Hour. If the handle does not include this hour, the value is 0. The Unit of Measure for the intervals must be KW or KWH; otherwise, the value returned is 0.
MULTIPLIER	Pulse multiplier*.
OFFSET	Pulse offset*.
RECORDER	Recorder identifier.
CHANNEL	Channel number.
RECORDERCHAN	Recorder,channel.

About cut start and stop times

This applies to all INTDLOADxxx functions, except INTDLOADLISTENERGY. The functions use the cuts whose start and stop time are closest to the bill period start and stop time, as specified for the billing cycle code that applies to the account. If the account itself has a start or stop time, that takes precedence. If the account's channels (or the list's channels for INTDLOADLIST) have different start and stop times, the program automatically applies the earliest start and the earliest stop among the channels.

Accessing Multiple Interval Databases from the same Rate Schedule

You can load interval data from more than one interval data source in the same rate schedule using either INTDOPEN or INTDLOADxxx. When interval data has been opened or loaded in the rate form, you can use other functions as normal on the data.

Using INTDOPEN:

The **INTDOPEN Function** on page 9-51 enables you to open multiple Interval Data Databases from a single rate form.

Using INTDLOADxxx functions:

To load interval data from a rate form, use the following format:

```
<interval_data_reference> = INTDLOADxxx("<file and path name to interval
database>;<determinant_identifier|recorder,channel>");
```

Where:

- <file and path name to interval database> is a string containing the absolute path and file name of the Interval Database, followed by a semicolon, and the <recorder,channel>. The string can have no spaces, but can name any supported file type. The interval database file can be in any of the following formats:

    - Enhanced Oracle Utilities Input/Output Format (*.lse)

    - Oracle Utilities Standard Format (*.inp)

    - Oracle Utilities Comma Separated Format (*.csv)

    - Oracle Utilities Standard XML Format (*.xml)

This applies to all INTDLOADxxx functions.

Example: Your Interval Data Options (see the *Data Manager User's Guide*) are set to retrieve interval data from: c:\lodestar\user\getwell.lse. Another interval data file is located at: d:\lodestar\user\getwell2.lse, and you need to load data from that file also. You could load interval data from the second file with the following statement:

```
HNDL_2 = INTDLOAD ("d:\lodestar\user\getwell2.lse;1700,1");
```

Another example might look like this:

```
// Load test data for the current bill period
CUTNAME = "d:\comndata\testfile.lse;RECORDER_TEST,1";
HNDL = INTDLOAD(CUTNAME);
```

Loading Interval Data from Relational Database Tables

You can also load interval data from multiple relational database tables in the Oracle Utilities Data Repository using the following functions:

- **INTDLOAD Function**

- **INTDLOADDATES Function**

- **INTDLOADHIST Function**

- **INTDLOADLIST Function**

- **INTDLOADLISTHIST Function**

- **INTDLOADLISTDATES Function**

To load interval data from the relational database, use the following format:

```
<interval_data_reference> = INTDLOADxxx("[QUAL/<alternate_qualifier>;]RDB/
<alternate_table>;<recorder,channel>");
```

Where:

- <alternate_qualifier> is a string containing the name of a alternate database qualifier containing the interval data to be loaded.

    - When an alternate qualifier is specified, all database calls for the function will be directed at the specified qualifier, with one exception. In the case of INTDLOADLISTxxx() functions, the list query alone will be fetched from the original qualifier.

    - The metadata of the alternate qualifier **must** be the same as the original qualifier.

    - When using an alternate qualifier and processing in the context of an Account (such as when running billing via Oracle Utilities Billing Component), the account **must** be present in both the qualifiers.

- <alternate_table> is a string containing the name of a table with the same schema as the LSCHANNELCUTHEADER table. The name of this table must begin with the letters "LSC". Also, this table must have two child tables, one with column VALUESCODES and one with column SEQUENCE, and which have the same schema as the LSCHANNELCUTDATA and LSCHANNELCUTEDITS tables, respectively. This table must also have one parent table, the CHANNEL table, which in turn also has one parent table, the RECORDER table.

- <recorder,channel> is an identifier for a particular recorder-ID, channel-number in the Interval Database.

For example:

```
// Header data is stored in LSCHANNELHEADERVERS table
CUTNAME = "RDB/LSCHANNELHEADERVERS;RECORDER_TEST,1";
HNDL = INTDLOADDATES(CUTNAME, BILL_START, BILL_STOP);
```

Saving Data

You can also save data to an alternate qualifier and/or table using the SAVE TO CHANNEL statement, using the following format:

```
SAVE <HNDL> TO CHANNEL "[QUAL/<alternate_qualifier>;]RDB/
<alternate_table>;<recorder,channel>");
```

Where:

- <HNDL> is the interval data handle you wish to save.

- <alternate_qualifier> is a string containing the name of a alternate database qualifier containing the interval data to be loaded (see above).

- <alternate_table> is a string containing the name of a table with the same schema as the LSCHANNELCUTHEADER table (see above). The name of this table must begin with the letters "LSC" (see above).

- <recorder,channel> is an identifier the recorder-ID, channel-number you wish to save the data to.

For example:

```
// Load data from LSCHVERS table
HNDL = INTDLOADDATES("RDB/LSCHVERS;TEST,1", BILL_START, BILL_STOP);
// Save data to LSCHVERS2 table in PRICING qualifier
SAVE HNDL TO CHANNEL "QUAL/PRICING;RDB/LSCHVERS2;TEST,1";
```

Deleting Data

You can also delete interval data from an alternate qualifier and/or table using the **INTDDELETE Function**, using the following format:

```
<interval_data_reference> = INTDDELETE("[QUAL/<alternate_qualifier>;]RDB/
<alternate_table>;<recorder,channel>");
```

Where:

- <alternate_qualifier> is a string containing the name of a alternate database qualifier containing the interval data to be loaded (see above).

- <alternate_table> is a string containing the name of a table with the same schema as the LSCHANNELCUTHEADER table (see above). The name of this table must begin with the letters "LSC" (see above).

- <recorder,channel> is an identifier the recorder-ID, channel-number you wish to delete.

For example:

```
// Save data to LSCHVERS2 table in PRICING qualifier
SAVE HNDL TO CHANNEL "QUAL/PRICING;RDB/LSCHVERS2;TEST,1";
// Delete cut "Test,1" from LSCHVERS table
DEL_HNDL = INTDDELETE("RDB/LSCHVERS;TEST,1", BILL_START, BILL_STOP);
```

INTDLOAD Function

Purpose

The INTDLOAD Function loads and totalizes all of an account's interval data for a user-specified determinant for the current bill period.

When specifying the account's interval data to be loaded using INTDLOAD, you have two choices: you can specify a particular channel by its recorder-ID, channel-number; or you can ask for all of the account's channels that collect data for a selected billing determinant. The first option is typically used in a contract that is specific to the account, because you are "hard coding" a particular recorder-ID, channel-number into the rate form. For the second option, if you select a billing determinant, the program looks up the unit of measure (UOM) associated with the bill determinant in the BILLDETERMINANT Table. Then, it looks at the CHANNELHISTORY Table to determine which of the channels that belong to the account currently being billed collect data in that UOM.

Format

```
<interval_data_reference> =
INTDLOAD(<determinant_identifier|recorder,channel>[,<string>])
```

Where

- <determinant_identifer> is an identifier for a billing determinant, as defined in the BILLDETERMINANT Table. Its UOM is retrieved, and the matching recorder,channels under the account are totalized to get the interval data cut.

- <recorder,channel> is the identifier (also called a "cut series key") for a particular recorder-ID, channel-number in the Interval Database. The format is 'recorder,channel'; e.g., '1701,1'

- <string> (*Optional)* is the second parameter, which can be "LAST_CUT" or "PURE_CUT". If it is "LAST_CUT", the last cut for the recorder,channel is loaded, preserving its cut start and stop times. If "PURE_CUT", the cuts that overlap the bill period are loaded, preserving the cut start and stop times.

Examples

Load interval data cuts based on KW, KVAR, and channel 80001,2.

```
INT_MY_HNDL = INTDLOAD(KW);
INT_KVAR_HNDL =INTDLOAD(KVAR);
INT_ANOTHER_HNDL = INTDLOAD('80001,2', "PURE_CUT");
```

INTDLOADACTUALCUT Function

Purpose

This INTDLOADACTUALCUT Function loads a specific, single interval data cut for a specified recorder,channel or billing determinant for a given start time. This function loads individual cuts as stored in the Oracle Utilities Data Repository. This function does NOT load multiple cuts if multiple cuts (or overlapping cuts) exist for the supplied parameters. Returns an interval data reference.

Format

```
<identifier> = INTDLOADACTUALCUT
(<determinant_identifier|recorder,channel>[,<date_identifier|expressio
n>]);
```

Where

- <determinant_identifer> is an identifier for a billing determinant, as defined in the BILLDETERMINANT table. Use of a billing determinant with this function is applicable only when running in the context of an account (such as when running in Oracle Utilities Billing Component).

- <recorder,channel> is the identifier (also called a "cut series key") for a particular recorder-ID, channel-number in the Interval Database. The format is 'recorder,channel'; e.g., '1701,1'

- <date_identifier> (*Optional)* is the start time of the cut as stored in the Oracle Utilities Data Repository. If not supplied, the default value is equal to BILL_START.

Example

Load the interval data cut for recorder,channel 80001,2 with a start time of 06/01/2005.

```
HNDL = INTDLOADACTUALCUT('80001,2', "06/01/2005");
```

INTDLOADDATES Function

Purpose

The INTDLOADDATES Function loads interval data for a user-specified date range. This function is similar to the **INTDLOAD Function** on page 9-34, except that you can specify a recorder,channel and a date range for the data.

Format

```
<interval_data_reference> =
INTDLOADDATES(<determinant_identifier|recorder,channel>,
<date_identifier|date_constant>,
<date_identifier|date_constant>[,<loadflag>][,<tzstd>][,<dst_flag>]);
```

Where

- <determinant_identifer> is an identifier for a billing determinant, as defined in the BILLDETERMINANT Table. Its UOM and End Use are retrieved, and the matching recorder,channels under the account are totalized to get the interval data cut.

- <recorder,channel> is the identifier (also called a "cut series key") for a particular recorder-ID,channel-number in the Interval Database. The format is 'recorder,channel'; e.g., '1701,1'.

- <date_identifier|date_constant>, <date_identifier|date_constant> are actual start and end dates. Acceptable formats are 'MM/DD/YYYY', 'MM/DD/YYYY HH:MM', and 'MM/DD/YYYY HH:MM:SS'.

  If you do not specify a time, the time defaults to 00:00:00.

- <loadflag> (optional) specifies the behavior of the function when loading cuts with mixed time zones and/or DST Participant flags. This parameter must be an integer, or an identifier or expression that resolves to an integer that is the sum of two values, one that specifies how to resolve mixed time zones (using the Time Zone Standard Name field), and one that specifies how to resolve mixed DST Participant flags.

  - Time Zones:

    - 0: If the Time Zone Standard Names are different, return an error. (Default)

    - 1: Use the first cut's Time Zone Standard Name (TZSN).

    - 2: Use the TZSN supplied in the function call.*

  - DST Participants

    - 4: If the DST Participant flags are different, return an error. (Default)

    - 8: Use the first cut's DST Participant flag.

    - 16: Use the DST Participant flag supplied in the function call.*

  For example, to return an error if either the TZSN or the DST Participant flags are different, you would set this parameter to 4 (0 for TZSN and 4 for DST) This is the default value for this parameter. To use both the TZSN and DST Participant flag from the first cut, you would set this parameter to 9 (1 for TZSN, 8 for DST).

  *When using **either** the TZSN **or** the DST Participant flag supplied in the function call, use 2 or 16 as normal. To use **both** the TZSN and DST Participant flag supplied in the function call, use 32.

The table below lists the possible combinations of the Time Zone and DST Participant values:

TZSN	DST Participant	Load Flag Parameter
Error if different (0)	Error if different (4)	4
Error if different (0)	From first cut (8)	8
Error if different (0)	From function call (16)	16
From first cut (1)	Error if different (4)	5
From first cut (1)	From first cut (8)	9
From first cut (1)	From function call (16)	17
From function call (2)	Error if different (4)	6
From function call (2)	From first cut (8)	10
From function call (2)*	From function call (16)*	32*

- <tzstd> (optional) is the TZSN for the handle. The supplied value must be one of "EST", "CST", "MST", "PST", or be defined in the LSCALENDAR.XML configuration file (if present). If empty, this is equal to the Time Zone Standard Name of the first cut. The Default value is the default time zone, as specified in the LSCALENDAR.XML file.

- <dst_flag> (optional) is the DST Participant flag for the handle. Must be either "Y" or "N". If empty, this is equal to the DST Participant flag of the first cut. The Default value is "N".

Examples

Load kWh interval data for the month of January, 2007:

```
KWH_HNDL = INTDLOADDATES(KWH, '01/01/2007', '01/31/2007');
```

OR

```
STARTDT = '01/01/2007';
STOPDT = '01/31/2007';
KWH_HNDL = INTDLOADDATES(KWH, STARTDT, STOPDT);
```

Load interval data for recorder,channel 'RECORDER01,1' for the month of January, 2007:

```
HNDL = INTDLOADDATES('RECORDER01,1', '01/01/2007', '01/31/2007');
```

OR

```
RECORDER = "RECORDER01";
CHANNEL = "1";
RECORDER_CHANNEL = RECORDER + "," + CHANNEL;
STARTDT = '01/01/2007';
STOPDT = '01/31/2007';
HNDL = INTDLOADDATES(RECORDER_CHANNEL, STARTDT, STOPDT);
```

Load kWh interval data for the month of January, 2006, and use the supplied Time Zone Standard Name and DST Participant flag.

```
STARTDT = '01/01/2006';
STOPDT = '01/31/2006';
KWH_HNDL = INTDLOADDATES(KWH, STARTDT, STOPDT, 32, "EST", "N");
```

INTDLOADHIST Function

Purpose

The INTDLOADHIST Function loads an account's interval data for a user-specified set of bill periods. This function is identical to the **INTDLOAD Function** on page 9-34, except that you can specify historical periods to load.

Format

```
<interval_data_reference> =
INTDLOADHIST(<determinant_identifier|recorder,channel>,
<start_bill_period_previous>, <end_bill_period_previous>);
```

Where

- <determinant_identifer> is an identifier for a billing determinant, as defined in the BILLDETERMINANT Table. Its UOM and End Use are retrieved, and the matching recorder,channels under the account are totalized to get the interval data cut.

- <recorder,channel> is the identifier (also called a "cut series key") for a particular recorder-ID,channel-number in the Interval Database. The format is 'recorder,channel'; e.g., '1701,1'.

- <start_bill_period_previous>, <end_bill_period_previous> specifies the bill periods to be loaded using the following convention: **0** is the current bill period, **1** is the previous bill period, and so on (the higher the number, the further back in time). The end_bill_period_previous must be greater than or equal to the start_bill_period_previous. The default start_bill_period _previous is 0 or the current period. The default end_bill_period_previous is the last period of data available for the account. If you specify a start but no end, the default end is the last period of data for that determinant stored for the account. See **Start and End Bill Period Parameters** in the section **Rules for Using Functions** in **Chapter 6**: **Rules Language Functions Overview** in the *Oracle Utilities Rules Language User's Guide* for additional details about specifying bill period parameters.

  If you omit both start and end bill period parameters from INTDLOADHIST, it is the same as INTDLOAD—only the data for the current bill period is loaded.

Examples

Load the account's KW data for the last 13 bill periods (including the current):

```
INT_MY_HNDL = INTDLOADHIST(KW,0,12);
```

Load the account's KVAR data for the current bill period.

```
INT_KVAR_HNDL =INTDLOADHIST(KVAR);
```

Load the interval data for recorder-id, channel-number 80001,2 for the bill period just before the current period.

```
INT_ANOTHER_HNDL = INTDLOADHIST('80001,2', 1, 1);
```

INTDLOADLIST Function

Purpose

The INTDLOADLIST Function totalizes the interval data for the current bill period for all channels in a TABLE.COLUMN list. (See the *Data Manager User's Guide* for information about creating TABLE.COLUMN lists.) The list must consist of a list of unique identifiers for the channels (UIDCHANNELs).

If the rate schedule is run in the context of an account, only channels in the list that are also in a CHANNELHISTORY record are loaded (CHANNELHISTORY STARTTIME and STOPTIME values are ignored; all matching channels are loaded for the full bill period). If there is no account (for example, for a rate schedule run using RUNRS.EXE) **ALL** channels in the list are loaded. In either case, all channels in the list should have the same UOM.

Note: This function returns summary values, and uses cut start and stop times. See **About cut start and stop times** on page 9-30 for more information.

Format

```
<interval_data_reference> = INTDLOADLIST(<list_identifier|list_name>);
```

Where

- <list_identifier | list_name> is an identifier, or literal constant of the form "listname", that identifies a list of UIDCHANNELs.

Example

These two statements would load and total all KWH channels for the account, given that the list (named ACCT_CHAN) is created with a TABLE.COLUMN query that targets the UIDCHANNEL column in the CHANNEL Table, and the query is: where ACCOUNTS.ID = ACCT_ID and CHANNELHISTORY.UNIT-OF-MEASURE CODE = 01.

```
ACCT_ID = ACCOUNT.ACCOUNTID;
KWH_HNDL = INTDLOADLIST ("ACCT_CHAN");
```

INTDLOADLISTDATES Function

Purpose

The INTDLOADLISTDATES Function totalizes the interval data for all channels in a list, over a user-specified date range. This function is identical to the **INTDLOADLIST Function** on page 9-39, except that you can specify a date range for the data.

Format

```
<interval_data_reference> =
INTDLOADLISTDATES(<list_identifier|list_name>,
<date_identifier|date_constant>, <date_identifier|date_constant>);
```

Where

*   <list_identifier | list_name> is an identifier, or literal constant of the form "listname", that identifies a list of UIDCHANNELs.

*   <date_identifier|date_constant>, <date_identifier|date_constant> are actual start and end dates. You can use any of the following formats:
    'MM/DD/YYYY', 'MM/DD/YYYY HH:MM', or
    'MM/DD/YYYY HH:MM:SS'.

    If you do not specify a time for the end date, the cut ends at midnight of the beginning of the specified date (default start time is 00:00:00; default end time is 23:59:59).

Example

Load and total all of the account's KWH data for the month of January 1997.

```
ACCT_ID = ACCOUNT.ACCOUNTID;
STARTDT = '01/01/1997';
STOPDT = '01/31/1997';
KWH_HNDL = INTDLOADLISTDATES ("ACCT_CHAN", STARTDT, STOPDT);
```

INTDLOADLISTENERGY Function

Purpose

The INTDLOADLISTENERGY Function finds the total kWh for all of an account's channels (or all channels in a list) that record billed kW or kWh, for the current bill period.

For this function, the listname is optional. It totalizes all (or all in the list) billed kWh or kW cuts (converting kW to kWh). It uses the entire cut, where the start date of the cut is on or after the bill start date and on or before the bill stop date. All intervals in each cut, including intervals after the bill stop date, are included.

Format

```
<interval_data_reference> =
INTDLOADLISTENERGY[(<list_identifier|list_name>)];
```

Where

- <list_identifier | list_name> *(Optional)* is an identifier, or literal constant of the form "listname", that identifies a list of UIDCHANNELs.

Examples

Load and total all KWH for an account that has only KWH cuts:

```
KWH_HNDL = INTDLOADLISTENERGY;
```

Loads and totals all KWH for an account that has KWH and KQH cuts (where the ACCT_CHAN query is ACCOUNTS.ID = ACCT_ID and CHANNELHISTORY.UNIT-OF-MEASURE CODE = 01):

```
KWH_HNDL = INTDLOADLISTENERGY ("ACCT_CHAN");
```

About cut start and stop times:

In the illustration below, INTDLOADLISTENERGY would use each cut's own recorded start and stop times, as noted.

Cut start and stop used by other INTDLOAD*xxx* functions
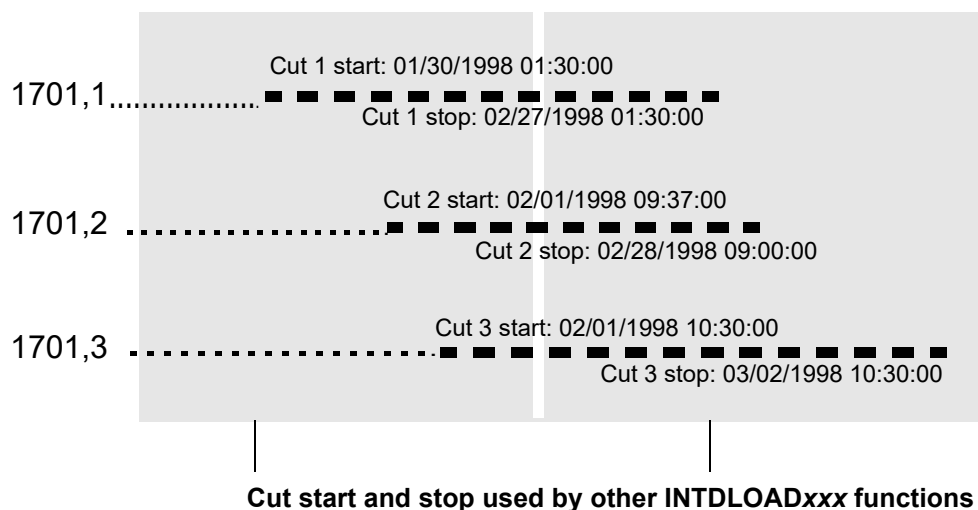
INTDLOADLISTHIST Function

Purpose

The INTDLOADLISTHIST Function totalizes the interval data for all channels in a list, over a user-specified set of bill periods. This function is identical to the **INTDLOADLIST Function** on page 9-39, except that you can specify historical periods to load.

Format

```
<interval_data_reference> =
INTDLOADLISTHIST(<list_identifier|list_name>,
<start_bill_period_previous>,<end_bill_period_previous>);
```

Where

- <list_identifier | list_name> is an identifier, or literal constant of the form "listname", that identifies a list of UIDCHANNELs.

- <start_bill_period_previous>, <end_bill_period_previous> specifies the bill periods to be loaded using the following convention: 0 is the current bill period, 1 is the previous bill period, and so on (the higher the number, the further back in time). The end period must be greater than or equal to the start bill period. The default start_bill_period_previous is 0 or the current period. The default end_bill_period_previous is the last period of data for the determinant. If you specify a start but no end, the default end is the last period of data for that determinant stored for the account. If you specify an end, you must specify a start. If you omit both start and end bill period from INTDLOADLISTHIST, it is the same as INTDLOADLIST—only the data for the current bill period is loaded.

Examples

In the following examples, the list is the same as that used in the description of the **INTDLOADHIST Function** on page 9-38.

Load and total all of the account's KWH data for the last 13 periods, including the current:

```
ACCT_ID = ACCOUNT.ACCOUNTID;
KWH_HNDL = INTDLOADLISTHIST("ACCT_CHAN", 0, 12);
```

Load and total all of the account's KWH data for the current bill period:

```
ACCT_ID = ACCOUNT.ACCOUNTID;
KWH_HNDL = INTDLOADLISTHIST("ACCT_CHAN");
```

Load and total all of the account's KWH data for the bill period just before the current period:

```
ACCT_ID = ACCOUNT.ACCOUNTID;
KWH_HNDL = INTDLOADLISTHIST("ACCT_CHAN", 1, 1);
```

INTDLOADRELATEDCHANNEL Function

Purpose

The INTDLOADRELATEDCHANNEL Function loads the interval data for the recorder,channel related to the interval data reference's recorder and channel through the CHANNELHISTORY Table, UIDRELATEDCHANNEL column, for the current account, for the record in effect on the reference's stop date. Returns an interval data reference.

Format

```
<identifier> = INTDLOADRELATEDCHANNEL(<interval_data_reference>);
```

Where

<interval_data_reference> is a reference to a loaded interval data cut, or a recorder,channel constant.

Example

Load the interval data for recorder,channel "1700,1".

```
INT_HNDL = INTDLOADRELATEDCHANNEL('1700,1');
```

INTDLOADSP Function

Purpose

The INTDLOADSP Totalizes the interval data for all channels belonging to a specified Aggregation Group, over a user-specified date range.

This function is typically used to totalize the interval data for all channels assigned to an Energy Service Provider. You can optionally filter the channel cuts by service code and factor code records.

Format

```
<interval_data_reference> = INTDLOADSP(<aggregation_group>,<service_code>,
<factor_code>, <start-date_identifier|date_constant>,
<end_date_identifier|date_constant>);
```

Where

- <aggregation_group> is an identifier or constant that contains an aggregation group name. Only channels belonging to this aggregation group are added. Typically this is the aggregation group for the service provider's Operating Company.

  To specify an alternate interval data table for the function call, include "RDB/<TABLE_NAME>;" immediately preceding the Aggregation group name, where <TABLE_NAME> is the name of the alternate interval data table. For example, "RDB/LSCHWEATHER;AGG_GROUP1" would load data from the LSCHWEATHER table for the AGG_GROUP1 aggregation group.

  To load from an alternate qualifier, include "QUAL/<QUALIFIER>; immediately before the table name, where <QUALIFIER> is the alternate qualifier. For example, "QUAL/PMQA;RDB/LSCHWEATHER;AGG_GROUP1" would load data from the LSCHWEATHER table in the PMQA qualifier for the AGG_GROUP1 aggregation group.

- <service_code> is an optional identifier or constant that contains a service code from the Service Table. If not specified, the default is all services. If the service code is not specified, or if the UOM associated with the Service Code is NULL, the program assumes UOMCODE "01" (kWh).

- <factor_code> is an optional identifier or constant that contains a key to the Factor Table. If supplied, only accounts with a matching ACCTFACTORHIST record are used. The actual key requires an operating company and jurisdiction; if not supplied, they default to the current rate form's. To explicitly specify an Operating Company and Jurisdiction, the code value must be "opcocode,juriscode,factorcode". To indicate that the factor is global (applies across all operating companies and jurisdictions), use the convention ",,factorcode".

- <start_date_identifier|date_constant>, <end_date_identifier|date_constant> are actual start and end dates. You can use any of these formats: 'MM/DD/YYYY', 'MM/DD/YYYY HH:MM', or 'MM/DD/YYYY HH:MM:SS'.

If you do not specify a time for the end date, the handle ends at midnight of the beginning of the specified date (default start time is 00:00:00; default end time is 23:59:59). If neither start date or end date is specified, the current bill period BILL_START and BILL_STOP is used.

If there are no accounts or channels that match the criteria, a handle of 0s is returned.

Examples

Example 1: *Load all of the energy from supplier ESP1 for the current bill period.*

```
ESP1_ENERGY_HNDL = INTDLOADSP("ESP1")
```

Example 2: *Load data from the LSCHWEATHER table for the AGG_GROUP1 aggregation group for the current bill period.*

```
WEATHER_HNDL = INTDLOADSP ("RDB/LSCHWEATHER;AGG_GROUP1");
```

INTDLOADSTAGING Function

Purpose

The INTDLOADSTAGING function loads interval data from the Interval Data Staging Tables (LSINTDSTAGING) for a user-specified date range. This function is similar to the **INTDLOADDATES Function** on page 9-36, except that it loads data from the Interval Data Staging tables.

Format

```
<interval_data_reference> = INTDLOADSTAGING(<recorder,channel>,
<date_identifier|date_constant>, <date_identifier|date_constant>);
```

Where

- <recorder,channel> is the identifier (also called a "cut series key") for a particular recorder-ID,channel-number in the Interval Database. The format is 'recorder,channel'; e.g., '1701,1'. Optionally, a table name and qualifier can also be specified, in the following format:

    [QUAL/<alternate_qualifier>,][RDB/<alternate_table>;]

    where:

    - <alternate_qualifier> is a string containing the name of a alternate database qualifier containing the interval data to be loaded.

        - The metadata of the alternate qualifier **must** be the same as the original qualifier.

        - When using an alternate qualifier and processing in the context of an Account (such as when running billing via Oracle Utilities Billing Component), the account **must** be present in both the qualifiers.

    - <alternate_table> is a string containing either LSINTDSTAGING or an equivalent.

- <date_identifier|date_constant>, <date_identifier|date_constant> are actual start and end dates. Acceptable formats are 'MM/DD/YYYY', 'MM/DD/YYYY HH:MM', and 'MM/DD/YYYY HH:MM:SS'.

    If you do not specify a time, the time defaults to 00:00:00.

Example

Load data for recorder,channel '1700,1' for the month of January, 2004 from the Interval Data Staging table:

```
STAG_HNDL = INTDLOADSTAGING('1700,1', '01/01/2004', '01/31/2004');
```

OR

```
STARTDT = '01/01/2004';
STOPDT = '01/31/2004';
STAG = INTDLOADSTAGING('1700,1', STARTDT, STOPDT);
```

INTDLOADUOM Function

Purpose

The INTDLOADUOM function loads the interval data for the specified UOM (and optional end use) for the current bill period.

This function is identical to the **INTDLOAD Function** on page 9-34, except that you specify the UOM and (optionally) end use of interest. INTDLOAD retrieves interval data according to the bill determinant you specify; this function retrieves interval data according to the UOM and end use you specify.

Format

```
<interval_data_reference> = INTDLOADUOM(<uom_code>, [<end_use_code>]);
```

Where

- <uom_code> is the code for a unit of measure in the UOM Lookup Code Table. All matching recorder,channels under the account are totalized to get the interval data handle.

- <end_use_code> *(Optional)* is the code for an end use from the ENDUSE Table. If specified, the recorder,channels must match both the UOM and End Use to be totalized to get the new interval data handle.

Examples

Load and totalize all of the account's KWH data for the current bill period:

```
KWH_HNDL = INTDLOADUOM ("01");
```

Load and totalize all of the account's KWH data for the end use "refrigeration," for the current bill period:

```
KWH_HNDL = INTDLOADUOM ("01", "REFRIGERATION");
```

INTDLOADUOMDATES Function

Purpose

The INTDLOADUOMDATES function loads the interval data for the specified UOM (and optional end use) over a specified date range. This function is identical to the **INTDLOADUOM Function** on page 9-47, except that it enables you to specify the date range of interest.

Format

```
<interval_data_reference> = INTDLOADUOMDATES(<uom_code>,
<end_use_code>, <date_identifier|date_constant>,
<date_identifier|date_constant>);
```

Where

- <uom_code> is the code for a unit of measure in the UOM Lookup Code Table. All matching recorder,channels under the account are totalized to get the interval data handle.

- <end_use_code> is the code for an end use from the ENDUSE Table. It is ignored if it is "" (Null). If specified, the recorder,channels must match both the UOM and End Use to be totalized to get the interval data handle.

- <date_identifier|date_constant>, <date_identifier|date_constant> are actual start and end dates. If no time is supplied with the end date, the handle ends at midnight of the beginning of the specified date.

Examples

Load and totalize all of the account's KW data for January 1997:

```
KW_HNDL = INTDLOADUOMDATES ("02", "", '01/01/97', '01/31/97');
```

Load and totalize all of the account's KW data for January, 1997 for the end use "refrigeration":

```
STARTDT = '01/01/1997';
STOPDT = '01/31/1997';
KW_HNDL = INTDLOADUOMDATES ("02", "REFRIGERATION", STARTDT, STOPDT);
```

INTDLOADUOMHIST Function

Purpose

The INTDLOADUOMHIST function loads the interval data for the specified UOM and (optionally) end use for the specified bill periods. This function is identical to the **INTDLOADUOM Function** on page 9-47, except that it enables you to specify the bill periods of interest.

Format

```
<interval_data_reference> = INTDLOADUOMHIST(<uom_code>,
<end_use_code>, <start_bill_period_previous>,
<end__bill_period_previous>);
```

Where

- <uom_code> is the code for a unit of measure in the UOM Lookup Code Table. All matching recorder,channels under the account are totalized to get the interval data handle.

- <end_use_code> is the code for an end use from the ENDUSE Table. It is ignored if it is " ". If specified, the recorder,channels must match both the UOM and End Use to be totalized to get the interval data handle.

- <start_bill_period_previous>, <end_bill_period_previous> specifies the bill periods to be loaded using the following convention: **0** is the current bill period, **1** is the previous bill period, and so on (the higher the number, the further back in time). The end_bill_period_previous must be greater than or equal to the start_bill_period_previous. The default start_bill_period_previous is 0 (the current period). The default end_bill_period_previous is the last period of data available for the account. If you specify a start but no end, the default end is the last period of data for that determinant stored for the account. If you specify an end, you must specify a start. If you omit both start and end bill period parameters from INTDLOADUOMHIST, it is the same as INTDLOADUOM—only the data for the current month is loaded.

Examples

Load and totalize all of the account's KWH data for the last 13 periods, including the current:

```
KWH_HNDL = INTDLOADUOMHIST("01"," ", 0, 12);
```

Load and totalize all of the account's KW data for the end use "refrigeration", for the current bill period:

```
KW_HNDL = INTDLOADUOMHIST("02", "REFRIGERATION");
```

Load and totalize all of the account's KWH data for the bill period just before the current period:

```
KWH_HNDL = INTDLOADUOMHIST("01"," ", 1, 1);
```

INTDLOADVERSION Function

Purpose

The INTDLOADVERSION Function loads interval data from the Interval Data Version Tables (LSCHVERSION and LSCDVERSION) for a specified versioned cut. This function is similar to the **INTDLOAD Function** on page 9-34, except that it loads data from the Interval Data Version tables.

Format

```
<interval_data_reference> = INTDLOADVERSION(<header table name>,
<recorder,channel>, <date_identifier|date_constant>[, <version
sequence number>);
```

Where

- <header table name> is the name of the Interval Data Version table where the cut to be loaded is located.

- <recorder,channel> is the identifier (also called a "cut series key") for a particular recorder-ID,channel-number in the Interval Data Version table. The format is 'recorder,channel'; e.g., '1701,1'.

- <date_identifier|date_constant>is the start time of the cut to be loaded. Acceptable formats are 'MM/DD/YYYY', 'MM/DD/YYYY HH:MM', and 'MM/DD/YYYY HH:MM:SS'. If you do not specify a time, the time defaults to 00:00:00.

- <version sequence number> Optional. is the version sequence (from the LSCHVERSIONS table) for the cut to be loaded.

Example

Load a versioned cut for recorder,channel '1700,1' with a start time of January 1, 2004, and version sequence of 3.

```
VER_HNDL = INTDLOADVERSION(LSCHVERSION, '1700,1', '01/01/2004', 3);
```

INTDOPEN Function

Purpose

The INTDOPEN Function opens an Interval Data Database in a user-specified mode. Returns an index to the file, which can be used in the INTDREADFIRST, INTDREADNEXT, INTDRECCOUNT, and INTDCLOSE functions.

> **Note:** A maximum number of **five** interval data files can be open at the same time.

Format

```
<identifier> = INTDOPEN(<interval_data_source_file_name>,
[<open_mode>]);
```

Where

- <interval_data_source_file_name> is the path and file name for the Interval Data Database. This file can be in any of the following formats:

    - Enhanced Oracle Utilities Input/Output Format (*.lse)

    - Oracle Utilities Standard Format (*.inp)

    - Oracle Utilities Comma Separated Format (*.csv)

    - Oracle Utilities Standard XML Format (*.xml)

    - Relational Database Table (see **Loading Interval Data from Relational Database Tables** on page 9-31 for more information)

- <open_mode> *(Optional)* specifies the mode in which the file is opened:

    - "A" - open the file for writing only; create the file if it does not exist.

    - "U" - open the file for reading and writing; create the file if it does not exist.

    - "R" - open the file for reading only.

Example

Open the interval data file "MYDATA.LSE" in read-only mode.

```
INTD_FILE = INTDOPEN("C:\LODESTAR\USER\MYDATA.LSE", "R");
INTD_COUNT = INTDRECCOUNT(INTD_FILE);
```

INTDREADFIRST Function

Purpose

The INTDREADFIRST Function returns a reference to the first record in an Interval Data Database. Records in the Oracle Utilities Data Repository are ordered by recorder ID, channel, and starttime. Records in files are ordered as they appear in the file. Returns an interval data reference.

Format

```
<identifier> = INTDREADFIRST(<interval_data_source_index>);
```

Where

- <interval_data_source_index> An index from a previously loaded interval data handle. If the parameter is omitted, the interval data source that is selected under the **Interval Data Source** tab of the **Default Options** dialog is used. If present, this parameter must be the result of an **INTDOPEN Function** call.

Example

Load a reference to the first record in "MYDATA.LSE".

```
INTD_FILE = INTDOPEN("C:\LODESTAR\USER\MYDATA.LSE");
INTD_FIRST = INTDREADFIRST(INTD_FILE);
```

INTDREADNEXT Function

Purpose

The INTDREADNEXT Function returns a reference to the next record in an Interval Data Database. Records are ordered by recorder ID, channel, and starttime. Records in files are ordered as they appear in the file. If there are no more records, the function returns a 0. Returns an interval data reference.

Format

```
<identifier> = INTDREADNEXT(<interval_data_source_index>);
```

Where

- <interval_data_source_index> An index from a previously loaded interval data handle. If the parameter is omitted, the interval data source that is selected under the **Interval Data Source** tab of the **Default Options** dialog is used. If present, this parameter must be the result of an **INTDOPEN Function** call.

Example

Load a reference to the next record in "MYDATA.LSE".

```
INTD_FILE = INTDOPEN("C:\LODESTAR\USER\MYDATA.LSE");
INTD_NEXT = INTDREADNEXT(INTD_FILE);
```

INTDRECCOUNT Function

Purpose

The INTDRECCOUNT Function returns the number of records in an Interval Data Database. Returns a scalar numeric value.

Format

```
<identifier> = INTDRECCOUNT(<interval_data_source_index>);
```

Where

- <interval_data_source_index> An index from a previously loaded interval data handle. If the parameter is omitted, the interval data source that is selected under the **Interval Data Source** tab of the **Default Options** dialog is used. If present, this parameter must be the result of an **INTDOPEN Function** call.

Example

Find the number of records in "MYDATA.LSE".

```
INTD_FILE = INTDOPEN("C:\LODESTAR\USER\MYDATA.LSE");
INTD_COUNT = INTDRECCOUNT(INTD_FILE);
```

INTDRELEASE Function

Purpose

The INTDRELEASE Function releases an interval data reference. It can be used to free up resources used by the reference; for example, to conserve memory, you can release the interval data as soon as the rate form is done with it. The resources are automatically released at the end of the rate form; however, resources should be freed as soon as possible if complex calculations are performed over large sets of interval data. This also frees any Time-of-Use references that were created from this interval data reference. This function always returns the integer 0.

Note that you can release a 'recorder,channel' constant; if you refer to it again, the interval data will be reloaded.

Format

```
<identifier> = INTDRELEASE(<interval_data_reference>);
```

Where

- <interval_data_reference> is a reference to a loaded interval data handle, or a recorder,channel constant.

Example

Release the KWH handle.

```
KWH_REL = INTDRELEASE(KWH_HNDL);
```

INTDREPLACE Function

Purpose

The INTDREPLACE Function replaces a range of intervals in an interval data handle with another previously loaded interval data handle of the same SPI. Intervals are replaced based on the Start and Stop time of the replacement handle only (not indexes). Replaced intervals have a status code of "A". Additionally, the "Edited by Rules Language" flag for the handle will be set to "Y" and the Descriptor of the handle be set to "Computed". This function returns 0 if successful, 1 if an error occurs.

Format

```
<identifier> = INTDREPLACE(<original_handle>,
<replacement_handle>[,<update_flag>);
```

Where

- <original_handle> is a reference to a loaded interval data handle.

- <replacement_handle> is a reference to a loaded interval data handle that will replace intervals in the <original_handle>. This must have the same SPI as the <original_handle>. The Start Time and Stop Time of the <replacement_handle> must fall completely within the Start Time and Stop Time of the <original_handle>. In other words, the Start Time of the <replacement_handle> must be greater than or equal to the Start Time of the <original_handle>, and the Stop Time of the <replacement_handle> must be less than or equal to the Stop Time of the <original_handle>.

- <update_flag> is an optional flag that indicates (Y = yes, N = no) if the statistics for the handle should be automatically updated when the intervals are replaced by the function. If set to "N", statistics for the handle can be updated using the **INTDUPDATESTATS Function**. The default is "Y."

Example

Replace the intervals for June 2 in the KWH_HNDL with intervals from a profile meter, without updating statistics.

```
//Load KWH_HNDL
START = '06/01/2005 00:00:00';
STOP = '06/30/2005 23:59:59';
KWH_HNDL = INTDLOADDATES ('METER,1', START, STOP);
//
//Load PROFILE_HNDL
REP_START = '06/02/2005 00:00:00';
REP_STOP = '06/02/2005 23:59:59';
PROFILE_HNDL = INTDLOADDATES ('PROFILE,1', REP_START, REP_STOP);
//
//Replace intervals
REPLACE = INTDREPLACE(KWH_HNDL, PROFILE_HNDL, "N");
```

INTDROLLAVG Function

Purpose

The INTDROLLAVG Function calculates the rolling average (or total) in interval data.

Each interval in the target handle is totaled or averaged over the specified number of intervals from the source handle. For example, if the original handle contains 15-minute data and you want a 60-minute rolling average, you would specify a "run length" of 4 and the AVERAGE method. To create the new handle, the program replaces each value from the original handle with a value that is the sum of itself and the preceding three values, divided by 4.

If you specify a run length of 2 and AVERAGE, the program gets each new value by adding the original value to the preceding value, and dividing by 2.

If you specify the TOTAL method, the result is a rolling sum instead of a rolling average. If you specify a run length of 4 and TOTAL, the program gets each new value by adding the original value to the preceding three values (and does *not* divide by 4).

Note: All interval data values whose quality is worse than the quality code specified under **Options** are ignored by this function. (To view this setting, select **Tools**, then **Options**, then **Interval Data Options** from the Oracle Utilities application's menu bar. For more information, see the *Data Manager User's Guide*.)

Format

```
<interval_data_reference> = INTDROLLAVG(<interval_data_reference>,
<number_intervals>, <type>);
```

Where

- <interval_data_reference> is a reference to a loaded interval data handle, or a 'recorder,channel' constant.

- <number_intervals> is the number of intervals to roll (the "run length" in the description above).

- <type> is the way to roll the intervals, where type is one of the following:

  - **AVERAGE**: Averages the values for the number of intervals. This is the default.

  - **TOTAL**: Totals the values for the number of intervals.

Example

Roll a handle with 15-minute date to an hourly average.

```
ROLL_60_HNDL = INTDROLLAVG(KWH_HNDL, 4, "AVERAGE");
```

INTDROLLPEAK Function

Purpose

The INTDROLLPEAK Function identifies the peak interval from a specified number of intervals from the source handle, and creates a new handle consisting of those peak intervals.

For example, if the original handle contains 15-minute data and you want a 60-minute rolling peak, you would specify a "run length" of 4. To create the new handle, the program replaces each value from the original handle with a value that is the peak among itself and the preceding three values.

Note: All interval data values whose quality is worse than the quality code specified under **Options** are set to 0 by this function. (To view this setting, select **Tools**, then **Options**, then **Interval Data Options** from the Oracle Utilities application's menu bar. For more information, see the *Data Manager User's Guide.*)

Format

```
<interval_data_reference> = INTDROLLPEAK(<interval_data_reference>,
<scalar_value>);
```

Where

- <interval_data_reference> is a reference to a loaded interval data handle, or a 'recorder,channel' constant.

- <scalar_value> is the rolling window size in number of intervals (2, 3, ...).

Example

Roll a handle with 15-minute data to an hourly peak.

```
ROLL_60_HNDL = INTDROLLPEAK(KWH_HNDL, 4);
```

INTDSCALAROP Function

Purpose

The INTDSCALAROP Function performs scalar operation on each interval value in the handle. The scalar value may be any numeric constant or identifier.

Format

```
<interval_data_reference> = INTDSCALAROP(<interval_data_reference>,
<operation>, <scalar_value>);
```

Where

- <interval_data_reference> is a reference to a loaded interval data handle, or a recorder,channel constant.

- <operation> is one of "ADD", "SUBTRACT", "MULTIPLY", "DIVIDE_BY", "NORMALIZE", "NORM_100", "MAXIMUM", "MINIMUM", "ABSOLUTE", any of the Math Functions described in **Chapter Eleven**: **Math Function Descriptions**, or any of the following special operations:

  - **NORMALIZE**: Set a divisor equal to the maximum value of the cut, and then divides every value in the cut by this divisor. In other words, it sets the maximum value of the cut to 1 and adjust the remaining values proportionally.

  - **NORM_100**: Set a divisor equal to the maximum value of the cut divided by a 100, and then divides every value in the cut by this divisor. In other words, it sets the maximum value of the cut to 100 and sets remaining values proportionally.

  - **QUANTITY**: If the handle has a UOM that is a Rate UOM, multiply every interval by SPI / Seconds Per Unit, and change the UOM to the corresponding Quantity UOM.

  - **RATE**: If the handle has a UOM that is a Quantity UOM, multiply every interval by Seconds Per Unit / SPI, and change the UOM to the corresponding Rate UOM.

  - **ROUND_AND_CARRY**: Round each value, carry the remainder (positive or negative) forward, and add to the next non-missing interval value before rounding it.

  - **SC2NUM**: Convert the status codes to numbers, from 0 to the number of different status codes possible. Space gets the highest number, then A, ... with 9 assigned 1. Zero is set if the status code is not recognized.

  - **SC2NORM**: Convert the status codes to numbers, scaled to a maximum of 2/3rds the handle maximum. Space gets the highest number, then A, ... with 9 assigned the lowest nonzero number. Zero is set if the status code is not recognized.

  - **SETALLVALUES**: Assign the scalar to every interval.

  If the operation is "MAXIMUM" or "MINIMUM", the result element is the maximum or minimum, respectively, of the corresponding interval value and the scalar value. If the operation is "SQROOT" (square root), the scalar value is ignored, and the square root of each interval is computed. If the operation is "ABSOLUTE", the scalar value is ignored, and the absolute value of each interval is computed. "SC2NUM" converts status codes to numbers ranging from 0 to the number of different status codes possible. "SC2NORM" converts status codes to numbers scaled to a maximum of 2/3rds of the handle maximum. **SETALLVALUES** assigns the scalar value to every interval. If the operation is **QUANTITY** or **RATE**, the scalar value is ignored. If the operation is **QUANTITY**, the handle will be converted to its quantity UOM. If the operation is **RATE**, the handle will be converted to its rate UOM. If the handle's UOM is not mapped or is the same as the operation UOM, the handle will be copied unchanged.

