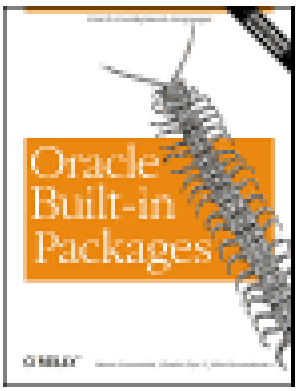

Oracle Built-in Packages

By Steven Feuerstein, Charles Dye & John Beresniewicz
1st Edition April 1998
1-56592-375-8, Order Number: 3758
956 pages, $49.95, Includes diskette

Chapter 6.
Generating Output from PL/SQL Programs

In this chapter:

The built-in packages offer a number of ways to generate output from within your PL/SQL program. While updating a database table is, of course, a form of "output" from PL/SQL, this chapter shows you how to use two packages that explicitly generate output. UTL_FILE reads and writes information in server-side files, and DBMS_OUTPUT displays information to your screen.

DBMS_OUTPUT: Displaying Output

DBMS_OUTPUT provides a mechanism for displaying information from your PL/SQL program on your screen (your session's output device, to be more specific). As such, it serves as just about the only immediately accessible (meaning "free with PL/SQL") means of debugging your PL/SQL stored code.[1] It is certainly your "lowest common denominator" debugger, similar to the used-and-abused MESSAGE built-in of Oracle Forms. DBMS_OUTPUT is also the package you are most likely to use to generate reports from PL/SQL scripts run in SQL*Plus.

Of all the built-in packages, the DBMS_OUTPUT package (and its PUT_LINE procedure, in particular) is likely to be the one you will find yourself using most frequently. You may therefore find it strange that I never call DBMS_OUTPUT.PUT_LINE. I find the design and functionality of DBMS_OUTPUT to be substandard and very frustrating.

In fact, I recommend that you never use this package--at least, not directly. You should instead encapsulate calls to DBMS_OUTPUT (and the PUT_LINE procedure, in particular) inside a package of your own construction. This technique is discussed in the "DBMS_OUTPUT Examples" section later in this chapter.

Getting Started with DBMS_OUTPUT

The DBMS_OUTPUT package is created when the Oracle database is installed. The *dbmsoutp.sql* script (found in the built-in packages source code directory, as described in Chapter 1, *Introduction*) contains the source code for this package's specification. This script is called by the *catproc.sql* script, which is normally run immediately after database creation. The script creates the public synonym DBMS_OUTPUT for the package. Instance-wise access to this package is provided on installation, so no additional steps should be necessary in order to use DBMS_OUTPUT. As far as package usage is concerned, you will almost always be using only the DBMS_OUTPUT.PUT_LINE procedure and only in SQL*Plus. The section "Enabling and Disabling Output" later in this chapter shows how you set up DBMS_OUTPUT for use in SQL*Plus.

DBMS_OUTPUT programs

Table 6-1 shows the DBMS_OUTPUT program names and descriptions.

Table 6-1: DBMS_OUTPUT Programs

Name	Description	Use in SQL?
DISABLE	Disables output from the package; the DBMS_OUTPUT buffer will not be flushed to the screen	Yes
ENABLE	Enables output from the package	Yes
GET_LINE	Gets a single line from the buffer	Yes
GET_LINES	Gets specified number of lines from the buffer and passes them into a PL/SQL table	Yes
NEW_LINE	Inserts an end-of-line mark in the buffer	Yes
PUT	Puts information into the buffer	Yes
PUT_LINE	Puts information into the buffer and appends an end-of-line marker after that data	Yes

> **NOTE:** All procedures in DBMS_OUTPUT have been enabled for indirect usage in SQL (that is, they can be called by a function that is then executed in a SQL statement), but only for Oracle 7.3 and later.

DBMS_OUTPUT concepts

Each user has a DBMS_OUTPUT buffer of up to 1,000,000 bytes in size. Write information to this buffer by calling the DBMS_OUTPUT.PUT and DBMS_OUTPUT.PUT_LINE programs. If you are using DBMS_OUTPUT from within SQL*Plus, this information will be displayed automatically when your program terminates. You can (optionally) explicitly retrieve information from the buffer with calls to DBMS_OUTPUT.GET and DBMS_OUTPUT.GET_LINE.

The DBMS_OUTPUT buffer can be set to a size between 2,000 and 1,000,000 bytes with the DBMS_OUTPUT.ENABLE procedure. If you do not enable the package, no information will be displayed or be retrievable from the buffer.