- <scalar_value> is a numeric constant or an identifier that has a numeric value. The scalar value may be any numeric constant or identifier.

Example

Multiply each interval value in the handle by 2 (both of the following statements perform the same function).

KWH_HNDL_2 = INTDSCALAROP(KWH_HNDL, "MULTIPLY", 2);

KWH_HNDL_2 = KWH_HNDL * 2;

INTDSCALE Function

Purpose

The INTDSCALE Function creates a new interval data handle, where each interval has the IPH of the specified period. It aggregates the intervals in the input handle to determine each new interval, based on the aggregation type.

Format

```
<interval_data_reference> = INTDSCALE(<interval_data_reference>,
<period>, < type>);
```

Where

- <interval_data_reference> is a reference to a loaded interval data handle, or a recorder,channel constant.

- <period> is the new interval period; "MIN15", "MIN30", "HOUR", "DAY", "WEEK", "MONTH", "QUARTER", "YEAR", or in Seconds. Valid seconds values ("1" through "3600") must be divisible by the SPI (seconds per interval).

  Note: Scaling up to and/or down from "WEEK", "MONTH", "QUARTER", or "YEAR" periods can produce unexpected results.

- <type> is one of the following. The default is the AGGREGATE value for the corresponding UOM record. In the following, IPH is the original IPH (Intervals per Hour); tIPH is the new IPH.

  - **TOTAL**: Add the values for each interval in the <period> (or TOU period in the <period>).

  - **HOUR_TOTAL**: If AGGREGATE=Average, then TOTAL/IPH, else TOTAL.

  - **MAXIMUM**: The maximum of the interval values.

  - **HOUR_MAXIMUM**: If AGGREGATE=Total, then MAXIMUM*IPH, else MAXIMUM.

  - **MINIMUM**: The minimum of the interval values.

  - **MINIMUM_NZ**: The minimum of the nonzero values.

  - **AVERAGE**: The average of the interval values.

  - **AVERAGE_NZ**: The average of the nonzero values.

  - **LF**: AVERAGE/MAXIMUM.

  - **HOURS**: The number of hours in <period> for nonzero interval values (or TOU period in <period>).

  - **IPH_TOTAL**: If AGGREGATE=Average, then (TOTAL/IPH)*$tIPH$, else TOTAL.

  - **IPH_MAXIMUM**: If AGGREGATE=Total, then (MAXIMUM*IPH)/$tIPH$, else MAXIMUM.

  - **MAXIMUMn:** The nth maximum of the interval values.

  - **HOUR_MAXIMUMn:** If AGGREGATE=Total then MAXIMUMn * IPH, else MAXIMUMn.

  - **MAXDATEn:** Set the nth maximum, and its date and time.

Examples

Aggregate a previously loaded 15-minute-per-interval handle to a 60-minute-per-interval handle, using the method appropriate to the handle's UOM.

```
KWH_60_HNDL = INTDSCALE(KWH_15_HNDL, "HOUR");
```

Scale a 15-minute-per-interval interval data handle to a 5-minute-per-interval handle using the method appropriate to the handle's UOM.

```
// Scale 15 Minute Data to 300 Seconds (5 Minutes)
GG_HNDL = INTDLOADDATES('DAY,1', '01/21/1999 19:45:00',
'01/22/1999 19:44:59');
GG_HNDL = INTDSCALE(GG_HNDL, "300");
```

INTDSETATTRIBUTE Function

Purpose

The INTDSETATTRIBUTE Function sets the attributes of a specified data handle. Returns an integer; zero if successful, not zero if an error (for example, if this attribute cannot be modified).

Format

```
<identifier> = INTDSETATTRIBUTE(<interval_data_reference>,
<attribute>, <identifier|expression>);
```

Where

- <interval_data_reference> is a reference to a loaded interval data handle, or a recorder,channel constant.

- <attribute> is one of the following:

    - **CALC_CONSTANT**: the CALC value.

    - **CATEGORY**: the category of the handle. Used only with enhanced interval data tables.

    - **CHANNEL**: the channel number.

    - **DATA_STATUS_FLAG**: the data status flag.

    - **DATE**: the date and time of the handle.

    - **DATE_INDEX**: the index of the interval containing that time.

    - **DC_FLOW**: the DC Flow (power flow) of the handle.

    - **DESCRIPTOR**: the descriptor field.

    - **DSTPARTICIPANT**: the DST Participant Flag for the handle.

    - **EDIT_FLAG_CHAR**: the edit flag.

    - **EXTERNAL_VAL_FLAG_CHAR**: the external validation status flag.

    - **INTERNAL_VAL_FLAG_CHAR**: the internal validation status flag.

    - **IPH**: the intervals per hour.

    - **MASK_FLAG**: a flag that indicates if the handle is a mask or not.

    - **MERGE_FLAG_CHAR**: the merge flag.

    - **MESSAGE01**, ..., **MESSAGE10**: the corresponding message field.

    - **METER_MULT**: the meter multiplier for the readings.

    - **METER_OFFSET**: the meter offset for the readings.

    - **MULTIPLIER**: the pulse multiplier.

    - **OFFSET**: the pulse offset.

    - **ORIGIN**: the origin of the handle.

    - **PARENTKEY**: the identity of the parent of the handle. Used only with enhanced interval data tables.

    - **POPULATION**: the population value. Used with Statistical records only.

    - **QUANTITY_MAX**: the maximum if UOM is not a rate, else maximum * SPI / Seconds-Per-Unit.

    - **QUANTITY_TOTAL**: the total if UOM is not a rate, else total * SPI / Seconds-Per-Unit.

- **RATE_MAX**: the maximum if UOM is not a quantity, else maximum * Seconds-Per-Unit / SPI.

- **REALSTARTTIME**: the date and time of the first interval.

- **REALSTOPTIME**: the date and time of the last interval.

- **RECORDER**: the recorder ID.

- **START_READING**: the start reading.

- **STOP_READING**: the stop reading.

- **STOPTIME**: the date and time of the last interval.

- **TIMESTAMP**: the date and time the handle was input.

- **TIMEZONES**: the number of timezones of the handle dates from GMT (-1 is unknown). Each timezone is 1/2 hour long; i.e., EST is 10 timezones from GMT.

- **TZSTDNAME**: the Time Zone Standard Name for the handle.

- **UOM**: the numeric UOM code for the handle.

- **VAL_FLAG_E**: the value of the validation flag E.

- **VAL_FLAG_I**: the value of the validation flag I.

- **VAL_FLAG_N**: the value of the validation flag N.

- **VAL_FLAG_O**: the validation flag O.

- **WEIGHT**: the weight value. Used with Statistical records only.

- <identifier|expression> is either an identifier or an expression that sets the values of the attribute. If an identifier, it must have been assigned earlier in the rate form. The type of identifier or expression must match the return type listed above.

Examples

Set the Meter Multiplier (METER_MULT) of 'MY_HNDL' to 1.1.

```
MY_HNDL_MM = INTDSETATTRIBUTE(MY_HNDL, METER_MULT, 1.1);
```

OR

```
MY_MM = 1.1;
MY_HNDL_MM = INTDSETATTRIBUTE(MY_HNDL, METER_MULT, MY_MM);
```

INTDSETDSTPARTICIPANT Function

Purpose

The INTDSETDSTPARTICIPANT Function changes the DST Participant flag on an interval data handle. Optionally the function can also adjust the Start Time and Stop Time of the handle as appropriate for the new DST Participant flag (forward one hour when changing from "N" to "Y", and back one hour when changing from "Y" to "N" when the handle's Start Time falls during Daylight Savings Time). Returns an interval data reference.

Note: This function creates a new handle. If the user saves the new handle to the same data source as the original cut, an interval data overlap situation will occur. Therefore Oracle Utilities recommends that users recognize and understand that when performing DST conversions using this function and the cut's start time has changed, if they wish to save the new handle to the original data source, they should delete the original cut from the data source.

Format

```
<identifier> = INTDSETDSTPARTICIPANT(<interval_data_reference>,
<DST_Participant>, <Convert_Flag>);
```

Where

- <interval_data_reference> is a reference to a loaded interval data handle, or a recorder,channel constant.

- <DST_Participant> is the value ("Y" or "N") to which the DST Participant flag on the handle will be set.

- <Convert_Flag> whether to change only the DST Participant flag or to also convert the handle as appropriate. The two options include:

    - **N**: Change only the DST Participant flag and do not adjust the Start Time and Stop Time of the handle (Default).

    - **Y**: Change the DST Participant flag and adjust the Start Time and Stop Time of the handle.

Example

Change the DST Participant flag from "N" to "Y", and convert the handle's Start Time and Stop Time appropriately.

```
HNDL = INTDLOAD('1700,1');
SET_DST_Y = INTDSETDSTPARTICIPANT(HNDL, Y, Y);
```

INTDSETSTRING Function

Purpose

The INTDSETSTRING Function sets the status code value of all non-missing intervals in an interval data handle to a user-specified character.

INTDSETSTRING is often used with the **IDATTR Function** to assign a status code to all of the non-missing intervals in a Profiled handle (a "Profiled" handle is one that was created by taking an account's bill determinant scalar value for the bill period and "spreading" it out into a load shape that matches a class profile template). The status code assigned to each interval can be the same as that stored in the Data Repository for the scalar value.

Format

```
<interval_data_reference> = INTDSETSTRING(<interval_data_reference>,
"STATUSCODE", <status_code>);
```

Where

- <interval_data_reference> is a reference to a loaded interval data handle, or a recorder,channel constant.

- <status_code> is an identifier, expression, or string constant whose value is a string consisting of a single character.

Example

Set the status code of each non-missing interval in the original handle to the value assigned to the temporary identifier SC, and store the new version of the handle with the handle MY_PROFILED_handle.

```
SC = IDATTR(KWH, "STATUSCODE");
MY_PROFILED_HNDL = INTDSETSTRING(OLD_PROF_HNDL, "STATUSCODE," SC);
```

In the first statement, the IDATTR function gets the status code stored in the Bill History Table's KWHSC column for the current account, and assigns it to the temporary identifier SC.

INTDSETVALUE Function

Purpose

The INTDSETVALUE Function sets an interval value for a previously loaded interval data handle. It creates a new (temporary) handle based on a set of supplied parameters. The index argument is one-based, and indicates which interval to set. The value argument indicates what value to set it to. *The status code is always set to A.* This function returns 0 if successful, nonzero if an error occurred.

This function will not change the current value of original cut. The maximum can be retrieved using the **INTDVALUE Function** on page 9-79.

To use this function to change an existing value, you need to assign the new handle to the identifier of the original cut via an **Assignment Statement**) (i.e., HNDL=HNDL2) or use the **INTDVALUE Function** to get new maximum values.

Format

```
<identifier> = INTDSETVALUE(<interval_data_reference>, <index>,
<value>);
```

Where

- <interval_data_reference> is a reference to a loaded interval data handle, or a recorder,channel constant.

- <index> is the interval to be set by the function.

- <value> is the value to set the particular interval to.

Example

Assign the interval values of HNDL to 5.

```
HNDL = INTDLOAD('1700,1');
COUNT_INTERVALS = INTDCOUNT(HNDL, "ALL");
FOR EACH I IN NUMBER COUNT_INTERVALS
    SET_VALUE = INTDSETVALUE(HNDL, I, 5);
END FOR;
```

INTDSETVALUESTATUS Function

Purpose

The INTDSETVALUESTATUS Function changes the status code and/or value of some intervals in a handle, based on a comparison of their current status code with a supplied status code, a date range, or a range of indices. It creates an interval data handle that matches the referenced handle (same IPH, UOM, and number of intervals).

Formats

```
<identifier> = INTDSETVALUESTATUS(<interval_data_reference>,
<comparison>, <status_code>[, <result_value>], <result_status_code>);
```

or

```
<identifier> = INTDSETVALUESTATUS(<interval_data_reference>,
<start_datetime>, <end_datetime>[, <result_value>],
<result_status_code>);
```

or

```
<identifier> = INTDSETVALUESTATUS(<interval_data_reference>,
<start_index>, <end_index>[, <result_value>], <result_status_code>);
```

Where

- <interval_data_reference> is a reference to a loaded interval data handle, or a recorder,channel constant.

- <comparison> is one of the following:

    - "=" - Equal

    - "<>" - Not equal

    - "<" - Less than

    - ">" - Greater than

    - "<=" - Less than or equal

    - ">=" - Greater than or equal

    - "IN" - IN the status code string

    - "NOT IN" - NOT IN the status code string.

    The comparison order is (from highest to lowest): (space) A B C...Z 0 1 2...8 9.

- <status_code> is an identifier or string constant that specifies a valid status code, or, for "IN" and "NOT IN", a string of status codes (no separator; "ABC" for codes A, B, and C). To set the status code to NULL, enter " " (double-quote-space-double-quote).

- <start_datetime>, <end_datetime> are identifiers or constants that specify dates and times.

- <start_index>, <end_index> are identifiers or constants that specify integer values.

- <result_value> *(Optional)* is an identifier or constant with a numeric value. If not supplied, the value is unchanged.

- <result_status_code> is an identifier or constant with a string value that specifies a valid status code. If not supplied, the status code is unchanged.

For the first format: The value or status code is set in any interval where the status code comparison is true. If the comparison is false, the value and status code are unchanged.

For the second and third formats: The value or status code is set in any interval whose date or index overlaps in any way the supplied date-time range or range of indices. All other values and status codes are unchanged.

Example

Set the status codes of all intervals in 'MY_HNDL' that fall between START_DATE and STOP_DATE to "A" and assign the resulting handle to 'NEW_STATUS_HANDLE'.

```
NEW_STATUS_HNDL = INTDSETVALUESTATUS(MY_HNDL, START_DATE, STOP_DATE,
"A");
```

INTDSHIFTSTARTTIME Function

Purpose

The INTDSHIFTSTARTTIME Function shifts the start (and stop) time of an interval data handle. The time identifier selects either the "STARTTIME" or "REALSTARTTIME" as the date from which the start time is shifted. The date time is the new start date/time. This function shifts the stop time of the handle by the same amount of time as the start time. If you shift the start time of a handle by 5 days, the stop date of the handle will also be shifted by 5 days.

Format

```
<interval_data_reference> =
INTDSHIFTSTARTTIME(<interval_data_reference>, <time_identifier>,
<date_time>);
```

Where

- <interval_data_reference> is a reference to a loaded interval data handle, or a recorder,channel constant.

- <time_identifier> is a string value that is either "STARTTIME" or "REALSTARTTIME".

- <date_time> is the new start date/time of the handle.

Example

Shift the start time of MY_HNDL from 05/01/93 to 05/08/93:

```
MY_HNDL_SHIFT = INTDSHIFTSTARTTIME(MY_HNDL, "STARTTIME", '05/08/93');
```

INTDSMOOTH Function

Purpose

The INTDSMOOTH Function smoothes gaps in interval data based on the type value. The type value must be "VALUE", "AVERAGE", or "PATCH". If "VALUE", all gaps are filled with the specified smooth value. If "AVERAGE", all gaps are filled with the average of all non-gaps. If "PATCH", gaps are filled with corresponding intervals from a supplied interval data handle.

Format

```
<interval_data_reference> = INTDSMOOTH(<interval_data_reference>,
<type>[, <scalar_value>][,<patch_interval_data_reference>);
```

Where

- <interval_data_reference> is a reference to a loaded interval data handle, or a recorder,channel constant.

- <type> is either **VALUE**, **AVERAGE**, or **PATCH**.

- <scalar_value> is a numeric constant, or an identifier that has a numeric value. It is required for type **VALUE**, and optional (ignored) for type **AVERAGE**.

- <patch_interval_data_reference> is a reference to loaded interval data handle that will be used to patch missing intervals in the <interval_data_reference>. This handle must contain at least the same number of intervals (or more) than the first handle.

Examples

Fill in all missing intervals in KWH_RAW_HNDL with the average value of non-missing intervals:

```
KWH_RAW_HNDL = INTDLOAD (KWH);
KWH_HNDL = INTDSMOOTH(KWH_RAW_HNDL, "AVERAGE");
```

Fill in all missing intervals in KWH_RAW_HNDL with the corresponding intervals from KWH_PATCH_HNDL:

```
KWH_RAW_HNDL = INTDLOAD (KWH);
KWH_PATCH_HNDL = INTDLOAD ('PATCH,1');
KWH_HNDL = INTDSMOOTH(KWH_RAW_HNDL, "PATCH", KWH_PATCH_HNDL);
```

INTDSORT Function

Purpose

The INTDSORT Function sorts the values in an interval data handle. The type value must be either "ASCENDING" or "DESCENDING". If "ASCENDING", the values are sorted from smallest to largest. If "DESCENDING", the values are sorted from largest to smallest. The default is "ASCENDING". This function returns an interval data reference.

Format

```
<interval_data_reference> = INTDSORT(<interval_data_reference>,
<type>);
```

Where

- <interval_data_reference> is a reference to a loaded interval data handle, or a recorder,channel constant.

- <type> is either **ASCENDING** or **DESCENDING**.

Example

Sort the intervals, from smallest to largest, in KWH_RAW_HNDL:

```
KWH_HNDL_SORT = INTDSORT(KWH_RAW_HNDL, "ASCENDING");
```

INTDSPIKETEST Function

Purpose

The INTDSPIKETEST Function examines the interval data for spikes as defined by the two parameters N and P. A spike is defined as any interval that exceeds a percent (P) of the average of the highest N number of intervals in the handle. This function returns a stem component variable that is the name of the variable in the Assignment statement (STEM in the example below). If no spikes are found in the handle of interval data, the SPIKECOUNT value will be set to zero. The stem will be set to "" if the function was successful, else it will be set to the integer zero. The component variables will include:

- STEM.SPIKECOUNT contains an integer count of the number of spikes found, up to 500.

- STEM.SPIKE1 contains the index of the first interval defined as a spike.

- STEM.SPIKE2 contains the index of the second interval defined as a spike.

- STEM.SPIKEn contains the index of the nth interval defined as a spike.

Format

```
<stem> = INTDSPIKETEST(<interval_data_reference>, <N>, <P>,
<status_code>);
```

Where

- <interval_data_reference> is a reference to the loaded interval data handle to be tested

- <N> is number of peaks to average

- <P> is the percent higher than the average of the specified peaks that any interval must be to be considered a spike

- <status_code> *Optional* is the status code that all intervals must be above or better than to be included in the validation. If not supplied, the status code will default to "9".

Example

Identify the intervals in CUT_HNDL that are more than 40 percent higher than the average of the top 5 peaks, and that have a Status Code of "L" or better:

```
SPIKETEST = INTDSPIKETEST(CUT_HNDL, 5, 40, "L" );
```

Result :

```
SPIKETEST.SPIKECOUNT = 2
SPIKETEST.SPIKE1 = 96
SPIKETEST.SPIKE2 = 225
```

INTDSUBSET Function

Purpose

The INTDSUBSET Function returns a new interval data reference whose values are those between the two dates or the supplied range of indices. The first date/index is the beginning date and time (or index) of an interval in the input interval data reference; the second date/index is the end date and time (or index) of an interval. You must use either start and stop dates OR start and stop indices. You cannot combine a start date with a stop index or a start index with a stop date.

Format

```
<identifier> = INTDSUBSET(<interval_data_reference>,
<date_identifier|expression>, <date_identifier|expression>);
```

OR

```
<identifier> = INTDSUBSET(<interval_data_reference>, <start_index>, <stop_index>);
```

Where

- <interval_data_reference> is a reference to a loaded interval data handle or a recorder,channel constant.

- <date_identifier|expression> is either an identifier that contains a date (such as BILL_PERIOD and BILL_START) or a date expression.

- <start_index> is an identifier that specifies an integer value. This value is one-based.

- <stop_index> is an identifier that specifies an integer value. This value is one-based.

Example

Return a subset of HNDL_1 that falls between 5/1/1993 and 5/4/1993:

```
HNDL_1 = INTDLOAD (KWH);
GET_SUBSET = INTDSUBSET (HNDL_1, "5/1/1993," "5/4/1993");
```

Return a subset of HNDL_1 that falls between index 5 and 15.

```
HNDL_1 = INTDLOAD (KWH);
GET_SUBSET = INTDSUBSET (HNDL_1, 5, 15);
```

INTDTOU Function

Purpose

The INTDTOU Function computes time-of-use values for the interval data handle, based on the given TOU schedule. If a season is specified, the computation is done only over those months within the season (in schedule SEASON_SCHEDULE_NAME). Values are computed for each period in the TOU schedule. The holidays are retrieved from the set named in the HOLIDAY_NAME.

Format

```
<interval_data_reference> = INTDTOU(<interval_data_reference>,
<schedule_name> [, <holiday_list_name>]);
```

Where

- <interval_data_reference> is a reference to a loaded interval data handle or a recorder,channel constant.

- <schedule_name> is an identifier with a string value, or a string constant, that is the name of a Time-of-Use Schedule.

- <holiday_list_name> (*Optional)* is a holiday list to be used with the TOU Schedule. The default is the default holiday schedule for the rate schedule's operating company, jurisdiction, which is opcocode|juriscode (e.g., GECO|MA).

Example

Create a Time-of-Use handle for the account:

```
TOU_HNDL = INTDTOU(KWH_HNDL, "MYTOUSCHED", "MYHOLIDAY");
```

Note

The INTDTOU function supports handles with up to 2 Daylight Savings Time shifts. Applying this function to a handle with more than 2 Daylight Savings Time shifts will result in incorrect values.

INTDTOURELEASE Function

Purpose

The INTDTOURELEASE Function releases a time-of-use reference set with INTDTOU. This function is similar to the **INTDRELEASE Function** on page 9-55, but is used specifically with TOU references. This function should be used to free resources used by TOU references. There can be up to four TOU references open that were created from the same interval data reference. The resources are also released when the interval data reference is released. This function always returns the integer 0.

Format

```
<identifier> = INTDTOURELEASE(<tou_handle_reference>);
```

Where

• <tou_handle_reference> is a reference returned by INTDTOU.

Example

Release the 'TOU_HNDL' reference.

```
TOU_REL = INTDTOURELEASE(TOU_HNDL);
```

INTDTOUVALUE Function

Purpose

The INTDTOUVALUE Function returns a value computed for a time-of-use period.

Format

```
<identifier> = INTDTOUVALUE(<interval_data_reference>, <period>,
<type>);
```

Where

- <interval_data_reference> is a reference returned by INTDTOU.

- <period> is an identifier with a string value, or a string constant (that is the name of a period in the time-of-use schedule), exactly as it appears in the schedule screen (trailing blanks are ignored).

- <type> is one of the following:

  - **TOTAL**: Adds the values for each interval in the period. This is the default.

  - **ENERGY**: Finds total energy for the period. This option is available only if interval data is kW or kWh. The values are combined appropriately based on IPH to compute the total energy in the period.

  - **MAXIMUM**: Finds maximum of the interval values during the period.

  - **MAXIMUM*n***: Finds *n*th maximum of the interval values during the period. For example, "MAXIMUM3" would find third highest value. You can specify the value of *n* as an integer from 2 through 10.

  - **KW_MAXIMUM**: If the UOM is KWH, this option returns the KW maximum by multiplying the actual maximum KWH value by the IPH (intervals per hour).

  - **MAXDATE*n***: Find the date and time of the *n*th maximum interval value during the period. For example, "MAXDATE3" would return the date and time of the third highest peak. You can specify the value of *n* as an integer from 2 through 10.

  - **MINIMUM**: Finds the minimum of the nonzero values during the period.

  - **AVERAGE**: Returns the average of the interval values during the period.

  - **LF**: Finds the load factor using the formula LF=(average/maximum) * 100%

  - **HOURS**: Finds the number of hours (rounded) in the bill period that were in the specified TOU period.

  - **DAYS**: Finds the number of days that have at least one interval in the specified period (**not** the number of hours divided by 24).

Example

Get the on-peak and off-peak energy for billing at different rates:

```
ONPEAK_KWH = INTDTOUVALUE(TOU_HNDL, "ONPEAK", "ENERGY");
OFFPEAK_KWH = INTDTOUVALUE(TOU_HNDL, "OFFPEAK", "ENERGY");
```

INTDUPDATESTATS Function

Purpose

This INTDUPDATESTATS Function updates statistics for an interval data handle that has had a range of intervals replaced via the **INTDREPLACE Function**. The default for the **INTDREPLACE Function** is to update statistics automatically. This function should only be used when statistics are explicitly NOT updated when using the **INTDREPLACE Function**.

Format

```
<identifier> = INTDUPDATESTATS(<interval_data_reference>);
```

Where

- <interval_data_reference> is an interval data handle previously operated on by the **INTDREPLACE Function**.

Example

Update statistics for KWH_HNDL.

```
//Load KWH_HNDL
START = '06/01/2005 00:00:00';
STOP = '06/30/2005 23:59:59';
KWH_HNDL = INTDLOADDATES ('METER,1', START, STOP);
//
//Load PROFILE_HNDL
REP_START = '06/02/2005 00:00:00';
REP_STOP = '06/02/2005 23:59:59';
PROFILE_HNDL = INTDLOADDATES ('PROFILE,1', REP_START, REP_STOP);
//
//Replace intervals
REPLACE = INTDREPLACE(KWH_HNDL, PROFILE_HNDL, "N");
//
...
//Update KWH_HNDL statistics
UPDATE = INTDUPDATESTATS (KWH_HNDL);
```

INTDVALUE Function

Purpose

The INTDVALUE Function computes a value for an interval data handle. The returned value is the result of adding, averaging, or taking the maximum of all the interval values. If "ENERGY" is specified, the interval data must be KW or KWH. It is added appropriately to compute the total energy in the handle. If "KW_MAXIMUM" is specified and the UOM is KWH, the actual maximum is multiplied by the IPH (intervals per hour) to get the KW maximum. The "AVERAGE" is over the non-missing values; the "AVERAGE_NZ" is over nonzero values.

If "QUANTITY_TOTAL" and a rate UOM are specified, the total is converted to a quantity; otherwise, a total is returned. If "QUANTITY_MAX" and a rate UOM are specified, the maximum is converted to a quantity; otherwise, a maximum is returned. If "RATE_MAX" and a quantity UOM are specified, the maximum is converted to a quantity; otherwise, a maximum is returned.

"QUANTITY_TOTAL" is the same as "ENERGY" for UOMs 01 and 02. "RATE_MAX" is the same as "KW_MAXIMUM" for UOMs 01 and 02. "MAX", "QUANTITY_MAX", and "RATE_MAX" are all equal for UOMs that do not appear in the UOMRATEQUANTITY Table.

Note: Whenever an interval data handle is loaded, the programs automatically compute all of the values that you can calculate using INTDVALUE, *except* INDEX and DATE. DATE enables you to retrieve the value for a specified date and time. INDEX enables you to retrieve the value for a specific interval, such interval #29.

Format

```
<identifier> = INTDVALUE(<interval_data_reference>, <type>,
<identifier|constant>);
```

Where

- <interval_data_reference> is a reference to a loaded interval data handle, or a recorder,channel constant.

- <type> must be one of the following:

    - **ABS_MAXIMUM**: returns a float that is the maximum of all the absolute values (ignores positive or negative sign).

    - **ABS_MAXDATE**: returns the date and time of the absolute maximum.

    - **AVERAGE**: returns a float that is TOTAL / COUNT.

    - **AVERAGE_NZ**: returns a float that is TOTAL / COUNT_NZ.

    - **CALC_CONSTANT**: returns a float that is the CALC value.

    - **CATEGORY**: returns a string containing the category of the cut. Used only with enhanced interval data tables.

    - **CHANNEL**: returns an integer that is the channel number.

    - **CHNSTATUS**: returns the channel status for the handle. **Note**: Only available when loading data from enhanced interval data tables.

    - **CMBSTATUS**: returns the combined status code for the handle (based on channel status and extended status codes). **Note**: Only available when loading data from enhanced interval data tables.

    - **COUNT**: returns an integer that is the total number of values.

    - **COUNT_NZ**: returns an integer that is the total number of values that are not zero or missing.

- **DATA_STATUS_FLAG**: returns a character (Y or N, depending on whether the data and status codes are both present). This type is Oracle Utilities Load Analysis specific.

- **DATE**: the identifier or expression must be present (containing a date and time). The value at that time is returned. It is an error if the date is outside the range of the interval data reference.

- **DATE_INDEX**: the identifier or expression must be present (containing a date and time). The index of the interval containing that time is returned. It is an error if the date is outside the range of the interval data reference.

- **DC_FLOW**: returns a character: D (Delivered), R (Received), or a space if unknown.

- **DELETE_FLAG**: returns 1 if the Delete flag is On (Y); else returns 0.

- **DELETE_FLAG_CHAR**: returns Delete flag; Y or N.

- **DESCRIPTOR**: returns a string that is the descriptor field.

- **DST_OFFSET**: returns an integer that is DST_STOP-DST_START, the number of intervals in the first fall DST hour.

- **DST_START**: returns an integer that is the index of the first interval in the first fall DST hour if the handle contains the fall DST transition; else zero.

- **DST_STOP**: returns an integer that is the index of the first interval in the second fall DST hour if the handle contains the fall DST transition; else zero.

- **DSTENERGY**: returns a float that is the sum of the values in the single Fall DST hour, IPH and UOM corrected (but is nonzero only if UOM is 01 or 02). This value is nonzero only if the handle includes 2:00 AM in the Fall DST shift day, and DSTPARTICIPANT is not "Y".

- **DSTPARTICIPANT**: returns a string that is "Y" if this handle includes a Daylight Savings Time transition and is not 24-hour adjusted.

- **DSTTOTAL**: returns a float that is the sum of the values in the single Fall DST hour, not IPH or UOM corrected. This value is nonzero only if the handle includes 2:00 AM in the Fall DST shift day, and DSTPARTICIPANT is not "Y".

- **EDIT_FLAG**: returns 1 if the Edit flag is On (Y); else returns 0.

- **EDIT_FLAG_CHAR**: returns Edit flag; Y or N.

- **END_TIME**: returns an integer that is 0 or the SPI-1, depending on the setting of the "User interval end time..." option.

- **ENERGY**: returns a float that is the sum of all the values, IPH and UOM corrected (but is nonzero only if UOM is 01 or 02).

- **EX_STATUS**: returns the extended status code at a specified index. The <identifier|constant> parameter must be present (containing a one-based integer) to use this option. It is an error if the index is outside the range of the interval data reference. **Note**: Only available when loading data from enhanced interval data tables.

- **EXTERNAL_VAL_FLAG**: returns 1 if the external validation status flag is On (Y); else returns 0.

- **EXTERNAL_VAL_FLAG_CHAR**: returns the external validation status flag; Y or N.

- **INDEX**: returns the value at a specified index. The <identifier|constant> parameter must be present (containing a one-based integer) to use this option. It is an error if the index is outside the range of the interval data reference.

- **INDEX_START**: returns the start date and time of a specified index. The <identifier|constant> parameter must be present (containing a one-based integer) to use this option. It is an error if the index is outside the range of the interval data reference.

- **INDEX_STATUS**: returns the status code at a specified index. The <identifier|constant> parameter must be present (containing a one-based integer) to use this option. It is an error if the index is outside the range of the interval data reference.

- **INDEX_STOP**: returns the stop date and time of a specified index. The <identifier|constant> parameter must be present (containing a one-based integer) to use this option. It is an error if the index is outside the range of the interval data reference.

- **INTERNAL_VAL_FLAG**: returns 1 if the internal validation status flag is On (Y); else returns 0.

- **INTERNAL_VAL_FLAG_CHAR**: returns internal validation status flag; Y or N.

- **IPH**: returns an integer that is the intervals per hour. This is zero if the SPI is greater than 3600.

- **KW_MAXIMUM**: returns a float that is the maximum value, corrected for IPH if the UOM is 01 and IPH is not zero.

- **LF**: The LF attribute returns the load factor, where the average is defined as the total divided by the count of intervals with non-9 status code. See **Interval Data Mask Functions** on page 9-11 for the correct way to compute the load factor on masked handles.

- **MASK_FLAG**: returns 1 if the handle is a mask; else returns 0.

- **MAXDATE**: returns the date and time of the maximum (the first one, if there are more than one with the same value).

- **MAXIMUM**: (or "MAX" or ">=") - returns a float that is the maximum value.

- **MAXINDEX**: returns and integer that is the 1-based index of maximum.

- **MERGE_FLAG**: returns 1 if the Merge flag is On (Y); else returns 0.

- **MERGE_FLAG_CHAR**: returns Merge flag; Y or N.

- **MESSAGE01**, ..., **MESSAGE10**: returns a string that is the value of the corresponding message field.

- **METER_MULT**: returns a float that is the meter multiplier for the readings.

- **METER_OFFSET**: returns a float that is the meter offset for the readings.

- **MINDATE**: returns the date and time of the first minimum.

- **MINIMUM**: (or "MIN" or "<=") - returns a float that is the smallest value.

- **MINIMUM_NZ**: returns a float that is the smallest nonzero value.

- **MULTIPLIER**: returns the pulse multiplier as a float.

- **NEG_VALUES_FLAG**: returns 1 if the handle has negative values; else returns 0.

- **NON_9_VALUE**: returns an integer that is the total number of values that are not missing.

- **OFFSET**: returns the pulse offset as a float.

- **ORIGIN**: returns a string that is one of: "M" = metered, "C" = computed, "P" = profiled, or "S" = statistical.

- **PARENTKEY**: returns a string containing the identity of the parent of the handle. Used only with enhanced interval data tables.

- **POPULATION**: Returns the float equal to the population value. This is used with Statistical records only.

- **QUANTITY_MAX**: returns the maximum if UOM is not a rate; else returns maximum * SPI / Seconds-Per-Unit.

- **QUANTITY_TOTAL**: returns the total if UOM is not a rate; else returns total * SPI / Seconds-Per-Unit.

- **RATE_MAX**: returns the maximum if UOM is not a quantity; else returns maximum * Seconds-Per-Unit / SPI.

- **READING_VALUE**: returns a float that is ((start_reading - stop_reading) * meter_mult) + meter_offset, corrected for rollover (stop_reading < start_reading).

- **REALSTARTTIME**: returns the date and time of the first interval.

- **REALSTOPTIME**: returns the date and time of the last interval.

- **RECORDER**: returns a string that is the recorder ID.

- **RECORDERCHAN**: returns a string that is "recorder,channel".

- **SPI**: returns an integer that is the seconds per interval.

- **SPRING_DST**: returns an integer that is the index of the first interval in the spring DST hour if the handle contains the spring DST transition; else returns 0.

- **START_OFFSET_FLAG**: returns a character; "Y" or "N", depending on whether the start date and time are both present (Oracle Utilities Load Analysis specific).

- **START_READING**: returns a float that is the start reading.

- **STARTTIME**: returns the date and time of the first interval, rounded to the beginning of the interval.

- **STOP_READING**: returns a float that is the stop reading.

- **STOPTIME**: returns the date and time of the last interval, rounded to the end of the interval.

- **TIMESTAMP**: returns date and time the handle was input.

- **TIMEZONES**: returns the number of timezones of the handle dates from GMT (-1 is unknown). Each timezone is 1/2 hour long, so EST is 10 timezones from GMT.

- **TIMEZONES_FLAG**: returns 1 if the timezones flag is On; else returns 0.

- **TZSTDNAME**: returns the Time Zone Standard Name for the handle.

- **TOTAL**: returns a float that is the sum of all the values, not IPH or UOM corrected.

- **UOM**: returns the numeric UOM code as an integer.

- **UOMAGGREGATE**: returns a character that is the UOM Aggregate value: T (Total), M (Maximum), or A (Average).

- **UOMCODE**: returns the UOM code as a string.

- **UOMNAME**: returns the name of the UOM as a string.

- **UOMRATEQUANTITY**: returns a character that indicates whether the UOM is a quantity (Q) or a rate (R). It returns " " if neither.

- **UOMRELATEDUOM**: returns a string that is the UOM code related to this UOM.

- **UOMSECSPERUNIT**: if the UOM is a rate, returns an integer that is the number of seconds per unit; else returns zero.

- **UOMTOTALIZE**: returns a character that is the UOM Totalize value: T (Total), M (Maximum), or A (Average).

- **UOMUNIT**: returns a string that is the unit of the UOM.

- **VAL_FLAG_E**: returns a character that is the value of the validation flag E.

- **VAL_FLAG_I**: returns a character that is the value of the validation flag I.

- **VAL_FLAG_N**: returns a character that is the value of the validation flag N.

- **VAL_FLAG_O**: returns a character that is the value of the validation flag O.

- **WEIGHT**: Returns the float equal to the weight value. Used with Statistical records only.

The default is **TOTAL**. In addition, the second through tenth peaks can be retrieved and reported using the types "MAXIMUM2", "MAXDATE2", "QUANTITY_MAX2", "RATE_MAX2", …, "MAXIMUM10", "MAXDATE10", "QUANTITY_MAX10", "RATE_MAX10".

- <identifier|constant> is only required if the type is **DATE** or **INDEX**. If **DATE**, the value at the specified date and time is retrieved. If the date and time is outside the handle, 0 is returned. If **INDEX**, it must evaluate to a number from 1 to the number of intervals in the handle (COUNT). If the number is less than 1 or greater than COUNT, 0 is returned.

Examples

To get the KWH for a previously loaded handle:

```
KWH = INTDVALUE(KWH_HNDL, "ENERGY");
```

To get the KW for a previously loaded handle:

```
KW = INTDVALUE(KWH_HNDL, "KW_MAXIMUM");
```

STDEV Function

Purpose

The STDEV Function returns the standard deviation from a previously loaded interval data handle. This function can also return the standard deviation for time of use periods within an interval data handle by supplying a Time of Use schedule and optional holiday list.

Format

```
<identifier> = STDEV(<interval_data_reference>[, <schedule_name>[,
<holiday_list_name>]]);
```

Where

- <interval_data_reference> is a reference to a loaded interval data handle

- <schedule_name> *Optional* is the name of a Time-of-Use schedule stored in the Data Repository

- <holiday_list_name> *Optional*. A holiday list to be used with the TOU schedule.

Examples

Get the standard deviation from HNDL.

```
HNDL = INTDLOADDATES ('RECORDER,1", START, STOP);
SD = STDEV (HNDL);
```

Get the standard deviations for OFF_PEAK and ON_PEAK time of use periods from HNDL.

```
HNDL = INTDLOADDATES ('RECORDER,1", START, STOP);
SD = STDEV (HNDL, TOU_SCHEDULE, 2007_HOLIDAYs);
ONPK_SD = SD.ON_PEAK;
OFFPK_SD = SD.OFF_PEAK;
```

> **Note**: In this example, ON_PEAK and OFF_PEAK are time of use periods defined for the TOU_SCHEDULE time of use schedule.

Enhanced Interval Data Functions

This section includes interval data functions that specifically work with interval data stored in Enhanced Interval Data tables. Rules Language functions used with enhanced/generic interval data use a specific set of parameters that differ slightly from other interval data functions. These include:

- **<parent_identity>** - the identity of the parent record. This can be in the form of a string that contains the identity or a database identifier that contains the identity. This is used by all enhanced interval data functions.

- **<parent_stem>** - a stem identifier that contains the parent record, including all required columns. Used by the INTDSAVEEXP function.

- **<table_name>** - the name of the interval data table in which the data is stored. This is used by all enhanced interval data functions.

- **<category>** - the optional category code associated with the interval data. This can be in the form of a string that contains the category or a database identifier that contains the category. This is used by all enhanced interval data functions.

Oracle Utilities Meter Data Management Interval Data

When using these functions with the Oracle Utilities Meter Data Management application, the following apply:

- The <parent_identity> parameter comes from the MDM Meter table, and comprises the meter ID, UOM, and channel number of the meter for which you wish to load/save/delete data, separated by commas.(example: "METER,01,1").

- The <category> parameter comes from the Usage Category.

- The <table_name> parameter is the Meter Data Channel Cut (LSMDMTRDATACUT) table.

- The <identity> parameter used by the INTDLOADEXCUT function comprises the following:

  - Meter ID

  - UOM

  - Channel Number

  - Category Code

  - Start Time

  - Version Sequence

  - **Example**: "METER,01,1, FINAL, 04/01/2006 00:00:00, 2").

INTDDELETEEX Function

Purpose

The INTDDELETEEX Function deletes an interval data cut from a specified Enhanced Interval Data table.

Format

```
<identifier> = INTDDELETEEX(<parent_identity|parent_db_identifier> ,
[<category|category_db_identifier>,][QUAL/<alternate_qualifier>,]
<table_name>, <date_identifier>);
```

Where

- <parent_identity> is a string containing the identity of the parent of the data to be deleted.

- <parent_db_identifier> is a database identifier that contains the database record for the Enhanced Interval Data table from which to delete the data. See **Database Identifiers** on page 4-5 in the *Oracle Utilities Rules Language User's Guide* for more information about using database identifiers.

- <category> (*Optional*) is a string containing the category for the data to be deleted.

- <category_db_identifier> (*Optional*) is a database identifier that contains the database record for the category for the data to be deleted. See **Database Identifiers** on page 4-5 in the *Oracle Utilities Rules Language User's Guide* for more information about using database identifiers.

- <alternate_qualifier> is a string containing the name of an alternate database qualifier containing the interval data to be deleted. When specifying an alternate database qualifier, the <alternate_qualifier> and <table_name> arguments should be concatenated, separated by a comma (Ex: "QUAL/TEST,LSMDMTRDATACUT").

- <table_name> is the name of the Enhanced Interval Data table from which to delete the data.

- <date_identifier> is the start time of the cut to be deleted as stored in the Oracle Utilities Data Repository.

Example

Delete the interval data cut for meter ID 80001, UOM 01, channel 2 with a start time of 06/01/2005 with a category of "FINAL" from the Meter Data Channel Cut (LSMDMTRDATACUT) table.

```
METER = "80001,01,2";
CATEGORY = "FINAL";
TABLE_NAME = "LSMDMTRDATACUT";
DATE = '06/01/2005 00:00:00';
DELETED = INTDDELETEEX(METER, CATEGORY, TABLE_NAME, DATE);
```

Delete data from LSINTERVALDATACUT

```
//Deleting Enhanced Interval Data Tables:   LSINTERVALDATACUT
//Set parameters
IDENTITY = "Cut1,2";
CATEGORY = "Type1,09/01/2006 00:00:00";
TABLE_NAME = "LSINTERVALDATACUT";
STARTTIME = DATE("01/01/2006 00:00:00");
//Delete data
DELETE_HNDL = INTDDELETEEX(IDENTITY, CATEGORY, TABLE_NAME, STARTTIME);
```

INTDGETATTREXALL Function

Purpose

The INTDGETATTREXALL function gets multiple custom and parent attributes of a specified enhanced interval data handle.

This function is used to get multiple custom and parent attributes of an enhanced interval data handle in a single function call. Returns an integer; zero if successful, not zero if an error (for example, if this attribute cannot be modified).

Format:

```
<identifier> = INTDGETATTREXALL(<interval_data_reference>,
<intd_stem>, <parent_stem>);
```

Where

- <interval_data_reference> is a reference to a loaded interval data handle.

- <intd_stem> a stem identifier whose tails will be assigned from custom columns in the loaded handle.

- <parent_stem> - a stem identifier whose tails will be assigned from parent columns in the loaded handle.

Example:

Report the C1 and C2 custom columns, and P1 and P2 parent columns in the #SAVE_AR array.

```
FOR EACH I IN NUMBER 100
    CUSTOM.C1 = "C" + I;
    CUSTOM.C2 = I;
    PARENT.P1 = "Parent" + I;
    PARENT.P2 = "Parent2_" + I;
    RET = INTDSETATTREXALL (#SAVE_AR[I], CUSTOM, PARENT);
END FOR;

FOR EACH I IN NUMBER 100
    RET = INTDGETATTREXALL (#SAVE_AR[I], CUSTOM, PARENT);
    REPORT CUSTOM.C1;
    REPORT CUSTOM.C2;
    REPORT PARENT.P1;
    REPORT PARENT.P2;
END FOR;
```

INTDLOADEXACTUAL Function

Purpose

The INTDLOADEXACTUAL Function loads a specific interval data cut from a specified Enhanced Interval Data table. It loads a specific interval data cut for a given start time. This function loads cuts as stored in the Oracle Utilities Data Repository. Returns an interval data reference.

Format

```
<interval_data_reference> = INTDLOADEXACTUAL(
<parent_identity|parent_db_identifier> ,
[<category|category_db_identifier>,][QUAL/<alternate_qualifier>,]
<table_name>, <date_identifier> );
```

Where

- <parent_identity> is a string containing the identity of the parent of the data to be loaded.

- <parent_db_identifier> is a database identifier that contains the database record for the Enhanced Interval Data table from which to load the data. See **Database Identifiers** on page 4-5 in the *Oracle Utilities Rules Language User's Guide* for more information about using database identifiers.

- <category> (*Optional*) is a string containing the category for the data to be loaded.

- <category_db_identifier> (*Optional*) is a database identifier that contains the database record for the category for the data to be loaded. See **Database Identifiers** on page 4-5 in the *Oracle Utilities Rules Language User's Guide* for more information about using database identifiers.

- <alternate_qualifier> is a string containing the name of a alternate database qualifier containing the interval data to be loaded. When specifying an alternate database qualifier, the <alternate_qualifier> and <table_name> arguments should be concatenated, separated by a comma (Ex: "QUAL/TEST,LSMDMTRDATACUT").

- <table_name> is the name of the Enhanced Interval Data table from which to load the data.

- <date_identifier> (*Optional)* is the start time of the cut as stored in the Oracle Utilities Data Repository. If not supplied, the default value is equal to BILL_START.

Example

Load the interval data cut for meter ID 80001, UOM 01, channel 2 with a start time of 06/01/2005 with a category of "FINAL" from the Meter Data Channel Cut (LSMDMTRDATACUT) table.

```
METER = "80001,01,2";
CATEGORY = "FINAL";
TABLE_NAME = "LSMDMTRDATACUT";
HNDL = INTDLOADEXACTUAL(METER, CATEGORY, TABLE_NAME, "06/01/2005");
```

INTDLOADEXCUT Function

Purpose

The INTDLOADEXCUT Function loads a specific interval data cut from an Enhanced Interval Data table, including versioning tables. It returns an interval data reference.

Format

```
<interval_data_reference> = INTDLOADEXCUT(<identity>, [QUAL/
<alternate_qualifier>,] <table_name> );
```

Where

- <identity> is a string containing the identity of the interval data cut to be loaded. When loading from a versioning table, this must include the parent identity of the interval data cut, plus the start time and version sequence (separated by commas).

- <alternate_qualifier> is a string containing the name of an alternate database qualifier containing the interval data to be loaded. When specifying an alternate database qualifier, the <alternate_qualifier> and <table_name> arguments should be concatenated, separated by a comma (Ex: "QUAL/TEST,LSMDMTRDATACUT").

- <table_name> is the name of the Enhanced Interval Data Versioning table from which to load the data.

Example

Load an interval data cut for meter ID 80001, UOM 01, channel 2 with a start time of 06/01/2005 and a category of "FINAL" from the Meter Data Channel Cut (LSMDMTRDATACUT) table.

```
METER = "80001"
UOM = "01";
CHAN = "1";
CATEGORY = "FINAL";
START = "06/01/2005 00:00:00"
ID = METER + "," + UOM + "," + CHAN + "," + CATEGORY + "," + START;
TABLE_NAME = "LSMDMTRDATACUT";
HNDL = INTDLOADEXCUT(ID, TABLE_NAME);
```

Load the versioned interval data cut for meter ID 80001, UOM 01, channel 2 with a start time of 06/01/2005 and a version sequence of 3 with a category of "FINAL" from the Meter Data Channel Cut Version (LSMDMTRDATACUTV) table.

```
METER = "80001"
UOM = "01";
CHAN = "1";
CATEGORY = "FINAL";
START = "06/01/2005 00:00:00"
SEQ = "3"
ID = METER + "," + UOM + "," + CHAN + "," + CATEGORY + "," + START + ","
+ SEQ;
TABLE_NAME = "LSMDMTRDATACUTV";
HNDL = INTDLOADEXCUT(ID, TABLE_NAME);
```

INTDLOADEXDATES Function

Purpose

The INTDLOADEXDATES Function loads interval data for a user-specified date range from a specified Enhanced Interval Data table. This function is similar to the **INTDLOADEX Function** on page 9-93, except that you specify a parent and date range for the data.

Format

```
<interval_data_reference> =
INTDLOADEXDATES(<parent_identity|parent_db_identifier> ,
[<category|category_db_identifier>,][QUAL/<alternate_qualifier>,]
<table_name>, <date_identifier|date_constant>,
<date_identifier|date_constant>[,<loadflag>][,<tzstd>][,<dst_flag>]);
```

Where

- <parent_identity> is a string containing the identity of the parent of the data to be loaded.

- <parent_db_identifier> is a database identifier that contains the database record for the Enhanced Interval Data table from which to load the data. See **Database Identifiers** on page 4-5 in the *Oracle Utilities Rules Language User's Guide* for more information about using database identifiers.

- <category> (*Optional*) is a string containing the category for the data to be loaded.

- <category_db_identifier> (*Optional*) is a database identifier that contains the database record for the category for the data to be loaded. See **Database Identifiers** on page 4-5 in the *Oracle Utilities Rules Language User's Guide* for more information about using database identifiers.

- <alternate_qualifier> is a string containing the name of an alternate database qualifier containing the interval data to be loaded. When specifying an alternate database qualifier, the <alternate_qualifier> and <table_name> arguments should be concatenated, separated by a comma (Ex: "QUAL/TEST,LSMDMTRDATACUT").

- <table_name> is the name of the Enhanced Interval Data table from which to load the data.

- <date_identifier|date_constant>, <date_identifier|date_constant> are actual start and end dates. Acceptable formats are 'MM/DD/YYYY', 'MM/DD/YYYY HH:MM', and 'MM/DD/YYYY HH:MM:SS'.

  If you do not specify a time, the time defaults to 00:00:00.

- <loadflag> (optional) specifies the behavior of the function when loading cuts with mixed time zones and/or DST Participant flags. This parameter must be an integer, or an identifier or expression that resolves to an integer that is the sum of two values, one that specifies how to resolve mixed time zones (using the Time Zone Standard Name field), and one that specifies how to resolve mixed DST Participant flags.

  - Time Zones:

    - 0: If the Time Zone Standard Names are different, return an error. (Default)

    - 1: Use the first cut's Time Zone Standard Name (TZSN).

    - 2: Use the TZSN supplied in the function call.*

  - DST Participants

    - 4: If the DST Participant flags are different, return an error. (Default)

    - 8: Use the first cut's DST Participant flag.

    - 16: Use the DST Participant flag supplied in the function call.*

*When using **either** the TZSN **or** the DST Participant flag supplied in the function call, use 2 or 16 as normal. To use **both** the TZSN and DST Participant flag supplied in the function call, use 32.

The table below lists the possible combinations of the Time Zone and DST Participant values:

TZSN	DST Participant	Load Flag Parameter
Error if different (0)	Error if different (4)	4
Error if different (0)	From first cut (8)	8
Error if different (0)	From function call (16)	16
From first cut (1)	Error if different (4)	5
From first cut (1)	From first cut (8)	9
From first cut (1)	From function call (16)	17
From function call (2)	Error if different (4)	6
From function call (2)	From first cut (8)	10
From function call (2)*	From function call (16)*	32*

For example, to return an error if either the TZSN or the DST Participant flags are different, you would set this parameter to 4 (0 for TZSN and 4 for DST) This is the default value for this parameter. To use both the TZSN and DST Participant flag from the first cut, you would set this parameter to 9 (1 for TZSN, 8 for DST).

- <tzstd> (optional) is the TZSN for the handle. The supplied value must be one of "EST", "CST", "MST", "PST", or be defined in the LSCALENDAR.XML configuration file (if present). If empty, this is equal to the Time Zone Standard Name of the first cut. The Default value is the default time zone, as specified in the LSCALENDAR.XML file.

- <dst_flag> (optional) is the DST Participant flag for the handle. Must be either "Y" or "N". If empty, this is equal to the DST Participant flag of the first cut. The Default value is "N".

Examples

Load the interval data cut for meter ID 80001, UOM 01, channel 2 with a category of "FINAL" from the Meter Data Channel Cut (LSMDMTRDATACUT) table for the month of January, 2006:

```
METER = "80001,01,2";
CATEGORY = "FINAL";
TABLE_NAME = "LSMDMTRDATACUT";
KWH_HNDL = INTDLOADEXDATES(METER, CATEGORY, TABLE_NAME, '01/01/2006',
'01/31/2006 23:59:59');
```

OR

```
STARTDT = '01/01/2006';
STOPDT = '01/31/2006 23:59:59';
METER = "80001,01,2";
CATEGORY = "FINAL";
TABLE_NAME = "LSMDMTRDATACUT";
KWH_HNDL = INTDLOADEXDATES(METER, CATEGORY, TABLE_NAME, STARTDT,
STOPDT);
```

Load the interval data cut for meter ID 80001, UOM 01, channel 2 from the Meter Data Channel Cut (LSMDMTRDATACUT) table for the month of January, 2006, and use the supplied Time Zone Standard Name and DST Participant flag.

```
STARTDT = '01/01/2006';
STOPDT = '01/31/2006 23:59:59';
METER = "80001,01,2";
TABLE_NAME = "LSMDMTRDATACUT";
KWH_HNDL = INTDLOADEXDATES(METER, TABLE_NAME, STARTDT, STOPDT, 34,
"EST", "N");
```

Load interval data from the LSINTERVALDATACUT table.

```
//Loading Enhanced Interval Data Tables:  LSINTERVALDATACUT,
//Set parameters
IDENTITY = "Cut1,2";
CATEGORY = "Type1,09/01/2006 00:00:00";
TABLE_NAME = "LSINTERVALDATACUT";
STARTTIME = DATE("01/01/2006 00:00:00");
STOPTIME = DATE("01/05/2006 23:59:59");
//Load HNDL
LOAD_HNDL = INTDLOADEXDATES(IDENTITY , CATEGORY , TABLE_NAME ,
STARTTIME , STOPTIME);
```

INTDLOADEX Function

Purpose

The INTDLOADEX Function loads and totalizes an account's interval data for a user-specified determinant for the current bill period from a specified Enhanced Interval Data table. It looks up the unit of measure (UOM) associated with the bill determinant in the Bill Determinant table. Then, it looks at the Meter Configuration table to determine which of the meters that belong to the account currently being billed collect data in that UOM. This function can only be used with Oracle Utilities Billing Component and Oracle Utilities Meter Data Management.

Format

```
<interval_data_reference> = INTDLOADEX(<determinant_identifer>,
[<category|category_db_identifier>,][QUAL/<alternate_qualifier>,]
<table_name>);
```

Where

- <determinant_identifer> is an identifier for a billing determinant, as defined in the Bill Determinant table. Its UOM is retrieved, and the matching meters for the account are totalized to get the interval data cut.

- <category> (*Optional*) is a string containing the category for the data to be loaded.

- <category_db_identifier> (*Optional*) is a database identifier that contains the database record for the category for the data to be loaded. See **Database Identifiers** on page 4-5 in the *Oracle Utilities Rules Language User's Guide* for more information about using database identifiers.

- <alternate_qualifier> is a string containing the name of a alternate database qualifier containing the interval data to be loaded. When specifying an alternate database qualifier, the <alternate_qualifier> and <table_name> arguments should be concatenated, separated by a comma (Ex: "QUAL/TEST,LSMDMTRDATACUT").

- <table_name> is the name of the Enhanced Interval Data table from which to load the data.

Examples

Load interval data cuts based on KW and KVAR with a category of "FINAL" from the Meter Data Channel Cut (LSMDMTRDATACUT) table.

```
CATEGORY = "FINAL";
TABLE_NAME = "LSMDMTRDATACUT";
INT_KW_HNDL = INTDLOADEX(KW, CATEGORY, TABLE_NAME);
INT_KVAR_HNDL =INTDLOADEX(KVAR, CATEGORY, TABLE_NAME);
```

INTDLOADEXLIST Function

Purpose

Totalizes the interval data stored in an enhanced interval data table for the current bill period for all parent records in a list.

This function totalizes the interval data for all parent records in a TABLE.COLUMN list. (See the *Data Manager User's Guide* for information about creating TABLE.COLUMN lists.) The list must consist of a list of unique identifiers for the parents.

Format

```
<interval_data_reference> = INTDLOADEXLIST([QUAL/<alternate_qualifier>,]
<table_name>, <list_identifier|list_name>,
[<category|category_db_identifier>]);
```

Where

- <alternate_qualifier> is a string containing the name of an alternate database qualifier containing the interval data to be loaded. When specifying an alternate database qualifier, the <alternate_qualifier> and <table_name> arguments should be concatenated, separated by a comma (Ex: "QUAL/TEST,LSMDMTRDATACUT").

- <table_name> is the name of the Enhanced Interval Data table from which to load the data.

- <list_identifier> is an identifier that contains the name of a list of parent records.

- <list_name> is a literal constant of the form "listname", that identifies a list of parent records.

- <category> is a string containing the category for the data to be loaded.

- <category_db_identifier> (*Optional*) is a database identifier that contains the database record for the category for the data to be loaded. See **Database Identifiers** on page 4-5 in the *Oracle Utilities Rules Language User's Guide* for more information about using database identifiers.

Example

Load all interval data for all weather stations returned by the "GET_TX_WEATHER_STATIONS" list (includes all weather stations with a Jurisdiction of TEXAS) with a category of "FINAL."

```
TABLE_NAME = "LSWEATHERDATA";
LIST_NAME = "GET_TX_WEATHER_STATIONS";
CATEGORY = "FINAL";
LST_HNDL = INTDLOADEXLIST (TABLE_NAME, LIST_NAME, CATEGORY);
```

INTDLOADEXLISTDATES Function

Purpose

The INTDLOADEXLISTDATES function totalizes the interval data stored in an enhanced interval data table for all parent records in a list over a specified time range.

This function totalizes the interval data for all parent records in a TABLE.COLUMN list. (See the *Data Manager User's Guide* for information about creating TABLE.COLUMN lists.) The list must consist of a list of unique identifiers for the parents.

Format

```
<interval_data_reference> = INTDLOADEXLISTDATES([QUAL/
<alternate_qualifier>,] <table_name>, <list_identifier|list_name>,
[<category|category_db_identifier>,]<date_identifier|date_constant>,
<date_identifier|date_constant>);
```

Where

- <alternate_qualifier> is a string containing the name of an alternate database qualifier containing the interval data to be loaded. When specifying an alternate database qualifier, the <alternate_qualifier> and <table_name> arguments should be concatenated, separated by a comma (Ex: "QUAL/TEST,LSMDMTRDATACUT").

- <table_name> is the name of the Enhanced Interval Data table from which to load the data.

- <list_identifier> is an identifier that contains the name of a list of parent records.

- <list_name> is a literal constant of the form "listname", that identifies a list of parent records.

- <category> is a string containing the category for the data to be loaded.

- <category_db_identifier> (*Optional*) is a database identifier that contains the database record for the category for the data to be loaded. See **Database Identifiers** on page 4-5 in the *Oracle Utilities Rules Language User's Guide* for more information about using database identifiers.

- <date_identifier|date_constant>, <date_identifier|date_constant> are actual start and end dates. Acceptable formats are 'MM/DD/YYYY', 'MM/DD/YYYY HH:MM', and 'MM/DD/YYYY HH:MM:SS'. If you do not specify a time, the time defaults to 00:00:00.

Example

Load all interval data for all weather stations returned by the "GET_TX_WEATHER_STATIONS" list (includes all weather stations with a Jurisdiction of TEXAS) with a category of "FINAL" for the month of January 2007.

```
TABLE_NAME = "LSWEATHERDATA";
LIST_NAME = "GET_TX_WEATHER_STATIONS";
CATEGORY = "FINAL";
START = DATE ('01/01/2007 00:00:00');
STOP = DATE ('01/31/2007 23:59:59');
LST_HNDL = INTDLOADEXLISTDATES (TABLE_NAME, LIST_NAME, CATEGORY,
START, STOP);
```

INTDLOADEXRELATEDCHANNEL Function

Purpose

The INTDLOADEXRELATEDCHANNEL Function loads the interval data for the related meter specified in the MDM Meter table from a specified Enhanced Interval Data table. It loads the interval data for the meter related to the interval data reference's meter through the MDM Meter table. This function is used with Oracle Utilities Meter Data Management ONLY. Returns an interval data reference.

Format

```
<identifier> = INTDLOADEXRELATEDCHANNEL
(<parent_identity|parent_db_identifier> ,
[<category|category_db_identifier>,][QUAL/<alternate_qualifier>,]
<table_name>, <date_identifier|date_constant>,
<date_identifier|date_constant> );
```

Where

- <parent_identity> is a string containing the identity of the parent of the meter to which the data to be loaded is related.

- <parent_db_identifier> is a database identifier that contains the database record for the Enhanced Interval Data table from which to load the data. See **Database Identifiers** on page 4-5 in the *Oracle Utilities Rules Language User's Guide* for more information about using database identifiers.

- <category> (*Optional*) is a string containing the category for the data to be loaded.

- <category_db_identifier> (*Optional*) is a database identifier that contains the database record for the category for the data to be loaded. See **Database Identifiers** on page 4-5 in the *Oracle Utilities Rules Language User's Guide* for more information about using database identifiers.

- <alternate_qualifier> is a string containing the name of an alternate database qualifier containing the interval data to be loaded. When specifying an alternate database qualifier, the <alternate_qualifier> and <table_name> arguments should be concatenated, separated by a comma (Ex: "QUAL/TEST,LSMDMTRDATACUT").

- <table_name> is the name of the Enhanced Interval Data table from which to load the data.

- <date_identifier|date_constant>, <date_identifier|date_constant> are actual start and end dates. Acceptable formats are 'MM/DD/YYYY', 'MM/DD/YYYY HH:MM', and 'MM/DD/YYYY HH:MM:SS'. If you do not specify a time, the time defaults to 00:00:00.

Example

Load related interval data for for January 2006 for meter ID 80001, UOM 01, channel 2.

```
STARTDT = '01/01/2006';
STOPDT = '01/31/2006 23:59:59';
METER = "80001,01,2";
CATEGORY = "FINAL";
TABLE_NAME = "LSMDMTRDATACUT";
INT_HNDL = INTDLOADEXRELATEDCHANNEL(METER, CATEGORY, TABLE_NAME,
STARTDT, STOPDT);
```

INTDSAVEEX Function

Purpose

The INTDSAVEEX Function saves an interval data handle to a specified Enhanced Interval Data table. It can save individual handles, or can save multiple handles with a single function call (known as "bulk saves").

Format

```
<identifier> = INTDSAVEEX( (<parent_identity|parent_db_identifier> ,
[<category|category_db_identifier>,][QUAL/<alternate_qualifier>,]
<table_name>, <intd_hndl|hndl_array> );
```

Where

- <parent_identity> is a string containing the identity of the parent of the data to be saved.

- <parent_db_identifier> is a database identifier that contains the database record for the Enhanced Interval Data table to which the data is to be saved. See **Database Identifiers** on page 4-5 in the *Oracle Utilities Rules Language User's Guide* for more information about using database identifiers.

    Note: When performing bulk saves with this function, this parameter should be an empty string (""). In this case, the parent identity is derived from the <hndl_array> parameter.

- <category> (*Optional*) is a string containing the category for the data to be saved.

- <category_db_identifier> (*Optional*) is a database identifier that contains the database record for the category for the data to be saved. See **Database Identifiers** on page 4-5 in the *Oracle Utilities Rules Language User's Guide* for more information about using database identifiers.

    Note: Even if a category is not used, the <category> parameter is required when using this function. In this case, specify an empty string ("").

- <alternate_qualifier> is a string containing the name of an alternate database qualifier containing the table where the interval data is to be saved. When specifying an alternate database qualifier, the <alternate_qualifier> and <table_name> arguments should be concatenated, separated by a comma (Ex: "QUAL/TEST,LSMDMTRDATACUT").

- <table_name> is the name of the Enhanced Interval Data table to which to the data is to be saved.

- <intd_hndl> is a reference to the loaded interval data handle to be saved.

- <hndl_array> is an array identifier that contains multiple handles to be saved. This is used when performing bulk saves.

Examples

Save the interval data in KWH_HNDL to meter ID 80001, UOM 01, channel 2 with a with a category of "FINAL" to the Meter Data Channel Cut (LSMDMTRDATACUT) table.

```
METER = "80001,01,2";
CATEGORY = "FINAL";
TABLE_NAME = "LSMDMTRDATACUT";
SAVE_HNDL = INTDSAVEEX(METER, CATEGORY, TABLE_NAME, KWH_HNDL);
```

Save the interval data in LOAD_HNDL to the LSINTERVALDATACUT table

```
//Saving Enhanced Interval Data Tables:   LSINTERVALDATACUT
IDENTITY = "Cut1,2";
CATEGORY = "Type1,09/01/2006 00:00:00";
TABLE_NAME = "LSINTERVALDATACUT";
LOAD_HNDL.READSTATUSCODE = "VALID";
```

```
SHIFT_STARTTIME = DATE("09/10/2006 00:00:00");
SHIFT_WEEK = INTDSHIFTSTARTTIME(LOAD_HNDL , "STARTTIME" ,
SHIFT_STARTTIME);
//Save shifted HNDL
SAVE_HNDL = INTDSAVEEX(IDENTITY , CATEGORY , TABLE_NAME , SHIFT_WEEK);
```

When saving existing interval data loaded from standard interval data tables (such as Channel Cut Header), there are additional steps required. A parent record, if it is not already in the database, must be created before the data can be saved. Once the parent record is in place, the loaded interval data can be saved using the INTDSAVEEX function.

```
//Load data to be saved from LSCHANNELCUTHEADER
IDENTITY_OLD = "000004_COM_DP,100";
SAVE_OLD_HNDL = INTDLOADDATES(IDENTITY_OLD , DATE("07/16/2002
00:00:00") , DATE("07/16/2002 23:59:59"));
//
//Save meter info to parent LSINTERVALDATA first if not in that table
LSINTD.CUTNAME = "000004_COM_DP";
LSINTD.CUTNUM = "100";
LSINTD.DATA_ID = "TEST";
LSINTD.BILLDETERMCODE = "1";
SAVE LSINTD TO TABLE "LSINTERVALDATA";
//
//Save data to new table
CATEGORY = "Type1,09/01/2006 00:00:00";
TABLE_NAME = "LSINTERVALDATACUT";
SAVE_HNDL = INTDSAVEEX(IDENTITY_OLD , CATEGORY , TABLE_NAME ,
SAVE_OLD_HNDL);
```

INTDSAVEEXP Function

Purpose

The INTDSAVEEXP function saves an interval data handle and its parent to specified Enhanced Interval Data tables.

This function can save individual handles/parents, or can save multiple handles/parents with a single function call (known as "bulk saves").

Format

```
<identifier> = INTDSAVEEXP( (<parent_stem|parent_array> ,
[<category|category_db_identifier|<category_array>,][QUAL/
<alternate_qualifier>,] <table_name>, <intd_hndl|hndl_array> );
```

Where

- <parent_stem> is a stem identifier containing the parent record of the data to be saved. This stem must contain all required columns for the parent record, not only those used by the identity.

- <parent_array> is an array identifier containing the parent records that correspond to the handles to be saved.

- <category> (*Optional*) is a string containing the category for the data to be saved.

- <category_db_identifier> (*Optional*) is a database identifier that contains the database record for the category for the data to be saved. See **Database Identifiers** on page 4-5 in the *Oracle Utilities Rules Language User's Guide* for more information about using database identifiers.

- <category_array> (*Optional*) is an array identifier containing categories that correspond to the handles to be saved.

- **Note**: Even if a category is not used, the <category> parameter is required when using this function. In this case, specify an empty string ("").

- <alternate_qualifier> is a string containing the name of a alternate database qualifier containing the table where the interval data is to be saved. When specifying an alternate database qualifier, the <alternate_qualifier> and <table_name> arguments should be concatenated, separated by a comma (Ex: "QUAL/TEST,LSMDMTRDATACUT").

- <table_name> is the name of the Enhanced Interval Data table to which to the data is to be saved.

- <intd_hndl> is a reference to the loaded interval data handle to be saved.

- <hndl_array> is an array identifier that contains multiple handles to be saved. This is used when performing bulk saves.

Examples

Save the interval data in KWH_HNDL to meter ID 45678, UOM 01, channel 2 with a with a category of "FINAL" to the Meter Data Channel Cut (LSMDMTRDATACUT) table.

```
MTR.METERID = "45678";
MTR.EXPECTEDUOMCODE = "01";
MTR.CHANNELID = "2";
CATEGORY = "FINAL";
TABLE_NAME = "LSMDMTRDATACUT";
SAVE_HNDL = INTDSAVEEXP(MTR, CATEGORY, TABLE_NAME, KWH_HNDL);
```

Save the interval data in LOAD_HNDL to the LSINTERVALDATACUT table

```
//Saving Enhanced Interval Data Tables:   LSINTERVALDATACUT
ID.CUTNAME = "CUT1";
ID.CUTNUMBER = "2";
CATEGORY = "Type1,09/01/2006 00:00:00";
TABLE_NAME = "LSINTERVALDATACUT";
LOAD_HNDL.READSTATUSCODE = "VALID";
SHIFT_STARTTIME = DATE("09/10/2006 00:00:00");
SHIFT_HNDL = INTDSHIFTSTARTTIME(LOAD_HNDL , "STARTTIME" ,
SHIFT_STARTTIME);
//Save shifted HNDL
SAVE_HNDL = INTDSAVEEXP(ID , CATEGORY , TABLE_NAME , SHIFT_HNDL);
```

Load data for weather stations WS_1 through WS_30 in Texas (TEXAS) that have a category of "INITIAL" for the month of June 2007 and save them to new weather stations WSN_1 through WSN_30 with a category of "FINAL."

```
TABLE_NAME = "LSWEATHERDATA";
START = DATE ('06/01/2007 00:00:00');
STOP = DATE ('06/30/2007 23:59:59');
FOR EACH REC IN LIST "GET_WS_DATA"
   X = X + 1;
   WS = REC.STATIONCODE;
   JURIS = REC.JURISCODE;
   WEATHER_STATION = WS + "," + JURIS;
   #ARR[X] = INTDLOADEXDATES (WEATHER_STATION, "INITIAL", TABLE_NAME,
START, STOP);
   WST = "WSN_" + X;
   PARENT_KEY = WST + "," + JURIS
   #PAR[X] = PARENT_KEY;
   CATEGORY = "FINAL";
   SET_PK = INTDSETATTRIBUTE (#ARR[X], "PARENTKEY", PARENT_KEY);
END FOR;
SAVE_HNDL = INTDSAVEEXP(#PAR[], CATEGORY, TABLE_NAME, #ARR[]);
```

INTDSETATTREX Function

Purpose

The INTDSETATTREX function sets an attribute of a specified enhanced interval data handle.

This function is used to set attributes of enhanced interval data handles. This function is similar to the INTDSETATTRIBUTE function. Returns an integer; zero if successful, not zero if an error (for example, if this attribute cannot be modified).

Format

```
<identifier> = INTDSETATTREX(<interval_data_reference>, <attribute>,
<identifier|expression>[,<field_type>]);
```

Where

- <interval_data_reference> is a reference to a loaded interval data handle.

- <attribute> is the custom attribute to be set.

- <identifier|expression> is either an identifier or an expression that sets the values of the attribute. If an identifier, it must have been assigned earlier in the rate form. The type of identifier or expression must match the return type listed above.

- <field_type> is either an identifier or an expression that sets the type of attribute to be set. Valid values are "P" (parent) or "C" (custom). The default is "C."

Examples

Set the Related Profile (REL_PRF) of 'HNDL' to RES_PROFILE_1.

```
HNDL_PF = INTDSETATTREX(HNDL, REL_PRF, "RES_PROFILE_1");
```

OR

```
PROFILE = "RES_PROFILE_1";
HNDL_PF = INTDSETATTRIBUTE(HNDL, REL_RPF, PROFILE);
```

INTDSETATTREXALL Function

Purpose

The INTDSETATTREXALL function sets multiple custom and parent attributes of a specified enhanced interval data handle.

This function is used to set multiple custom and parent attributes of an enhanced interval data handle in a single function call. This function is similar to the INTDSETATTRIBUTE and INTDSETATTREX functions. Returns an integer; zero if successful, not zero if an error (for example, if this attribute cannot be modified).

Format

```
<identifier> = INTDSETATTREXALL(<interval_data_reference>,
<intd_stem>, <parent_stem>);
```

Where

- <interval_data_reference> is a reference to a loaded interval data handle.

- <intd_stem> a stem identifier that contains custom colums.

- <parent_stem> - a stem identifier that contains the parent record, including all required columns.

Example

Set the C1 and C2 custom columns, and P1 and P2 parent columns in the #SAVE_AR array.

```
FOR EACH I IN NUMBER 100
    CUSTOM.C1 = "C" + I;
    CUSTOM.C2 = I;
    PARENT.P1 = "Parent" + I;
    PARENT.P2 = "Parent2_" + I;
    RET = INTDSETATTREXALL (#SAVE_AR[I], CUSTOM, PARENT);
END FOR;
```

INTDVALUEEX Function

Purpose

The INTDVALUEEX function returns an attribute of a specified enhanced interval data handle.

This function is used to return attribute values of enhanced interval data handles. This function is similar to the INTDVALUE function.

Format

```
<identifier> = INTDVALUEEX(<interval_data_reference>,
<type>[,<field_type>);
```

Where

- <interval_data_reference> is a reference to a loaded interval data handle.

- <type> is the name of the custom attribute to be returned

- <field_type> is either an identifier or an expression that sets the type of attribute to be set. Valid values are "P" (parent) or "C" (custom). The default is "C."

Example

Get the Related Profile (REL_PRF) for a previously loaded handle:

```
PROFILE = INTDVALUEEX(HNDL, "REL_PRF");
```

Enhanced Interval Data Functional Differences

The following table lists each of the Enhanced Interval Data functions, its corresponding standard function, and a brief description of the differences between the two.

Enhanced Interval Data Function	Standard Interval Data Function	Difference
INTDDELETEEX	INTDDELETE	Enhanced function deletes data from enhanced interval data tables.
INTDLOADEX	INTDLOAD	Enhanced function loads data from specified enhanced interval data table.
INTDLOADEXACTUAL	INTDLOADACTUALCUT	Enhanced function loads data from specified enhanced interval data table.
INTDLOADEXCUT	INTDLOADVERSION	Enhanced function loads data from specified enhanced interval data version table.
INTDLOADEXDATES	INTDLOADDATES	Enhanced function loads data from specified enhanced interval data table.
INTDLOADEXLIST	INTDLOADLIST	Enhanced function loads data from specified enhanced interval data table.
INTDLOADEXLISTDATES	INTDLOADLISTDATES	Enhanced function loads data from specified enhanced interval data table.
INTDLOADEXRELATEDCHANNEL	INTDLOADRELATEDCHANNEL	Enhanced function loads data from specified enhanced interval data table.
INTDSETATTREX	INTDSETATTRIBUTE	Enhanced function sets attributes for enhanced interval data handle. Supports attributes based on Required, Optional, and Custom columns.*
INTDVALUEEX	INTDVALUE	Enhanced function gets values from enhanced interval data handles. Supports values based on Required, Optional, and Custom columns.*
INTSAVEEX	SAVE TO CHANNEL	Enhanced function saves interval data handle to an enhanced interval data table.
INTDSAVEEXP	SAVE TO CHANNEL	Enhanced function saves interval data handle and it's corresponding parent record to an enhanced interval data table.

* Available in the Patch 9 release, scheduled for August 10, 2007.

Interval Data Functions and Enhanced Interval Data Handles

The following table lists the interval data functions not listed above, and whether or not the function can be used with enhanced interval data.

Other Interval Data Functions	Applicable to Enhanced Interval Data
INTDADDATTRIBUTE	Yes
INTDADDVMSG	Yes
INTDBLOCKOP	Yes
INTDBLOCKOPNA	Yes
INTDCLOSE	N/A
INTDCOUNT	Yes
INTDCOUNTSTATUSCODE	Yes
INTDCREATEDAYMASK	Yes
INTDCREATEFACTORMASK	Yes
INTDCREATEHANDLE	Yes. Handles created using this function can be saved to an enhanced interval data table. Note that attributes based on required columns must be set using the INTDSETATTREX function prior to saving.
INTDCREATEMASK	Yes
INTDCREATEOVERRIDEDAYMASK	Yes
INTDCREATEOVERRIDEMASK	Yes
INTDCREATESTATUSCODEMASK	Yes
INTDCREATETOUPERIOD	Yes
INTDDIPTEST	Yes
INTDEXPORT	Yes. When enhanced interval data handles are exported to XML format, they are exported to the Compact XML format.
INTDGETERRORCODE	Yes
INTDGETERRORMESSAGE	Yes
INTDISEQUAL	Yes
INTDJOIN	Yes
INTDLOADHIST	N/A
INTDLOADLISTENERGY	N/A
INTDLOADLISTHIST	N/A
INTDLOADSP	N/A
INTDLOADSTAGING	N/A
INTDLOADUOM	N/A

INTDLOADUOMDATES	N/A
INTDLOADUOMHIST	N/A
INTDOPEN	N/A
INTDREADFIRST	N/A
INTDREADNEXT	N/A
INTDRECCOUNT	N/A
INTDRELEASE	Yes
INTDREPLACE	Yes
INTDROLLAVG	Yes
INTDROLLPEAK	Yes
INTDSCALAROP	Yes
INTDSCALE	Yes
INTDSETDSTPARTICIPANT	Yes
INTDSETSTRING	Yes
INTDSETVALUE	Yes
INTDSETVALUESTATUS	Yes
INTDSHIFTSTARTTIME	Yes
INTDSMOOTH	Yes
INTDSORT	Yes
INTDSPIKETEST	Yes
INTDSUBSET	Yes
INTDTOU	Yes
INTDTOURELEASE	Yes
INTDTOUVALUE	Yes
INTDUPDATESTATS	Yes
STDEV	Yes

Chapter 10

Meter Value Function Descriptions

This chapter describes all of the meter value functions available with the Oracle Utilities Rules Language.

Meter Value Functions

MVLOAD Function

Purpose

Loads and totalizes meter values for a specified bill determinant and (optionally) billing entity for the current bill period.

The Meter Value Table is used to store bill determinant values for account "sub-entities"—CIS accounts, individual channels, or channel groups that belong to a parent account. The MVLOAD function loads and totalizes either all of an account's values for a specified bill determinant that are stored in the Meter Value Table for the current bill period, or just those bill determinant values that belong to one of the account's sub-entities (that is, a particular CIS account, channel, or channel group) for the current bill period.

Format

```
<stem_identifier> = MVLOAD(<determinant_identifier>
[,<name_identifier|name_literal>]);
```

Where

- <determinant_identifier> is a billing determinant identifier defined in the BILLDETERMINANT Table. The matching meter value records under the account are totalized to get the determinant value.

- <name_identifier | name_literal> *(Optional)* is the name of the account "sub-entity" in the Meter Value Table whose bill determinant values you want to load and totalize.

Note: The <stem_identifier> on the left side of the equal sign is the name you assign to the temporary record created by the function (see "About records loaded by the MVLOADxxx functions," below).

Examples

Load and totalize all KW data stored in the Meter Value Table for the account for the current bill period:

```
MV = MVLOAD(KW);
```

Load and totalize the KW data stored in the Meter Value Table for the channel group named "CHGRP" for the current bill period:

```
MV = MVLOAD(KW, "CHGRP");
```

About records loaded by the MVLOAD*xxx* functions

The following applies to all MVLOADxxx functions.

When the billing program executes the MVLOAD function, it automatically creates a temporary record that includes the desired bill determinant value, as well as other information from the Meter Value Table. The temporary record includes the scheduled read date, the earliest start time among totaled values, the latest stop time among totaled values, and a string value. Which of these values are loaded depends upon the function. When loaded, these records are automatically assigned the identifier *stem.component,* where "stem" is the identifier for the entire temporary record and *component* refers to an individual field in the record. See **Record Identifiers (stem.component)** on page 4-14 of the *Oracle Utilities Rules Language User's Guide* for more information. You assign the "stem" by supplying an identifier on the left side of the ASSIGNMENT Statement that contains the function*;* the system automatically assigns the names to the components, as shown:

*   **NAME**: The recorder-ID/channel, CIS account, or group name this data is for.

*   **VAL**: Value, or the accumulated values, if loaded.

*   **READDATE**: Scheduled read date. This ties to the Billing Cycle Read Date and identifies the Billing Cycle this data is for.

*   **STARTTIME**: Earliest start time of any totaled value.

*   **STOPTIME**: Last stop time of any totaled value.

*   **STRVAL**: Its string value (only assigned if there is one meter value; they are not totaled).

*   **CODE**: The bill determinant code for the data represented in the record, such as kW or kWh.

When they have been loaded, you can apply other statements in the rate form to these record values by using <stem.component> to identify them. If you use MV as the identifier on the left side of the statement (as shown in the preceding examples), you would use the identifier MV.NAME to refer to the value in the temporary record's NAME field.

Each record identifier may have historical values, depending on the function.

MVLOADACCT Function

Purpose

The MVLOADACCT Function loads and totalizes meter values for a specified bill determinant and (optionally) billing entity for a specified account (typically the SYSTEM) for the current bill period.

This function is similar to the **MVLOAD Function** on page 10-2, except that it enables you to load bill determinant values for a specific account other than the one being billed. This is typically used to access information about the entire utility System, which is stored in the Meter Value Table under an "account" named SYSTEM. The "account's" records selected for the current bill period are those whose READDATE is in the billed account's bill period (may not match the current account's read date).

Format

```
<stem_identifier> = MVLOADACCT(<determinant_identifier>, <accountid>
[,<name_identifier|name_literal>]);
```

Where

- <determinant_identifier> is a billing determinant identifier, as defined in the BILLDETERMINANT Table. The matching meter value records under the "account" are totalized to get the determinant value.

- <accountid> identifies the account whose values you want to load and totalize, typically SYSTEM.

- <name_identifier | name_literal> *(Optional)* is the name of the account "sub-entity" in the Meter Value Table whose bill determinant values you want to load and totalize. This is typically the name of values of interest, such as ACTUAL_PEAK or BILLED_PEAK.

The <stem_identifier> on the left side of the equal sign is the name you assign to the temporary record created by the function. See **About records loaded by the MVLOADxxx functions** on page 10-3 for more information.

Example

Get the actual system peak (value, start and stop times) for the current bill period, then get the interval value at that time from previously loaded current account's cut in KW_HNDL:

```
SYSPEAK = MVLOADACCT(KW, "SYSTEM", "ACTUAL_PEAK");
VALUE_AT_PEAK = INTDVALUE(KW_HNDL, "DATE", SYSPEAK.STARTIME);
```

MVLOADACCTDATES Function

Purpose

The MVLOADACCTDATES Function loads and totalizes meter values for a specified bill determinant and (optionally) billing entity for a specified account (typically the System) for a user-specified time period. This function is identical to the **MVLOADACCT Function** on page 10-4, except that it loads and totalizes bill determinant values for a user-specified time period.

Format

```
<stem_identifier> = MVLOADACCTDATES(<determinant_identifier>,
<accountid> [,<name_identifier|name_literal>],
<date_identifier|date_constant>, <date_identifier|date_constant>);
```

Where

- <determinant_identifier> is a billing determinant identifier defined in the BILLDETERMINANT Table. The matching meter value records under the "account" are totalized to get the determinant value.

- <accountid> identifies the account whose values you want to load and totalize, typically SYSTEM.

- <name_identifier | name_literal> *(Optional)* is the name of the account "sub-entity" in the Meter Value Table whose bill determinant values you want to load and totalize. This is typically the name of values of interest, such as ACTUAL_PEAK or BILLED_PEAK.

- <date_identifier|date_constant>, <date_identifier|date_constant> are actual start and end dates. You can use the following formats: 'MM/DD/YYYY' , 'MM/DD/YYYY HH:MM' or 'MM/DD/YYYY HH:MM:SS'.

The <stem_identifier> on the left side of the equal sign is the name you assign to the temporary record created by the function. See **About records loaded by the MVLOADxxx functions** on page 10-3 for more information.

Example

Get the actual system peak (value, start and stop times) for the period between BILL_START and BILL_STOP, then get the interval value at that time from previously loaded current account's cut in KW_HNDL:

```
SYSPEAK = MVLOADACCT(KW, "SYSTEM", "ACTUAL_PEAK", BILL_START,
BILL_STOP);
VALUE_AT_PEAK = INTDVALUE(KW_HNDL, "DATE", SYSPEAK.STARTIME);
```

MVLOADACCTHIST Function

Purpose

The MVLOADACCTHIST Function loads and totalizes meter values for a specified bill determinant for a specified account (typically the SYSTEM) for a user-specified set of bill periods. This function is identical to the **MVLOADACCT Function** on page 10-4, except that you can specify a desired set of billing periods.

Format

```
<stem_identifier> = MVLOADACCTHIST(<determinant_identifier>, <accountid>
[,<name_identifier|name_literal>], <start_bill_period_previous>,
<end_bill_period_previous>);
```

Where

- <determinant_identifier> is a billing determinant identifier defined in the BILLDETERMINANT Table. The matching meter value records under the "account" are totalized to get the determinant value.

- <accountid> identifies the account whose values you want to load and totalize, typically SYSTEM.

- <name_identifier | name_literal> *(Optional)* is the name of the account "sub-entity" in the Meter Value Table whose bill determinant values you want to load and totalize. This is typically the name of values of interest, such as ACTUAL_PEAK or BILLED_PEAK.

- <start_bill_period_previous>, <end_bill_period_previous> specify the bill periods to be loaded using the following convention: **0** is the current bill period, **1** is the previous bill period, and so on (the higher the number, the further back in time). The end_bill_period_previous must be greater than or equal to the start_bill_period_previous. The default start_bill_period_previous is 0 or the current period. The default end_bill_period_previous is the last period of data available for the account. If you specify a start but no end, the default end is the last period of data for that determinant stored for the account. (See "Rules for Using Functions" in *Chapter Six* of the *Oracle Utilities Rules Language User's Guide* for additional details about specifying bill period parameters.)

  If you omit both start and end bill period parameters from MVLOADACCTHIST, it is the same as MVLOADACCT—only the data for the current bill period is loaded.

The <stem_identifier> on the left side of the equal sign is the name you assign to the temporary record created by the function. See **About records loaded by the MVLOADxxx functions** on page 10-3 for more information.

Example

Get the actual system peak (value, start and stop times) for the past 2 bill periods, then get the interval value at that time from the previously loaded current account's cut in KW_HNDL:

```
SYSPEAK = MVLOADACCTHIST(KW, "SYSTEM", "ACTUAL_PEAK" 1, 2);
VALUE_AT_PEAK = INTDVALUE(KW_HNDL, "DATE", SYSPEAK.STARTIME);
```

MVLOADDATES Function

Purpose

The MVLOADDATES Function loads and totalizes meter values for a specified bill determinant and (optionally) billing entity for a user-specified time period. This function is identical to the **MVLOAD Function** on page 10-2, except that you can specify a date range for the data.

Format

```
<stem_identifier> = MVLOADDATES (<determinant_identifier>
[,<name_identifier|name_literal>], <date_identifier|date_constant>,
<date_identifier|date_constant>);
```

Where

- <determinant_identifier> is a billing determinant identifier defined in the BILLDETERMINANT Table. The matching meter value records under the account are totalized to get the determinant value.

- <name_identifier | name_literal> *(Optional)* is the name of the account "sub-entity" in the Meter Value Table whose bill determinant values you want to load and totalize.

- <date_identifier|date_constant>, <date_identifier|date_constant> are actual start and end dates. You can use the following formats: 'MM/DD/YYYY', 'MM/DD/YYYY HH:MM', or 'MM/DD/YYYY HH:MM:SS'.

The <stem_identifier> on the left side of the equal sign is the name you assign to the temporary record created by the function. See **About records loaded by the MVLOADxxx functions** on page 10-3 for more information.

Examples

Load and totalize all KW data stored in the Meter Value Table for the account for the month of January 1997.

```
MV_JAN = MVLOADDATES(KW, '01/01/1997', '01/31/1997');

STARTDT = '01/01/1997';
STOPDT = '01/31/1997';
MV_JAN = MVLOADDATES(KW, STARTDT, STOPDT);
```

Load and totalize the KW data stored in the Meter Value Table for the channel group named "CHGRP" for the month of January 1997:

```
MV_JAN = MVLOADDATES(KW, "CHGRP", '01/01/1997', '01/31/1997');
```

MVLOADHIST Function

Purpose

The MVLOADHIST Function loads and totalizes meter values for a specified bill determinant and (optionally) billing entity for a user-specified set of bill periods. This function is identical to the **MVLOAD Function** on page 10-2, except that you can specify a desired set of billing periods.

Format

```
<stem_identifier> = MVLOADHIST (<determinant_identifier>
[,<name_identifier|name_literal>], <start_bill_period_previous>,
<end_bill_period_previous>);
```

Where

- <determinant_identifier> is a billing determinant identifier defined in the BILLDETERMINANT Table. The matching meter value records under the account are totalized to get the determinant value.

- <name_identifier | name_literal> *(Optional)* is the name of the account "sub-entity" in the Meter Value Table whose bill determinant values you want to load and totalize.

- <start_bill_period_previous>, <end_bill_period_previous> specifies the bill periods to be loaded, using the following convention: **0** is the current bill period, **1** is the previous bill period, and so on (the higher the number, the further back in time). The end_bill_period_previous must be greater than or equal to the start_bill_period_previous. The default start_bill_period_previous is 0 or the current period. The default end_bill_period_previous is the last period of data available for the account. If you specify a start but no end, the default end is the last period of data for that determinant stored for the account. See **Start and End Bill Period Parameters** under **Rules for Using Functions** in **Chapter 6**: **Rules Language Functions Overview** of the *Oracle Utilities Rules Language User's Guide* for additional details about specifying bill period parameters.)

  If you omit both start and end bill period parameters from MVLOADHIST, it is the same as MVLOAD—only the data for the current bill period is loaded.

The <stem_identifier> on the left side of the equal sign is the name you assign to the temporary record created by the function. See **About records loaded by the MVLOADxxx functions** on page 10-3 for more information.

Examples

Load and totalize all KW data stored in the Meter Value Table for the account for the last 13 bill periods, including the current bill period:

```
MV_HIST = MVLOADHIST(KW, 0, 12);
```

Load and totalize the KW data stored in the Meter Value Table for the channel group named "CHGRP" for the bill period just before the current bill period:

```
MV_HIST = MVLOADHIST(KW, "CHGRP", 1, 1);
```

MVLOADLIST Function

Purpose

The MVLOADLIST Function loads and totalizes meter values for a specified bill determinant for the current bill period for all entities whose NAME appears in a specified Table.Column list. This function totalizes the bill determinant values for all entities in a TABLE.COLUMN list. See **Chapter 8**: **Working with Lists and Queries** in the *Data Manager User's Guide* for information about creating TABLE.COLUMN lists.

Format

```
<stem_identifier> = MVLOADLIST(<determinant_identifier>,
<list_identifier|list_name>);
```

Where

- <determinant_identifier> is a billing determinant identifier defined in the BILLDETERMINANT Table. The matching meter value records under the list are totalized to get the determinant value.

- <list_identifier | list_name> is an identifier, or a literal constant of the form "listname", that identifies a list of NAMES from the Meter Value Table.

The <stem_identifier> on the left side of the equal sign is the name you assign to the temporary record created by the function. See **About records loaded by the MVLOADxxx functions** on page 10-3 for more information.

Example

Load KWH for the 'METER_VAL_LIST' list for the current bill period.

```
MY_LIST = MVLOADLIST(KWH, "METER_VAL_LIST");
```

MVLOADLISTDATES Function

Purpose

The MVLOADLISTDATES Function loads and totalizes meter values for a specified bill determinant for all entities whose NAME appears in a specified Table.Column list, for a user-specified time period. This function is identical to the **MVLOADLIST Function** on page 10-10, except that you can specify a date range for the data.

Format

```
<stem_identifier> = MVLOADLISTDATES(<determinant_identifier>,
<list_identifier|list_name>, <date_identifier|date_constant>,
<date_identifier|date_constant>);
```

Where

- <determinant_identifier> is a billing determinant identifier defined in the BILLDETERMINANT Table. The matching meter value records under the list are totalized to get the determinant value.

- <list_identifier | list_name> is an identifier, or a literal constant of the form "listname", that identifies a list of NAMES from the Meter Value Table.

- <date_identifier|date_constant>, <date_identifier|date_constant> are actual start and end dates. You can use the following formats: 'MM/DD/YYYY', 'MM/DD/YYYY HH:MM', or 'MM/DD/YYYY HH:MM:SS'.

The <stem_identifier> on the left side of the equal sign is the name you assign to the temporary record created by the function. See **About records loaded by the MVLOADxxx functions** on page 10-3 for more information.

Example

Load KWH for the 'METER_VAL_LIST' list for the time period that falls between 5/1/1993 and 5/8/1993.

```
MY_LIST = MVLOADLISTDATES(KWH, "METER_VAL_LIST", '5/1/1993', '5/8/
1993');
```

MVLOADLISTHIST Function

Purpose

The MVLOADLISTHIST Function loads and totalizes meter values for a specified bill determinant for all entities whose NAME appears in a specified Table.Column list, for a user-specified set of bill periods. This function is identical to the **MVLOADLIST Function** on page 10-10, except that you can specify a desired set of bill periods.

Format

```
<stem_identifier> = MVLOADLISTDATES(<determinant_identifier>,
<list_identifier|list_name>, <start_bill_period_previous>,
<end_bill_period_previous>);
```

Where

- <determinant_identifier> is a billing determinant identifier defined in the BILLDETERMINANT Table. The matching meter value records under the list are totalized to get the determinant value.

- <list_identifier | list_name> is an identifier, or a literal constant of the form "listname", that identifies a list of NAMES from the Meter Value Table.

- <start_bill_period_previous>, <end_bill_period_previous> specifies the bill periods to be loaded using the following convention: 0 is the current bill period, 1 is the previous bill period, and so on (the higher the number, the further back in time). The end period must be greater than or equal to the start bill period. The default start_bill_period_previous is 0 or the current period. The default end_bill_period_previous is the last period of data for the determinant. If you specify a start but no end, the default end is the last period of data for that determinant stored for the account. If you specify an end, you must specify a start. If you omit both start and end bill period from MVLOADLISTHIST, it is the same as MVLOADLIST—only the data for the current bill period is loaded.

The <stem_identifier> on the left side of the equal sign is the name you assign to the temporary record created by the function. See **About records loaded by the MVLOADxxx functions** on page 10-3 for more information.

Example

Load KWH for the 'METER_VAL_LIST' list for the last three bill periods, including the current bill period.

```
MY_LIST = MVLOADLISTHIST(KWH, "METER_VAL_LIST", 0, 2);
```

Chapter 11

Math Function Descriptions

This chapter describes all of the math functions available with the Oracle Utilities Rules Language.

Math Functions

ACOS Function

Purpose

The ACOS Function returns the arccosine value of an input. The return value is in the range 0 to PI radians. If the input value is less than -1 or greater than 1, ACOS returns 0.

Format

```
<identifier> = ACOS(<identifier|expression>);
```

Where

- <identifier|expression> is either an identifier or expression. If an identifier, it must have been assigned earlier in the rate form.

Example

Find the arccosine value of 0.5.

```
ARC_COSINE = ACOS(.5);
```

Result:

```
ARC_COSINE = 1.047197551
```

ASIN Function

Purpose

The ASIN Function returns the arcsine value of an input.The return value is in the range -PI/2 to PI/2 radians. If the input is less than -1 or greater than 1, ASIN returns 0.

Format

```
<identifier> = ASIN(<identifier|expression>);
```

Where

* <identifier|expression> is either an identifier or expression. If an identifier, it must have been assigned earlier in the rate form.

Example

Find the arcsine value of 0.5.

```
ARC_SIN = ASIN(.5);
```

Result:

```
ARC_SIN = -.5235987756
```

ATAN Function

Purpose

The ATAN Function returns the arctangent value of an input. The return value is in the range -PI/2 to PI/2 radians.

Format

```
<identifier> = ATAN(<identifier|expression>);
```

Where

- <identifier|expression> is either an identifier or expression. If an identifier, it must have been assigned earlier in the rate form.

Example

Find the arctangent values of 0.75 and -4.

```
ARC_TAN = ATAN(.75);
ARC_TAN2 = ATAN(-4);
```

Result:

```
ARC_TAN = .6435011088
ARC_TAN2 = -1.325817664
```

ATAN2 Function

Purpose

The ATAN2 Function divides the first input by the second input, then returns the arctangent value of the result. The return value is in the range -PI to PI radians, using the signs of both parameters to determine the quadrant of the return value.

Format

```
<identifier> = ATAN2(<identifier|expression>,
<identifier|expression>);
```

Where

- <identifier|expression> is either an identifier or expression. If an identifier, it must have been assigned earlier in the rate form.

Example

Find the arctangent value of the result BILL_DTMT_VAL1 divided by BILL_DTMT_VAL2.

```
BILL_DTMT_VAL1 = 2;
BILL_DTMT_VAL2 = 8;
ARCTANGENT2 = ATAN2(BILL_DTMT_VAL1, BILL_DTMT_VAL2);
```

Process:

```
2 ÷ 8 = .25
ATAN2 (.25) =.2449786631
```

Result:

```
ARCTANGENT2 =.2449786631
```

BITAND Function

Purpose

The BITAND Function returns an integer that is the result of a bitand operation on two supplied integer values. The function converts the integers to binary data, performs the bitand operation, and returns the result as an integer. Input values are rounded to the nearest integer before the operation is performed. Returns 0 if an error occurs.

Format

```
<identifier> = BITAND(<identifier|expression>,
<identifier|expression>)
```

Where:

• <identifier|expression> is either an identifier, or an expression that sets the (non-negative) values of the integer. If an identifier, it must have been assigned earlier in the rate form.

Examples

```
RC = BITAND(1, 3);
```

Result: RC = 1

```
RC = BITAND(1, 4);
```

Result: RC = 0

```
EX_ST = INTDVALUE(HNDL, "EX_STATUS", 5);
//Check the first bit
RC = BITAND(EX_ST, 1);
//
//Check the first and second bit
RC = BITAND(EX_ST, 3);
```

Notes:

This function can be used to obtain the combined status code (combination of channel status code and extended status code) of interval data stored in the Enhanced Interval Data tables. See **Enhanced Interval Data Functions** on page 9-85 for more information about loading interval data from these tables using the Rules Language.

CEIL Function

Purpose

The CEIL Function returns a scalar numeric value that is the smallest integer greater than or equal to the value.

Format

```
<identifier> = CEIL(<identifier|constant>, <places>);
```

Where

- <identifier|constant > is either an identifier that contains a floating-point number (such as a bill determinant identifier) or a floating-point constant.

- <places> *(Optional)* specifies number of places to ceil. **0** directs the program to ceil to an integer; **1**, ceil to tenths (0.1); **2**, ceil to hundredths (0.01); **-2**, ceil to hundreds (100), and so on. The default is **0** (ceil to an integer).

Example

Return smallest integer greater than or equal to MY_VAL, rounded to 0, 1, and 3 decimal places.

```
MY_VAL = 1.54763
CEILING1 = CEIL(MY_VAL);
CEILING2 = CEIL(MY_VAL, 1);
CEILING3 = CEIL(MY_VAL, 3);
```

Result:

```
CEILING1 = 2
CEILING2 = 1.6
CEILING3 = 1.548
```

COS Function

Purpose

The COS Function Returns the cosine value of an input. Input and return values are in radian measure.

Format

```
<identifier> = COS(<identifier|expression>);
```

Where

*   <identifier|expression> is either an identifier or expression. If an identifier, it must have been assigned earlier in the rate form.

Example

Find the cosine value of 1.

```
COS_VAL = COS(1);
```

Result:

```
COS_VAL = .877582562
```

COSECANT Function

Purpose

Returns the cosecant (1/sin) value of an input. On error, returns 0.

Input and return values are in radian measure.

Format

```
<identifier> = COSECANT(<identifier|expression>);
```

Where

*   <identifier|expression> is either an identifier or expression. If an identifier, it must have been assigned earlier in the rate form.

Example

Find the cosecant value of 1.

```
COSECANT_VAL = COSECANT(1);
```

Result:

```
COSECANT_VAL = 1.188395105
```

COSH Function

Purpose

The COSH Function returns the hyperbolic cosine value of an input. If the result is too large, the function returns 0.

Format

```
<identifier> = COSH(<identifier|expression>);
```

Where

- <identifier|expression> is either an identifier or expression. If an identifier, it must have been assigned earlier in the rate form.

Example

Find the hyperbolic cosine value of 0.

```
HYP_COS = COSH(0);
```

Result:

```
HYP_COS = 1
```

COTANGENT Function

Purpose

The COTANGENT Function returns the cotangent (1/tan) value of an input. On error, returns 0. Input and return values are in radian measure.

Format

```
<identifier> = COTANGENT(<identifier|expression>);
```

Where

• <identifier|expression> is either an identifier or expression. If an identifier, it must have been assigned earlier in the rate form.

Example

Find the cotangent value of 1.

```
COTANGENT_VAL = COTANGENT(1);
```

Result:

```
COTANGENT_VAL = .6420926159
```

DIVQUOT Function

Purpose

The DIVQUOT Function divides the first input by the second input, and returns the integral quotient. Both values are rounded to integers before the operation is performed. If the value of the second input is 0, returns 0.

Format

```
<identifier> = DIVQUOT(<identifier|expression>,
<identifier|expression>);
```

Where

- <identifier|expression> is either an identifier or expression. If an identifier, it must have been assigned earlier in the rate form.

  Note: A bill month returned as a date can be used as the first parameter if the second parameter is 16. In this case, the function returns the year and month, respectively.

Example

Find the integral quotient of: 17 divided by 3, 16 divided by 3.2, and 16 divided by 3.7.

```
DIVQUOT_VAL1 = DIVQUOT(17, 3);
DIVQUOT_VAL2 = DIVQUOT(16, 3.2);
DIVQUOT_VAL3 = DIVQUOT(16, 3.7);
```

Result:

```
DIVQUOT_VAL1 = 5
DIVQUOT_VAL2 = 5
DIVQUOT_VAL3 = 4
```

DIVREM Function

Purpose

The DIVREM divides the first input by the second input, and returns the integral remainder. Both the input values are rounded to integers before the operation is performed. If the value of the second input is 0, returns 0.

Format

```
<identifier> = DIVREM(<identifier|expression>,
<identifier|expression>);
```

Where

- <identifier|expression> is either an identifier or expression. If an identifier, it must have been assigned earlier in the rate form.

Example

Find the integral remainder of: 17 divided by 3, 16 divided by 3.2, and 16 divided by 3.7.

```
DIVREM1 = DIVREM(17, 3);
DIVREM2 = DIVREM(16, 3.2);
DIVREM2 = DIVREM(176, 3.7);
```

Result:

```
DIVREM1 = 2
DIVREM2 = 1
DIVREM2 = 0
```

EXP Function

Purpose

The EXP Function returns the exponential value of an input on success, or 0 on overflow (input > 709.782712893) and underflow (input < -708.396418532264).

Format

```
<identififier> = EXP(<identifier|expression>);
```

Where

- <identifier|expression> is either an identifier or expression. If an identifier, it must have been assigned earlier in the rate form.

Example

Find the exponential values of 1 and 1.36.

```
EXP_VAL1 = EXP(1);
EXP_VAL2 = EXP(1.36);
```

Result:

```
EXP_VAL1 = 2.72
EXP_VAL1 = 3.90
```

FABS Function

Purpose

The FABS Function returns the absolute value of its input.

Format

```
<identifier> = FABS(<identifier|expression>);
```

Where

* <identifier|expression> is either an identifier or expression. If an identifier, it must have been assigned earlier in the rate form.

Example

Return the absolute value of -25.

```
FABS_VAL = FABS(-25);
```

Result:

```
FABS_VAL = 25
```

FLOOR Function

Purpose

The FLOOR Function returns a scalar numeric value that is the largest integer (or optionally, real number) not greater than a supplied value. The optional "places" argument lets you specify the precision of the returned value.

Format

```
<identifier> = FLOOR(<identifier|constant>, <places>);
```

Where

- <identifier|constant> is either an identifier that contains a floating-point number (such as a determinant identifier) or a floating-point constant.

- <places> *(Optional)* specifies the precision of the returned value. 0 means floor to an integer, 2 means floor to hundredths (0.01), −2 means floor to hundreds (100). The default is 0.

Example

Return largest integer less than or equal to MY_VAL2, rounded to tenths, and the nearest multiple of 10.

```
MY_VAL2 = 132.548
FLOOR_VAL1 = FLOOR(MY_VAL2, 1);
FLOOR_VAL2 = FLOOR(MY_VAL2, -1);
```

Result:

```
FLOOR_VAL1 = 132.50
FLOOR_VAL2 = 130.00
```

FMOD Function

Purpose

The FMOD Function returns the remainder of the first input divided by the second input. If the value of the second input is 0, FMOD returns 0.

Format

```
<identifier> = FMOD(<identifier|expression>, <identifier|expression>);
```

Where

*   <identifier|expression> is either an identifier or expression. If an identifier, it must have been assigned earlier in the rate form.

    Note: A bill month returned as a date can be used as the first parameter if the second parameter is 16. In this case, the function returns the year and month, respectively.

Example

Find the remainder of 17 divided by 3 and -17 divided by 3.

```
FMOD_VAL1 = FMOD(17, 3);
FMOD_VAL2 = FMOD(-17, 3);
```

Result:

```
FMOD_VAL1 = 2
FMOD_VAL2 = -2
```

FREXPM Function

Purpose

The FREXPM Function breaks down an input value into a mantissa (m) and an exponent (n), and returns m.

The absolute value of m is greater than or equal to 0.5 and less than or equal to 1.0, and the input value is equal to $m*(2$ to the power of integer n). If the input value is 0, 0 is returned.

Format

```
<identifier> = FREXPM(<identifier|expression>);
```

Where

- <identifier|expression> is either an identifier or expression. If an identifier, it must have been assigned earlier in the rate form.

Example

Find the mantissa (m) of: 2, 1, and 7.

```
FREXPM1 = FREXPM(2);
FREXPM2 = FREXPM(1);
FREXPM3 = FREXPM(7);
```

Result:

```
FREXPM1 = 0.5
FREXPM2 = 0.5
FREXPM3 = 0.88
```

FREXPN Function

Purpose

The FREXPN Function breaks down an input value into a mantissa (*m*) and an exponent (*n*), and returns *n*. The absolute value of m is greater than or equal to 0.5 and less than or equal to 1.0, and the input value is equal to $m*(2$ to the power of integer *n*). If the input value is 0, 0 is returned.

Format

```
<identifier> = FREXPM(<identifier|expression>);
```

Where

- <identifier|expression> is either an identifier or expression. If an identifier, it must have been assigned earlier in the rate form.

Example

Find the exponent (n) of: 2, 1, and 7.

```
FREXPN1 = FREXPN(2);
FREXPN2 = FREXPN(1);
FREXPN3 = FREXPN(7);
```

Result:

```
FREXPN1 = 2
FREXPN2 = 1
FREXPN3 = 3
```

LOG Function

Purpose

The LOG Function returns the base e logarithm value of an input on success. If the input value is negative or 0, the function returns 0.

Format

```
<identifier> = LOG(<identifier|expression>);
```

Where

- <identifier|expression> is either an identifier or expression. If an identifier, it must have been assigned earlier in the rate form.

Example

Find the base e logarithm value of 2.

```
LOGARITHM = LOG(2);
```

Result:

```
LOGARITHM = .6931471806
```

LOG10 Function

Purpose

The LOG10 Function returns the base 10 logarithm value of an input on success. If the input value is negative or 0, the function returns 0.

Format

```
<identifier> = LOG10(<identifier|expression>);
```

Where

• <identifier|expression> is either an identifier or expression. If an identifier, it must have been assigned earlier in the rate form.

Example

Find the base 10 logarithm value of 100.

```
LOG10_VAL = LOG10(100);
```

Result:

```
LOG10_VAL = 2
```

MAX Function

Purpose

The MAX Function returns the maximum value of two or more parameters. You can compare database values, constants, and/or values defined elsewhere in the schedule. You can supply as many parameters as you wish, but there must be at least two. Returns a scalar numeric value.

Format

```
<identifier> = MAX(<identifier|expression>, <identifier|expression>,
<...>);
```

Where

• <identifier|expression> is either an identifier or expression. If an identifier, it must have been assigned earlier in the rate form.

Example

Find the maximum value among the VAL1 ... VAL5.

```
VAL1 = 12
VAL2 = -8
VAL3 = 0
VAL4 = 18
VAL5 = 3
MAX_VAL = MAX(VAL1, VAL2, VAL3, VAL4, VAL5);
```

Result:

```
MAX_VAL = 18
```

MAXN Function

Purpose

The MAXN Function finds the *nth* maximum value of the parameters supplied. *N* can be an expression, or the value of an identifier. The return value is 0 if *n* is less than 1 or greater than the count of values. Returns a scalar numeric value.

Format

```
<identifier> = MAXN(<n>, <identifier|expression>,
<identifier|expression>, <...>);
```

Where

- `<n>` is an identifier or integer constant that indicates which peak to find; e.g., first, second, third, etc.

- `<identifier|expression>` is either an identifier or expression. If an identifier, it must have been assigned earlier in the rate form.

Example

Find the 4th maximum value among the VAL1 ... VAL6.

```
VAL1 = 7
VAL2 = 8
VAL3 = 1
VAL4 = 5
VAL5 = 3
VAL6 = 9
MAX_4 = MAXN(4, 7, 8, 1, 5, 3, 9);
```

Result:

```
MAX_4 = 5
```

MIN Function

Purpose

The MIN Function returns the minimum of two or more parameters.

Format

```
<identifier> = MIN(<identifier|constant>, <identifier|constant>,
<...>);
```

Where

- <identifier|constant> is either an identifier that contains a floating-point number (such as a determinant identifier) or a floating-point constant.

Example

Get the minimum demand from a set of values.

```
MINKW = MIN(CONTRACT_KW, 100);
```

MINNZ Function

Purpose

The MINNZ Function finds nonzero minimum value. This function is identical to the MIN function, except that it excludes zero values from the comparison.

Format

```
<identifier> = MINNZ(<identifier|constant>, <identifier|constant>,
<...>);
```

Where

- <identifier|constant> is either an identifier that contains a floating-point number (such as a determinant identifier) or a floating-point constant.

Example

Get the minimum nonzero demand from a set of values:

```
MINKW = MINNZ(CONTRACT_KW, KW);
```

MODF Function

Purpose

The MODF Function returns the signed fractional portion of an input value.

Format

```
<identifier> = MODF(<identifier|expression>);
```

Where

- <identifier|expression> is either an identifier or expression. If an identifier, it must have been assigned earlier in the rate form.

Example

Find the signed fractional portion of HNDL_VAL.

```
HNDL_VAL = -14.876543;
MODULES = MODF(HNDL_VAL);
```

Result:

```
MODULES = -0.876543
```

POW Function

Purpose

The POW Function returns the value of the first input raised to the power of the second input value. It returns 1 if the second value is 0, and 0 on overflow or underflow.

Format

```
<identifier> = POW(<identifier|expression>, <identifier|expression>);
```

Where

- <identifier|expression> is either an identifier or expression. If an identifier, it must have been assigned earlier in the rate form.

Example

Raise 3 to the 2nd power:

```
NEW_NUM = POW(3, 2);
REPORT NEW_NUM LABEL "THREE TO THE SECOND POWER = ";
```

Result:

```
THREE TO THE SECOND POWER = 9.00
```

Raise 2 to the 4th power:

```
NEW_NUM = POW(2, 4);
REPORT NEW_NUM LABEL "TWO TO THE FOURTH POWER = ";
```

Result:

```
TWO TO THE FOURTH POWER = 16.00
```

ROUND Function

Purpose

The ROUND Function rounds a value to user-specified decimal place.

Format

```
<identifier> = ROUND(<identifier|constant>, <places>);
```

Where

- <identifier|constant> is either an identifier that contains a floating-point number (such as a determinant identifier) or a floating-point constant.

- <places> is an integer that indicates how many decimal places to keep. A value of **0** means round to an integer, **2** means round to hundredths (0.01), **-2** means round to nearest hundred (100).

Example

Round the ENERGY.CHG charge to dollars and cents:

```
ENERGY_CHG = 105.132057;
MY_CHARGE = ROUND(ENERGY_CHG,2);
```

Result:

```
MY_CHARGE = 105.13
```

ROUND2VALUE Function

Purpose

The ROUND2VALUE Function rounds a value to the nearest multiple of another value. Returns a scalar numeric value. If the second value is 0, the first value is returned unchanged.

Format

```
<identifier> = ROUND2VALUE(<identifier1|constant1>,
<identifier2|constant2>);
```

Where

- <identifier1|constant1> is either an identifier that contains a floating-point number (such as a determinant identifier) or a floating-point constant.

- <identifier2|constant2> is either an identifier that contains an integer or floating-point number, or an integer or floating-point constant.

Example

Round the value 55.36 to the nearest multiple of 50, then to the nearest multiple of 0.11:

```
VALUE1 = 55.36;
ROUND2V_1 = ROUND2VALUE (VALUE1, 50);
ROUND2V_2 = ROUND2VALUE (VALUE1, 0.11);
```

Result:

```
ROUND2V_1 = 50.00
ROUND2V_2 = 55.33
```

ROUNDINT Function

Purpose

The ROUNDINT Function rounds a value to the nearest *n* number of digits, where *n* is a user-specified number of places.

The function returns an integer if places is less than or equal to 0; else returns a float.

Format

```
<identifier> = ROUNDiNT(<identifier|constant>[, <places>]);
```

Where

- <identifier|constant> is an identifier that contains a floating-point number (such as a determinant identifier) or a floating-point constant.

- <places> *(Optional)* is an integer that indicates how many decimal places to keep. The default is **0** (round to an integer); **2** means round to hundredths (0.01), and **-2** means round to nearest hundred (100).

Example

Round the ENERGY.CHG charge to the nearest dollar:

```
ENERGY_CHG = 105.132057;
MY_CHARGE_ROUND = ROUNDINT(ENERGY_CHG);
```

Result:

```
MY_CHARGE_ROUND = 105
```

SECANT Function

Purpose

The SECANT Function returns the secant (1/cos) value of an input. On error, returns 0.

Format

```
<identifier> = SECANT(<identifier|expression>);
```

Where

- <identifier|expression> is either an identifier or expression. If an identifier, it must have been assigned earlier in the rate form.

Example

Find the secant value of 1.

```
SEC = SECANT(1);
```

Result:

```
SEC = 1.850815718
```

SIN Function

Purpose

The SIN Function returns the sine value of an input.

Format

```
<identifier> = SIN(<identifier|expression>);
```

Where

- <identifier|expression> is either an identifier or expression. If an identifier, it must have been assigned earlier in the rate form.

Example

Find the sine value of 1.

```
SIN_VAL = SIN(1);
```

Result:

```
SIN_VAL = .8414709848
```

SINH Function

Purpose

The SINH Function returns the hyperbolic sine value of an input. If the result is too large, returns 0.

Format

```
<identifier> = SINH(<identifier|expression>);
```

Where

• <identifier|expression> is either an identifier or expression. If an identifier, it must have been assigned earlier in the rate form.

Example

Find the hyperbolic sine value of 15.

```
HYP_SINE = SINH(15);
```

Result:

```
HYP_SINE = 1,634,508.69
```

SQROOT Function

Purpose

The SQROOT Function returns the square root of a non-negative value. If value is negative, returns 0.

The value may be a database value, a constant, or a value defined elsewhere in the schedule. Returns a scalar numeric value.

Format

```
<identifier> = SQROOT(<identifier|constant>);
```

Where

- <identifier|constant> is either an identifier that contains a floating-point number (such as a determinant identifier) or a floating-point constant.

Example

Find the square root of NUMBER.

```
NUMBER = 16;
SQROOT = SQROOT(NUMBER);
```

Result:

```
SQROOT = 4.0
```

TAN Function

Purpose

The TAN Function returns the tangent value of an input.

Format

```
<identifier> = TAN(<identifier|expression>);
```

Where

• <identifier|expression> is either an identifier or expression. If an identifier, it must have been assigned earlier in the rate form.

Example

Find the tangent value of 8.

```
TAN = TAN(8);
```

Result:

```
TAN = -6.79971145
```

TANH Function

Purpose

The TANH Function returns the hyperbolic tangent value of an input value.

Format

```
<identifier> = TANH(<identifier|expression>);
```

Where

- <identifier|expression> is either an identifier or expression. If an identifier, it must have been assigned earlier in the rate form.

Example

Find the hyperbolic tangent values of 0.5 and -2.

```
HYP_TAN = TANH(.5);
HYP_TAN2 = TANH(-2);
```

Result:

```
TANH = .4621171573
TANH2 = -0.9640275801
```

Chapter 12

String Function Descriptions

This chapter describes all of the string functions available with the Oracle Utilities Rules Language.

String Functions

FLOAT2STRING Function

Purpose

The FLOAT2STRING Function converts the value of an identifier to a string.

The value must be an integer or a floating-point number. The second, optional parameter is an integer expression or identifier that must be equal to 0, 1, 2, 3, or 6. If specified, the string will have a number of decimal places equal to the integer expression. If not specified, trailing 0s are removed; if the decimal point ends up as the rightmost character, it is also removed. Returns a string value.

Format

```
<IDENTIFIER> = FLOAT2STRING(<identifier|expression>, <integer>);
```

Where

- <identifier|expression> is either an identifier or expression. If an identifier, it must have been assigned earlier in the rate form.

- <integer> *optional* is an integer expression or identifier. The possible expressions are 0, 1, 2, 3, and 6. If an identifier, it must have been assigned earlier in the rate form.

Format

Convert the value of ARC_TAN to a string with 3 decimal places.

```
ARC_TAN = 0.644
ARC_TAN_STR = FLOAT2STRING(ARC_TAN, 3);
```

Result:

```
ARC_TAN_STR = "0.644"
```

FLOAT2STRINGNC Function

Purpose

The FLOAT2STRINGNC Function converts the value of an identifier to a string, but without commas to mark the thousands.

This function is the same as FLOAT2STRING, except that no commas are inserted in the number.

Format

```
<IDENTIFIER> = FLOAT2STRINGNC(<identifier|expression>, <integer>);
```

Where

- <identifier|expression> is either an identifier or expression. If an identifier, it must have been assigned earlier in the rate form.

- <integer> *(Optional)* is an integer expression or identifier. The possible expressions are 0, 1, 2, 3, and 6. If an identifier, it must have been assigned earlier in the rate form.

Format

Convert the value of TOTAL_CHARGES to a string with no comma.

```
$TOTAL_CHARGES = 10440;
TOTAL_CHARGES_STR = FLOAT2STRINGNC(TOTAL_CHARGES);
```

Result:

```
TOTAL_CHARGES_STR = "10440"
```

INSTR Function

Purpose

The INSTR Function returns the position (denoted with an integer) of the first occurrence of string2 in string1.

Format

```
<identifier> = INSTR(<string1>, <string2>);
```

Where

- <string1> is a text string.

- <string2> is another text string; may consist of a single character.

Format

Find the first occurrence of "A" in the string "DATABASE".

STR_POSITION_A = INSTR("DATABASE", "A")
LABEL STR_POSITION_A "String Position of A";

Result:

```
String Position of A: 2
```

LEFT Function

Purpose

The LEFT Function returns the leftmost *n* characters of a string. If *n* is greater than the length of the string, the entire string is returned (not padded).

Format

```
<identifier> = LEFT(<string>, <n>);
```

Where

•   <string> is a text string.

•   <n> is the number of characters.

Format

To reformat a bill date from a "mm/dd/yyyy" format to a "yyyy/mm" format.

```
BILLING_MONTH = " " + RIGHT(BILL_DATE,4) + "/" + LEFT(BILL_DATE,2);
```

If the value for BILL_DATE was 11/20/1997, BILLING_MONTH would return "1997/11".

LEN Function

Purpose

The LEN Function returns the length of a string (an integer).

Format

```
<identifier> = LEN(<string>);
```

Where

*   <string> is a text string.

Format

Find the length of the string "DATABASE".

STR_LENGTH = LEN("DATABASE");
```
LABEL STR_LENGTH 'String Length";
```

Result:

```
String Length: 8
```

Find whether BILL_MONTH is between January and September (a string length of 6), or between October and December (a string length of 7):

```
BILL_MONTH = " " + MONTH(BILL_PERIOD) + "/" + YEARSTR(BILL_PERIOD);
CHARACTER_COUNT = LEN(BILL_MONTH);
IF CHARACTER_COUNT = 6 THEN
    /* BILL_MONTH is between Jan. and Sept. */
END IF;
IF CHARACTER_COUNT = 7 THEN
    /* BILL_MONTH is between Oct. and Dec. */
END IF;
```

LTRIM Function

Purpose
The LTRIM Function returns the string with leading spaces removed.

Format
```
<identifier> = LTRIM(<string>);
```

Where
•   <string> is a text string.

Format
Trim the leading spaces off the string "   DATABASE".

STR_TRIM = LTRIM("   **DATABASE");**
```
LABEL STR_TRIM "String";
```

Result:

```
String: DATABASE
```

MID Function

Purpose

The MID Function returns a specified number of characters, beginning at a user-specified start position.

Returns the part of the string beginning at start. If length is omitted or the length is greater than the number of characters in the string, all characters from the start position to the end of the string are returned.

Format

```
<identifier> = MID(<string>, <start>[, <length>]);
```

Where

- <string> is a text string.

- <start> is the start position in the string.

- <length> *(Optional)* is the number of characters to be returned.

Format

Find the first 5 characters of the string "DATABASE".

FIRST_FIVE = MID("DATABASE", 1, 5)
LABEL FIRST_FIVE "First Five Characters"

Result:

First Five Characters: DATAB

RIGHT Function

Purpose

The RIGHT Function returns the rightmost *n* characters of a string. If *n* is greater than the length of the string, the entire string is returned (not padded).

Format

```
<identifier> = RIGHT(<string>, <n>);
```

Where

- <string> is a text string.

- <n> is the number of characters.

Format

To reformat a bill date from a "mm/dd/yyyy" format to a "yyyy/mm" format.

```
BILLING_MONTH = " " + RIGHT(BILL_DATE,4) + "/" + LEFT(BILL_DATE,2);
```

For example, if the value for BILL_DATE were 11/20/1997, then BILLING_MONTH would return "1997/11".

RTRIM Function

Purpose

The RTRIM Function returns the string with trailing spaces removed.

Format

```
<identifier> = RTRIM(<string>);
```

Where

- <string> is a text string.

Format

Trim the trailing spaces off the string "DATABASE   ".

```
STR_TRIM = RTRIM("DATABASE      ");
LABEL STR_TRIM "String";
```

Result:

```
String: DATABASE
```

STRING Function

Purpose

The STRING Function converts the value of a constant or identifier to a string.

You can specify the number of characters in the string to be returned. Numbers are converted with commas to mark the thousands.

Note: If you do not want the commas, such as for a year, use the **STRINGNC Function** on page 12-12 instead. Date/times are converted to the date/time display format.

Format

```
<identifier> = STRING(<identifier|constant>[, <length>]);
```

Where

- <identifier|constant> is either an identifier that contains a floating-point number (such as a determinant identifier) or a floating-point constant.

- <length> *(Optional)* number of characters to be returned.

Format

Convert the constant 104040 to a string 5 characters long.

CONST_STR = STRING('104040', 5);
LABEL CONST_STR "Constant to String";

Result:

Constant to String: "10,404"

STRINGNC Function

Purpose

The STRINGNC Function is similar to the **STRING Function** on page 12-11 except that STRINGNC converts numbers without commas to mark the thousands. This is desirable for formatting years, for example.

Format

```
<identifier> = STRINGNC(<identifier|constant>, <length>);
```

Where

- <identifier|constant> is either an identifier that contains a floating-point number (such as a determinant identifier) or a floating-point constant.

- <length> *(Optional)* the number of characters to be returned.

Format

To get the current bill year:

```
BILLING_YEAR= YEAR(BILL_PERIOD);
BILL_YEAR = STRINGNC(BILLING_YEAR);
LABEL BILL_YEAR "Bill Year";
```

Result:

The above would return the year in BILL_PERIOD as a string with no comma:

```
Bill Year: "1998"
```

TOLOWER Function

Purpose

The TOLOWER Function returns the string with all uppercase letters converted to lowercase, and all other characters unchanged.

Format

```
<identifier> = TOLOWER(<string>);
```

Where

*   <string> is a text string.

Format

```
LOWER_ID = TOLOWER("DataBase");
LABEL LOWER_ID "Lowercase ID";
```

Result:

```
Lowercase ID: "database"
```

TOUPPER Function

Purpose

The TOUPPER Function returns the string with all lowercase letters converted to uppercase, and all other characters unchanged.

Format

```
<identifier> = TOUPPER(<string>);
```

Where

*   <string> is a text string.

Format

```
UPPER_ID = TOUPPER("DataBase");
LABEL UPPER_ID "Uppercase ID";
```

Result:

```
Uppercase ID: "DATABASE"
```

TRIM Function

Purpose

The TRIM Function returns the string with leading and trailing spaces removed.

Just as the **LTRIM Function** on page 12-7 removes leading spaces, and the **RTRIM Function** on page 12-10 removes trailing spaces, TRIM removes both.

Format

```
<identifier> =TRIM (<string>);
```

Where

- <string> is a text string.

Format

```
TEST = "     TEST     ";
TRIM_TEST = TRIM (TEST);
LABEL TRIM_TEST "Trim Test";
```

Result:

```
Trim Test: "TEST"
```

Chapter 13

Other Function Descriptions

This chapter describes "other" functions available with the Oracle Utilities Rules Language, including:

- **Database Functions**

- **Date/Time Functions**

- **Historical-Data Functions**

- **Internal Functions**

- **Season-Based Functions**

- **Term Functions**

- **Miscellaneous Functions**

Database Functions

Database functions are used to obtain information about and perform operations on records in the Oracle Utilities Data Repository.

ACCOUNTFACTOR Function

Purpose

The ACCOUNTFACTOR function returns a value that indicates whether a particular factor was in effect for an account on the end date of the current bill period or user-specified historical bill period.

This function enables you to selectively apply a factor to accounts—that is, to some accounts but not others, even though the accounts are on the same rate code.

If you include this function in the rate form, the program automatically checks the Factor History Table (ACCTFACTORHIST) to see if the specified factor was in effect for the account on the end date of the specified bill period. If so, the function returns a value of 1; if not, it returns 0.

Format

```
<identifier>= ACCOUNTFACTOR(<factor_code|identifier>[,
<bill_period_previous>]);
```

Where

- <factor_code | identifier> specifies the factor to be found. You can specify a factor code that is listed in the Factors Lookup Code Table, or an identifier that contains a factor code.

  Factors are identified by the following key from the Factor Lookup Code Table: "operating_company_code,jurisdiction_code,factor_code". To specify the desired factor, you need only specify the factor_code; the operating company and jurisdiction are automatically assumed by the program to be that of the current account. For example, to apply this function to a factor that has the lookup code "STATETAX", you would specify ACCOUNTFACTOR("STATETAX"). To indicate that the factor is global (applies across all operating companies and jurisdictions), use the convention ",,factorcode".

  You can also use a simple identifier that you assigned to a factor code elsewhere in the rate form.

- <bill_period_previous> - *(Optional)* specifies the billing period. Use **0** to specify the current bill period, **1** for the previous bill period, and so on (the higher the number, the further back in time). If not supplied, the current bill period is used.

Example

Determine whether the 'FACTOR_TAX_A' factor was in effect for the current bill period.

```
FACTOR_CHECK = ACCOUNTFACTOR("FACTOR_TAX_A");
```

ARRAYUPPERBOUND Function

Purpose

The ARRAYUPPERBOUND function returns the upper bound of the array identifier. The upper bound is the highest index of the array that has been assigned a value. Returns a scalar numeric value.

Format

```
<identifier> = ARRAYUPPERBOUND(<array_identifier>);
```

Where

- <array_identifier> is an array identifier previously assigned in the rate form. See **Array Identifiers** on page 4-20 in the *Oracle Utilities Rules Language User's Guide* for more information about using array identifiers.

    Note: the array should NOT contain an identifier in the index position (between the brackets [X])

Example

Return the upper bound of the #ARR [ ] array identifier.

```
#ARR [INDEX].ACCT = "1234";
#ARR [INDEX].TYPE = "RES";
#ARR [INDEX].STATUS = "ACTIVE";
ARR_SIZE = ARRAYUPPERBOUND(#ARR []);
```

CALLSTOREDPROC Function

Purpose

The CALLSTOREDPROC function calls a stored procedure.

Rules for writing stored procedures to be called by the CALLSTORPROC function include:

- Stored procedure names should be 18 character or less, and should be qualified in the same way as table names in the database are qualified.

- The COMMIT Statement is not allowed inside the stored procedure.

- The procedure must take at least one input parameter.

- The last parameter to the procedure must be an output-only parameter, and must be assigned a value within the procedure.

Format

```
<identifier> = CALLSTOREDPROC(<identifier|expresssion>
[,<identifier|expresssion> ... ]);
```

OR

```
<identifier> = CALLSTOREDPROC(<storedProcName>, <InputStem>,
<OutputStem>);
```

Where

- <identifier|expression> is an identifier or expression that is either the name of the stored procedure or one of the parameters used by the stored procedure. The first parameter to this function must be the stored procedure name. This can be followed by any number of parameters that the stored procedure accepts. The last parameter specifies the return data type; its actual value is ignored.

  - To return a string, the last parameter must be empty double quotes ("").

  - To return an integer, the last parameter must be an integer (0).

  - To return a float, the last parameter must be a float (0.0).

  - To return a date, the last parameter must be a valid date (for example, 01/01/2000 or BILL_START).

- <storedProcName> is the name of the stored procedure.

- <InputStem> is a stem identifier whose tail identifiers correspond to the input parameters for the stored procedure. The first tail identifier MUST be named "P1", the next "P2" and so on.

- <OutputStem> is a stem identifier whose tail identifiers correspond to the output parameters of the stored procedure. The first tail identifier MUST be named "P1", the next "P2" and so on.

Examples

Invoke a stored procedure named "RETURNCUSTOMERID" and return a string value. The stored procedure is written with one input parameter (the ACCOUNTID) and one output parameter. The output parameter is assigned a value within the procedure.

```
ACCT_ID = ACCOUNT.ACCOUNTID
CUSTOMERID = CALLSTOREDPROC("RETURNCUSTOMERID", ACCT_ID, "");
```

Invoke a stored procedure named "RETURNCUSTOMERINFO" and set a string value and an integer value in CUSTOMERINFO.P1 and CUSTOMERINFO.P2 respectively. The stored procedure is written with two input parameters (ACCOUNTINFO.P1 and ACCOUNTINFO.P2) and two output parameters

*(CUSTOMERINFO.P1 and CUSTOMERINFO.P2). The output parameters are assigned values within the
procedure. The function returns CUSTOMERINFO.P1.*

```
ACCOUNTINFO.P1 = "Account1";
ACCOUNTINFO.P2 = '01/01/2001';
CUSTOMERINFO.P1 = "";
CUSTOMERINFO.P2 = 1;
CUSTOMERID = CALLSTOREDPROC("RETURNCUSTOMERINFO", ACCOUNTINFO,
CUSTOMERINFO);
```

Notes for using Stem.Tail idenifiers

- The type of the output identifiers determine the type of the values passed back in them.

- The function will return the value of the first output parameter.

- The minimum requirement is that there should be at least one input and one output
  parameter (P1).

- The stored procedure must be written in such a way that all the input parameters come first
  followed by the output parameters.

- No In/Out parameters are allowed.

GETADOCONNECTION Function

Purpose

The GETADOCONNECTION function gets the ADO database connection used by the Rules Language. This function obtains the ADO database connection used by the Rules Language and makes it available to third-party components, such as COM objects created through use of the **CREATEOBJECT Function**. This function returns the ADO database connection interface.

See **COM Object Functions** in **Chapter 8**: **Working with COM Components** in the *Oracle Utilities Rules Language User's Guide* for more information about using this statement.

Format

```
<identifier> = GETADOCONNECTION();
```

Example

Invoke the "Execute Query" method of a LSDB DataSource COM object.

```
OBJECT = CREATEOBJECT ("LSDB.DataSource");
CON = GETADOCONNECTION ();
RES = OBJECT->ExecuteQuery(CON, XML_QUERY);
```

GETCONNECT Function

Purpose

The GETCONNECT function returns the data source used to log on to the Oracle Utilities application.

This function does not require parameters. It retrieves the appropriate information based on the current user: the connection string used to log on to the Oracle Utilities application.

Format

```
<identifier> = GETCONNECT ( );
```

Example

```
USER_INFO_CONNECT = GETCONNECT( );
LABEL USER_INFO_CONNECT "Data Source";
```

GETDATASOURCE Function

Purpose

The GETDATASOURCE function returns a variant (COM object) that contains the current database connection used by the Rules Language.

This function obtains the database connection used by the Rules Language and makes it available to third-party components, such as COM objects created through use of the **CREATEOBJECT Function**.

See **COM Object Functions** in **Chapter 8**: **Working with COM Components** in the *Oracle Utilities Rules Language User's Guide* for more information about using this statement.

Format

```
<identifier> = GETDATASOURCE();
```

Example

Get the current database connection.

```
DS = GETDATASOURCE ();
```

Notes

This function returns the database connection as a COM object, which can be converted into an XML string using the following syntax:

```
DS = GETDATASOURCE ();
DS_XML = DS->XML;
```

Example

Invoke the "ExecuteQuery" method of an LSDB DataSource COM object.

```
OBJECT = CREATEOBJECT ("LSDB.DataSource");
DS = GETDATASOURCE ();
DS_XML = DS->XML;
RES = OBJECT->ExecuteQuery(DS_XML, XML_QUERY);
```

GETQUALIFIER Function

Purpose

The GETQUALIFIER function gets the qualifier for the current database connection used by the Rules Language.

This function obtains the qualifier for the current database connection used by the Rules Language.

Format

```
<identifier> = GETQUALIFIER();
```

Example

Get the qualifier for the current database connection.

```
QUAL = GETQUALIFIER();
```

GETUSERID Function

Purpose

The GETUSERID function returns the user id used to log on to the Oracle Utilities application.

This function does not require parameters. It retrieves the appropriate information based on the current user: the user id used to log on to the Oracle Utilities application.

Format

```
<identifier> = GETUSERID ( );
```

Example

```
USER_ID_INFO = GETUSERID( );
LABEL USER_ID_INFO "User ID";
```

Result:

```
User ID: jqsmith
```

HASVALUE Function

Purpose

The HASVALUE function determines whether an identifier has a value in the database, has been assigned a value in the rate form, or has no value.

If the identifier has a value in the database, it is assigned that value. If the identifier does not have a value in the database, the function returns a code to indicate its status:

0 - the identifier does not have a value in the database, and it has not been assigned a value in the rate form. Also, any interval data handle with a value of 0

1 - value set during data loading

2 - value set via ASSIGNMENT Statement in rate form

3 - value set in NOVALUE Statement in rate form.

Format

```
<identifier> = HASVALUE(<identifier>);
```

Where

• <identifier> can be any identifier, including database identifiers.

> **Note**: When specifying an array identifier, you must include an index for the array. See **Array Identifiers** on page 4-20 for more information about array identifiers.

Example

Determine if the identifier "MY_HNDL" has a value in the database.

```
MY_HNDL_VAL = HASVALUE(MY_HNDL);
```

Determine if the array identifier "#MY_ARRAY" has a value in the database.

```
MY_ARRAY_VAL = HASVALUE(#MY_ARRAY[1]);
```

LISTCOUNT Function

Purpose

The LISTCOUNT function returns the number of items in a list.

The list must be a Table.Column list (see the *Data Manager User's Guide* for more information about Table. Column lists). The function returns 0 if there are no items in the list, or if an error occurred when the list query was run.

Format

```
<identifier> = LISTCOUNT(<identifier|literal>);
```

Where

- <identifier|literal> is an identifier or literal whose string value is the name of a Table.Column list.

Example

Return the number of items in the "ACCT_CHAN" list.

```
NUM_ITEM = LISTCOUNT("ACCT_CHAN");
```

LISTOP Function

Purpose

The LISTOP function performs column functions — AVG, COUNT, MAX, MIN, or SUM — on a Table.Column list.

Returns the result of applying the operation (column function) to a table-column list. The operation must be "AVG", "COUNT", "MAX", "MIN", or "SUM" (see a SQL Reference guide for definitions of these functions). Multiple operations can be listed. If the list is not correct, the same record may appear in the result set more than once. If this occurs, the repeated value will contribute to the result more than once. The optional 'DISTINCT' parameter cannot correct this, because DISTINCT applies to values, not records. See LSTRFRSH.EXE in **Chapter Eight**: **Working with Lists and Queries** in the *Data Manager User's Guide* for more information.

Format

```
<identifier> = LISTOP(<identifier|list_name>, <operation>
[,<operation> ...][, <distinct>]);
```

Where

- <identifier|list_name> is the name of a Table.Column list.

- <operation> is one of: "AVG", "COUNT", "MAX", "MIN", or "SUM." This can also be the result of another function or operation whose result is a string equal to one of the above values. The column must be numeric if the operation is "AVG" or "SUM."

- <distinct> *(Optional)* determines whether the function applies to all of the values on the table-column, or only distinct (unique) values. If used, this should be string with a value of "DISTINCT" (all other values are ignored). This parameter may be a string that evaluates to "DISTINCT".

Example

Return the number of distinct values in the FACTOR_VALUES list (comprised of values in the VAL (VALUE) column of the FACTORVALUE Table that are >=0 and <= 150).

DISTINCT_COUNT = LISTOP("FACTOR_VALUES", "COUNT", "DISTINCT");
LABEL DISTINCT_COUNT "Number of Distinct Values";

Result:

Number of Distinct Values: 26

LISTUPDATE Function

Purpose

The LISTUPDATE function updates one or more column values of every record in a Table.Column list.

The list must be a full record list, or a list of the primary keys in a table. This function requires selecting the 'Automatically save/approve each page if it is OK' Save option on the Advanced dialog in Trial Bill, or the -k (Save Results) switch if using RUNRS.EXE. Returns the number of rows updated.

Format

```
<identifier> = LISTUPDATE(<identifier|list_name>, <column_name>,
<value>);
```

OR

```
<identifier> = LISTUPDATE(<identifier|list_name>, <stem_identifier>);
```

Where

- <identifier|list_name> is the name of a table-column list.

- <column_name> is the name of the column to be updated.

- <value> is the value to which each column is updated.

- <stem_identifier> is a stem identifier whose corresponding tail identifiers are column names. Each stem.tail identifier should be assigned to the value to which the column is to be updated.

Note: The LISTUPDATE function cannot update columns whose values are lookups from another table. Also, this function only updates records in the specified list, and does not affect any related records or data.

Examples

Update the Effective Date (STARTTIME) column in the MARCH_ACCOUNTS list (comprised of values in the Effective Date (STARTTIME) column of the Account Table that are >=02/28/1999 and <= 03/02/1999) to 03/01/1999.

```
UPDATE_EFFECTIVE_DATE = LISTUPDATE(MARCH_ACCOUNTS, STARTTIME,
03/01/1999);
```

Update the Effective Date (STARTTIME) and Stop Time (STOPTIME) columns in the MARCH_ACCOUNTS list.

```
NEW_DATES.STARTTIME = '03/01/1999';
NEW_DATES.STOPTIME = '03/31/1999';
UPDATE_EFFECTIVE_DATES = LISTUPDATE(MARCH_ACCOUNTS, NEW_DATES);
```

LISTVALUE Function

Purpose

The LISTVALUE function returns the first element in a Table.Column list or Query list.

Returns the first element in a table-column list. If the list targets a UID, the full record is returned. Otherwise, just a value is returned.

Format

```
<identifier> = LISTVALUE(<identifier|list_name>);
```

Where

- <identifier|list_name> is the name of a table-column or query list. See **Lists** on page 7-63 in the *Oracle Utilities Energy Information Platform User's Guide* for more information about query lists.

  Note: Query lists created using the Lists function of the Energy Information Platform user interface cannot be selected using the Rules Language Elements Editor.

Example

Return the first value in the FACTOR_VALUES list (comprised of values in the VAL (VALUE) column of the FACTORVALUE Table that are >=0 and <= 150).

FIRST_VALUE = LISTVALUE(FACTOR_VALUES);
LABEL FIRST_VALUE "First Value in List";

Result:

```
First Value in List: 1
```

PRORATEFACTOR Function

Purpose

The PRORATEFACTOR function prorates a factor over a user-specified time period. It returns the value of a factor prorated over the time from the start time of START_BILL_ PERIOD through the stop time of END_BILL_PERIOD. This prorates over several periods; the individual periods' values are determined according to the corresponding PRORATEMETHOD in the FACTORVALUE Table.

Format

```
<identifier> = PRORATEFACTOR(<database_code>,
<start_bill_period_previous>, <end_bill_period_previous>);
```

Where

- <database_code> specifies the factor to be found; format is "factorcode".

- <start_bill_period_previous>, <end_bill_period_previous> specifies the bill periods to be loaded, using the following convention: **0** is the current bill period, **1** is the previous bill period, and so on (the higher the number, the further back in time). The end_bill_period_previous must be greater than or equal to the start_bill_period_previous. The default start_bill_period_previous is 0 (the current period). The default end_bill_period_previous is the last period of data available for the account. If you specify a start but no end, the default end is the last period of data for that determinant stored for the account. See **Start and End Bill Period Parameters** under **Rules for Using Functions** in **Chapter 6**: **Rules Language Functions Overview** of the *Oracle Utilities Rules Language User's Guide* for details about specifying bill period parameters.

Example

Prorate the "TAX" factor over the last three bill periods, including the current bill period.

```
FCTR_PRORATE = PRORATEFACTOR("TAX", 0, 2);
```

RSPRORATE Function

Purpose

The RSPRORATE function prorates a value based on the time that the rate schedule is in effect in the bill period. This function is for use with new accounts.

This function prorates a user-selected identifier or constant for the portion of the time that the rate schedule is in effect in the bill period, using the formula:

```
value * (RS_EFFECTIVE_STOP – RS_EFFECTIVE_START) / (BILL_ STOP – BILL_START).
```

The result is always less than or equal to the original value.

Note: This function is intended for prorating charges for a new account whose bill covers less than a full bill period. For accounts switching rates in a bill period, factor prorating is recommended. See the description of the **For Each x in Factor Statement** on page 3-8.

Format

```
<identifier> = RSPRORATE(<identifier|constant>);
```

Where

- <identifier|constant> is either an identifier that contains a floating-point number (such as a determinant identifier) or a floating-point constant.

Example

Prorate the value of KWH for the account.

```
KWH_PRORATE = RSPRORATE(KWH);
```

SETBINPATH Function

Purpose

The SETBINPATH returns the path to the LODESTAR bin directory.

This function returns the path to a file in the LODESTAR bin directory. The function returns a fully qualified file name.

Format

```
<identifier> = SETBINPATH(<identifier|expression>);
```

Where

- <identifier|expression> is an identifier or expression that evaluates to a string that is a file name. If " " is provided, the function returns the name of the application executing the Rules Language.

Example

Return the path to the LODESTAR\Bin directory.

```
BIN_PATH = SETBINPATH("datamgr.exe");
```

Return the name of the program executing the Rules Language .

```
BIN_PATH_EXE = SETBINPATH("");
```

Result:

```
BIN_PATH_EXE = "C:\LODESTAR\Bin\datamgr"
```

SETDBMONITOR Function

Purpose

The SETDBMONITOR function turns the Database Monitor on or off.

This function turns the Database Monitor (used with early versions of the Oracle Utilities Transaction Management) on or off. The function returns the previous state of the database monitor 0 = off, 1 = on.

Format

```
<identifier> = SETDBMONITOR(<identifier|expression>);
```

Where

- <identifier|expression> is an identifier or expression that evaluates to zero or one. If one, database monitoring is turned on, if zero database monitoring is turned off.

Example

Turn on the Database Monitor.

```
DBMON_ON = SETDBMONITOR("1");
```

WQ_OPEN Function

Purpose

The WQ_OPEN function opens a work queue item.

This function opens a work queue item record in the Work Queue Open Item table used by the Work Queues functionality. The function returns an XML document containing the opened work queue item.

Format

```
<identifier> = WQ_OPEN(<stem_identifier>);
```

Where

- <stem_identifier> is a stem identifier or expression that evaluates to a stem identifier. The corresponding tail identifiers provide the values for the work queue item. Available tail identifiers include:

  - **TYPE** (used for WQTYPECODE): The work queue item type. The type defines a number of attributes for the work queue item. Required.

  - **QUEUE** (used for WQQUEUECODE): The work queue. If not supplied, the default queue for the TYPE is used.

  - **PRODUCT** (used for PRODUCTCODE): The Oracle Utilities product associated with the work queue item. If supplied, the PRODUCT must have a corresponding record in the LODESTAR Product table. If not supplied, the default product for the TYPE is used.

  - **PRIORITYLEVEL**: The priority level for the work queue item. If not supplied, the default PRIORITYLEVEL for the TYPE is used.

  - **WORKBYTIME**: *Optional.* The time by which the item is expected to be resolved. If not supplied, it is calculated from the Default Work By Hours values of the TYPE.

  - **ASSIGNEDTOUSERID**: The User Id of the user to which the item is assigned. This value will change each time the item is unassigned, reassigned, resolved, approved, rejected, or closed. If supplied, the User ID must have a corresponding record in the Users table in the Security database.

  - **PROCESSNAME**: The business process associated with the work queue item. If supplied, the PRODUCT must have a corresponding record in the Business Process table. If not supplied, the default process for the TYPE is used.

  In addition to these, any custom parameters may be specified. See the *Oracle Utilities Energy Information Platform Configuration Guide* for more information about custom parameters.

Example

Open an ERROR type work queue item in the "QUEUE_1" work queue.

```
ERROR.TYPE = "ERROR";
ERROR.QUEUE = "QUEUE_1";
ERROR.PRODUCT = "BX";
ERROR.ASSIGNEDTOUSERID = "lou_p";
OPEN_ERROR_WQ = WQ_OPEN(ERROR);
```

Date/Time Functions

Date/Time functions are used to obtain date and time related information from the Oracle Utilities Data Repository, and to perform date and time related operations.

BILLINGHOURS Function

Purpose

The BILLINGHOURS returns number of hours in one or more user-specified billing periods.

This function returns the number of hours in one or more billing periods for an account. One value is returned (that is, the total for all specified billing periods, rather than one value for each) as a scalar numeric value.

Format

```
<identifier> = BILLINGHOURS(<start_bill_period_previous>,
<end_bill_period_previous>);
```

Where

*   <start_bill_period_previous>, <end_bill_period_previous> specifies the desired billing periods. **0** specifies the current bill period, **1** the previous bill period, and so on (the higher the number, the further back in time). If you omit both start- and stop-periods, the function returns the number of hours in the current period. See **Start and End Bill Period Parameters** under **Rules for Using Functions** in **Chapter 6**: **Rules Language Functions Overview** of the *Oracle Utilities Rules Language User's Guide* for details about specifying bill period parameters.

Examples

Get the total number of hours in the last three bill periods, including the current bill period:

```
BH = BILLINGHOURS(0, 2);
```

Get the total number of hours in the previous bill period:

```
BH = BILLINGHOURS(1);
```

Get the number of hours in the current bill period:

```
BH = BILLINGHOURS;
```

DATE Function

Purpose

The DATE function converts a date expressed as a string value into a format that the programs recognize as a date.

This function converts a date expressed in a text string or as a number into a format that the programs recognize as a date.

Formats

```
<identifier_date> = DATE(<date_identifier|date_constant>);
<identifier_date> = DATE(<number>, "GMT");
<identifier_date> = DATE(<number>, <input offset>, <return offset>);
<identifier_date> = DATE(<year>,<month>,<day>);
```

Where

- <date_identifier|date_constant> is either an identifier that contains a date (such as BILL_PERIOD or BILL_START), a constant in the format 'mm/dd/yyyy', 'mm/dd/yyyy hh:mm', 'yyyy-mm-dd' , or 'yyyy-mm-dd hh:mm', or an identifier or constant that contains a value that is the number of seconds since 01/01/1970 00:00:00 GMT. If a number is provided as the only parameter, the date is converted to the local timezone.

- <number> is the number of seconds since 01/01/1970 00:00:00 GMT.

- "GMT" is a keyword that indicates that the date returned is represented as the date/time in GMT. That is, the date returned is not converted to the local timezone.

- <input offset> is the number of hours added or subtracted from the <number> parameter, represented as "GMT +/- X", where X is the number of hours added (+) or subtracted (-). For example, "GMT+5" would add 5 hours to the <number> parameter. This parameter should be used if the <number> parameter is based on different timezone than GMT. If this parameter is included, the <return offset> parameter is required.

- <return offset> is the timezone that the returned date is to be converted to, represented as "GMT +/- X", where X is the number of hours plus (+) or minus (-) GMT 0. For example, Eastern Standard Time would be represented as "GMT-5". This parameter is required if the <input offset> parameter is used.

- <year> is either an identifier or a constant that contains a 4-digit year.

- <month> is either an identifier or a constant that contains a 2-digit month.

- <day> is either an identifier or a constant that contains a 2-digit day.

Example

The first four statements below assign a date constant to a date identifier. The last two statements apply the DATE function to convert the data type for the assigned values from STRING to DATE.

```
DTSTRG = "05/01/1997";
YRSTRG = "1997";
MOSTRG = "05";
DYSTRG = "01";
DATE1 = DATE(DTSTRG);
LABEL DATE1 "Date 1"
DATE2 = DATE(YRSTRG, MOSTRG, DYSTRG);
LABEL DATE2 "Date 2";
```

Result:

```
Date 1: 05/01/1997
Date 2: 05/01/1997
```

DATEFROMFLOAT Function

Purpose

The DATEFROMFLOAT function converts a float value to a date/time.

This function converts a float value into a format that the programs understand as a date, inverting the **DATETOFLOAT Function** on page 13-26. Returns a date.

Format

```
<identifier_float> = DATEFROMFLOAT(<identifier|float_expression>);
```

Where

• <identifier|float_expression> is either an identifier that contains a float value, or a float expression.

Example

Convert the float value assigned to the FLOAT identifier into a date.

```
FLOAT = 134264872460
FLOAT_DATE = DATEFROMFLOAT(FLOAT);
```

Result:

```
FLOAT_DATE = 11/07/2000 21:56:12
```

DATETIMEFROMSTRING Function

Purpose

The DATETIMEFROMSTRING function converts a string value into a format that the programs understand as a date, inverting the **DATETIMETOSTRING Function** on page 13-25. Returns a date/time.

Format

```
<identifier> = DATETIMEFROMSTRING(<date/time string>, <date/time
format string>);
```

Where

- <date/time string> is a string value that evaluates to a date and time in the same format as the <date/time string format string> parameter.

- <date/time format string> is valid date/time format. These include: "MM/dd/yyyy HH:mm:ss", "dd/MM/yyyy HH:mm:ss", "yyyy/MM/dd HH:mm:ss", "MM/dd/yy HH:mm:ss", "dd/MM/yy HH:mm:ss", "yy/MM/dd HH:mm:ss". Where "MM" = Month, "dd" = Day, "yy" or "yyyy" = Year, "HH" = Hour, "mm" = Minutes, "ss" = seconds. Only the above formats are supported.

Example

Convert the string value assigned to the STRING identifier into a date.

```
STRING = "11/07/2000 21:56:12"
STRING_DATE = DATETIMEFROMSTRING(STRING, "MM/dd/yyyy HH:mm:ss");
```

Result:

```
STRING_DATE = 11/07/2000 21:56:12
```

DATETIMETOSTRING Function

Purpose

The DATETIMETOSTRING function converts a date/time value into a string. Returns a string.

Format

```
<identifier> = DATETIMETOSTRING(<date_identifier|expression>, <date/
time format string>);
```

Where

- <date_identifier|expression> is a date identifier or expression.

- <date/time format string> is valid date/time format. These include: "MM/dd/yyyy HH:mm:ss", "dd/MM/yyyy HH:mm:ss", "yyyy/MM/dd HH:mm:ss", "MM/dd/yy HH:mm:ss", "dd/MM/yy HH:mm:ss", "yy/MM/dd HH:mm:ss". Where "MM" = Month, "dd" = Day, "yy or yyyy" = Year, "HH" = Hour, "mm" = Minutes, "ss" = seconds. Other formats can also be used. The slash ("/") can be substituted for another character such as "-". "MMMM" will return the full month name. "MMM" returns the abbreviated month name. Day formats work the same as month formats.

  Note: Only the six formats specified above can be converted back into a Date/Time using the **DATETIMEFROMSTRING Function** if needed.

Example

Convert the date value assigned to the DATE_TIME identifier into a string.

```
DATE_TIME = CURRENT_DATE
DATE_STRING = DATETIMETOSTRING(DATE_TIME, "MM/dd/yyyy HH:mm:ss");
```

Result:

```
DATE_STRING = "11/07/2000 21:56:12"
```

DATETOFLOAT Function

Purpose

The DATETOFLOAT function converts a date to a floating point number that can be stored as a determinant. It converts a date into a format that the programs understand as a floating point number. The float value is $(((((((((year * 16) + month) * 32) + day) * 32) + hour) * 64) + minute) * 64) + seconds)$. Returns a float.

Format

```
<identifier_date> = DATETOFLOAT(<identifier|date_expression>);
```

Where

- <identifier|date_expression> is either an identifier that contains a date (such as BILL_PERIOD or BILL_START) or a date expression in the format 'mm/dd/yyyy', 'mm/dd/yyyy hh:mm', 'yyyy-mm-dd' , or 'yyyy-mm-dd hh:mm'.

Example

Convert the current date into a float.

```
DATE_FLOAT = DATETOFLOAT(CURRENT_DATE);
```

Result:

```
DATE_FLOAT = 134264872540
```

DAY Function

Purpose

The DAY function returns the number of the day of the month: 1–31.

Finds the number of the day in the month for a specified date identifier (such as BILL_STOP), or for a user-specified date.

Format

```
<identifier> = DAY(<date_identifier|date_constant>);
```

Where

- <date_identifier|date_constant> is either an identifier that contains a date (such as BILL_PERIOD or BILL_START) or a date constant in the format 'mm/dd/yyyy'. If you include a time, it will be ignored.

Example

Add a surcharge if BILL_STOP falls in the first half of the month.

```
BILL_DATE = "05/15/1999"
MDAY = DAY(BILL_DATE);

/* Add high surcharge for use in beginning of the month */
IF (MDAY <= 15) THEN
    $SURCHARGE = $10;
ELSE
    $SURCHARGE = $5;
END IF;
```

DAYDIFF Function

Purpose

The DAYDIFF function returns number of days separating two dates.

The result of DAYDIFF is the first date minus the second date, in number of days. The result can be positive or negative. The difference between the same day is 0. The time of day is ignored (midnight of both dates is used). The result is not rounded, and may differ from the value of NUMDAYS. (NUMDAYS is a predefined identifier that automatically contains the number of days between the specified BILL_START and BILL_STOP for the billing period, rounded to the nearest day.)

Format

```
<identifier> = DAYDIFF(<date_identifier|date_constant>,
<date_identifier|date_constant>);
```

Where

- <date_identifier|date_constant> is either an identifier that contains a date (such as BILL_PERIOD or BILL_START) or a date constant in the form 'mm/dd/yyyy'.

Example

Determine how many days are between the account's first and second peaks:

```
PEAK1_DATE = "01/01/1999"
PEAK2_DATE = "01/15/1999"
NDAYS = DAYDIFF(PEAK2_DATE, PEAK1_DATE);
```

Result:

```
NDays = 14
```

DAYNAME Function

Purpose

The DAYNAME function returns the name of the day of the week, expressed as a text string (upper and lower case, initial capital letter); for example, "Sunday," "Monday," etc.

Format

```
<identifier> = DAYNAME(<date_identifier|date_constant>);
```

Where

- <date_identifier|date_constant> is either an identifier that contains a date (such as BILL_PERIOD or BILL_START) or a date constant in the form 'mm/dd/yyyy'.

Example

Get the day of the week on which the READ_DATE occurred.

```
DAY = DAYNAME(READ_DATE);
```

DBDATETIME Function

Purpose

The DBDATETIME function converts a date or time value into a string that is suitable for use in a database record key. See **Database Identifiers** on page 4-5 in the *Oracle Utilities Rules Language User's Guide* for information about keys. The format of the returned string will match the format that is required by your database software.

Format

```
<identifier> = DBDATETIME(<date_identifier|date_constant>);
```

Where

- <date_identifier|date_constant> is either an identifier that contains a date (such as BILL_PERIOD or BILL_START) or a date constant in the format 'mm/dd/yyyy'.

Examples

Convert the BILL_START date to a string.

```
DBDT = DBDATETIME(BILL_START);
```

Convert the date '11/11/1997' to a string.

```
DBDT = DBDATETIME('11/11/1997');
```

DBDT may now be used in a Table.Column query where start date is part of the key to the desired record.

HOUR Function

Purpose

The HOUR function finds the number of the hour in the day (0 through 23) for a specified date identifier (such BILL_STOP), or for a user-specified date.

Format

```
<identifier> = HOUR(<date_identifier|expression>);
```

Where

- <date_identifier|expression> is either an identifier that contains a date (such as BILL_PERIOD or BILL_START), or a date constant in the format 'mm/dd/yyyy hh:mm:ss'. If the parameter is an identifier, it must have been assigned a date value.

Example

Return the number of the hour for the date/time constant: '10/27/1997 10:30:45'.

```
HOUR = HOUR('10/27/1997 10:30:45');
LABEL HOUR "Read Time Hour";
```

Result:

```
Read Time Hour: 10
```

MINUTE Function

Purpose

The MINUTE function finds the number of the minute in the hour (0 through 59) for a specified date identifier (such BILL_STOP), or for a user-specified date.

Format

```
<identifier> = MINUTE(<date_identifier|expression>);
```

Where

- <date_identifier|expression> is either an identifier that contains a date (such as BILL_PERIOD or BILL_START) or a date constant in the format 'mm/dd/yyyy hh:mm:ss'. If the parameter is an identifier, it must have been assigned a date value.

Example

Return the number of the minute for the date/time constant: '10/27/1997 10:30:45'.

MINUTE = MINUTE('10/27/1997 10:30:45');
LABEL MINUTE "Read Time Minute";

Result:

Read Time Minute: 30

MONTH Function

Purpose

The MONTH function returns number of the month: January is 1, December is 12.

Format

```
<identifier> =  MONTH(<date_identifier|date_constant>);
```

Where

- <date_identifier|date_constant> is either an identifier that contains a date (such as BILL_PERIOD and BILL_START) or a constant in the form 'mm/dd/yyyy'.

Example

Add a surcharge for the month of August (8):

```
BILL_STOP = "08/30/1999"
M = MONTH(BILL_STOP);

IF (M = 8) THEN
    $SURCHARGE = $10;
ELSE
    $SURCHARGE = $0
END IF;
```

MONTHDIFF Function

Purpose

The MONTHDIFF function returns the number of months separating two dates (first date minus second date).

The result is the number of months separating the two dates, ignoring the day in the month. The months are subtracted and added to the difference of the years multiplied by 12.

Format

```
<identifier> = MONTHDIFF(<date_identifier|date_constant>,
<date_identifier|date_constant>);
```

Where

*   <date_identifier|date_constant> is either an identifier that contains a date (such as BILL_PERIOD and BILL_START) or a constant in the form 'mm/dd/yyyy'.

Example

Apply a 20% discount to the first year, 10% to the second year. (The year is determined by the number of months between the BILL_STOP date and the SERVICE_START_DATE date.)

```
SERVICE_START_DATE = '06/15/1995';

MDIFF = MONTHDIFF(BILL_STOP, SERVICE_START_DATE);

/* Compute discount, 20% first year, 10% second year */
IF (MDIFF <= 12) THEN
    DISCOUNT = .20;
ELSE
    IF (MDIFF <= 24) THEN
    DISCOUNT = .10;
ELSE
    DISCOUNT = 0;
    END IF;
END IF;

$EFFECTIVE_REVENUE = $NET_BILL * (1 - DISCOUNT);
```

MONTHHOURS Function

Purpose

The MONTHHOURS function returns the number of hours in one or more calendar months. It is identical to the **BILLINGHOURS Function** on page 13-21, except that it applies to calendar months rather than billing periods.

Format

```
<identifier> = MONTHHOURS(<start_month_previous>,
<end_month_previous>);
```

Where

- <start_month_previous>, <end_month_previous> **0** specifies the current month, **1** the previous month, and so on (the higher the number, the further back in time). The default end_month_previous is the same as the start_month_previous. If neither is supplied, 0 is assumed for both. For example, to get the number of hours in the last three months, you would specify: 'MONTHHOURS(0, 2)'.

  If you specify just one month, the function will return the number of hours in that month; for example, MONTHHOURS(2). If you leave off both start and end months, the function will return the number of hours in the current month.

Example

Change the default 730 to the correct number of hours for this month.

```
HOURS_PER_MONTH = MONTHHOURS();
```

MONTHNAME Function

Purpose

The MONTHNAME function returns the name of the month of the year, expressed as a text string ("February", etc.).

Format

```
<identifier> = MONTHNAME(<date_identifier|date_constant>;
```

Where

- <date_identifier|date_constant> is either an identifier that contains a date (such as BILL_PERIOD or BILL_START) or a date constant in the form 'mm/dd/yyyy'.

Example

Return the name of the month for the date '10/27/1997'.

MONTH_NAME = MONTHNAME('10/27/1997')
```
LABEL MONTH_NAME "Month Name";
```

Result:

```
Month Name: October
```

ROUNDDATE Function

Purpose

The ROUNDDATE function rounds date to nearest specified unit. It returns a date rounded back to the nearest unit—HOUR, DAY, WEEK, MONTH, or YEAR. It can be rounded to the exact time, one second before it, or to the end of the period.

Format

```
<identifier> = ROUNDDATE(<date_identifier|date_constant>, <day>,
<time>);
```

Where

- <date_identifier|date_constant> is either an identifier that contains a date (such as BILL_PERIOD and BILL_START) or a constant in one of three forms: 'mm/dd/yyyy', 'mm/dd/yyyy hh:mm', or 'mm/dd/yyyy hh:mm:ss'.

- <day> is a string constant or identifier that contains a day of the week or other rounding value. Valid days are "Saturday," "Sunday," "Monday," "Tuesday," "Wednesday," "Thursday," and "Friday". Day names are case insensitive, and only the first three letters are actually checked. In addition, the following can be used: "HOUR", "DAY", "WEEK", "MONTH", or "YEAR". These must be completely spelled out.

  Note: "Mon" is Monday, not Month. A week is from Sunday midnight through the following Saturday.

- <time> is a string or time constant or identifier that contains the rounding time. Valid values are "START" (0 minutes, 0 seconds—the default), "END" (the rounded value plus one period minus one second) or a time constant of the form 'hh:mm:ss' or 'hh:mm'. The latter two explicitly specify the hour, minute, and second (ss is 0 in 'hh:mm').

  Note: "END" means the end of the period—end of day, or week, month. The others apply to the beginning of the period. The other allowed value is "1SEC-" (one second before rounded date).

Example

Get calendar day-based start and stop times.

```
BILL_START_ROUND = ROUNDDATE (BILL_START, "DAY");
BILL_STOP_ROUND = ROUNDDATE (BILL_STOP, "DAY", "1SEC-");
```

SAMEWEEKDAYLASTYEAR Function

Purpose

The SAMEWEEKDAYLASTYEAR function returns the closest date from a year before the supplied date that is on the same day of the week.

Format

```
<identifier> = SAMEWEEKDAYLASTYEAR(<date_identifier>);
```

Where

- <date_identifier> is an identifier that contains a date (such as BILL_PERIOD or BILL_START) or a date constant in the format 'mm/dd/yyyy'.

Example

Find the closest date from the last year to the BILL_START and BILL_STOP dates that are on the same days of the week.

```
HIST_START_DATE = SAMEWEEKDAYLASTYEAR(BILL_START);
HIST_STOP_DATE = SAMEWEEKDAYLASTYEAR(BILL_STOP);
```

SECOND Function

Purpose

The SECOND function finds the number of the second in the hour (0 through 59) for a specified date identifier (such BILL_STOP), or for a user-specified date.

Format

```
<identifier> = SECOND(<date_identifier|expression>);
```

Where

•   <date_identifier|expression> is either an identifier that contains a date (such as BILL_PERIOD or BILL_START) or a date constant in the format 'mm/dd/yyyy hh:mm:ss'. If the parameter is an identifier, it must have been assigned a date value.

Example

Return the number of the second for the date/time constant: 10/27/1997 10:30:45.

SECOND = SECOND('10/27/1997 10:30:45');
LABEL SECOND "Read Time Second";

Result:

Read Time Second: 45

WEEKDAY Function

Purpose

The WEEKDAY function returns the day of the week from Sunday *(Sunday=0, Monday=1, ..., Saturday=6)* expressed as a scalar integer value.

Format

```
<identifier> = WEEKDAY(<date_identifier|expression>);
```

Where

- <date_identifier|expression> is either an identifier that contains a date (such as BILL_PERIOD or BILL_START) or a date constant in the format 'mm/dd/yyyy'. If the parameter is an identifier, it must have been assigned a date value. The parameter can also be a date expression.

Example

Return the name of the weekday for the date 10/27/1997.

```
WEEKDAY = WEEKDAY('10/27/1997');
LABEL WEEKDAY "Weekday";
```

Result:

```
Weekday: 1 (Monday)
```

WEEKDIFF Function

Purpose

The WEEKDIFF function returns the number of weeks separating two dates (first date minus second date). The result ignores the day of the week—both dates are rounded down to Sunday, subtracted to get the number of days difference, which is divided by 7 to get the number of weeks difference.

Format

```
<identifier> = WEEKDIFF(<date_identifier|date_constant>,
<date_identifier|date_constant>);
```

Where

- <date_identifier|date_constant> is either an identifier that contains a date (such as BILL_PERIOD and BILL_START) or a constant in the format 'mm/dd/yyyy'.

Example

Compute the number of weeks in a bill period:

```
NUMWK = WEEKDIFF(BILL_STOP, BILL_START);
```

YEAR Function

Purpose

The YEAR function returns the number of the year in a date — all four digits.

Format

```
<identifier> = YEAR(<date_identifier|date_constant>);
```

Where

- <date_identifier|date_constant> is either an identifier that contains a date (such as BILL_PERIOD and BILL_START) or a constant in the form 'mm/dd/yyyy'.

Example

Label the current year "Year of this Bill".

```
Y = YEAR(BILL_START);
LABEL Y "Year of this Bill";
```

YEARDAY Function

Purpose

The YEARDAY function returns the number of days of the year since January 1, based on a date identifier or date string: 0 through 364 (365 for leap years). January 1 is 0. Returns a scalar integer value.

Format

```
<identifier> = YEARDAY(<date_identifier|expression>);
```

Where

- <date_identifier|expression> is either an identifier that contains a date (such as BILL_PERIOD or BILL_START) or a date constant in the format 'mm/dd/yyyy'. If the parameter is an identifier, it must have been assigned a date value. The parameter can also be a date expression.

Example

Return the number of the yearday for the date 10/27/1997.

```
YEARDAY = YEARDAY('10/27/1997');
LABEL YEARDAY "Yearday";
```

Result:

```
Yearday: 299
```

YEARSTR Function

Purpose

Like the **YEAR Function** on page 13-42, the YEARSTR function returns the number without the comma to mark the thousands.

Format

```
<identifier> = YEARSTR(<date_identifier|date_constant>);
```

Where

- <date_identifier|date_constant> is either an identifier that contains a date (such as BILL_PERIOD and BILL_START) or a constant in the form 'mm/dd/yyyy'.

Example

Get the current bill month with the year:

```
BILLING_MONTH= MONTH(BILL_PERIOD);
BILLING_YEAR= YEARSTR(BILL_PERIOD);
BILL_MONTH = " " + BILLING_MONTH + "/" + BILLING_YEAR;
```

If BILL_PERIOD is 10/18/1997, then BILL_MONTH returns "10/1997".

Historical-Data Functions

Historical-Data functions are used to obtain information about and perform operations on historical data stored in the Oracle Utilities Data Repository.

In many of the historical functions, the last two parameters are <start_month_previous>, <end_month_previous>. For these parameters, **0** signifies the current bill period, **1** the previous bill period, and so on (the higher the number, the further back in time). The end month must be greater than or equal to the start month. These are optional, though start month must be specified if end month is. The default start month is 0 or the current month. The default end month is the start month. If these two parameters are omitted the value is computed over the current month only. See **Start and End Bill Period Parameters** under **Rules for Using Functions** in **Chapter 6**: **Rules Language Functions Overview** of the *Oracle Utilities Rules Language User's Guide* for additional details about specifying bill period parameters.

There are many situations where an error is possible. In Oracle Utilities Rate Management and the Trial Calculation functions in Oracle Utilities Billing Component and Data Manager, zero is returned if values are missing or NULL. For example, in Oracle Utilities Rate Management if the start or stop month is greater than the number of months of bill history data available, their time is set to one second before the last bill start.

The determinant referencing rules in actual bill calculations in Oracle Utilities Billing Component are more strict:

*   If the start month to historical functions is one and there is only the current value, zero is returned. However, if the start month is greater than one and greater than the number of historical values, it is an error.

*   If the stop month is greater than the number of historical values, it is set to the last period with data.

*   If all values between the start and stop months, inclusive, are missing or NULL, and there are values after the stop month, it is an error.

The above rules do not apply to the **HASVALUE Function** and the **HISTCOUNT Function**.

Historical Values

Historical functions retrieve values from historical determinants and dates. The maximum number of values for a determinant or date retrieved from the database is 1200 values. The maximum number of values for a computed determinant or date is 36 values.

COMPSUM Function

Purpose

The COMPSUM totals values for a historical identifier over specified bill periods.

Format

```
<identifier> = COMPSUM(<historical_identifier>,
<start_bill_period_previous>, <end_bill_period_previous>);
```

Where

- <historical_identifier> is an identifier containing historical determinant values (computed, or loaded from the Oracle Utilities Data Repository).

- <start_bill_period_previous>, <end_bill_period_previous> **0** signifies the current bill period, **1** the previous bill period, and so on (the higher the number, the further back in time). If you omit both start and end periods, the value will be computed over the current month only. See **Start and End Bill Period Parameters** under **Rules for Using Functions** in **Chapter 6**: **Rules Language Functions Overview** of the *Oracle Utilities Rules Language User's Guide* for additional details about specifying bill period parameters.

Example

Compute an average for KWH over the past 12 bill periods (excluding the current):

```
CS = COMPSUM(KWH, 1, 12);
AVE_KWH = CS/12;
```

HISTCOUNT Function

Purpose

The HISTCOUNT function returns a count of the historical values, typically loaded by another function in the rate form. You can optionally exclude zero values from the count.

Any historical function, as well as the DETERMINANT Statement, can load historical values for up to 36 bill periods. If there are no historical functions or a DETERMINANT Statement present in the rate form to load historical values, the HISTCOUNT function returns a count of 1 (indicating just the current bill period).

Format

```
<identifier> =
HISTCOUNT(<historical_identifier|date_identifier>,<type>);
```

Where

- <historical_identifier> is the identifier containing historical values (computed, or loaded from the Oracle Utilities database).

- <date_identifier> is one of BILL_PERIOD, BILL_START, or BILL_STOP.

- <type> (*Optional*) is either "ALL" or "NON_ZERO". "NON_ZERO" excludes zero values from the count. "ALL" is the default. It is used only with determinant IDs.

Examples

Return a count of the number of historic, nonzero kWh values loaded.

```
NUMKW = HISTCOUNT(KWH,"NON_ZERO");
```

Count the number of kWh values loaded for the account (excluding that for the current bill period).

```
X = HISTCOUNT(KWH, "NON_ZERO") - 1;
TOTAL_KWH = COMPSUM(1,X)
```

The value returned then becomes a parameter in the COMPSUM function, which would sum the values for all bill periods loaded, excluding the current bill period.

HISTMAX Function

Purpose

The HISTMAX function compares two or more sets of historical values and/or constants, and returns the greater of each value in the comparison.

You can specify any number of historical determinants and/or constants as parameters, but you must specify at least two parameters. Any of the parameters can be a determinant with historical values.

The result is a determinant with historical values. (The result contains the same number of values as the parameter with the most historical values.) Each value in the result is the greatest value among those compared. The program compares each value in a set of historical determinants with the corresponding values in other historical determinants, or with the scalar value for a parameter that is not a historical determinant. If the number of historical values differs in two parameters, zeros are used as fillers for the missing values in the parameter with fewer values.

Format

```
<determinant_identifier> =  HISTMAX(<identifier|constant>,
<identifier|constant>, ...);
```

Where

- <identifier|constant> is either an identifier that contains a floating-point number (such as a determinant identifier) or a floating-point constant.

Example

Get the maximum of the on-peak peak and the off-peak peak for each bill period:

```
MAX_ON_OFF_KWH = HISTMAX(PKKWH, OPKWH);
```

Result:

If the values for PKKWH for the last three bill periods were 502, 712, and 499, and for OPKWH they were 652, 519, and 700, the result of the sample statement shown above would be:

```
MAX_ON_OFF_KWH = 652, 712, 700
```

HISTMIN Function

Purpose

The HISTMIN function compares two or more sets of historical values and/or constants, and returns the smallest of each value in the comparison.

This function is the same as HISTMAX, except that it finds the minimum of the corresponding elements of the parameters.

Format

```
<determinant_identifier> = HISTMIN(<identifier|constant>,
<identifier|constant>, ...);
```

Where

- <identifier|constant> is either an identifier that contains a floating-point number (such as a determinant identifier) or a floating-point constant.

Example

Get the minimum of the on-peak peak and the off-peak peak for each bill period:

```
MIN_ON_OFF_KWH = HISTMIN(PKKWH, OPKWH);
```

Result:

If the values for PKKWH for the last three bill periods were 502, 712, and 499, and for OPKWH they were 652, 519, and 700, the result of the sample statement shown above would be:

```
MIN_ON_OFF_KWH = 502, 519, 499
```

HISTMINNZ Function

Purpose

The HISTMINNZ function compares two or more sets of historical values and/or constants, and returns the smallest of each nonzero value in the comparison.

This function is the same as HISTMIN, except that it finds the nonzero minimum of the corresponding elements of the parameters.

Format

```
<determinant_identifier> =  HISTMINNZ(<identifier|constant>,
<identifier|constant>, ...);
```

Where

- <identifier|constant> is either an identifier that contains a floating-point number (such as a determinant identifier) or a floating-point constant.

Example

Get the nonzero minimum of the on-peak peak and the off-peak peak for each bill period:

```
MIN_ON_OFF_KWH = HISTMINNZ(PKKWH, OPKWH);
```

Result

If the values for PKKWH for the last four bill periods were 100, 50, 101, and 70, and for OPKWH they were 80, 0, 0, and 90, the result of the sample statement shown above would be:

```
MIN_ON_OFF_KWH = 80, 50, 101, 70
```

HISTVALUE Function

Purpose

The HISTVALUE function returns a specified historical determinant value for a specified bill period.

Format

```
<identifier> = HISTVALUE(<historical_identifier|date_identifier>,
<bill_period_previous>);
```

Where

- <historical_identifier> is an identifier containing historical values loaded from the Oracle Utilities database, or computed in the rate form.

- <date_identifier> is one of BILL_PERIOD, BILL_START, or BILL_STOP.

- <bill_period_previous> is the desired bill period. Use **0** to specify the current bill period, **1** for the previous bill period, and so on (the higher the number, the further back in time).

Example

Get the KWH value for the previous December (based on monthly bill periods with February as the current month).

```
DEC_KWH = HISTVALUE(KWH, 2);
```

MAXNRANGE Function

Purpose

The MAXNRANGE function returns the account's *n*th maximum value for the specified bill determinant over the specified date range. This is the same as the **MAXRANGE Function** on page 13-53, except that the *n*th maximum is returned. The return value is **0** if *n* is less than 1 or greater than the number of historic values.

Format

```
<identifier> = MAXNRANGE(<n>, <historical_identifier>,
<start_bill_period_previous>, <end_bill_period_previous>);
```

Where

- <n> is an identifier or integer constant that indicates which peak to find; e.g., first, second, third, etc.

- <historical_identifier> is a determinant identifier that contains historical values (computed, or loaded from the Oracle Utilities database).

- <start_bill_period_previous>, <end_bill_period_previous> specifies the bill periods to be loaded using the following convention: **0** is the current bill period, **1** is the previous bill period, and so on (the higher the number, the further back in time). The end_bill_period_previous must be greater than or equal to the start_bill_period_previous. The default start_bill_period_previous is 0 (the current period). The default end_bill_period_previous is the last period of data available for the account. If you specify a start but no end, the default end is the last period of data for that determinant stored for the account. See **Start and End Bill Period Parameters** under **Rules for Using Functions** in **Chapter 6**: **Rules Language Functions Overview** of the *Oracle Utilities Rules Language User's Guide* for additional details about specifying bill period parameters.

Example

Get the 3rd maximum historic demand from the last 12 bill periods:

```
THIRD_RATCH_KW = MAXRANGE(3, KW, 1, 12);
```

MAXRANGE Function

Purpose

The MAXRANGE function returns the account's maximum value for the specified bill determinant over the specified date range.

Format

```
<identifier> = MAXRANGE(historical_identifier>,
<start_bill_period_previous>, <end_bill_period_previous>);
```

Where

- <historical_identifier> is a determinant identifier that contains historical values (computed, or loaded from the Oracle Utilities database).

- <start_bill_period_previous>, <end_bill_period_previous> specifies the bill periods to be loaded, using the following convention: **0** is the current bill period, **1** is the previous bill period, and so on (the higher the number, the further back in time). The end_bill_period_previous must be greater than or equal to the start_bill_period_previous. The default start_bill_period_previous is 0 (the current period). The default end_bill_period_previous is the last period of data available for the account. If you specify a start but no end, the default end is the last period of data for that determinant stored for the account. See **Start and End Bill Period Parameters** under **Rules for Using Functions** in **Chapter 6**: **Rules Language Functions Overview** of the *Oracle Utilities Rules Language User's Guide* for additional details about specifying bill period parameters.

Example

Get the maximum historic demand from the last 12 bill periods:

```
RATCH_KW = MAXRANGE(KW, 1, 12);
```

MINRANGE Function

Purpose

The MINRANGE function finds the minimum of the specified billing determinant between the start bill period specified through the end bill period specified (inclusive).

Format

```
<identifier> = MINRANGE(<historical_identifier>,
<start_bill_period_previous>, <end_bill_period_previous>);
```

Where

- <historical_identifier> is the identifier for a determinant containing historical values (computed or loaded from the Oracle Utilities database).

- <start_bill_period_previous>, <end_bill_period_previous> specifies the bill periods to be loaded, using the following convention: **0** is the current bill period, **1** is the previous bill period, and so on (the higher the number, the further back in time). The end_bill_period_previous must be greater than or equal to the start_bill_period_previous. The default start_bill_period_previous is 0 (the current period). The default end_bill_period_previous is the last period of data available for the account. If you specify a start but no end, the default end is the last period of data for that determinant stored for the account. See **Start and End Bill Period Parameters** under **Rules for Using Functions** in **Chapter 6**: **Rules Language Functions Overview** of the *Oracle Utilities Rules Language User's Guide* for additional details about specifying bill period parameters.

Example

Get the minimum historical demand for the last 12 months.

```
MIN_KW_DEMAND = MINRANGE(KW, 1, 12);
```

Internal Functions

Internal functions compute different types of values from billing determinant records stored in the Oracle Utilities Data Repository.

COMPIKVA Function

Purpose

The COMPIKVA function computes the inverse of kVA.

IKVA stands for *inverse kVA*. This function returns square root of *kVA* squared, minus *rkva_kw* squared, where *rkva_kw* is either *rkVA* or *kW*. If *rkva_kw* is *rkVA*, the result of the function is *kW*; if *rkva_kw* is *kW*, the result is *rkVA*. Returns zero if *rkva_kw* is greater than *kVA*. This function returns a scalar numeric value.

Format

```
<identifier> = COMPIKVA(<kva>, <rkva_kw>);
```

Where

- <kva> is either an identifier that contains a floating-point number (such as the KVA bill determinant identifier) or a floating-point constant.

- <rkva_kw> is either an identifier that contains a floating-point number (such as the KW determinant identifier) or a floating-point constant.

Examples

Get RKVA from KVA and KW:

```
RKVA = COMPIKVA (KVA, KW);
```

Get KW from KVA and RKVA:

```
KW = COMPIKVA (KVA, RKVA);
```

COMPKVA Function

Purpose

The COMPKVA function computes kVA.

This function returns the square root of the sum of the squares, computing *kVA* from *kVAR* and *kW*. This function returns a scalar numeric value.

Format

```
<identifier> = COMPKVA(<kvar>, <kw>);
```

Where

- <kvar> is either an identifier that contains a floating-point number (such as the kVAR bill determinant identifier) or a floating-point constant.

- <kw> is either an identifier that contains a floating-point number (such as the kW bill determinant identifier) or a floating-point constant.

Example

Get kVA from kVAR and kW.

```
KVA = COMPKVA(KVAR, KW);
```

COMPKVARHFROMKQKW Function

Purpose

The COMPKVARHFROMKQKW function computes *kVARh* from *kQh* and *kWh*.

This function returns the value of $(2*kQh - kWh)/\text{sqrt}(3)$, computing *kVARh* from *kQh* and *kWh*. This function returns a scalar numeric value.

Format

```
<identifier> = COMPKVARHFROMKQKW(<kQh>, <kWh>)
```

Where

- <kQh> is either an identifier that contains a floating-point number (such as the kQh bill determinant identifier) or a floating-point constant.

- <kWh> is either an identifier that contains a floating-point number (such as the kWh bill determinant identifier) or a floating-point constant.

Example

Get kVARh from kQh and kWh.

```
KVARH = COMPKVARHFROMKQKW(KQH, KWH);
```

COMPLF Function

Purpose

The COMPLF function computes load factor using the formula: ((kWh/HOURS_PER_MONTH)/kW). Returns a scalar numeric value.

Format

```
<identifier> = COMPLF(<kWh>, <kW>);
```

Where

- <kWh> is either an identifier that contains a floating-point number (usually a bill determinant identifier representing an energy value) or a floating-point constant.

- <kW> is either an identifier that contains a floating-point number (usually a bill determinant identifier representing a demand value) or a floating-point constant.

Example

Get Load Factor based on kWh and kW.

```
LF = COMPLF(KWH, KW);
```

Note about HOURS_PER_MONTH: The function automatically uses the actual number of hours in the current billing period for the account. To specify a particular number of hours in the rate schedule, use the predefined identifier HOURS_PER_MONTH in an **Assignment Statement** to set HOURS_PER_MONTH equal to a desired constant value that represents the number of hours.

IDATTR Function

Purpose

The IDATTR function returns an attribute value for a selected identifier.

All identifiers have a set of attributes that define how you can use them in rate forms ("type," "datatype," "label," and so on.) IDATTR returns the value stored for a selected attribute associated with a specified identifier.

Format

```
<identifier> = IDATTR(<identifier>, <attribute>);
```

Where

- <identifier> is any identifier.

- <attribute > is any one of the attribute names listed below ("TYPE", "DATATYPE", etc.).

Attribute_Name	Values that can be returned
"TYPE"	"TEMP", "DETERMN", "DBASE", "INPUT", "REVENUE", "INTDATA", "REPORT"
"DATATYPE"	"NONE", "FLOAT", "STRING", "INTEGER", "DATE", "HANDLE"
"LABEL"	Assigned label of the identifier; if no label, the name of the identifier.
"SET"	**0** (Not set) **1** (Set via initial input) **2** (Set via initial assignment (ASSIGNMENT, ALL, BLOCK, or other statements) **3** Set via NOVALUE Statement **4** Database value equal to NULL
"FIXED"	**0** (cannot be changed in computation) **1** (can be changed in computation).
"TOTAL"	**0** (not the TOTAL revenue identifier) **1** (is the TOTAL revenue identifier).
"HISTCOUNT"	Number of values, current plus historical. 0 if not set.
"STATUSCODE"	Applies only to bill determinant identifiers that have an associated Status Code column in the Bill History Table (e.g., KWH). Returns the status code stored in the column if it exists; otherwise, returns "".

Revenue identifiers used in ALL or BLOCK statements have additional values that can be returned via this function. These are:.

Revenue Identifiers*	Value Returned
"CHARGE"	The value for a charge from an ALL or BLOCK statement.
"USAGE"	The number of billing units from an ALL or BLOCK statement.
"BDNAME"	The name of the billing determinant used in an ALL or BLOCK statement.

*Return 0 or "" if the revenue identifier is or was not assigned via an ALL or BLOCK statement.

Example

Find the "DATATYPE" of the KW identifier.

```
KW_DATATYPE = IDATTR(KW, "DATATYPE");
```

FLAG Function

Purpose

The FLAG function returns the setting for analysis flags.

The two flags currently supported are SAVE and INTD. The function returns 1 if the flag is on, 0 if it is off. The SAVE flag indicates that data—determinants, interval data, etc.—will be saved. It is off in all Oracle Utilities Rate Management analyses. The INTD flag indicates that interval data operations are performed. It is off for Bill Frequency and Typical Bill calculations.

Format

```
<identifier> = FLAG(<"SAVE"|"INTD">);
```

Where

- <"SAVE"> returns 1 if SAVEs are enabled; else, returns 0.

- <"INTD"> returns 1 if interval data functions are enabled; else, returns 0.

Example

Are interval data functions enabled?

```
INTD_ENABLE = FLAG("INTD");
```

LF2KW Function

Purpose

The LF2KW function computes kW from kWh and load factor.

Computes kW using the formula ((kWh/HOURS_PER_MONTH) / lf). It is the inverse of the **LF2KWH Function** on page 13-63.

Format

```
<identifier> = LF2KW(<lf>, <kwh>);
```

Where

*   <lf> is either an identifier that contains a floating-point number or a floating-point constant.

*   <kwh> is either an identifier that contains a floating-point number (usually a determinant identifier representing an energy value) or a floating-point constant.

Example

Get kW, given load factor and kWh:

```
KW = LF2KW(LOADFCTR, KWH);
```

Note about HOURS_PER_MONTH: The function automatically uses the actual number of hours in the current billing period for the account. To specify a particular number of hours in the rate schedule, use the predefined identifier HOURS_PER_MONTH in an **Assignment Statement** to set HOURS_PER_MONTH equal to a desired constant value that represents the number of hours.

LF2KWH Function

Purpose

The LF2KWH function computes kWh from kW and load factor.

Computes kWh using the formula ((lf * kW) * HOURS_PER_MONTH).

Format

```
<identifier> = LF2KWH(<lf>, <kw>);
```

Where

- <lf> is either an identifier that contains a floating-point number or a floating-point constant.

- <kw> is either an identifier that contains a floating-point number (usually a determinant identifier representing a demand value) or a floating-point constant.

Typically, the value for load factor <lf> is a constant specified in a contract.

Example

If the contract called for a 35% load factor, you would supply:

```
CONTRACT_KWH = LF2KWH(.35, KW);
```

Note about HOURS_PER_MONTH: The function automatically uses the actual number of hours in the current billing period for the account. To specify a particular number of hours in the rate schedule, use the predefined identifier HOURS_PER_MONTH in an **Assignment Statement** to set HOURS_PER_MONTH equal to a desired constant value that represents the number of hours.

MAXKW Function

Purpose

The MAXKW function sets a value for the special identifier AUXILIARY_DEMAND.

This function indicates which determinant was used to compute a customer's bill. It sets the value of the special identifier AUXILIARY_DEMAND. A value of **1** means that the kW determinant was used to compute the bill; **2** means contract kW, **3** means historical kW, and **4** means a minimum was used. Its value is **0** if not set. These values appear on reports. The return value is the same as if the **MAX Function** on page 11-22 was used.

Format

```
<identifier> = MAXKW(KW, CKW, HKW, <min_kw>);
```

Where

- kW, CkW, HkW supply the determinant metered kW for the first parameter, contract kW for the second, and historical kW for the third, as shown. Alternative values or variables are not allowed for the first three parameters. If you do not want to use one of them, use 0.

- The fourth parameter <min_kw> is optional. Supply a constant or variable that represents the minimum, if desired.

Example

Determine the determinant used to calculate the customer's bill.

```
BILL_DETER = MAXKW(KW, CKW, HKW)
```

POWERFACTOR Function

Purpose

The POWERFACTOR function returns the ratio of real power (kWh) to apparent power (kVARh) for any given load and time.

Returns the value of $1/sqrt(1 + (kVARh/kWh)^{**}2)$, computing the power factor from kVARh and kWh. Returns a scalar numeric value.

Format

```
<identifier> = POWERFACTOR(<kVARh>, <kWh>)
```

Where

- <kVARh> is either an identifier that contains a floating-point number (such as the kVARh bill determinant identifier) or a floating-point constant.

- <kWh> is either an identifier that contains a floating-point number (such as the kWh bill determinant identifier) or a floating-point constant.

Example

Determine the ratio of real power to apparent power.

```
RP_RATIO = POWERFACTOR(KVARH, KWH)
```

READING2USAGE Function

Purpose

The READING2USAGE function returns the computed usage of a selected billing determinant.

Returns the usage for a selected billing determinant computed as:
Usage = ((current identifier value - previous identifier value) * meter multiplier) + meter offset.
Returns a scalar numeric value.

Format

```
<identifier> = READING2USAGE(<determinant_identifier> [,
<meter_multiplier>[,<meter_offset>]])
```

Where

- <determinant_identifier> is a determinant identifier that contains a current value and at least one historical value. If the current value is less than the previous value (meter rollover), the first power of ten greater than the historical value is added to the current value. For example, if the current value is 444 and the historical value is 1333, the function would add 10,000 (or 10 to the power of 4, which is the first power of ten greater than 1333) to the current value before calculating the usage.

- <meter_multiplier> (*Optional)* is an appropriate meter multiplier value. The default value is 1.0.

- <meter_offset> (*Optional)* is an appropriate meter offset value. The default value is 0.0.

Example

Determine the usage of kWh, based on a meter multiplier of 100 and a meter offset of 2.

```
KWH_USAGE = READING2USAGE(KWH, 100, 2)
```

Season-Based Functions

Season-based functions are used to obtain information about and perform operations on season periods stored in the Oracle Utilities Data Repository.

AVGSEASON Function

Purpose

The AVGSEASON function finds the average value for a historical determinant over a specified season period for a given account. It returns a scalar numeric value. Only the values that fall within the season and (optionally) the specified bill periods are used in the averaging; because the periods that make up a season are not necessarily consecutive, the values used in averaging may not always be consecutive.

Format

```
<identifier> = AVGSEASON(<season_period_name>, <historical_identifier>,
<start_bill_period_previous>, <end_bill_period_previous>;
```

Where

- <season_period_name> is the name of a season period, as defined in a season schedule. The season period must belong to the season schedule in effect. See "How to Specify a Season Schedule," below.

- <historical_identifier> is the identifier for a determinant containing historical values (computed, or loaded from the Oracle Utilities Data Repository).

- <start_bill_period_previous>, <end_bill_period_previous> **0** indicates the current bill period, **1** the previous bill period, and so on (the higher the number, the further back in time). To get the average for the last three billing periods, including the current one, you would specify: AVGSEASON(0, 2). If you omit both start and end, the value will be computed over all values available. See **Start and End Bill Period Parameters** under **Rules for Using Functions** in **Chapter 6**: **Rules Language Functions Overview** of the *Oracle Utilities Rules Language User's Guide* for additional details about specifying bill period parameters.

Example

Find the average kWh value for the SUMMER season using all available values.

```
SUMMER_AVE_KWH = AVGSEASON("SUMMER", KWH);
```

How to Specify a Season Schedule

Season schedules are stored in the Oracle Utilities Data Repository. You can view, create, or modify a Season Schedule using Data Manager. See **Season Schedules** in **Chapter 7**: **Maintaining Data** in the *Data Manager User's Guide* for more information. Each season schedule consists of two or more periods, such as "Summer" and "Winter". Each period covers a specific date range, such as 04/01/1998 through 09/30/1998.

When you run a billing or analysis program using a rate form that includes a season reference, you must specify which season schedule in the database to apply.

There are two ways to do this:

- Specify a season schedule by selecting **Tools-›Options-›Rate Analysis**, then making your selection under **Default Season Schedule**.

- Specify it in the rate form by assigning the season schedule name to the special identifier SEASON_SCHEDULE_NAME. For example, if the database contained a season schedule called SEASON1, you could include the following Assignment Statement near the beginning of your rate form:

```
SEASON_SCHEDULE_NAME = SEASON1;
```

The season schedule specified via Options is the default. A season schedule specified in a rate form overrides the Options value.

Oracle Utilities Billing Component and Oracle Utilities Rate Management use the bill period's stop date to determine which season period the bill period belongs to (that's the default). To specify a different date, apply the BILL_PERIOD_SELECT identifier (for more information about BILL_PERIOD_SELECT, see the *Oracle Utilities Rules Language User's Guide*).

MAXSEASON Function

Purpose

The MAXSEASON function finds the maximum value for a historical determinant during a specified season period for an account. It returns a scalar numeric value. Only the values that fall within the season and (optionally) the specified bill periods are used in the evaluation; because the periods that make up a season are not necessarily consecutive, the values used may not always be consecutive.

Format

```
<identifier> = MAXSEASON(<season_period_name>,
<historical_identifier>, <start_bill_period_previous>,
<end_bill_period_previous>);
```

Where

- <season_period_name> is the name of a season period, as defined in a season schedule. The season period must belong to the season schedule in effect. See **How to Specify a Season Schedule** on page 13-67, under the description for the AVGSEASON function, for important details about season schedules.

- <historical_identifier> is a determinant identifier that contains historical values (computed, or loaded from the Oracle Utilities database).

- <start_bill_period_previous>, <end_bill_period_previous> specifies the bill periods to be loaded using the following convention: **0** is the current bill period, **1** is the previous bill period, and so on (the higher the number, the further back in time). The end_bill_period_previous must be greater than or equal to the start_bill_period_previous. The default start_bill_period_previous is 0 (the current period). The default end_bill_period_previous is the last period of data available for the account. If you specify a start but no end, the default end is the last period of data for that determinant stored for the account. See **Start and End Bill Period Parameters** under **Rules for Using Functions** in **Chapter 6**: **Rules Language Functions Overview** of the *Oracle Utilities Rules Language User's Guide* for additional details about specifying bill period parameters.

Example

Find the maximum kWh value for the SUMMER season for the last three bill periods, including the current one.

```
SUMMER_MAX_KWH = MAXSEASON("SUMMER", KWH, 0, 2);
```

MINSEASON Function

Purpose

The MINSEASON function finds the minimum historical value of the specified billing determinant for the bill periods in the season, from the start bill period specified through the end bill period specified, inclusive.

Format

```
<identifier> = MINSEASON(<season_period_name>,
<historical_identifier>, <start_bill_period_previous>,
<end_bill_period_previous>);
```

Where

- <season_period_name> is the name of a season period, as defined in a season schedule. The season period must belong to the season schedule in effect. See "Specifying a Season Schedule," under the description for the AVGSEASON function, for important details about the Season Schedules.

- <historical_identifier> is the identifier for a determinant containing historical values (computed, or loaded from the Oracle Utilities database).

- <start_bill_period_previous>, <end_bill_period_previous> specifies the bill periods to be loaded, using the following convention: **0** is the current bill period, **1** is the previous bill period, and so on (the higher the number, the further back in time). The end_bill_period_previous must be greater than or equal to the start_bill_period_previous. The default start_bill_period_previous is 0 (the current period). The default end_bill_period_previous is the last period of data available for the account. If you specify a start but no end, the default end is the last period of data for that determinant stored for the account. See **Start and End Bill Period Parameters** under **Rules for Using Functions** in **Chapter 6**: **Rules Language Functions Overview** of the *Oracle Utilities Rules Language User's Guide* for additional details about specifying bill period parameters.

Example

Find the minimum kWh value for the SUMMER season for the last three bill periods, including the current one.

```
SUMMER_MIN_KWH = MINSEASON("SUMMER", KWH, 0, 2);
```

MONTHLYMERGE Function

Purpose

The MONTHLYMERGE function combines all values in each bill month according to type.

This is used when there are multiple bill periods in a bill month, and values for the bill months are needed. Returns a set of historical values, one per bill month. After the result has been assigned to an identifier, the identifier can be used in the **MAXRANGE Function** on page 13-53 or the **SEASONVALUE Function** on page 13-72.

Format

```
<identifier> = MONTHLYMERGE(<determinant_identifier|date_identifier>,
<type>)
```

Where

- <determinant_identifier> is an identifier containing determinant values (computed, or loaded from the Oracle Utilities Data Repository).

- <date_identifier> is an identifier that contains a date (such as BILL_PERIOD or BILL_START). Must be an identifier assigned a date value.

- <type> is one of the following: "TOTAL", "AVERAGE", "AVG", "MAXIMUM", "MAX", "START", or "END". Only "START" and "END" apply to date identifiers. If the type is TOTAL, the values in the same bill month are added; if MAXIMUM, the maximum is taken.

Example

Return the total of the kWh for all bill history records within the billing month:

```
MERGE_MONTH_TOTAL = MONTHLYMERGE (KWH, "TOTAL")
```

SEASONVALUE Function

Purpose

The SEASONVALUE function returns a new value for a billing determinant based on the portion of the current billing period that falls within a specified season schedule.

This function prorates the bill determinant value based the portion of the current billing period that falls within a specified season schedule.

Format

```
<identifier> = SEASONVALUE(<season_period_name>,
<historical_identifier>, <type>);
```

Where

- <season_period_name> is the name of a season period, as defined in a season schedule. The season period must belong to the season schedule in effect. See **How to Specify a Season Schedule** on page 13-67, under the description for the AVGSEASON function, for important details about the season schedules.

- <historical_identifier> is a determinant identifier that contains historical values (computed, or loaded from the Oracle Utilities database).

- <type> is a string that determines how much of the current bill period is in the season. The options are "BILL_START", "BILL_STOP", and "PRORATE". "PRORATE" is the default. The first two return 0 if the BILL_START or BILL_STOP date (respectively) are not in the season; if they are, the full determinant values are returned. If the type is PRORATE, the value is prorated based on the proportion of days in the current bill period in the season. The value returned may vary from 0 (if the whole bill period is outside the season) to the full determinant value (if the whole bill period is within the season).

Example

Find the prorated kWh value for the current bill period that falls within the SUMMER season.

```
SUMMER_PRORATE_KWH = SEASONVALUE("SUMMER", KWH, "PRORATE");
```

SUMSEASON Function

Purpose

The SUMSEASON function sums the maximum monthly values in a season. Similar to the **MAXSEASON Function** on page 13-69, except the determinant is totaled so that the result is a scalar numeric value.

Format

```
<identifier> = SUMSEASON(<season_period_name>,
<historical_identifier>, <start_bill_period_previous>,
<end_bill_period_previous>);
```

Where

- <season_period_name> is the name of a season period, as defined in a season schedule. The season period must belong to the season schedule in effect. See **How to Specify a Season Schedule** on page 13-67, under the description for the AVGSEASON function, for details about the season schedules.

- <historical_identifier> is a determinant identifier that contains historical values (computed, or loaded from the Oracle Utilities database).

- <start_bill_period_previous>, <end_bill_period_previous> specifies the bill periods to be loaded, using the following convention: **0** is the current bill period, **1** is the previous bill period, and so on (the higher the number, the further back in time). The end_bill_period_previous must be greater than or equal to the start_bill_period_previous. The default start_bill_period_previous is 0 (the current period). The default end_bill_period_previous is the last period of data available for the account. If you specify a start but no end, the default end is the last period of data for that determinant stored for the account. See **Start and End Bill Period Parameters** under **Rules for Using Functions** in **Chapter 6**: **Rules Language Functions Overview** of the *Oracle Utilities Rules Language User's Guide* for additional details about specifying bill period parameters.

Example

Find the sum of the maximum KWH value for the SUMMER season for the last three bill periods, including the current bill period.

```
MAXTOTAL_SUMMER_KWH = SUMSEASON("SUMMER", KWH, 0, 2);
```

Term Functions

Term functions are used to retrieve and save terms and term details to and from the Oracle Utilities Data Repository.

Term Function Tail Identifiers

Term functions use the following tail identiers for database columns:

Tail Identifier	Table	Field
CONTRACTID	LSCMCONTRACT	CONTRACTID
REVISION	LSCMCONTRACT	REVISION
ACCOUNTID	ACCOUNT	ACCOUNTID
SERVICEPOINT	LSSERVICEPOINT	SERVICEPOINTID
MARKETID	LSMARKET	MARKETID
SERVICETYPE	LSSERIVICETYPE	SERVICETYPE
PRODUCTID	LSCMPRODUCT	PRODUCTID
PRODUCTSTART	LSCMPRODUCT	STARTTIME
PRODUCTSTOP	LSCMPRODUCT	STOPTIME
GROUPID	LSCMCONITEMGROUP	GROUPID
TERMTYPE	LSTERMTYPE	TERMTYPECODE
TERMCATEGORY	LSTERMCATEGORY	TERMCATEGORYCODE
TERMSTART	LSTERM	STARTTIME
TERMSTOP	LSTERM	STOPTIME
STARTTIME	Term Table*	STARTTIME
STOPTIME	Term Table*	STOPTIME
VAL	Term Table*	VAL
VALNUM	Term Table*	VALNUM
VALDATE	Term Table*	VALDATE
ISSTANDARD	LSCMCONTRACTTERM	ISSTANDARD
ISREQUIRED	LSCMCONTRACTTERM	ISREQUIRED
ISCALCULATED	LSCMCONTRACTTERM	ISCALCULATED
PERIOD	LSCMCONITEMDTLS	PERIOD

*Any of the following tables used to store terms:

- LSCMCONTRACTTERM (Contract Terms)

- LSCMCONITEMTERM, LSCMCONITEMPRDTERM, LSCMCONITEMDTLS (Contract Item Terms)

• LSCMCONITEMGROUPPRDTERM (Contract Group Terms)

LOADCONTRACTTERM Function

Purpose

The LOADCONTRACTTERM function loads a single contract term from the Contract Terms table in the Oracle Utilities Data Repository. The function returns a stem identifier containing the retrieved term.

Format

```
<output_stem> = LOADCONTRACTTERM(<input_stem>);
```

Where

• <input_stem> is a stem identifier with following tail identifiers that specify the term to retrieve:

  • CONTRACTID: The contract ID of the contract for the term to be retrieved

  • REVISION: The revision number of the contract for the term to be retrieved

  • TERMTYPE: The term type for the term to be retrieved

  • TERMCATEGORY: The term category for the term to be retrieved

  • STARTTIME: The start time of the term to be retrieved

  See **Term Function Tail Identifiers** on page 13-74 for a list of the database table-columns that correspond to these identifiers. If a tail identifier is missing or has previously been cleared (via the CLEAR statement), a NULL value will be used.

• <output_stem> is a stem identifier with following tail identifiers representing columns from the Contract Term table:

  • CONTRACTID: The contract ID of the contract for the retrieved term

  • REVISION: The revision number of the contract for the retrieved term

  • TERMSTART: The start time of the term

  • TERMSTOP: The stop time of the term

  • TERMTYPE: The term type for the retrieved term

  • TERMCATEGORY: The term category for the retrieved term

  • STARTTIME: The start time of the retrieved term

  • STOPTIME: The stop time of the retrieved term

  • VAL: The text value of the retrieved term

  • VALNUM: The numeric value of the retrieved term

  • VALDATE: The date value of the retrieved term

  • ISSTANDARD: The Is Standard flag of the retrieved term

  • ISREQUIRED: The Is Required flag of the retrieved term

  • ISCALCULATED: The Is Calculated flag of the retrieved term

  • Any custom columns on the Contract Term table. The tail identifier used will be the same as the custom column name.

  • ERRORRETURN: Return code of the function call. An error occurs if no term record is found, if multiple terms records are found, or if a custom column on the Contract Term

table has the same name as one of the pre-defined tail identifiers (see **Term Function Tail Identifiers** on page 13-74). Return codes are as follows:

- 0 - Success

- 1 - No Record Found

- 2 - Multiple Records Found

- 3 - Column name conflict with pre-defined tail

Note: Any errors returned will also appear on the output report.

See **Term Function Tail Identifiers** on page 13-74 for a list of the database table-columns that correspond to these identifiers. If a value for a column is NULL, the tail identifier will be cleared if it already exists. If the tail identifier that corresponds to a NULL column value does not already exist, it will not be created.

Example

Retrieve the MARGIN,CONTRACT term with a start date of 01/01/2008 00:00:00 for contact "Customer_Pricing_01", revision 1.

```
LOAD_TERM.CONTRACTID = "Customer_Pricing_01";
LOAD_TERM.REVISION = "1";
LOAD_TERM.TERMTYPE = "MARGIN";
LOAD_TERM.TERMCATEGORY = "CONTRACT";
LOAD_TERM.STARTTIME = "01/01/2008 00:00:00";
CONTRACT_TERM_DTLS = LOADCONTRACTTERM(LOAD_TERM);
```

This function would return the following tail identifiers for the "CONTRACT_TERM_DTLS" stem identifier:

Tail Identifiers	Value
CONTRACTID	"Customer_Pricing_01"
REVISION	"1"
TERMSTART	"01/01/2006 00:00:00"
TERMSTOP	NULL
TERMTYPE	"MARGIN"
TERMCATEGORY	"CONTRACT"
STARTTIME	"01/01/2006 00:00:00"
STOPTIME	NULL
VAL	NULL
VALNUM	"5"
VALDATE	NULL
ISSTANDARD	"Yes"
ISREQUIRED	"Yes"
ISCALCULATED	"No"

LOADCONTRACTTERMALL Function

Purpose

The LOADCONTRACTTERMALL function loads all contract terms for a specified contract from the Contract Terms table in the Oracle Utilities Data Repository. This function creates one or more stem identifiers containing the retrieved terms. The function returns zero if successful, and returns an integer (1, 2, 3, or 4) if an error occurs.

Format

```
<error_code> = LOADCONTRACTTERMALL(<input_stem>);
```

Where

- <input_stem> is a stem identifier with following tail identifiers that specify the contract from which to retrieve the terms:

  - CONTRACTID: The contract ID of the contract for the term to be retrieved

  - REVISION: The revision number of the contract for the term to be retrieved

  See **Term Function Tail Identifiers** on page 13-74 for a list of the database table-columns that correspond to these identifiers. If a tail identifier is missing or has previously been cleared (via the CLEAR statement), a NULL value will be used.

- <error_code> is the return code of the function call. An error occurs if no term record is found, if multiple terms records are found, if a custom column on the Contract Term table has the same name as one of the pre-defined tail identifiers (see **Term Function Tail Identifiers** on page 13-74), or if the 64-character Rules Language identifier length is exceeded. Return codes are as follows:

  - 0 - Success

  - 1 - No Record Found

  - 2 - Multiple Records Found

  - 3 - Column name conflict with pre-defined tail

  - 4 - Identifier 64-character limit exceeded

  Note: Any errors returned will also appear on the output report.

Stem Identifiers: This function creates one or more stem identifiers that contain the retrieved terms. The stem identifiers are created by concatenating the table name prefix ("CONT"), an underscore, the Term Type Code, an underscore, and the Category Code. For example, the stem identifier for contract terms of type "MARGIN" of category "CONTRACT" would be as follows:

```
CONT_MARGIN_CONTRACT
```

If the Category Code is null, then the stem identifier will be created by concatenating the table name prefix, an underscore, and the Term Type Code. For example, the stem identifier for contract terms of type "MARGIN" with a Null category would be as follows:

```
CONT_MARGIN
```

Stem identifiers are made into array identifiers if there are multiple values of STARTTIME (in the term table) for the same term for the specified contract.

Tail Identifiers: Each stem identifier has the following tail identifiers:

- TERMSTART: The start time of the term

- TERMSTOP: The stop time of the term

- STARTTIME: The start time of the retrieved term

- STOPTIME: The stop time of the retrieved term

- VAL: The text value of the retrieved term

- VALNUM: The numeric value of the retrieved term

- VALDATE: The date value of the retrieved term

- ISSTANDARD: The Is Standard flag of the retrieved term

- ISREQUIRED: The Is Required flag of the retrieved term

- ISCALCULATED: The Is Calculated flag of the retrieved term

- Any custom columns on the Contract Term table. The tail identifier used will be the same as the custom column name.

See **Term Function Tail Identifiers** on page 13-74 for a list of the database table-columns that correspond to these identifiers. If a value for a column is NULL, the tail identifier will be cleared if it already exists. If the tail identifier that corresponds to a NULL column value does not already exist, it will not be created.

Example

Retrieve all terms for contact "Customer_Pricing_01", revision 1.

```
LOAD_ALL_TERMS.CONTRACTID = "Customer_Pricing_01";
LOAD_ALL_TERMS.REVISION = "1";
ALL_CONTRACT_TERM = LOADCONTRACTTERMALL(LOAD_ALL_TERMS);
```

This function would return a number of stem identifiers, one for each combination of Term Type and Category. For example:

```
CONT_MARGIN_CONTRACT
CONT_DEPOSITAMOUNT_CONTRACT
CONT_EFFECTIVEPRICESFROM_CONTRACT
CONT_ONPEAKMARGIN_PRODUCT
CONT_OFFPEAKMARGIN_PRODUCT
CONT_CUST_CHARGE_TYPE_PRODUCT
...
```

Each of these stem identifiers would contain the above listed tail identifiers.

If there were multiple Start Time values for the same term, these stem identifiers would be array identifiers, each with an upper bound equal to the number of term records returned.

LOADGROUPTERM Function

Purpose

The LOADGROUPTERM function loads a single contract group term from the Contract Item Group Terms table in the Oracle Utilities Data Repository. The function returns a stem identifier containing the retrieved term.

Format

```
<output_stem> = LOADGROUPTERM(<input_stem>);
```

Where

- <input_stem> is a stem identifier with following tail identifiers that specify the term to retrieve:

    - CONTRACTID: The contract ID of the contract for the term to be retrieved

    - REVISION: The revision number of the contract for the term to be retrieved

    - TERMTYPE: The term type for the term to be retrieved

    - TERMCATEGORY: The term category for the term to be retrieved

    - STARTTIME: The start time of the term to be retrieved

    - PRODUCTID: The Product ID for the contract item group

    - PRODUCTSTART: The Product start time for the contract item group

    - PRODUCTSTOP: The Product stop time for the contract item group

    - GROUPID: The Group ID for the contract item group

    See **Term Function Tail Identifiers** on page 13-74 for a list of the database table-columns that correspond to these identifiers. If a tail identifier is missing or has previously been cleared (via the CLEAR statement), a NULL value will be used.

- <output_stem> is a stem identifier with following tail identifiers representing columns from the Contract Item Group Product Term table:

    - CONTRACTID: The contract ID of the contract for the retrieved term

    - REVISION: The revision number of the contract for the retrieved term

    - TERMSTART: The start time of the term

    - TERMSTOP: The stop time of the term

    - TERMTYPE: The term type for the retrieved term

    - TERMCATEGORY: The term category for the retrieved term

    - PRODUCTID: The Product ID for the contract item group

    - PRODUCTSTART: The Product start time for the contract item group

    - PRODUCTSTOP: The Product stop time for the contract item group

    - GROUPID: The Group ID for the contract item group

    - STARTTIME: The start time of the retrieved term

    - STOPTIME: The stop time of the retrieved term

    - VAL: The text value of the retrieved term

    - VALNUM: The numeric value of the retrieved term

    - VALDATE: The date value of the retrieved term

- Any custom columns on the Contract Item Group Product Term table. The tail identifier used will be the same as the custom column name.

- ERRORRETURN: Return code of the function call. An error occurs if no term record is found, if multiple terms records are found, or if a custom column on the Contract Term table has the same name as one of the pre-defined tail identifiers (see **Term Function Tail Identifiers** on page 13-74). Return codes are as follows:

  - 0 - Success

  - 1 - No Record Found

  - 2 - Multiple Records Found

  - 3 - Column name conflict with pre-defined tail

  Note: Any errors returned will also appear on the output report.

See **Term Function Tail Identifiers** on page 13-74 for a list of the database table-columns that correspond to these identifiers. If a value for a column is NULL, the tail identifier will be cleared if it already exists. If the tail identifier that corresponds to a NULL column value does not already exist, it will not be created.

Example

Retrieve the DEPOSITAMOUNT,CONTRACT group term with a start date of 01/01/2008 00:00:00 for contact "Customer_Pricing_01", revision 1, for group "GROUP_01" and Product "GENERAL_SERVICE" (for 01/01/2007 00:00:00 through 12/31/2010 23:59:59).

```
LOAD_GROUP_TERM.CONTRACTID = "Customer_Pricing_01";
LOAD_GROUP_TERM.REVISION = "1";
LOAD_GROUP_TERM.TERMTYPE = "DEPOSITAMOUNT";
LOAD_GROUP_TERM.TERMCATEGORY = "CONTRACT";
LOAD_GROUP_TERM.STARTTIME = "01/01/2008 00:00:00";
LOAD_GROUP_TERM.PRODUCTID = "GENERAL_SERVICE";
LOAD_GROUP_TERM.PRODUCTSTART = "01/01/2007 00:00:00";
LOAD_GROUP_TERM.PRODUCTSTOP = "12/31/2010 23:59:59";
LOAD_GROUP_TERM.GROUPIP = "GROUP_01";
GROUP_TERM_DTLS = LOADCONTRACTTERM(LOAD_GROUP_TERM);
```

This function would return the following tail identifiers for the "GROUP_TERM_DTLS" stem identifier:

Tail Identifiers	Value
CONTRACTID	"Customer_Pricing_01"
REVISION	"1"
TERMSTART	"01/01/2006 00:00:00"
TERMSTOP	NULL
TERMTYPE	"DEPOSITAMOUNT"
TERMCATEGORY	"CONTRACT"
PRODUCTID	"GENERAL_SERVICE"
PRODUCTSTART	"01/01/2007 00:00:00"
PRODUCTSTOP	"12/31/2010 23:59:59"
GROUPID	"GROUP_01"

Tail Identifiers	Value
STARTTIME	"01/01/2006 00:00:00"
STOPTIME	NULL
VAL	NULL
VALNUM	"100"
VALDATE	NULL

LOADGROUPTERMALL Function

Purpose

The LOADGROUPTERMALL function loads all contract group terms for a specified contract and group from the Contract Item Group Terms table in the Oracle Utilities Data Repository. This function creates one or more stem identifiers containing the retrieved terms. The function returns zero if successful, and returns an integer (1, 2, 3, or 4) if an error occurs.

Format

```
<error_code> = LOADGROUPTERMALL(<input_stem>);
```

Where

- <input_stem> is a stem identifier with following tail identifiers that specify the contract from which to retrieve the terms:

    - CONTRACTID: The contract ID of the contract for the term to be retrieved

    - REVISION: The revision number of the contract for the term to be retrieved

    - PRODUCTID: The Product ID for the contract item group

    - PRODUCTSTART: The Product start time for the contract item group

    - PRODUCTSTOP: The Product stop time for the contract item group

    - GROUPID: The Group ID for the contract item group

    See **Term Function Tail Identifiers** on page 13-74 for a list of the database table-columns that correspond to these identifiers. If a tail identifier is missing or has previously been cleared (via the CLEAR statement), a NULL value will be used.

- <error_code> is the return code of the function call. An error occurs if no term record is found, if multiple terms records are found, if a custom column on the Contract Term table has the same name as one of the pre-defined tail identifiers (see **Term Function Tail Identifiers** on page 13-74), or if the 64-character Rules Language identifier length is exceeded. Return codes are as follows:

    - 0 - Success

    - 1 - No Record Found

    - 2 - Multiple Records Found

    - 3 - Column name conflict with pre-defined tail

    - 4 - Identifier 64-character limit exceeded

    Note: Any errors returned will also appear on the output report.

Stem Identifiers: This function creates one or more stem identifiers that contain the retrieved terms. The stem identifiers are created by concatenating the table name prefix ("GRUP"), an underscore, the Term Type Code, an underscore, and the Category Code. For example, the stem identifier for contract terms of type "MARGIN" of category "CONTRACT" would be as follows:

```
GRUP_MARGIN_CONTRACT
```

If the Category Code is null, then the stem identifier will be created by concatenating the table name prefix, an underscore, and the Term Type Code. For example, the stem identifier for contract terms of type "MARGIN" with a Null category would be as follows:

```
GRUP_MARGIN
```

Stem identifiers are made into array identifiers if there are multiple values of STARTTIME (in the term table) for the same term for the specified contract.

Tail Identifiers: Each stem identifier has the following tail identifiers:

- TERMSTART: The start time of the term

- TERMSTOP: The stop time of the term

- STARTTIME: The start time of the retrieved term

- STOPTIME: The stop time of the retrieved term

- VAL: The text value of the retrieved term

- VALNUM: The numeric value of the retrieved term

- VALDATE: The date value of the retrieved term

- Any custom columns on the Contract Term table. The tail identifier used will be the same as the custom column name.

See **Term Function Tail Identifiers** on page 13-74 for a list of the database table-columns that correspond to these identifiers. If a value for a column is NULL, the tail identifier will be cleared if it already exists. If the tail identifier that corresponds to a NULL column value does not already exist, it will not be created.

Example

Retrieve all group terms for contact "Customer_Pricing_01", revision 1 for group "GROUP_01" and Product "GENERAL_SERVICE" (for 01/01/2007 00:00:00 through 12/31/2010 23:59:59).

```
LOAD_ALL_GROUP_TERMS.CONTRACTID = "Customer_Pricing_01";
LOAD_ALL_GROUP_TERMS.REVISION = "1";
LOAD_ALL_GROUP_TERMS.PRODUCTID = "GENERAL_SERVICE";
LOAD_ALL_GROUP_TERMS.PRODUCTSTART = "01/01/2007 00:00:00";
LOAD_ALL_GROUP_TERMS.PRODUCTSTOP = "12/31/2010 23:59:59";
LOAD_ALL_GROUP_TERMS.GROUPIP = "GROUP_01";
ALL_CONTRACT_GROUP_TERM = LOADGROUPTERMALL(LOAD_ALL_GROUP_TERMS);
```

This function would return a number of stem identifiers, one for each combination of Term Type and Category. For example:

```
GRUP_MARGIN_CONTRACT
GRUP_DEPOSITAMOUNT_CONTRACT
GRUP_EFFECTIVEPRICESFROM_CONTRACT
GRUP_ONPEAKMARGIN_PRODUCT
GRUP_OFFPEAKMARGIN_PRODUCT
GRUP_CUST_CHARGE_TYPE_PRODUCT
...
```

Each of these stem identifiers would contain the above listed tail identifiers.

If there were multiple Start Time values for the same term, these stem identifiers would be array identifiers, each with an upper bound equal to the number of term records returned.

LOADITEMTERM Function

Purpose

The LOADITEMTERM function loads a single contract item term from the Oracle Utilities Data Repository. This function can retrieve terms from the Contract Item Term table, the Contract Item Product Term table, or the Contract Item Details table. The function returns a stem identifier containing the retrieved term.

Format

```
<output_stem> = LOADITEMTERM(<input_stem>, <table>);
```

Where

- <input_stem> is a stem identifier with following tail identifiers that specify the term to retrieve:

  - CONTRACTID: The contract ID of the contract for the term to be retrieved

  - REVISION: The revision number of the contract for the term to be retrieved

  - TERMTYPE: The term type for the term to be retrieved

  - TERMCATEGORY: The term category for the term to be retrieved

  - STARTTIME: The start time of the term to be retrieved

  - ACCOUNTID: The Account ID for the contract item (if applicable)

  - PRODUCTID: The Product ID for the contract item (if applicable)

  - PRODUCTSTART: The Product start time for the contract item (if applicable)

  - PRODUCTSTOP: The Product stop time for the contract item (if applicable)

  - SERVICEPOINT: The Service Point ID for the contract item (if applicable)

  - MARKETID: The Market ID related to the Service Point ID for the contract item (if applicable)

  - SERVICETYPE: The service type related to the Service Point ID for the contract item (if applicable)

  - STOPTIME: The stop time of the term to be retrieved. This tail only applies when retrieving term details from Contract Item Details table.

  See **Term Function Tail Identifiers** on page 13-74 for a list of the database table-columns that correspond to these identifiers. If a tail identifier is missing or has previously been cleared (via the CLEAR statement), a NULL value will be used.

- <table> is a string that specifies the table from which the term details are to be retrieved:

  - "ITEM": retrieve the term details from the Contract Item Term table

  - "PRODUCT": retrieve the term details from the Contract Item Product Term table

  - "DETAILS" retrieve the term details from the Contract Item Details table

- <output_stem> is a stem identifier with following tail identifiers representing columns from the specified table:

  - CONTRACTID: The contract ID of the contract for the retrieved term

  - REVISION: The revision number of the contract for the retrieved term

  - TERMSTART: The start time of the term

  - TERMSTOP: The stop time of the term

  - TERMTYPE: The term type for the retrieved term

- TERMCATEGORY: The term category for the retrieved term

- ACCOUNTID: The Account ID for the contract item

- PRODUCTID: The Product ID for the contract item

- PRODUCTSTART: The Product start time for the contract item

- PRODUCTSTOP: The Product stop time for the contract item

- SERVICEPOINT: The Service Point ID for the contract item

- MARKETID: The Market ID related to the Service Point ID for the contract item

- SERVICETYPE: The service type related to the Service Point ID for the contract item

- STARTTIME: The start time of the retrieved term

- STOPTIME: The stop time of the retrieved term

- VAL: The text value of the retrieved term

- VALNUM: The numeric value of the retrieved term

- VALDATE: The date value of the retrieved term

- PERIOD: The period for the retrieved term. This tail is only returned when retrieving terms from the Contract Item Details table.

- Any custom columns on the specified table. The tail identifier used will be the same as the custom column name.

- ERRORRETURN: Return code of the function call. An error occurs if no term record is found, if multiple terms records are found, or if a custom column on the specified table has the same name as one of the pre-defined tail identifiers (see **Term Function Tail Identifiers** on page 13-74). Return codes are as follows:

  - 0 - Success

  - 1 - No Record Found

  - 2 - Multiple Records Found

  - 3 - Column name conflict with pre-defined tail

  Note: Any errors returned will also appear on the output report.

See **Term Function Tail Identifiers** on page 13-74 for a list of the database table-columns that correspond to these identifiers. If a value for a column is NULL, the tail identifier will be cleared if it already exists. If the tail identifier that corresponds to a NULL column value does not already exist, it will not be created.

Example

Retrieve the MARGIN,CONTRACT term with a start date of 01/01/2008 00:00:00 for contact "Customer_Pricing_01", revision 1, for Account "ACCT_01" and Product "STANDARD_SERVICE" (for 01/01/2007 00:00:00 through 12/31/2010 23:59:59) from the Contract Item Terms table.

```
LOAD_ITEM_TERM.CONTRACTID = "Customer_Pricing_01";
LOAD_ITEM_TERM.REVISION = "1";
LOAD_ITEM_TERM.TERMTYPE = "MARGIN";
LOAD_ITEM_TERM.TERMCATEGORY = "CONTRACT";
LOAD_ITEM_TERM.STARTTIME = "01/01/2008 00:00:00";
LOAD_ITEM_TERM.ACCOUNTID = "ACCT_01";
LOAD_ITEM_TERM.PRODUCTID = "STANDARD_SERVICE";
LOAD_ITEM_TERM.PRODUCTSTART = "01/01/2007 00:00:00";
LOAD_ITEM_TERM.PRODUCTSTOP = "12/31/2010 23:59:59";
ITEM_TERM_DTLS = LOADCONTRACTTERM(LOAD_ITEM_TERM, "ITEM");
```

This function would return the following tail identifiers for the "ITEM_TERM_DTLS" stem identifier:

Tail Identifiers	Value
CONTRACTID	"Customer_Pricing_01"
REVISION	"1"
TERMSTART	"01/01/2006 00:00:00"
TERMSTOP	NULL
TERMTYPE	"MARGIN"
TERMCATEGORY	"CONTRACT"
ACCOUNTID	"ACCT_01"
PRODUCTID	"STANDARD_SERVICE"
PRODUCTSTART	"01/01/2007 00:00:00"
PRODUCTSTOP	"12/31/2010 23:59:59"
STARTTIME	"01/01/2006 00:00:00"
STOPTIME	NULL
VAL	NULL
VALNUM	"5"
VALDATE	NULL

LOADITEMTERMALL Function

Purpose

The LOADITEMTERMALL function loads all contract item terms for a specified contract, contract item, or contract item product from the Oracle Utilities Data Repository. This function can retrieve terms from the Contract Item Term table, the Contract Item Product Term table, or the Contract Item Details table. This function creates one or more stem identifiers containing the retrieved terms. The function returns zero if successful, and returns an integer (1, 2, 3, or 4) if an error occurs.

Format

```
<error_code> = LOADITEMTERMALL(<input_stem>, <table>);
```

Where

- <input_stem> is a stem identifier with following tail identifiers that specify the terms to retrieve:

    - CONTRACTID: The contract ID of the contract for the term to be retrieved

    - REVISION: The revision number of the contract for the term to be retrieved

    - ACCOUNTID: The Account ID for the contract item (if applicable)

    - PRODUCTID: The Product ID for the contract item (if applicable)

    - PRODUCTSTART: The Product start time for the contract item (if applicable)

    - PRODUCTSTOP: The Product stop time for the contract item (if applicable)

    - SERVICEPOINT: The Service Point ID for the contract item (if applicable)

    - MARKETID: The Market ID related to the Service Point ID for the contract item (if applicable)

    - SERVICETYPE: The service type related to the Service Point ID for the contract item (if applicable)

    See **Term Function Tail Identifiers** on page 13-74 for a list of the database table-columns that correspond to these identifiers. If a tail identifier is missing or has previously been cleared (via the CLEAR statement), a NULL value will be used.

- <table> is a string that specifies the table from which the term details are to be retrieved:

    - "ITEM": retrieve the term details from the Contract Item Term table

    - "PRODUCT": retrieve the term details from the Contract Item Product Term table

    - "DETAILS" retrieve the term details from the Contract Item Details table

- <error_code> is the return code of the function call. An error occurs if no term record is found, if multiple terms records are found, if a custom column on the Contract Term table has the same name as one of the pre-defined tail identifiers (see **Term Function Tail Identifiers** on page 13-74), or if the 64-character Rules Language identifier length is exceeded. Return codes are as follows:

    - 0 - Success

    - 1 - No Record Found

    - 2 - Multiple Records Found

    - 3 - Column name conflict with pre-defined tail

    - 4 - Identifier 64-character limit exceeded

    Note: Any errors returned will also appear on the output report.

Stem Identifiers: This function creates one or more stem identifiers that contain the retrieved terms.

- For terms retrieved from the **Contract Item Term** table, stem identifiers are created by concatenating the table name prefix ("ITEM"), an underscore, the Term Type Code, an underscore, and the Category Code. For example, the stem identifier for contract terms of type "MARGIN" of category "CONTRACT" would be as follows:

  `ITEM_MARGIN_CONTRACT`

  If the Category Code is null, then the stem identifier will be created by concatenating the table name prefix, an underscore, and the Term Type Code. For example, the stem identifier for contract terms of type "MARGIN" with a Null category would be as follows:

  `ITEM_MARGIN`

- For terms retrieved from the **Contract Item Product Term** table, stem identifiers are created by concatenating the table name prefix ("IPRD"), an underscore, the Term Type Code, an underscore, and the Category Code. For example, the stem identifier for contract terms of type "MARGIN" of category "CONTRACT" would be as follows:

  `IPRD_MARGIN_CONTRACT`

  If the Category Code is null, then the stem identifier will be created by concatenating the table name prefix, an underscore, and the Term Type Code. For example, the stem identifier for contract terms of type "MARGIN" with a Null category would be as follows:

  `IPRD_MARGIN`

- For terms retrieved from the **Contract Item Details** table, stem identifiers are created by concatenating the table name prefix ("DTLS"), an underscore, the Term Type Code, an underscore, and the Category Code. For example, the stem identifier for contract terms of type "MARGIN" of category "CONTRACT" would be as follows:

  `DTLS_MARGIN_CONTRACT`

  If the Category Code is null, then the stem identifier will be created by concatenating the table name prefix, an underscore, and the Term Type Code. For example, the stem identifier for contract terms of type "MARGIN" with a Null category would be as follows:

  `DTLS_MARGIN`

Stem identifiers are made into array identifiers if there are multiple values of STARTTIME (in the term table) for the same term for the specified contract.

Tail Identifiers: Each stem identifier has the following tail identifiers:

- TERMSTART: The start time of the term
- TERMSTOP: The stop time of the term
- STARTTIME: The start time of the retrieved term
- STOPTIME: The stop time of the retrieved term
- VAL: The text value of the retrieved term
- VALNUM: The numeric value of the retrieved term
- VALDATE: The date value of the retrieved term
- PERIOD: The period for the retrieved term. This tail is only returned when retrieving terms from the Contract Item Details table.
- Any custom columns on the specified table. The tail identifier used will be the same as the custom column name.

See **Term Function Tail Identifiers** on page 13-74 for a list of the database table-columns that correspond to these identifiers. If a value for a column is NULL, the tail identifier will be cleared if it already exists. If the tail identifier that corresponds to a NULL column value does not already exist, it will not be created.

Example

Retrieve all item terms for contact "Customer_Pricing_01", revision 1 for Account "ACCT_01" and Product "STANDARD_SERVICE" (for 01/01/2007 00:00:00 through 12/31/2010 23:59:59) from the Contract Item Terms table.

```
LOAD_ALL_ITEM_TERMS.CONTRACTID = "Customer_Pricing_01";
LOAD_ALL_ITEM_TERMS.REVISION = "1";
LOAD_ALL_ITEM_TERMS.ACCOUNTID = "ACCT_01";
LOAD_ALL_ITEM_TERMS.PRODUCTID = "STANDARD_SERVICE";
LOAD_ALL_ITEM_TERMS.PRODUCTSTART = "01/01/2007 00:00:00";
LOAD_ALL_ITEM_TERMS.PRODUCTSTOP = "12/31/2010 23:59:59";
ALL_CONTRACT_ITEM_TERMS = LOADITEMTERMALL(LOAD_ALL_ITEM_TERMS,
"ITEM");
```

This function would return a number of stem identifiers, one for each combination of Term Type and Category. For example:

```
ITEM_MARGIN_CONTRACT
ITEM_DEPOSITAMOUNT_CONTRACT
ITEM_EFFECTIVEPRICESFROM_CONTRACT
ITEM_ONPEAKMARGIN_PRODUCT
ITEM_OFFPEAKMARGIN_PRODUCT
ITEM_CUST_CHARGE_TYPE_PRODUCT
...
```

Each of these stem identifiers would contain the above listed tail identifiers.

If there were multiple Start Time values for the same term, these stem identifiers would be array identifiers, each with an upper bound equal to the number of term records returned.

SAVECONTRACTTERM Function

Purpose

The SAVECONTRACTTERM function saves a single contract term to the Contract Term table in the Oracle Utilities Data Repository. The function returns zero (0) if successful, and 1 if an error occurs.

Format

```
<return_code> = SAVECONTRACTTERM(<input_stem>);
```

Where

- <input_stem> is a stem identifier with following tail identifiers that specify the term to save:

    - CONTRACTID: The contract ID of the contract for the term to be saved

    - REVISION: The revision number of the contract for the term to be saved

    - TERMSTART: The start time of the term to be saved

    - TERMSTOP: The stop time of the term to be saved

    - TERMTYPE: The term type for the term to be saved

    - TERMCATEGORY: The term category for the term to be saved

    - STARTTIME: The start time of the term to be saved

    - STOPTIME: The stop time of the term to be saved

    - VAL: The text value of the term to be saved

    - VALNUM: The numeric value of the term to be saved

    - VALDATE: The date value of the term to be saved

    - ISSTANDARD: The Is Standard flag of the term to be saved

    - ISREQUIRED: The Is Required flag of the term to be saved

    - ISCALCULATED: The Is Calculated flag of the term to be saved

    - Any custom columns on the Contract Term table. The tail identifier used will be the same as the custom column name.

    See **Term Function Tail Identifiers** on page 13-74 for a list of the database table-columns that correspond to these identifiers. If a tail identifier is missing or has previously been cleared (via the CLEAR statement), a NULL value will be used.

- <return_code> is the return code of the function call. An error occurs if the term record cannot be saved. Return codes are as follows:

    - 0 - Success

    - 1 - Record could not be saved

    Note: Any errors returned will also appear on the output report.

Example

Save the MARGIN,CONTRACT term with a start date of 01/01/2008 00:00:00 for contact "Customer_Pricing_01", revision 1 with a numeric value of 10.

```
SAVE_TERM.CONTRACTID = "Customer_Pricing_01";
SAVE_TERM.REVISION = "1";
SAVE_TERM.TERMTYPE = "MARGIN";
SAVE_TERM.TERMCATEGORY = "CONTRACT";
SAVE_TERM.STARTTIME = "01/01/2008 00:00:00";
SAVE_TERM.STOPTIME = NULL;
SAVE_TERM.VAL = NULL;
SAVE_TERM.VALNUM = "10";
SAVE_TERM.VALDATE = NULL;
SAVE_TERM.ISSTANDARD = "Yes";
SAVE_TERM.ISREQUIRED = "Yes";
SAVE_TERM.ISCALCULATED = "No";
SAVE_TERM_RETURN = SAVECONTRACTTERM(SAVE_TERM);
```

SAVECONTRACTTERMALL Function

Purpose

The SAVECONTRACTTERMALL function saves all contract terms for a specified contract to the Contract Term table in the Oracle Utilities Data Repository. The function returns zero (0) if successful, and 1 if an error occurs.

Format

```
<return_code> = SAVECONTRACTTERMALL(<input_stem>[, <clear_flag>]);
```

Where

- <input_stem> is a stem identifier with following tail identifiers that specify the contract for which terms are to be saved:

    - CONTRACTID: The contract ID of the contract for the term to be saved

    - REVISION: The revision number of the contract for the term to be saved

- <clear_flag> is an optional flag that specifies whether or not to clear all stem and tail identifiers associated with the specified contract. A value of "Y" indicates that all stem and tail identifiers be cleared. Any other value indicates that stem and tail identifiers should NOT be cleared.

- <return_code> is the return code of the function call. An error occurs if the term record cannot be saved. Return codes are as follows:

    - 0 - Success

    - 1 - Record could not be saved

    Note: Any errors returned will also appear on the output report.

Example

Save all terms for contact "Customer_Pricing_01", revision 1 to the Contract Terms table, and clear all associated stem and tail identifiers.

```
SAVE_ALL_TERMS.CONTRACTID = "Customer_Pricing_01";
SAVE_ALL_TERMS.REVISION = "1";
SAVE_ALL_TERMS_RETURN = SAVECONTRACTTERMALL(SAVE_ALL_TERMS, "Y");
```

Notes

This function saves one or more stem identifiers (and their corresponding tail identifiers) that contain previously created or retrieved contract terms (see **LOADCONTRACTTERMALL Function** on page 13-77) based on the contract specified in the function call.

The stem identifiers to be saved are the concatenation of the table name prefix ("CONT"), an underscore, the Term Type Code, an underscore, and the Category Code. For example, the stem identifier for contract terms of type "MARGIN" of category "CONTRACT" would be as follows:

```
CONT_MARGIN_CONTRACT
```

If the Category Code is null, then the stem identifier is the concatenation of the table name prefix, an underscore, and the Term Type Code. For example, the stem identifier for contract terms of type "MARGIN" with a Null category would be as follows:

```
CONT_MARGIN
```

> **Note**: Because the CONTRACTID and REVISION can be different than the contract used to initially create the stem identifiers, not all of the identifiers initially loaded will necessarily be saved.

SAVEGROUPTERM Function

Purpose

The SAVEGROUPTERM function saves a single contract item group term to the Contract Item Group Term table in the Oracle Utilities Data Repository. The function returns zero (0) if successful, and 1 if an error occurs.

Format

```
<return_code> = SAVEGROUPTERM(<input_stem>);
```

Where

- <input_stem> is a stem identifier with following tail identifiers that specify the term to save:

  - CONTRACTID: The contract ID of the contract for the term to be saved

  - REVISION: The revision number of the contract for the term to be saved

  - TERMSTART: The start time of the term to be saved

  - TERMSTOP: The stop time of the term to be saved

  - TERMTYPE: The term type for the term to be saved

  - TERMCATEGORY: The term category for the term to be saved

  - PRODUCTID: The Product ID for the contract item group

  - PRODUCTSTART: The Product start time for the contract item group

  - PRODUCTSTOP: The Product stop time for the contract item group

  - GROUPID: The Group ID for the contract item group

  - STARTTIME: The start time of the term to be saved

  - STOPTIME: The stop time of the term to be saved

  - VAL: The text value of the term to be saved

  - VALNUM: The numeric value of the term to be saved

  - VALDATE: The date value of the term to be saved

  - Any custom columns on the specified table. The tail identifier used will be the same as the custom column name.

  See **Term Function Tail Identifiers** on page 13-74 for a list of the database table-columns that correspond to these identifiers. If a tail identifier is missing or has previously been cleared (via the CLEAR statement), a NULL value will be used.

- <return_code> is the return code of the function call. An error occurs if the term record cannot be saved. Return codes are as follows:

  - 0 - Success

  - 1 - Record could not be saved

  Note: Any errors returned will also appear on the output report.

Example

Save the DEPOSITAMOUNT,CONTRACT group term with a start date of 01/01/2008 00:00:00 for contact "Customer_Pricing_01", revision 1 with a numeric value of 100, for group "GROUP_01" and Product "GENERAL_SERVICE" (for 01/01/2007 00:00:00 through 12/31/2010 23:59:59) to the Contract Item Group Terms table.

```
SAVE_GROUP_TERM.CONTRACTID = "Customer_Pricing_01";
SAVE_GROUP_TERM.REVISION = "1";
SAVE_GROUP_TERM.TERMTYPE = "DEPOSITAMOUNT";
SAVE_GROUP_TERM.TERMCATEGORY = "CONTRACT";
SAVE_GROUP_TERM.GROUPID = "GROUP_01";
SAVE_GROUP_TERM.PRODUCTID = "GENERAL_SERVICE";
SAVE_GROUP_TERM.PRODUCTSTART = "01/01/2007 00:00:00";
SAVE_GROUP_TERM.PRODUCTSTOP = "12/31/2010 23:59:59";
SAVE_GROUP_TERM.STARTTIME = "01/01/2008 00:00:00";
SAVE_GROUP_TERM.STOPTIME = NULL;
SAVE_GROUP_TERM.VAL = NULL;
SAVE_GROUP_TERM.VALNUM = "100";
SAVE_GROUP_TERM.VALDATE = NULL;
SAVE_GROUP_TERM_RETURN = SAVEGROUPTERM(SAVE_GROUP_TERM);
```

SAVEGROUPTERMALL Function

Purpose

The SAVEGROUPTERMALL function saves all contract group terms for a specified contract to the Contract Item Group Term table in the Oracle Utilities Data Repository. The function returns zero (0) if successful, and 1 if an error occurs.

Format

```
<return_code> = SAVEGROUPTERMALL(<input_stem>[, <clear_flag>]);
```

Where

- <input_stem> is a stem identifier with following tail identifiers that specify the contract group for which terms are to be saved:

    - CONTRACTID: The contract ID of the contract for the term to be saved

    - REVISION: The revision number of the contract for the term to be saved

    - PRODUCTID: The Product ID for the contract item group

    - PRODUCTSTART: The Product start time for the contract item group

    - PRODUCTSTOP: The Product stop time for the contract item group

    - GROUPID: The Group ID for the contract item group

- <clear_flag> is an optional flag that specifies whether or not to clear all stem and tail identifiers associated with the specified contract group. A value of "Y" indicates that all stem and tail identifiers be cleared. Any other value indicates that stem and tail identifiers should NOT be cleared.

- <return_code> is the return code of the function call. An error occurs if the term record cannot be saved. Return codes are as follows:

    - 0 - Success

    - 1 - Record could not be saved

    Note: Any errors returned will also appear on the output report.

Example

Save all group terms for contact "Customer_Pricing_01", revision 1, for group "GROUP_01" and Product "GENERAL_SERVICE" (for 01/01/2007 00:00:00 through 12/31/2010 23:59:59) to the Contract Item Group Terms table, and clear all associated stem and tail identifiers.

```
SAVE_GROUP_TERMS.CONTRACTID = "Customer_Pricing_01";
SAVE_GROUP_TERMS.REVISION = "1";
SAVE_GROUP_TERMS.PRODUCTID = "GENERAL_SERVICE";
SAVE_GROUP_TERMS.PRODUCTSTART = "01/01/2007 00:00:00";
SAVE_GROUP_TERMS.PRODUCTSTOP = "12/31/2010 23:59:59";
SAVE_GROUP_TERMS.GROUPID = "GROUP_01";
SAVE_GROUP_TERMS_RETURN = SAVECGROUPTERMALL(SAVE_GROUP_TERMS, "Y");
```

Notes

This function saves one or more stem identifiers (and their corresponding tail identifiers) that contain previously created or retrieved group terms (see **LOADGROUPTERMALL Function** on page 13-82) based on the contract group specified in the function call.

The stem identifiers to be saved are the concatenation of the table name prefix ("GRUP"), an underscore, the Term Type Code, an underscore, and the Category Code. For example, the stem identifier for contract terms of type "MARGIN" of category "CONTRACT" would be as follows:

```
GRUP_MARGIN_CONTRACT
```

If the Category Code is null, then the stem identifier is the concatenation of the table name prefix, an underscore, and the Term Type Code. For example, the stem identifier for contract terms of type "MARGIN" with a Null category would be as follows:

```
GRUP_MARGIN
```

> **Note**: Because the CONTRACTID and REVISION can be different than the contract used to initially create the identifiers, not all of the identifiers initially loaded will necessarily be saved.

SAVEITEMTERM Function

Purpose

The SAVEITEMTERM function saves a single contract item term to the Oracle Utilities Data Repository. This function can save terms to the Contract Item Term table, the Contract Item Product Term table, or the Contract Item Details table. The function returns zero (0) if successful, and 1 if an error occurs.

Format

```
<return_code> = SAVEITEMTERM(<input_stem>, <table>);
```

Where

- <input_stem> is a stem identifier with following tail identifiers that specify the term to save:

    - CONTRACTID: The contract ID of the contract for the term to be saved

    - REVISION: The revision number of the contract for the term to be saved

    - TERMSTART: The start time of the term to be saved

    - TERMSTOP: The stop time of the term to be saved

    - TERMTYPE: The term type for the term to be saved

    - TERMCATEGORY: The term category for the term to be saved

    - ACCOUNTID: The Account ID for the contract item

    - PRODUCTID: The Product ID for the contract item

    - PRODUCTSTART: The Product start time for the contract item

    - PRODUCTSTOP: The Product stop time for the contract item

    - SERVICEPOINT: The Service Point ID for the contract item

    - MARKETID: The Market ID related to the Service Point ID for the contract item

    - SERVICETYPE: The service type related to the Service Point ID for the contract item

    - STARTTIME: The start time of the term to be saved

    - STOPTIME: The stop time of the term to be saved

    - VAL: The text value of the term to be saved

    - VALNUM: The numeric value of the term to be saved

    - VALDATE: The date value of the term to be saved

    - PERIOD: The period for term to be saved. This tail is only returned when saving terms to the Contract Item Details table.

    - Any custom columns on the specified table. The tail identifier used will be the same as the custom column name.

    See **Term Function Tail Identifiers** on page 13-74 for a list of the database table-columns that correspond to these identifiers. If a tail identifier is missing or has previously been cleared (via the CLEAR statement), a NULL value will be used.

- <table> is a string that specifies the table from to the term details are to be saved:

    - "ITEM": save the term to the Contract Item Term table

    - "PRODUCT": save the term to the Contract Item Product Term table

    - "DETAILS": save the term to the Contract Item Details table

- • <return_code> is the return code of the function call. An error occurs if the term record cannot be saved. Return codes are as follows:

  - • 0 - Success

  - • 1 - Record could not be saved

  Note: Any errors returned will also appear on the output report.

Example

Save the MARGIN,CONTRACT item term with a start date of 01/01/2008 00:00:00 for contact "Customer_Pricing_01", revision 1 with a numeric value of 10, for Account "ACCT_01" and Product "STANDARD_SERVICE" (for 01/01/2007 00:00:00 through 12/31/2010 23:59:59) to the Contract Item Terms table.

```
SAVE_ITEM_TERM.CONTRACTID = "Customer_Pricing_01";
SAVE_ITEM_TERM.REVISION = "1";
SAVE_ITEM_TERM.TERMTYPE = "MARGIN";
SAVE_ITEM_TERM.TERMCATEGORY = "CONTRACT";
SAVE_ITEM_TERM.ACCOUNTID = "ACCT_01";
SAVE_ITEM_TERM.PRODUCTID = "STANDARD_SERVICE";
SAVE_ITEM_TERM.PRODUCTSTART = "01/01/2007 00:00:00";
SAVE_ITEM_TERM.PRODUCTSTOP = "12/31/2010 23:59:59";
SAVE_ITEM_TERM.STARTTIME = "01/01/2008 00:00:00";
SAVE_ITEM_TERM.STOPTIME = NULL;
SAVE_ITEM_TERM.VAL = NULL;
SAVE_ITEM_TERM.VALNUM = "10";
SAVE_ITEM_TERM.VALDATE = NULL;
SAVE_ITEM_TERM_RETURN = SAVEITEMTERM(SAVE_ITEM_TERM,"ITEM");
```

SAVEITEMTERMALL Function

Purpose

The SAVEITEMTERMALL function saves all contract item terms for a specified contract to the Oracle Utilities Data Repository. This function can save terms to the Contract Item Term table, the Contract Item Product Term table, or the Contract Item Details table. The function returns zero (0) if successful, and 1 if an error occurs.

Format

```
<return_code> = SAVEITEMTERMALL(<input_stem>, <table>[,
<clear_flag>]);
```

Where

- <input_stem> is a stem identifier with following tail identifiers that specify the contract and contract item for which terms are to be saved:

    - CONTRACTID: The contract ID of the contract for the term to be saved

    - REVISION: The revision number of the contract for the term to be saved

    - ACCOUNTID: The Account ID for the contract item

    - PRODUCTID: The Product ID for the contract item

    - PRODUCTSTART: The Product start time for the contract item

    - PRODUCTSTOP: The Product stop time for the contract item

    - SERVICEPOINT: The Service Point ID for the contract item

    - MARKETID: The Market ID related to the Service Point ID for the contract item

    - SERVICETYPE: The service type related to the Service Point ID for the contract item

- <table> is a string that specifies the table to which the term details are to be saved:

    - "ITEM": save the term to the Contract Item Term table

    - "PRODUCT": save the term to the Contract Item Product Term table

    - "DETAILS": save the term to the Contract Item Details table

- <clear_flag> is an optional flag that specifies whether or not to clear all stem and tail identifiers associated with the specified contract. A value of "Y" indicates that all stem and tail identifiers be cleared. Any other value indicates that stem and tail identifiers should NOT be cleared.

- <return_code> is the return code of the function call. An error occurs if the term record cannot be saved. Return codes are as follows:

    - 0 - Success

    - 1 - Record could not be saved

    Note: Any errors returned will also appear on the output report.

Example

Save all terms for contact "Customer_Pricing_01", revision 1 to the Contract Terms table, and clear all associated stem and tail identifiers.

```
SAVE_ALL_TERMS.CONTRACTID = "Customer_Pricing_01";
SAVE_ALL_TERMS.REVISION = "1";
SAVE_ALL_TERMS_RETURN = SAVECONTRACTTERMALL(SAVE_ALL_TERMS, "ITEM",
"Y");
```

Notes

This function saves one or more stem identifiers (and their corresponding tail identifiers) that contain previously created or retrieved contract item terms (see **LOADITEMTERMALL Function** on page 13-87) based on the contract and contract item specified in the function call.

> **Note**: Because the CONTRACTID and REVISION can be different than the contract used to initially create the identifiers, not all of the identifiers initially loaded will necessarily be saved.

- For terms to be saved to the **Contract Item Term** table, stem identifiers are the concatenation of the table name prefix ("ITEM"), an underscore, the Term Type Code, an underscore, and the Category Code. For example, the stem identifier for contract terms of type "MARGIN" of category "CONTRACT" would be as follows:

  ITEM_MARGIN_CONTRACT

  If the Category Code is null, then the stem identifier is the concatenation of the table name prefix, an underscore, and the Term Type Code. For example, the stem identifier for contract terms of type "MARGIN" with a Null category would be as follows:

  ITEM_MARGIN

- For terms to be saved to the **Contract Item Product Term** table, stem identifiers are the concatenation of the table name prefix ("IPRD"), an underscore, the Term Type Code, an underscore, and the Category Code. For example, the stem identifier for contract terms of type "MARGIN" of category "CONTRACT" would be as follows:

  IPRD_MARGIN_CONTRACT

  If the Category Code is null, then the stem identifier is the concatenation of the table name prefix, an underscore, and the Term Type Code. For example, the stem identifier for contract terms of type "MARGIN" with a Null category would be as follows:

  IPRD_MARGIN

- For terms to be saved to the **Contract Item Details** table, stem identifiers are the concatenation of the table name prefix ("DTLS"), an underscore, the Term Type Code, an underscore, and the Category Code. For example, the stem identifier for contract terms of type "MARGIN" of category "CONTRACT" would be as follows:

  DTLS_MARGIN_CONTRACT

  If the Category Code is null, then the stem identifier is the concatenation of the table name prefix, an underscore, and the Term Type Code. For example, the stem identifier for contract terms of type "MARGIN" with a Null category would be as follows:

  DTLS_MARGIN

Miscellaneous Functions

ACCTREADDATES Function

Purpose

The ACCTREADDATES function returns read dates for the specified account.

The first value returned (HISTVALUE(0)) is the first read date on or after the BILL_STOP. Subsequent read dates are the ones before the BILL_STOP, with most recent first.

Format

```
<identifier> = ACCTREADDATES(<account_id>)
```

Where

<account_id> is the account id number from the Oracle Utilities Data Repository.

Example

Return a list of read dates for account # 800001

```
READ_DATES = ACCTREADDATES("800001")
```

ACCTTABLELOAD Function

Purpose

The ACCTTABLELOAD function returns all specified records within specified date range.

Database records returned contain the STARTTIME and STOPTIME, as well as VAL and STRVAL if in the table. A record is in effect if the account is related to the record key through the table any time between the supplied dates. If there are no records, returns 0.

Format

```
<identifier> = ACCTTABLELOAD(<table_name>, <record_key>, <startdate>,
<stopdate>)
```

Where

- <table_name> is one of: "ACCTRIDERHIST", "ACCTRATECODEHIST", "ACCTOVERRIDEHIST", "ACCTNAMEOVERHIST", or "ACCTFACTORHIST".

- <record_key> is a string that is the non-account and non-start time part of a record's key.

- <startdate> is a date constant that is the start of the date range the records are taken from.

- <stopdate> is a date constant that is the end of the date range the records are taken from.

Example

Return the records from the ACCTRIDERHIST Table that have a record key of 'SPECIAL_RIDER'.

```
RH = ACCTTABLELLOAD("ACCTRIDERHIST", "SPECIAL_RIDER");
```

CONFIGADD Function

Purpose

The CONFIGADD function adds the value of parameters within a configuration file to the internal configuration settings.

The values in the configuration file are added to the internal configuration settings. The values in the next CONFIGADD replace any previous CONFIGADD. See **LODESTAR.CFG** on page 2-2 in **Chapter 2**: **Configuration Files** in the *Oracle Utilities Energy Information Platform Configuration Guide* for details on creating a LODESTAR.CFG configuration file.

Format

```
<identifier> = CONFIGADD(<identifier|expression>);
```

Where

• <identifier|expression> is a string that is a fully qualified path and file name.

Example

Add the parameters in LODESTAR.CFG to the internal configuration settings.

```
CONFIG = CONFIGADD("C:\LODESTAR\USER\LODESTAR.CFG");
```

CONFIGGET Function

Purpose

The CONFIGGET function returns the value of a configuration file parameter from the LODESTAR.CFG configuration file. See **LODESTAR.CFG** on page 2-2 in **Chapter 2**: **Configuration Files** in the *Oracle Utilities Energy Information Platform Configuration Guide* for details on creating a LODESTAR.CFG configuration file.

Format

```
<identifier> = CONFIGGET(<identifier|expression>);
```

Where

- <identifier|expression> is a string that is a configuration file parameter.

Example

Get the string value of the 'CISFILENAME' parameter.

```
CONFIGPARAM = CONFIGGET("CISFILENAME");
```

CREATEOBJECT Function

Purpose

The CREATEOBJECT function creates a COM object, based on the object's ProgID.

This function creates a COM object. Once created, the properties and methods of the COM object are available to the Rules Language using COM expressions. The function returns a reference to the COM object which can be used in COM expressions.

See **Working with COM Objects** in **Chapter 8**: **Working with COM Components** in the *Oracle Utilities Rules Language User's Guide* for more information about using this function.

Format

```
<identifier> = CREATEOBJECT(<ProgID>);
```

Where

- <ProgID> is a string that contains the ProgID of the COM object to be created.

Example

Create a DOMDocument COM object.

```
OBJECT = CREATEOBJECT ("MSXML.DOMDocument");
```

CREATEREPORT Function

Purpose

The CREATEREPORT function generates a report (including Oracle BI Publisher reports) based on parameters passed from the Rules Language. Returns the GUID (global unique identifier) that uniquely identifies the report instance created by the function.

Format

```
<identifier> = CREATEREPORT(<stem_identifier>, <input_parameters>);
```

Where

- <stem_identifier> is a stem identifier with the following structure:

    - <STEM>.RPTTYPE is a string which specifies the Report Type of the report to be generated. Can be one of the following:

        - Oracle BI Publisher Reports ("BIPublisher")

        - Oracle Utilities Rules Language Reports ("LSRate")

    - <STEM>.RPTNAME is a string that specifies the Report Name. This is the name from the Report Templates table.

    - <STEM>.USERID is a string that specifies the User ID for the user generating the report. The default value is the application's user id. To obtain the current user ID, you can also use the **GETUSERID Function**.

    - <STEM>.TITLE is an optional string specifying the title for the report. For Crystal Reports, this is the title that will appear in the Report Title column on the View Reports screen in the Energy Information Platform user interface.

    - <STEM>.INPUTPARAMS is a string or XML document that contains input parameters. **Note**: The INPUTPARAMS tail must be used when passing parameters to reports such as Oracle Utilities Rules Language reports.

    - <STEM>.ISSHARED an optional string that specifies if the report is to be shared. Can be "Y" (shared) or "N" (not shared, default).

    - <STEM>.AUTO_SAVE is a flag that designates if saves should be enabled when executing a Rules Language report. Setting this to "false" disables all saves. Setting this to "true" is required when saving interval data.

    - <STEM>.CUSTOMINPUT is an optional string that contains custom input used by the report.

- <input_parameters> is a stem identifier or an identifier containing an XML document containing input parameters used by the report. If the <stem_identifier> does not contain an INPUTPARAMS tail, this second parameter will be used as report input parameters. **Note**: This second parameter can be used **only** with Crystal Reports.

Working with Input Parameters

If supplied, the INPUTPARAMS stem will be transformed into XML with the following structure:

```
<CRParameters>
  <CRParameter ParameterFieldName= "Tail name" CurrentValue= "Tail value"/>
</CRParameters>
```

where:

- "Tail Name" is the tail name (in quotes)

- "Tail Value" is the tail value (in quotes)

When specifying a Table.Column value, use the following naming convention:

```
INPUTPARAMS.<TABLE>__<COLUMN>
```

In this case, **two** underscores separate <TABLE> from <COLUMN>,which will be converted to <TABLE>.<COLUMN>

Example:

```
INPUTPARAMS.RECORDER = "1700";
INPUTPARAMS.CHANNEL = 1;
INPUTPARAMS.STARTTIME = "03/08/2004 00:00:00";
INPUTPARAMS.STOPTIME = "03/09/2004 23:59:59";
INPUTPARAMS.LSCHANNELCUTHEADER__STARTTIME  = "03/08/2004 00:00:00";
INPUTPARAMS.LSCHANNELCUTHEADER__STOPTIME="03/09/2004 00:00:00";
INPUTPARAMS.LSCHANNELCUTHEADER__RECORDER = "1700";
INPUTPARAMS.LSCHANNELCUTHEADER__CHANNEL ="1";
INPUTPARAMS.PARAM_NAME1 = "PARAM_VALUE1";
INPUTPARAMS.PARAM_NAME2= "PARAM_VALUE2";
INPUTPARAMS.PARAM_NAME3= "PARAM_VALUE3";
```

would be converted into the following input parameters XML:

```
<CRParameters>
  <CRParameter ParameterFieldName="RECORDERID" CurrentValue="1700"/>
  <CRParameter ParameterFieldName="CHANNEL" CurrentValue="1"/>
  <CRParameter ParameterFieldName="LSCHANNELCUTHEADER.STARTTIME"
CurrentValue="03/08/2004 00:00:00"/>
  <CRParameter ParameterFieldName= "PARAM_NAME1" CurrentValue="PARAM_VALUE1"/>
.
.
</CRParameters>
```

Examples

Run the Account Notes report (Oracle BI Publisher).

```
//Set up Input Parameters
INPUTPARAMS.ACCOUNT__ACCOUNTID = "800001";
INPUTPARAMS.ACCTNOTETYPE__ACCTNOTETYPECODE = "EXCEPTION";
INPUTPARAMS.RESOLVED= "Y";
//Set up Report Parameters
STEM.RPTTYPE = "BIPublisher";
STEM.RPTNAME = "Account Notes";
STEM.TITLE = "Account Notes Report";
STEM.ISSHARED = "Y";
//Run Report
GUID_1 = CREATEREPORT (STEM, INPUTPARAMS);
```

Run the Account Notes report (Crystal Report).

```
//Set up Input Parameters
INPUTPARAMS.ACCOUNT__ACCOUNTID = "800001";
INPUTPARAMS.ACCTNOTETYPE__ACCTNOTETYPECODE = "EXCEPTION";
INPUTPARAMS.RESOLVED= "Y";
//Set up Report Parameters
STEM.RPTTYPE = "Crystal";
STEM.RPTNAME = "Account Notes";
STEM.TITLE = "Account Notes Report";
STEM.ISSHARED = "Y";
//Run Report
GUID_1 = CREATEREPORT (STEM, INPUTPARAMS);
```

Run a Rules Language Report (Save Interval Data to Staging).

```
//Set up Input Parameters
INPUTPARAMS.RECORDERID = "RECORDER1";
INPUTPARAMS.CHANNELNUM = "1";
INPUTPARAMS.STARTTIME= "01/01/2008 00:00:00";
INPUTPARAMS.STOPTIME= "01/31/2008 23:59:59";

//Set up Report Parameters
STEM.RPTTYPE = "LSRate";
STEM.RPTNAME = "Save Interval Data to Staging";
STEM.TITLE = "Interval Data Save - 10011";
STEM.AUTO_SAVE = "true";
STEM.ISSHARED = "Y";
STEM.USERID = GETUSERID();
//Run Report
GUID_1 = CREATEREPORT (STEM, INPUTPARAMS);
```

Note

This function executes in all modes, and is **not** disabled if saves are disabled. Also, if used with Oracle Utilities Billing Component, this function executes upon when the rate schedule is processed, and even if the bill report is rejected.

To override this default behavior, you can use an IF THEN statement and the LSRSENV.COMMIT Rate Schedule Environment identifier to make sure the rate schedule being processed is in "commit" mode (that is, saves are enabled), as follows:

Run the Account Notes report if in "commit" mode.

```
//Set up Input Parameters
INPUTPARAMS.ACCOUNT__ACCOUNTID = "800001";
INPUTPARAMS.ACCTNOTETYPE__ACCTNOTETYPECODE = "EXCEPTION";
INPUTPARAMS.RESOLVED= "Y";
//Set up Report Parameters
STEM.RPTTYPE = "Crystal";
STEM.RPTNAME = "Account Notes";
STEM.TITLE = "Account Notes Report";
STEM.ISSHARED = "Y";
//Verify "commit" mode
IF LSRSENV.COMMIT = 1
    THEN
    //Run Report
    GUID_1 = CREATEREPORT (STEM, INPUTPARAMS);
END IF;
```

EMAILCLIENT Function

Purpose

The EMAILCLIENT function sends an email to a specified recipient or group. This can be used to send an email notice, or to send a file created by the Rules Language or other application (such as a report). The message is sent when the EMAILCLIENT statement is executed in the rate schedule. Sending email to contacts and groups requires that the specified contact(s) or group(s) be previously defined in the Oracle Utilities Data Repository. Returns 0 if successful. See **Return Codes** on page 13-111 for details concerning error-based return codes.

Format

```
<identifier> = EMAILCLIENT(<stem_identifier>);
```

Where

- <stem_identifier> is a stem identifier or expression that evaluates to a stem identifier. The corresponding tail identifiers provide the values for the email to be sent. Available tail identifiers include:

  - MAILTO_*: an email address stored in the TO field.

  - MAILCC_*: an email address stored in the CC field.

  - MAILBCC_*: an email address stored in the BCC field.

  - CONTACT_TO_LASTNAME_*: Last Name contact attribute (TO).

  - CONTACT_TO_FIRSTNAME_*: First Name contact attribute (TO).

  - CONTACT_TO_MIDDLENAME_*: Middle Name contact attribute (TO).

  - CONTACT_TO_SUFFIX_*: Suffix Name contact attribute (TO).

  - CONTACT_TO_TITLE_*: Title Name contact attribute (TO).

  - CONTACT_TO_UNIQUEID_*: Unique identifier contact attribute (TO).

  - CONTACT_TO_OWNERID_*: Owner identifier contact attribute (TO).

  - CONTACT_CC_LASTNAME_*: Last Name contact attribute (CC).

  - CONTACT_CC_FIRSTNAME_*: First Name contact attribute (CC).

  - CONTACT_CC_MIDDLENAME_*: Middle Name contact attribute (CC).

  - CONTACT_CC_SUFFIX_*: Suffix Name contact attribute (CC).

  - CONTACT_CC_TITLE_*: Title Name contact attribute (CC).

  - CONTACT_CC_UNIQUEID_*: Unique identifier attribute (CC).

  - CONTACT_CC_OWNERID_*: Owner identifier attribute (CC).

  - CONTACT_BCC_LASTNAME_*: Last Name contact attribute (BCC).

  - CONTACT_BCC_FIRSTNAME_*: First Name contact attribute (BCC).

  - CONTACT_BCC_MIDDLENAME_*: Middle Name contact attribute (BCC).

  - CONTACT_BCC_SUFFIX_*: Suffix Name contact attribute (BCC).

  - CONTACT_BCC_TITLE_*: Title Name contact attribute (BCC).

  - CONTACT_BCC_UNIQUEID_*: Unique identifier contact attribute (BCC).

  - CONTACT_BCC_OWNERID_*: Owner identifier contact attribute (BCC).

  - GROUP_TO_NAME_*: a group name attribute stored in the TO field.

  - GROUP_TO_OWNERID_*: the owner identifier attribute for the TO field.

- GROUP_CC_NAME_*: a group name attribute stored in the CC field.

- GROUP_CC_OWNERID_*: the owner identifier attribute for the CC field.

- GROUP_BCC_NAME_*: a group name attribute stored in the BCC field.

- GROUP_BCC_OWNERID_*: the owner identifier attribute for the BCC field.

- SUBJECT: the subject of the email.

- TEXT: the body of the email. Note that the buffer size of the text is limited to 64k. If the amount of text to be sent is greater than 64k, then sending an attachment can be used.

- FILE_*: path and file name of file attachment.

- GUID_*: a report identifier found in the LSRFRPTINSTANCE table. Translates to a file stored on the file system.

- IMPORT_*: path and file name of a text file (*.txt) to be imported into the body of the email. Note that the buffer size of the text is limited to 64k.

- EMAILFROM: a required name that resolves to a single email address for where the email came from.

*indicates that the tail uses a three digit number scheme to indicate multiple entries. Note that if the field name is not unique, then the last entry made is used. Also, for contact and group names this three digit number is used to relate all the common fields together.

Example

Send an email.

```
//Set up email attributes
EMAIL_XML.MAILTO_001 = "don_Ho@HoStarInternational.com";
EMAIL_XML.MAILCC_001 = "alex_Maker@HoStarInternational.com";
EMAIL_XML.CONTACT_TO_LASTNAME_001 = "Meister";
EMAIL_XML.CONTACT_TO_LASTNAME_002 = "Gordon";
EMAIL_XML.CONTACT_TO_FIRSTNAME_001 ="Burger";
EMAIL_XML.CONTACT_TO_TITLE_001 = "Mayor";
EMAIL_XML.CONTACT_TO_TITLE_002 = "Action Hero";
EMAIL_XML.GROUP_TO_GROUPNAME_001 = "HoSpecialGroup";
EMAIL_XML.GROUP_TO_OWNERID_001 = "SpecialName";
EMAIL_XML.SUBJECT = "The latest news on Performance Numbers!";
EMAIL_XML.TEXT = "Here are the latest numbers!";
EMAIL_XML.FILE_001 = "c:\HoStatusReport.xls";
EMAIL_XML.EMAILFROM = "MimiHo@HoStarInternational.com";
//Send email
EMAIL_STAT = EMAILCLIENT(EMAIL_XML);
```

Notes

The EMAILCLIENT function requires the presence of the LSRELAY.CFG.XML file in the C:\LODESTAR\CFG directory on the executing computer. See **LSRELAY.CFG.XML** on page 2-34 in the *Oracle Utilities Energy Information Platform Configuration Guide* for more information about this file.

This function executes in all modes, and is **not** disabled if saves are disabled. Also, if used with Oracle Utilities Billing Component, this function executes upon when the rate schedule is processed, and even if the bill report is rejected.

To override this default behavior, you can use an IF THEN statement and the LSRSENV.COMMIT Rate Schedule Environment identifier to make sure the rate schedule being processed is in "commit" mode (that is, saves are enabled), as follows:

Send an email if in "commit" mode.

```
//Set up email attributes
```

```
EMAIL_XML.MAILTO_001 = "don_Ho@HoStarInternational.com";
EMAIL_XML.SUBJECT = "The latest news on Performance Numbers!";
EMAIL_XML.TEXT = "Here are the latest numbers!";
EMAIL_XML.FILE_001 = "c:\HoStatusReport.xls";
EMAIL_XML.MAILFROM = "MimiHo@HoStarInternational.com";
//Verify "commit" mode
IF LSRSENV.COMMIT = 1
   THEN
   //Send email
   EMAIL_STAT = EMAILCLIENT(EMAIL_XML);
END IF;
```

Return Codes

The function returns a 0 (zero) if successful. If an error occurs, the return code provided by the function indicates the nature of the error. The table below outlines the function's return codes and probable causes for each.

Code	Description	Probable Cause
-1	EMAIL_SERVER_CONNECTION_FAILED	Unable to initialize the socket sub-system. Unable to connect to the mail server. May need to increase EMAIL_TIMEOUT.
-2	INVALID_EMAIL_ADDRESS	Missing a valid 'Reply' address. Missing a valid 'From' address. An invalid address was encountered.
-3	EMAIL_SEND_FAILED	Missing a valid 'From' address. Missing a valid 'Recipient', 'CC' or 'BCC' address. Unable to send the email. May need to increase EMAIL_TIMEOUT.
-4	ATTACHMENT_NOT_FOUND	Attachment %s is missing. Make sure GUID/path/file exists.
-5	AUTH_EMAIL_FAILED	Unable to connect to the mail server. May need to increase EMAIL_TIMEOUT.
-6	DISCONNECT_FAILED	Disconnect failed. Socket not closed.
-99	(others)	Missing configuration item %s. Unable to locate a valid lsrelay.cfg.xml configuration file. Unable to locate PLBX report. Check config file for report listing.

EXPBLKMDMUSAGE Function

Purpose

The EXPBLKMDMUSAGE function exports usage for a specified account or service point over a specified date range to a Oracle Utilities Meter Data (*.lsm) file. Usage exported with this function can be consumption, interval, or time-of-use usage. Returns 0 if successful.

Format

```
<identifier> = EXPBLKMDMUSAGE(<stem_identifier>);
```

Where

- <stem_identifier> is a stem identifier or expression that evaluates to a stem identifier. The corresponding tail identifiers provide the values for the usage to be exported. Available tail identifiers include:

    - ACCOUNTID: The Account ID that corresponds to the usage to export. If not provided, SERVICEPOINTID is required.

    - SERVICEPOINTID: The Service Point ID that corresponds to the usage to export. If not provided, ACCOUNTID is required.

    - CHANNELID: The Channel ID of the usage to export. If not provided, all usage for all channels for the specified Account or Service Point will be exported.

    - UOM: The Expected Unit of Measure of the usage to export. If not provided, all usage for all UOMs for the specified Account or Service Point will be exported

    - USAGETYPECODE (Required): The Usage Type Code of the usage to export.

    - STARTDATE (Required): The start date of the usage to export

    - STOPDATE (Required): The stop date of the usage to export

    - USAGECATEGORY: The Usage Category (RAW, STAGING, or FINAL) of the usage to export

    - OUTPUTFILE (Required): The path and file name for the file to which the usage is exported

Examples

Export all "FINAL" KWH (01) interval usage for January 2006 for account "ACCOUNT-5A".

```
//Set up export parameters
MDM_READ.ACCOUNTID = "ACCOUNT-5A";
MDM_READ.UOM = "01";
MDM_READ.USAGETYPECODE = "INTERVAL";
MDM_READ.STARTDATE = "01/01/2006";
MDM_READ.STOPDATE = "01/31/2006 23:59:59";
MDM_READ.USAGECATEGORY = "FINAL";
MDM_READ.OUTPUTFILE = "C:\LODESTAR\USER\ACCOUNT-5A_USAGE.lsm";
//
//Export Reading
EXPORT = EXPBLKMDMUSAGE(MDM_READ);
```

Export all "FINAL" KWH (01) consumption usage for January 2006 for service point "SP-5B".

```
//Set up export parameters
MDM_READ.SERVICEPOINTID = "SP-5B";
MDM_READ.UOM = "01";
MDM_READ.OUTPUTFILE = "C:\LODESTAR\USER\SP-5B_USAGE.lsm";
MDM_READ.STARTDATE = "01/01/2006";
MDM_READ.STOPDATE = "01/31/2006 23:59:59";
MDM_READ.USAGETYPECODE = "CONSUMPTION";
```

```
MDM_READ.USAGECATEGORY = "FINAL";
//
//Export Reading
EXPORT = EXPBLKMDMUSAGE(MDM_READ);
```

Notes

This function executes in all modes, and is **not** disabled if saves are disabled. Also, if used with Oracle Utilities Billing Component, this function executes upon when the rate schedule is processed, and even if the bill report is rejected.

To override this default behavior, you can use an IF THEN statement and the LSRSENV.COMMIT Rate Schedule Environment identifier to make sure the rate schedule being processed is in "commit" mode (that is, saves are enabled), as follows:

Export usage if in "commit" mode.

```
//Set export attributes
//Set up export parameters
MDM_READ.SERVICEPOINTID = "SP-5B";
MDM_READ.UOM = "01";
MDM_READ.OUTPUTFILE = "C:\LODESTAR\USER\SP-5B_USAGE.lsm";
MDM_READ.STARTDATE = "01/01/2006";
MDM_READ.STOPDATE = "01/31/2006 23:59:59";
MDM_READ.USAGETYPECODE = "CONSUMPTION";
MDM_READ.USAGECATEGORY = "FINAL";
//Verify "commit" mode
IF LSRSENV.COMMIT = 1
   THEN
   //Export data
   EXPORT = EXPBLKMDMUSAGE(MDM_READ);
END IF;
```

EXPMDMUSAGE Function

Purpose

The EXPMDMUSAGE function exports a specified usage reading to a Oracle Utilities Meter Data (*.lsm) file. Usage exported with this function can be consumption, interval, or time-of-use usage. Returns 0 if successful.

Format

```
<identifier> = EXPMDMUSAGE(<stem_identifier>);
```

Where

- <stem_identifier> is a stem identifier or expression that evaluates to a stem identifier. The corresponding tail identifiers provide the values for the usage reading to be exported. Available tail identifiers include:

    - METERID (Required): The Meter ID of the reading to export

    - CHANNELID: The Channel ID of the reading to export. Required for interval readings.

    - UOM (Required): The Expected Unit of Measure of the reading to export

    - USAGETYPECODE (Required): The Usage Type Code of the reading to export

    - STARTDATE (Required): The Start Read Time of the reading to export

    - STOPDATE (Required): The Stop Read Time of the reading to export

    - USAGECATEGORY (Required): The Usage Category (RAW, STAGING, or FINAL) of the reading to export

    - OUTPUTFILE (Required): The path and file name for the file to which the reading is exported

Example

Export the January 1 - 31 (FINAL) reading for meter "METER-5A", channel "1", and UOM "01" (KWH).

```
//Set up reading parameters
MDM_READ.METERID = "METER-5A";
MDM_READ.CHANNELID = "1";
MDM_READ.UOM = "01";
MDM_READ.USAGETYPECODE = "INTERVAL";
MDM_READ.STARTDATE = "01/01/2006";
MDM_READ.STOPDATE = "01/31/2006 23:59:59";
MDM_READ.USAGECATEGORY = "FINAL";
MDM_READ.OUTPUTFILE = "C:\LODESTAR\USER\MDM_USAGE.lsm";
//
//Export Reading
EXPORT = EXPMDMUSAGE(MDM_READ);
```

Notes

This function executes in all modes, and is **not** disabled if saves are disabled. Also, if used with Oracle Utilities Billing Component, this function executes upon when the rate schedule is processed, and even if the bill report is rejected.

To override this default behavior, you can use an IF THEN statement and the LSRSENV.COMMIT Rate Schedule Environment identifier to make sure the rate schedule being processed is in "commit" mode (that is, saves are enabled), as follows:

Export usage if in "commit" mode.

```
//Set export attributes
```

```
//Set up export parameters
MDM_READ.METERID = "METER-5A";
MDM_READ.CHANNELID = "1";
MDM_READ.UOM = "01";
MDM_READ.USAGETYPECODE = "INTERVAL";
MDM_READ.STARTDATE = "01/01/2006";
MDM_READ.STOPDATE = "01/31/2006 23:59:59";
MDM_READ.USAGECATEGORY = "FINAL";
MDM_READ.OUTPUTFILE = "C:\LODESTAR\USER\MDM_USAGE.lsm";
//Verify "commit" mode
IF LSRSENV.COMMIT = 1
    THEN
    //Export data
    EXPORT = EXPMDMUSAGE(MDM_READ);
END IF;
```

EXPORT_USAGE Function

Purpose

The EXPORT_USAGE function exports interval data associated to a supplied Account ID to a Microsoft Excel (*.xls) file. Interval data exported by this function must be associated to the supplied Account ID in the Entity Interval Data table. This function provides access to the Export Account Usage function of the Energy Information Platform from the Rules Language. Returns 0 if successful.

Format

```
<identifier> = EXPORT_USAGE(<stem_identifier>);
```

Where

- <stem_identifier> is a stem identifier or expression that evaluates to a stem identifier. The corresponding tail identifiers provide the values for the data to be exported. Available tail identifiers include:

  - PURPOSE (Required): Specifies which set of records for a given Account should be used when there is more than one set of records in the Entity Interval Data table for a given Account.

  - ENTITYTYPE: The type of entity for which data is exported. ACCOUNT is default if not supplied.

  - ACCOUNTID (Required): The Account ID for which data is to be exported.

  - SCALERESULTS : The manner in which the data is to be scaled (if needed). Can be one of the following: AUTO (automatic), 5MIN, 10MIN, 15MIN, 20MIN, 30MIN, HOUR, DAY, WEEK, MONTH, and YEAR. The default is AUTO (automatic).

  - STARTTIME (Required): Start time for the start of the date range for which data will be exported.

  - STOPTIME (Required): Stop time for the stop of the date range for which data will be exported.

  - OUTPUTFILE: Path and file name to the file where the data will be exported.

  - INTDLOCATION: Interval data location from which the data will be exported. The default is the table LSCHANNELCUTHEADER table.

  - INCLUDESTATUS: Indicates if any status codes present in the interval data will be exported. Valid values include "TRUE" or "FALSE".

Example

Export data for Account ID 80001 from 05/01/2005 through 05/31/2005 to a file called "80001_Export.xls" to the C:\LODESTAR\USER directory.

```
ACCOUNT_ID = 800001;
STARTTIME = '05/01/2005 00:00:00';
STOPTIME = '05/31/2005 23:59:59';
//Set export attributes
STEM.PURPOSE = "USAGE";
STEM.ENTITYTYPE = ACCOUNT;
STEM.ACCOUNTID = ACCOUNT_ID;
STEM.SCALERESULTS = AUTO;
STEM.STARTTIME = STARTTIME;
STEM.STOPTIME = STOPTIME;
STEM.OUTPUTFILE ="C:\LODESTAR\USER\80001_EXPORT.XLS";
STEM.INCLUDESTATUS ="TRUE";
//Export data
```

```
EXPORT = EXPORT_USAGE (STEM);
```

Notes

This function executes in all modes, and is **not** disabled if saves are disabled. Also, if used with Oracle Utilities Billing Component, this function executes upon when the rate schedule is processed, and even if the bill report is rejected.

To override this default behavior, you can use an IF THEN statement and the LSRSENV.COMMIT Rate Schedule Environment identifier to make sure the rate schedule being processed is in "commit" mode (that is, saves are enabled), as follows:

Export usage if in "commit" mode.

```
//Set export attributes
STEM.PURPOSE = "USAGE";
STEM.ENTITYTYPE = ACCOUNT;
STEM.ACCOUNTID = ACCOUNT_ID;
STEM.SCALERESULTS = AUTO;
STEM.STARTTIME = STARTTIME;
STEM.STOPTIME = STOPTIME;
STEM.OUTPUTFILE ="C:\LODESTAR\USER\80001_EXPORT.XLS";
STEM.INCLUDESTATUS ="TRUE";
//Verify "commit" mode
IF LSRSENV.COMMIT = 1
    THEN
    //Export data
    EXPORT = EXPORT_USAGE (STEM);
END IF;
```

FACTORINEFFECT Function

Purpose

The FACTORINEFFECT function checks to see if specified factor has a factor value on the given date. Returns 1 (true) or 0 (false).

Returns 1 if there is a FACTORVALUE record related to the FACTOR record on the supplied date, and has a non-null factor value whose start time is the first one on or before the supplied date. Otherwise, returns 0. A supplied date overrides a date in the factor code. If neither date is supplied, the bill period effective date is used.

Format

```
<identifier> = FACTORINEFFECT(<factor_code>, <date>)
```

Where

*   <factor_code> is a string that is a key to a record in the FACTOR Table.

*   <date> is a date string. If not supplied and the factor code does not have a date, the bill period effective date is used.

Example

Return a 1 (true) if there is a factor value record for factor "CCA_17" on 01/10/93. If not, return a 0 (false):

```
IN_EFFECT = FACTORINEFFECT ("CCA_17", "01/10/1993")
```

GETUSERSPECIFIEDSTOP Function

Purpose

The GETUSERSPECIFIEDSTOP function returns the "User Specified Stop" date (if supplied) or NULL (if not supplied). This function returns the "User Specified Stop" date when processing billing calculations using Current/Final Billing, or Trial Bill/Calculation. The function returns NULL in all other billing modes.

Format

```
<identifier> = GETUSERSPECIFIEDSTOP();
```

Example

Get the user specified stop.

```
USERSPECSTOP = GETUSERSPECIFIEDSTOP();
IF HASVALUE(USERSPECSTOP)
   THEN
   LABEL USERSPECSTOP "User Specified Stop";
   ELSE
   ...
```

INEFFECT Function

Purpose

The INEFFECT function indicates whether a specified tariff rider, rate code, or override was in effect for the account on a particular date.

The function returns 1 if yes, 0 if no.

Format

```
<identifier> = INEFFECT(<table_name>, <record_key>,
<date_identifier|date_constant>);
```

Where

- <table_name> is one of:
  - "ACCTRIDERHIST" (links tariff riders to accounts)
  - "ACCTRATECODEHIST" (links rate code to accounts)
  - "ACCTOVERRIDEHIST" (links overrides to accounts)
  - "ACCTNAMEOVERHIST" (links overrides to accounts at the meter level)
- <record_key> is an identifier for the rider, rate code, or override, depending on the table specified in the first parameter.

  Note: The <record_key> is the part of the key that doesn't include the account number or start time. For example, for a rate form it would be <"operating_company_code, jurisdication_code,rate_form_code">. See the example below.

- <date_identifier|date_constant> is either an identifier that contains a date (such as BILL_PERIOD or BILL_START) or a date constant in the format 'mm/dd/yyyy' or 'mm/dd/yyyy hh:mm'.

Example

Return a value (0 or 1) to indicate whether or not the rate 1-GL was in effect for the current account on its bill start date.

```
RC_INEF = INEFFECT("ACCTRATECODEHIST", "GECO,SF,1-GL", BILL_START);
```

ISHOLIDAY Function

Purpose

The ISHOLIDAY function returns a 1 if a specified date is in a specified holiday list; otherwise, it returns a 0.

Format

```
<identifier> = ISHOLIDAY(<date_identifier|date_constant>,
<holiday_list_name>);
```

Where

- <date_identifier|date_constant> is either an identifier that contains a date (such as BILL_PERIOD or BILL_START) or a constant in the format 'mm/dd/yyyy', 'mm/dd/yyyy hh:mm', 'yyyy-mm-dd', or 'yyyy-mm-dd hh:mm'.

- <holiday_list_name> is the name of a list in the HOLIDAYLIST Table.

Example

Return a 1 if 02/14/1998 is in the holiday list "STANDARD_HOLIDAYS". Otherwise, return a 0.

```
VDAY = ISHOLIDAY(02/14/1998, "STANDARD_HOLIDAYS")
```

RUNRATE Function

Purpose

The RUNRATE function executes a new rate, starting the rate as a new process and then continue with its own execution. The parameters for the rate will be passed in as parameters to this function. The UserID, Password, connect string, and Qualifier will default to the parent rate and must not be passed in. Returns an integer value which can be used to wait for the spawned rate to complete.

Format

```
<identifier> = RUNRATE(<parameter1>[,<parameter2>,...]);
```

Where

- <parameter1> is one of the parameters used by the rate being executed. The function can accept the same parameters as the RUNRS command line program. See **Executing Oracle Utilities Rules Language Rate Schedules** on page 8-41 of the *Oracle Utilities Energy Information Platform Configuration Guide* for more information about RUNRS.

Example

Execute rate "CODE1".

```
RATE_A = "-vOPCO1:JURIS1:CODE1";
START_A =  "-s05/01/1993";
STOP_A = "-t05/31/1993 23:59:59";
RUN_A = RUNRATE (RATE_A, START_A, STOP_A);
```

SAVE_PROFILE Function

Purpose

The SAVE_PROFILE function saves a Rules Language code profile to a specified file. It creates a code profile for the rate schedule in which it's used. This profile is similar to the profile available when running the Trial Calculation function in Data Manager or Oracle Utilities Billing Component. See **Rules Language Profiling** on page 14-2 in the *Oracle Utilities Energy Information Platform Configuration Guide* for more information about using this function.

Format

```
<identifier> = SAVE_PROFILE(<path and file name>);
```

Where

*   <path and file name> is the path and file name for the file that will contain the Rules Language code profile.

    Note: If the specified path does not exist, the rate schedule will run to completion, but the Rules Language will generate a warning message and the code profile will not be created.

Example

Save a Rules Language code profile to a file called "CODE_PROFILE.TXT" in the "C:\LODESTAR\LOG" directory.

```
CODE_PROFILE = SAVE_PROFILE("C:\LODESTAR\LOG\CODE_PROFILE.TXT");
```

NOTE: This function should only be used when troubleshooting Rules Language performance issues or other problems, as creating the code profile will have a negative impact on performance.

SETREPORTTITLE Function

Purpose

The SETREPORTTITLE function sets the report title for Rules Language reports run via the Report Framework of the Oracle Utilities Energy Information Platform. The title set by this function appears in the Report Title column on the View Reports screen.

Note: This function can be used in conjunction with Rules Language reports and Oracle Utilities Billing Component billing processes.

Format

```
<identifier> = SETREPORTTITLE(<REPORT_TITLE>);
```

where:

<REPORT_TITLE> is an identifier or string literal that contains the report title for the current running Rules Language report.

Example

Set the title for the current Rules Language report to "Validate 1700,1."

```
RECORDERID = "1700";
CHANNELNUM = "1";
RECORDER_CHANNEL = RECORDERID + "," + CHANNELNUM;
TITLE = "Validate " + RECORDER_CHANNEL;
RPT_TITLE = SETREPORTTITLE(TITLE);
```

USEREXIT Function

Purpose

The USEREXIT function calls a user-written function. Users may write their own functions for use in rate forms. This function is used in a rate form to call the user-written function.

Format

```
<identifier> = USEREXIT(<dll_name>, <function_name>[,<arg1>...]);
```

Where

- <dll_name> is the name of a Dynamic Link Library (DLL) that contains the function. Do not include the '.dll' extension.

- <function_name> is the name of a function in the DLL.

- <arg#> are arguments to the function.

Example

Call a user-written function "POWERFACTOR" from the UMSEXMPL DLL.

```
LABEL PF "POWER FACTOR";
PF = USEREXIT("UMSEXMPL", "POWERFACTOR", KVARH, KWH);
```

WAITFORRUNRATE Function

Purpose

The WAITFORRUNRATE function causes a rate to wait for rates created using the RUNRATE function.

This function causes a currently running rate to wait for any rates that it had created using the **RUNRATE Function**. This function takes zero (0) or more parameters, each of which must be integer values returned by the RUNRATE function. If no parameters are provided, then this function waits for all rates created by this rate, to complete. Return the number of rates that it waited for.

Format

```
<identifier> = WAITFORRUNRATE(<parameter1>[,<parameter2>,...]);
```

Where

• <parameter1> is an integer value returned by the **RUNRATE Function**.

Example

Run rates CODE1 and CODE2 and wait for them to complete before continuing processing.

```
//Set up rate parameters
RATE_A = "-vOPCO1:JURIS1:CODE1";
START_A =  "-s05/01/1993";
STOP_A = "-t05/31/1993 23:59:59";
RATE_B = "-vOPCO2:JURIS2:CODE2";
START_B =  "-s05/01/1993";
STOP_B = "-t05/31/1993 23:59:59";
//Run Rates A + B
RUN_A = RUNRATE (RATE_A, START_A, STOP_A);
RUN_B = RUNRATE (RATE_B, START_B, STOP_B);
//Wait for Rates A+ B
WAIT = WAITFORRUNRATE (RUN_A, RUN_B);
```

Appendix A

Reserved Words

This appendix lists the "reserved words"—that is, words that have special meaning to the programs and therefore can be used only as specified. This includes:

- **Statement Keywords**

- **Function Keywords**

- **Interval Data Function Keywords**

- **Meter Value Function Keywords**

- **Predefined Identifiers**

- **Predefined, Assignable Identifiers**

Statement Keywords

Statement keywords cannot be used as identifiers; the predefined identifiers can be used only as described in this manual.

ABORT	LEAVE
ADDITIONAL	LIST
ALL	NEXT
AND	NOVALUE
AS	OR
BLOCK	OTHERWISE
CHANNEL	OVERRIDE
CHARGE	REPORT
CLEAR	REVENUE
DETERMINANT	SAVE
DONE	SECTION
EACH	SELECT
ELSE	SET
END	TABLE
FACTOR	THEN
FIRST	TO
FOR	TOTAL
FROM	UNBILLED
IF	WARN
IGNORE	WEEK
INCLUDE	WHEN
INTO	

Function Keywords

Function keywords cannot be used as identifiers; the predefined identifiers can be used only as described in this manual.

ACCOUNTFACTOR	MAXKW
AVGSEASON	MAXRANGE
BILLINGHOURS	MAXSEASON
CEIL	MIN
COMPIKVA	MINNZ
COMPKVA	MINRANGE
COMPLF	MINSEASON
COMPSUM	MONTH
DAY	MONTHDIFF
DBDATETIME	MONTHHOURS
FLAG	PRORATEFACTOR
FLOOR	ROUND
HASVALUE	ROUND2VALUE
HISTCOUNT	ROUNDDATE
HISTMAX	RSPRORATE
HISTMIN	SEASONVALUE
HISTMINNZ	SQROOT
HISTVALUE	STRING
IDATTR	SUMSEASON
LF2KW	USEREXIT
LF2KWH	WEEKDIFF
LISTCOUNT	YEAR
MAX	YEARSTR
INTO	WHEN

Interval Data Function Keywords

Interval Data function keywords cannot be used as identifiers; the predefined identifiers can be used only as described in this manual.

INTDBLOCKOP	INTDLOADHISTLIST
INTDCOUNT	INTDLOADUOM
INTDCREATEDAYMASK	INDTLOADUOMDATES
INTDCREATEFACTORMASK	INTDLOADUOMHIST
INTDCREATEMASK	INTDRELEASE
INTDCREATEOVERRIDEDAYMASK	INTDSCALAROP
INTDCREATEOVERRIDEMASK	INTDSCALE
INDTCREATETOUPERIOD	INTDSMOOTH
INTDEXPORT	INTDTOU
INTDLOADDATES	INTDTOURRELEASE
INTDLOADHIST	INTDVALUE
INTDLOADLIST	
INTDLOADLISTDATES	

Meter Value Function Keywords

Meter Value function keywords cannot be used as identifiers; the predefined identifiers can be used only as described in this manual

MVLOAD
MVLOADDATES
MVLOADHIST
MVLOADLIST
MVLOADLISTDATES
MVLOADLISTHIST

Predefined Identifiers

AUXILIARY_DEMAND	RS_EFFECTIVE_STOP
BILL_PERIOD	RS_JURIS_CODE
BILL_START	RS_OPCO_CODE
BILL_STOP	UIDACCOUNT
NUMDAYS	RS_OPCO_CODE
RATE_CODE	RS_JURIS_CODE
READ_DATE	UIDACCOUNT
RS_EFFECTIVE_START	

Predefined, Assignable Identifiers

BILL_PERIOD_SELECT
HOURS_PER_MONTH
INTD_ERROR_STOP
SEASON_SCHEDULE_NAME

Appendix B

XML Statements and Functions

This chapter describes XML statements and functions provided by the Oracle Utilities Rules Language, including:

- **XML Overview**

- **XML Statements**

- **XML/Document Object Management Functions**

- **Using the XML Statements and Functions**

XML Overview

The Oracle Utilities Rules Language provides two mechanisms for processing XML: a declarative approach using the XML_ELEMENT and XML_OP statements, and a functional approach using the XML/Document Object Management functions.

The declarative approach allows the user to specify a known XML format and have the underlying Rules Language processor assign values appropriately. This approach essentially "flattens" the nested XML structure so that every element or sub-element is uniquely represented by one Rules Language identifier.

The functional approach gives the user more flexibility in handling an unknown format, but requires detailed knowledge of Document Object Management (DOM).

XML Data Types

The two new data types introduced with these functions are XML Document and XML Node. The same XML document or node may be assigned to several identifiers; care should be exercised when using these functions, particularly the delete functionality.

The main uses of the XML document format are to load and save data, and to retrieve the root element in the document. The root element is an XML node. You can retrieve the type and value of a node and its siblings and, for nodes that are elements, its attributes and children. If a child node is not actually in the XML document, the node and all its attributes will be cleared.

There are no operations allowed on an XML document. XML documents can only be used as a parameter to one of the XML functions described in this appendix (see **XML/Document Object Management Functions** on page B-12). The only operations allowed on a XML node are comparison (= or <>) to zero. Otherwise, nodes must be used in one of the statements or functions described in this appendix.

Using Stem.Tail XML Identifiers

If an XML node is assigned to an identifier that is a stem, its Stem.Tail identifiers with tails NODENAME, NODETYPE and NODEVALUE are also assigned their corresponding values. The Stem.Tail identifier with tail NODEPRESENT is assigned the integer 1. All other Stem.Tail identifiers are assigned the value of the node's child whose name is the tail.

If a Stem.Tail identifier whose tail is NODEVALUE is assigned a value, and the stem is an XML element with a node, the string representation of the value will be assigned as the node's value.

If an identifier that is an XML element with a node is assigned a value, the string representation of the value will be assigned as the node's value. However, if you want to use the node's value in an expression, you must use Stem.NODEVALUE.

If a Stem.Tail identifier whose tail is not NODEVALUE is assigned a value, and the stem is an XML element with a node, the tail is assumed to be the name of an attribute of the node, and the string representation of the value will be assigned as the value of this attribute.

To remove an attribute from a node, assign it an empty value:

```
// Remove the attribute Tail from the node STEM.
CLEAR X;
STEM.Tail = X;
```

XML Statements

This section provides detailed explanations of the XML statements available in the Oracle Utilities Rules Language. It also describes the formats and conventions used with statements in this manual, and the format in which the statement descriptions are presented.

Identifier Statement

Purpose

The IDENTIFIER Statement is used to define identifiers before they are used.

The order in which identifiers appear in a rate from determines several things, such as the order in which they appear in reports. In general, the earlier an identifer appears in a rate schedule, the earlier it appears in the report. This statement lets you determine ordering without executing any statements (the IDENTIFIER Statement has no run-time component; it only defines identifiers).

This statement can also be used to define a parent identifier before the identifier is used in the **XML_ELEMENT Statement** on page B-6.

Format

IDENTIFIER statements have this format:

```
IDENTIFIER <identifier>, <identifier> ...;
```

Where:

- <identifier> is the identifier you wish to define.

To Create

The IDENTIFIER statement can only be created from the Rules Language Text Editor. See **The Rules Language Text Editor** on page 2-11 in the *Oracle Utilities Rules Language User's Guide* for more information.

Example

Define the LS_INPUT identifier.

```
/* Predefine identifier */
IDENTIFIER LS_INPUT;
```

Notes

When using the IDENTIFIER statement to define XML element identifiers, the IDENTIFIER statement should only be used to define the root element.

OPTIONS Statement

Purpose

The OPTIONS Statement is used to specify that the case (UPPER or lower) of identifiers should remain as defined. If not present, all identifiers are converted to uppercase. If it is present the name of an identifier remains exactly as typed. With this option, if two identifiers differ only in the case of some of their letters, they are different identifiers.

This statement is useful when defining XML attributes and elements that may need to be either lower-cased or mixed-case.

Format

OPTIONS statements have this format:

```
IDENTIFIER MIXED_CASE_IDENTIFIERS_UNIQUE;
```

Where:

• MIXED_CASE_IDENTIFIERS_UNIQUE indicates that the case of identifiers remain unaltered.

To Create

The OPTIONS statement can only be created from the Rules Language Text Editor. See **The Rules Language Text Editor** on page 2-11 in the *Oracle Utilities Rules Language User's Guide* for more information.

Example

Allow mixed case XML attributes

```
/* Allow mixed-case XML attributes */
OPTIONS MIXED_CASE_IDENTIFIERS_UNIQUE;
```

Notes

Use of the OPTIONS statement affects identifiers that appear after this statement in the rate form. Identifiers that appear before it are uppercased.

All Oracle Utilities defined identifiers such as BILL_PERIOD, $EFFECTIVE_REVENUE, determinant identifiers, and interval data attributes must all be entered in upper case if the OPTIONS statement is used.

XML_ELEMENT Statement

Purpose

The XML_ELEMENT Statement lets you map an XML format into Rules Language identifiers. The XML format consists of elements and sub-elements, and this statement describes the relationship between a sub-element and its parent. If the parent element is assigned, all its attributes and children are automatically assigned their respective values, recursively. The defined identifier can also be used in the **FOR EACH x IN XML_ELEMENT_OF 0 Statement** on page B-8 to iterate over multiple sub-elements with the same name.

Format

XML_ELEMENT statements have this format:

```
XML_ELEMENT <identifier> NODENAME <symbol|literal> PARENT
<parent_identifier>;
```

Where:

- <identifier> is an identifier used to represent this child element of the parent. An identifier may appear at most once here. Attributes of the element can be represented using the `identifier.attribute` syntax.

- NODENAME is an optional keyword that allows you to define the node name of the element.

- <symbol|literal> is a symbol or literal that exactly matches an element name (case sensitive). There may be several identifiers with the same node name, but different parents.

- PARENT is an optional keyword that allows you to define the parent of the element.

- <parent_identifier> *Optional;* a previously defined identifier. When it is assigned, this identifier is also set if its element is a child of the parent element. If there is no parent assigned, the identifier is assumed to be the root element of the document.

To Create

The XML_ELEMENT Statement can only be created from the Rules Language Text Editor. See **The Rules Language Text Editor** on page 2-11 of the *Oracle Utilities Rules Language User's Guide* for more information.

Example

Set the structure of the Oracle Utilities Import format.

```
/* Set the XML structure of the Oracle Utilities Import XML Format */
/* Set the root element */
IDENTIFIER LS_IMPORT;
/* Declare the child tree */
XML_ELEMENT LS_IMPORT NODENAME "LS_IMPORT";
XML_ELEMENT CUST_DATA NODENAME "CUSTOMER_DATA" PARENT LS_IMPORT;
XML_ELEMENT REC_GROUP NODENAME "RECORD_GROUP_TRANSACTION" PARENT
CUST_DATA;
/* Declare a record */
XML_ELEMENT LS_RECORD NODENAME "LODESTAR_RECORD" PARENT REC_GROUP;
XML_ELEMENT TABLE_ID NODENAME "TABLE" PARENT LS_RECORD;
XML_ELEMENT TABLE_NAME NODENAME "NAME" PARENT TABLE_ID;
/* Declare a column */
XML_ELEMENT LS_COLUMN NODENAME "COLUMN" PARENT TABLE_ID;
XML_ELEMENT COLUMN_NAME NODENAME "COLUMN_NAME" PARENT LS_COLUMN;
XML_ELEMENT COLUMN_VALUE NODENAME "COLUMN_VALUE" PARENT LS_COLUMN;
```

Notes

If a parent identifier is cleared using the **Clear Statement**, all its children and attribute identifiers are also cleared, recursively. If an XML_ELEMENT identifier is assigned an XML node, all of its unassigned parent elements will be created as needed, so that its entire parent structure is assigned. The attributes of an element can be assigned any time after the element has been created.

FOR EACH x IN XML_ELEMENT_OF 0 Statement

Purpose

The FOR EACH x IN XML_ELEMENT_OF 0 Statement repeats a set of nested statements for each element defined in an XML structure. This statement iterates the nested statements over all matching elements, one by one. Matching elements have the same element name and the same parent element, as defined in the XML_ELEMENT Statement.

Format

FOR EACH x IN XML_ELEMENT_OF 0 statements have this format:

```
FOR EACH <xml_element_identifier> IN XML_ELEMENT_OF 0
    <nested_statements>
END FOR;
```

Where:

- <xml_element_identifier> is an identifier that appears in the IDENTIFIER clause of an XML_ELEMENT Statement.

To Create

The FOR EACH x IN XML_ELEMENT_OF 0 Statement can only be created from the Rules Language Text Editor. See **The Rules Language Text Editor** on page 2-11 of the *Oracle Utilities Rules Language User's Guide* for more information.

Example

Perform a set of operations on each LODESTAR_RECORD element.

```
/* Set the XML structure of the Oracle Utilities Import XML Format */
FOR EACH LS_RECORD IN XML_ELEMENT_OF 0;
    <operations>
END FOR
```

Notes

The 0 is required after XML_ELEMENT_OF.

XML_OP Statement

Purpose

The XML_OP Statement performs an operation on one or more XML elements (as defined using the XML_ELEMENT Statement). Supported operations include CREATE, INSERT, COPY, and DELETE.

Format

XML_OP statements have this format:

```
XML_OP <operation> <identifier> [,<identifier>...];
```

Where:

- <operation> is a literal or symbol that is one of the following:

    - **"CREATE" or CREATE:** Creates this node as a child of its parent. Each identifier's node is created, from left to right. If no parent node was specified in the XML_ELEMENT statement, this node is assumed to be the root element of an XML document. The document is created, with this identifier as its root. If the parent node exists, the sub-element is created and attached to the parent. If a node with this name already exists or the identifier is already assigned a node, a new node is still created. The new node will be a sibling of the previous node if the parent is unchanged; otherwise, it will be a new child of the parent node.

    - **"CREATE_ALL" or CREATE_ALL:** Creates this node as a child of its parent, and then creates all its children, recursively. Each identifier's node is created, from left to right. If no parent node was specified in the XML_ELEMENT statement, this node is assumed to be the root element of an XML document. The document is created, with this identifier as its root. If the parent node exists, the sub-element is created and attached to the parent. If a node with this name already exists or the identifier is already assigned a node, a new node is still created. The new node will be a sibling of the previous node if the parent is unchanged; otherwise, it will be a new child of the parent node.

    - **"INSERT" or INSERT:** Creates this node as a child of its parent, immediately after its previous instance. The node must have a parent. If it has not been created, it is created and appended to the end of the parent's nodes. Each identifier's node is created, from left to right.

    - **"INSERT_ALL" or INSERT_ALL:** Creates this node as a child of its parent, immediately after its previous instance, and then creates its children, recursively. The node must have a parent. If it has not been created, it is created and appended to the end of the parent's nodes. Each identifier's node is created, from left to right.

    - **"INSERT_UNUSED" or INSERT_UNUSED:** Creates this node as a child of its parent, immediately after its previous instance, if the identifier's node does not have a value or any attributes. The node must have a parent. If it has not been created, it is created and appended to the end of the parent's nodes. If it was created and has a value or attribute, a new node is created immediately after it. If it does not have a value or attribute, the identifier is unchanged. Each identifier's node is created, from left to right.

    - **"INSERT_UNUSED_ALL" or INSERT_UNUSED_ALL:** Same as INSERT_UNUSED, except that if a new node is created, all its children are also created, recursively.

    - **"DELETE" or DELETE:** Removes this node as a child of its parent, then deletes it and all its child nodes. Each identifier and all its children are cleared. Identifiers are deleted from left to right. If the node does not have a parent, it is assumed to be the root node. Its document and all related nodes are deleted. **This operation should not be**

used if the node or one of its children has been assigned to more than one identifier.

- \<identifier\> one or more identifiers that are XML elements (as defined by the **XML_ELEMENT Statement** on page B-6).

To Create

The XML_OP Statement can only be created from the Rules Language Text Editor. See **The Rules Language Text Editor** on page 2-11 of the *Oracle Utilities Rules Language User's Guide* for more information.

Example

Create an XML node called LS_IMPORT and insert a child node called LS_RECORD as a child of LS_IMPORT.

```
/* Set the root element */
IDENTIFIER LS_IMPORT;
/* Declare the child tree */
XML_ELEMENT LS_IMPORT NODENAME "LS_IMPORT";
XML_ELEMENT CUST_DATA NODENAME "CUSTOMER_DATA" PARENT LS_IMPORT;
XML_ELEMENT REC_GROUP NODENAME "RECORD_GROUP_TRANSACTION" PARENT
CUST_DATA;
/* Declare a record */
XML_ELEMENT LS_RECORD NODENAME "LODESTAR_RECORD" PARENT REC_GROUP;

/* Create the LS_IMPORT XML node */
XML_OP CREATE LS_IMPORT;
/* Insert the child LS_RECORD node */
XML_OP INSERT LS_RECORD;
```

XML/Document Object Management Functions

The functions in this section manipulate an XML string, using Document Object Management (DOM) functions.

Like all functions, you must assign the results of these functions to an identifier using an Assignment Statement. The format is:

<identifier> = FUNCTION(<parameters>);

Where:

- <identifier> is a temporary, determinant, or interval data identifier. The function description below indicates what each returns: a scalar numeric (should be assigned to a temporary identifier), historical values (should be assigned to a determinant identifier), or an interval data reference (should be assigned to an interval data identifier).

- FUNCTION is one of the functions described below.

- <parameters> are one or more expressions, identifiers, or constants, as described in each function listed below.

DOMDOCCREATE Function

Creates an XML document with a root element node.

This function creates an XML document with a specified root element node. Currently, any errors are fatal. Returns an XML document.

Format:

```
<identifier> = DOMDOCCREATE(<identifier|string expression>);
```

Where:

- <identifier|string expression> is either an identifier or a string expression that evaluates to a string that will be the root element name of the document.

Example:

Create an XML document with a root element name of LS_IMPORT.

LS_IMP_DOC = DOMDOCCREATE(LS_IMPORT);

DOMDOCLOADFILE Function

Loads and parses an XML file.

This function loads and parses an XML file, and returns the XML document contained in the file. Currently, any errors are fatal. Returns an XML document.

Format:

```
<identifier> = DOMDOCLOADFILE(<identifier|string expression>);
```

Where:

- <identifier|string expression> is either an identifier or a string expression that evaluates to a string that is name of a file containing XML. The default location of the file is the C:\LODESTAR\User directory, but a full path can be specified.

Example:

Load an XML file named LS_IMPORT.XML.

```
LS_IMP_FILE = DOMDOCLOADFILE("LS_IMPORT.XML");
```

DOMDOCLOADXML Function

Loads and parses an XML document.

This function loads and parses an XML document. Currently, any errors are fatal. Returns an XML document.

Format:

```
<identifier> = DOMDOCLOADXML(<identifier|string expression>);
```

Where:

- <identifier|string expression> is either an identifier or a string expression that evaluates to a string that is name of an XML document.

Example:

Load an XML document named LS_IMPORT.

LS_IMP_DOC = DOMDOCLOADXML("LS_IMPORT");

DOMDOCSAVEFILE Function

Saves an XML file based on a specified XML document.

This function creates an XML file based on a specified XML document. The document is written out as XML to the specified file, replacing its contents. Returns the integer 0.

Format:

```
<identifier> = DOMDOCSAVEFILE(<xml_document_identifier>,
<identifier|string expression>);
```

Where:

- <xml_document_identifier> is an XML document identifier.

- <identifier|string expression> is either an identifier or a string expression that evaluates to a string that is the name of a file that will contain XML. The default location of the file is the C:\LODESTAR\User directory, but a full path can be specified.

Example:

Save an XML document called LS_IMPORT to a file called LS_IMPORT.XML.

```
LS_IMP_SAVE = DOMDOCSAVEFILE(LS_IMPORT, "LS_IMPORT.XML");
```

DOMDOCGETROOT Function

Retrieves the root node of an XML document.

This function retrieves the root node of a specified XML document. Returns an XML node.

Format:

```
<identifier> = DOMDOCGETROOT(<xml_document_identifier>);
```

Where:

- <xml_document_identifier> is an identifier that is an XML document.

Example:

Get the root node of the LS_DATA XML document.

```
LS_IMP_ROOT = DOMDOCGETROOT(LS_DATA);
```

DOMDOCADDPI Function

Adds a processing instruction to an XML document.

This function adds a processing instruction to an XML document. Returns an XML node.

> **Note**: If using this function inside a FOR EACH in LIST statement, include the USE_DOMDOCADDAPI_IN_LOOP = 1 keyword and value in the LODESTAR.CFG file.

Format:

```
<identifier> = DOMDOCADDPI(<identifier|string expression>);
```

Where:

- <xml_node_identifier> is the root element of the XML document. The 'PI' is inserted before it.

- <node_name> is a valid XML node name.

- <node_value> is a literal or string value in the form "attribute=""value"" attribute=""value"" ..." (the double double-quotes will become single double-quotes).

Example:

Add a reference to an XSL style sheet

```
SS_OUT = DOMDOCADDPI(CUST_BILL, "xml-stylesheet", "type='text/xsl'
href='PSNH_BillReport.xsl'");
```

DOMNODEGETNAME Function

Retrieves the name of an XML node.

This function retrieves the name of an XML node. Returns a string.

Format:

```
<identifier> = DOMNODEGETNAME(<xml_node_identifier>);
```

Where:

- <xml_node_identifier> is an identifier that is an XML node.

Example:

Get the name of the LS_RECORD node.

LS_RECORD_NAME = DOMNODEGETNAME(LS_RECORD);

DOMNODEGETTYPE Function

Retrieves the type of an XML node.

This function retrieves the type of an XML node. Types may be "attribute", "element", "comment", "text", ... Returns a string.

Format:

```
<identifier> = DOMNODEGETTYPE(<xml_node_identifier>);
```

Where:

- <xml_node_identifier> is an identifier that is an XML node.

Example:

Get the node type for the LS_RECORD node.

LS_RECORD_NODE_TYPE = DOMNODEGETTYPE(LS_RECORD);

DOMNODEGETVALUE Function

Retrieves the value of an XML node.

This function retrieves the value of an XML node. Returns a string.

Format:

```
<identifier> = DOMNODEGETVALUE(<xml_node_identifier>);
```

Where:

-   <xml_node_identifier> is an identifier that is an XML node.

Example:

Get the value of the LS_RECORD node.

LS_RECORD_VAL = DOMNODEGETVALUE(LS_RECORD);

DOMNODEGETCHILDCT Function

Retrieves the number of child nodes of an XML node.

This function retrieves the number of child nodes of an XML node (may be 0). Returns an integer.

Format:

```
<identifier> = DOMNODEGETCHILDCT(<xml_node_identifier>);
```

Where:

- <xml_node_identifier> is an identifier that is an XML node.

Example:

Get the number of child nodes in the LS_RECORD node.

LS_RECORD_NUM_CHILDREN = DOMNODEGETCHILDCT(LS_RECORD);

DOMNODEGETFIRSTCHILD Function

Retrieves the first child of an XML node, if any.

This function retrieves the first child of an XML node, if any. If there are no child nodes, returns 0. Returns an XML node.

Format:

```
<identifier> = DOMNODEGETFIRSTCHILD(<xml_node_identifier>);
```

Where:

* <xml_node_identifier> is an identifier that is an XML node.

Example:

Get the first child node of the LS_RECORD node.

```
LS_RECORD_FIRSTCHILD = DOMNODEGETFIRSTCHILD(LS_RECORD);
```

DOMNODEGETSIBLING Function

Retrieves the next (right side) child of an XML node, if any.

This function retrieves the next child of an XML node, if any. If there is not another child, returns 0. Returns an XML node.

Format:

```
<identifier> = DOMNODEGETSIBLING(<xml_node_identifier>);
```

Where:

- <xml_node_identifier> is an identifier that is an XML node.

Example:

Get the next child of the LS_RECORD node.

LS_RECORD_NEXTCHILD = DOMNODEGETSIBLING(LS_RECORD);

DOMNODECREATECHILDELEMENT Function

Creates a child node in an XML node.

This function creates a child node in a specified XML node. The new element is appended as the last child node of the specified node. Currently, any errors are fatal. Returns an XML node that is the new element.

Format:

```
<identifier> = DOMNODECREATECHILDELEMENT(<xml_node_identifier>,
<identifier|string expression>);
```

Where:

- <xml_node_identifier> is an identifier that is an XML node.

- <identifier|string expression> is either an identifier or a string expression that evaluates to a string that is the name of the new element.

Example:

Add a new child node called ARRANGEMENT to the LS_ACCOUNT node.

```
LS_RECORD_NEW_NODE = DOMNODECREATECHILDELEMENT(LS_ACCOUNT,
"ARRANGMENT");
```

DOMNODESETATTRIBUTE Function

Sets the value of an attribute of an XML node.

This function sets the value of an attribute of a specified XML node. The attribute value is added to the element if the attribute does not exist; otherwise, it replaces the attribute's value. Currently, any errors are fatal. Returns the integer 0.

Format:

```
<identifier> = DOMNODESETATTRIBUTE(<xml_node_identifier>,
<identifier|string expression>, <identifier|string expression>);
```

Where:

- <xml_node_identifier> is an an XML node that is an element.

- <identifier|string expression> is either an identifier or a string expression that evaluates to a string that is the name of the attribute.

- <identifier|string expression> is either an identifier or a string expression that evaluates to a string that is the value of the attribute.

Example:

Set the value of the "Arrangement" attribute of the LS_ACCOUNT node to TRUE.

```
LS_RECORD_NEW_NODE = DOMNODESETATTRIBUTE(LS_ACCOUNT, "ARRANGEMENT",
"TRUE");
```

DOMNODEGETCHILDELEMENTCT Function

Retrieves the number of child nodes of an XML node that are elements.

This function retrieves the number of child nodes of an XML node that are elements; this may be 0. Returns an integer.

Format:

```
<identifier> = DOMNODEGETCHILDELEMENTCT(<xml_node_identifier>);
```

Where:

- <xml_node_identifier> is an identifier that is an XML node.

Example:

Get the number of child element nodes in the LS_RECORD node.

LS_RECORD_NUM_ELEM_CHILDREN = DOMNODEGETCHILDELEMENTCT(LS_RECORD);

DOMNODEGETFIRSTCHILDELEMENT Function

Retrieves the first child of an XML node that is an element, if any.

This function retrieves the first child of an XML node that is an element, if any. If there are no child nodes, returns 0. Returns an XML node.

Format:

```
<identifier> = DOMNODEGETFIRSTCHILDELEMENT(<xml_node_identifier>);
```

Where:

- <xml_node_identifier> is an identifier that is an XML node.

Example:

Get the first child element node of the LS_RECORD node.

LS_RECORD_FIRSTCHILD_ELEM = DOMNODEGETFIRSTCHILDELEMENT(LS_RECORD);

DOMNODEGETSIBLINGELEMENT Function

Retrieves the next (right side) child of an XML node that is an element, if any.

This function retrieves the next child of an XML node that is an element, if any. If there is not another child, returns 0. Returns an XML node.

Format:

```
<identifier> = DOMNODEGETSIBLINGELEMENT(<xml_node_identifier>);
```

Where:

- <xml_node_identifier> is an identifier that is an XML node.

Example:

Get the next child element node of the LS_RECORD node.

```
LS_RECORD_NEXTCHILD = DOMNODEGETSIBLINGELEMENT(LS_RECORD);
```

DOMNODEGETATTRIBUTECT Function

Retrieves the number of attribute nodes of an XML node, if any.

This function retrieves the number of attribute nodes of an XML node; this may be 0. If the node is not an attribute, returns 0. Returns an integer.

Format:

```
<identifier> = DOMNODEGETATTRIBUTECT(<xml_node_identifier>);
```

Where:

- <xml_node_identifier> is an identifier that is an XML node.

Example:

Get the number of attribute nodes in the LS_RECORD node.

```
LS_RECORD_ATT_NODES = DOMNODEGETATTRIBUTECT(LS_RECORD);
```

DOMNODEGETATTRIBUTEI Function

Retrieves the index'th attribute an XML node, if any.

This function retrieves the index'th attribute of a specified XML node, if any. If there is no such attribute, returns a NULL node. Returns an XML node.

Format:

```
<identifier> = DOMNODEGETATTRIBUTECT(<xml_node_identifier>, <index>);
```

Where:

- <xml_node_identifier> is an identifier that is an XML node.

- <index> an integer between 1 and the number of attributes in the node, inclusive.

Example:

Get the 4th attribute node in the LS_RECORD node.

```
LS_RECORD_ATTI_NODE_4 = DOMNODEGETATTRIBUTEI(LS_RECORD, 4);
```

DOMNODEGETATTRIBUTEBYNAME Function

Retrieves the attribute of an XML node with a specified name, if any.

This function retrieves the attribute of a specified XML node with this name, if any. If there is no such attribute, returns a NULL node. Returns an XML node.

Format:

```
<identifier> = DOMNODEGETATTRIBUTEBYNAME(<xml_node_identifier>,
<name>);
```

Where:

- <xml_node_identifier> is an identifier that is an XML node.

- <name> is a string that is the name of an attribute in the XML node.

Example:

Get the attribute in the LS_ACCOUNT node with the name ARRANGEMENT.

**LS_RECORD_ATT_ARRANGE = DOMNODEGETATTRIBUTEBYNAME(LS_RECORD,
"ARRANGEMENT");**

DOMNODEGETBYNAME Function

Retrieves the first node under a specified XML node with a specified name, if any.

This function retrieves the first node under the specified node with the specified name. If there is no such node, returns 0. Returns an XML node.

Format:

```
<identifier> = DOMNODEGETBYNAME(<xml_node_identifier>, <name>);
```

Where:

- <xml_node_identifier> is an identifier that is an XML node.

- <name> a string that is the name of an XML node.

Example:

Get the first node in the LS_ACCOUNT node with the name ARRANGEMENT.

LS_RECORD_FIRST_ARRANGE = DOMNODEGETBYNAME(LS_RECORD, "ARRANGEMENT");

Using the XML Statements and Functions

The XML statements and functions described in this appendix allow you to obtain data values from XML documents and files and assign those values to identifiers. These identifiers can be used in Rules Language processing, and the results can be saved back to the XML structure for use as output data.

You can also create XML documents and files and populate the nodes within documents and files with appropriate data and values.

This section provides step-by-step descriptions for these operations.

Reading from XML Documents and Files

The steps for reading data from existing XML documents and files are:

1. Define the XML structure.

    Defining the XML structure of the XML document or file defines the relationship between the elements and nodes in the XML document or file. To do this, use the **Identifier Statement** on page B-4 and the **XML_ELEMENT Statement** on page B-6 respectively.

2. Load the XML file or document.

    Loading the XML document or file creates an XML document, and allows the Rules Language to access the root element.

3. Get the root element of the XML document.

    Getting the root element of the XML document enables the Rules Language to access the XML elements and nodes. To do this, use the **DOMDOCGETROOT Function** on page B-17. The identifier assigned to the result of the DOMDOCGETROOT function **must** be the root element of the XML document.

4. Derive values from the XML, as specified in the XML structure.

    Deriving the data values from the XML document is done using either Stem.Tail identifiers or the DOM functions.

Example

The following example shows how data can be extracted from an XML structure. In this example, the XML structure is the context of an activity performed using the Oracle Utilities Billing Component - Workflow Management product. With the following activity context:

```
<CONTEXT>
    <ACCOUNTID>123</ACCOUNTID>
    <PASTDUEAMT>90.00</PASTDUEAMT>
    <OTHER />
</CONTEXT>
```

the following Rules Language statements could be used to extract data from the context.

```
/* Define the Context Structure */
IDENTIFIER CONTEXT;
XML_ELEMENT CONTEXT_ID NODENAME "CONTEXT"
XML_ELEMENT ACCOUNT_ID NODENAME "ACCOUNTID" PARENT CONTEXT
XML_ELEMENT PASTDUE_AMT NODENAME "PASTDUEAMT" PARENT CONTEXT
XML_ELEMENT OTHER_ID NODENAME "OTHER" PARENT CONTEXT
/* Load the XML document */
CONTEXT_DOC = DOMDOCLOADXML (RATE_SCHEDULE_INPUT_XML);
/* Obtain the Root Element */
CONTEXT = DOMDOCGETROOT (CONTEXT_DOC)
/* Get the Account ID */
ACCT_ID = ACCOUNT_ID.NODEVALUE;
/* Get the Past Due Amount */
PAST_DUE = PASTDUE_AMT.NODEVALUE;
```

Creating XML Documents and Files

The steps to create an XML document or file are:

1. Define the XML structure.

    Defining the XML structure of the XML document or file defines the relationship between the elements and nodes in the XML document or file. To do this, use the **Identifier Statement** on page B-4 and the **XML_ELEMENT Statement** on page B-6 respectively.

2. Create the XML document.

    Creating an XML document is performed using the CREATE operation of the **XML_OP Statement** on page B-9. The XML element identifier created via the XML_OP statement **must** be the root element of the XML document.

3. Create the nodes in the XML document, as specified in the XML structure.

    Creating the nodes within the XML document is performed using either the CREATE or INSERT operations of the **XML_OP Statement** on page B-9. The XML element identifier created via the XML_OP Statement **must** be the node names of the XML document.

4. Set node values using Stem.Tail identifiers.

    Setting the node values in the XML document can be done using either Stem.Tail identifiers or the DOM functions.

5. *Optional*. Save the XML document to a file.

    Use the **DOMDOCSAVEFILE Function** on page B-16 to save the XML document to a file.

Example

The following example shows how an XML structure can be created using the Rules Language. For this example, the XML structure is the context of an activity performed using the Oracle Utilities Billing Component - Workflow Management product.

```
/* Define the Context Structure */
```

```
                    IDENTIFIER CONTEXT;
                    XML_ELEMENT CONTEXT_ID NODENAME "CONTEXT"
                    XML_ELEMENT ACCOUNT_ID NODENAME "ACCOUNTID" PARENT CONTEXT;
                    XML_ELEMENT PASTDUE_AMT NODENAME "PASTDUEAMT" PARENT CONTEXT;
                    XML_ELEMENT OTHER_ID NODENAME "OTHER" PARENT CONTEXT;
                    /* Create the document */
                    XML_OP CREATE CONTEXT_ID;
                    /* Create the document nodes and assign values */
                    XML_OP INSERT ACCOUNT_ID;
                    ACCOUNT_ID.NODEVALUE = "123";
                    XML_OP INSERT PASTDUE_AMT;
                    PASTDUE_AMT.NODEVALUE = "90.00";
                    XML_OP INSERT OTHER_ID;
                    OTHER_ID.ARRANGEMENT = "TRUE";
```

The resulting XML structure would be:

```
<CONTEXT>
    <ACCOUNTID>123</ACCOUNTID>
    <PASTDUEAMT>90.00</PASTDUEAMT>
    <OTHER ARRANGEMENT='TRUE'/>
</CONTEXT>
```

The following statement could be used to save the XML to a file.

```
CONTEXT_FILE = DOMDOCSAVEFILE (CONTEXT, "CONTEXT.XML");
```

Index

W

X