The buffer stores three different types of data--VARCHAR2, NUMBER, and DATE--in their internal representations. These types match the overloading available with the PUT and PUT_LINE procedures. Note that DBMS_OUTPUT does *not* support Boolean data in either its buffer or its overloading of the PUT procedures.

The following anonymous PL/SQL block uses DBMS_OUTPUT to display the name and salary of each employee in department 10:

```
DECLARE
   CURSOR emp_cur
   IS
      SELECT ename, sal
        FROM emp
       WHERE deptno = 10
       ORDER BY sal DESC;
BEGIN
   FOR emp_rec IN emp_cur
   LOOP
      DBMS_OUTPUT.PUT_LINE
         ('Employee ' || emp_rec.ename || ' earns ' ||
          TO_CHAR (emp_rec.sal) || ' dollars.');
   END LOOP;
END;
/
```

This program generates the following output when executed in SQL*Plus:

```
Employee KING earns 5000 dollars.
Employee SCOTT earns 3000 dollars.
Employee JONES earns 2975 dollars.
Employee ADAMS earns 1100 dollars.
Employee JAMES earns 950 dollars.
```

DBMS_OUTPUT exceptions

DBMS_OUTPUT does not contain any declared exceptions. Instead, Oracle designed the package to rely on two error numbers in the -20 NNN range (usually reserved for Oracle customers). You may, therefore, encounter one of these two exceptions when using the DBMS_OUTPUT package (no names are associated with these exceptions).

The -20000 error number indicates that these package-specific exceptions were raised by a call to RAISE_APPLICATION_ERROR, which is in the DBMS_STANDARD package.

-20000
: ORU-10027: buffer overflow, limit of <buf_limit> bytes.

  If you receive the -10027 error, you should see if you can increase the size of your buffer with another call to DBMS_OUTPUT.ENABLE.

-20000
: ORU-10028: line length overflow, limit of 255 bytes per line.

  If you receive the -10028 error, you should restrict the amount of data you are passing to the buffer in a single call to PUT_LINE, or in a batch of calls to PUT followed by NEW_LINE.

You may also receive the ORA-06502 error:

ORA-06502
: Numeric or value error.

If you receive the -06502 error, you have tried to pass more than 255 bytes of data to DBMS_OUTPUT.PUT_LINE. You must break up the line into more than one string.

DBMS_OUTPUT nonprogram elements

The DBMS_OUTPUT package defines a PL/SQL table TYPE as follows:

```
TYPE chararr IS TABLE OF VARCHAR2(255) INDEX BY BINARY_INTEGER;
```

The DBMS_OUTPUT.GET_LINES procedure returns its lines in a PL/SQL table of this type.

Drawbacks of DBMS_OUTPUT

Before learning all about this package, and rushing to use it, you should be aware of several drawbacks with the implementation of this functionality:

- The "put" procedures that place information in the buffer are overloaded only for strings, dates, and numbers. You cannot request the display of Booleans or any other types of data. You cannot display combinations of data (a string and a number, for instance), without performing the conversions and concatentations yourself.
- You will see output from this package only after your program completes its execution. You *cannot* use DBMS_OUTPUT to examine the results of a program while it is running. And if your program terminates with an unhandled exception, you may not see anything at all!
- If you try to display strings longer than 255 bytes, DBMS_OUTPUT will raise a VALUE_ERROR exception.
- DBMS_OUTPUT is not a strong choice as a report generator, because it can handle a maximum of only 1,000,000 bytes of data in a session before it raises an exception.
- If you use DBMS_OUTPUT in SQL*Plus, you may find that any leading blanks are automatically truncated. Also, attempts to display blank or NULL lines are completely ignored.

There are workarounds for almost every one of these drawbacks. The solution invariably requires the construction of a package that encapsulates and hides DBMS_OUTPUT. This technique is explained in the "DBMS_OUTPUT Examples" section.

Enabling and Disabling Output

The ENABLE and DISABLE procedures enable and disable output from the DBMS_OUTPUT.PUT_LINE (and PUT and PUTF) procedure.

The DBMS_OUTPUT.ENABLE procedure

The ENABLE procedure enables calls to the other DBMS_OUTPUT modules. If you do not first call ENABLE, then any other calls to the package modules are ignored. The specification for the procedure is,

```
PROCEDURE DBMS_OUTPUT.ENABLE  (buffer_size IN INTEGER DEFAULT 20000);
```

where buffer_size is the size of the buffer that will contain the information stored by calls to PUT and

PUT_LINE. The buffer size can be as large as 1,000,000 bytes. You can pass larger values to this procedure without raising an error, but doing so will have no effect besides setting the buffer size to its maximum.

You can call ENABLE more than once in a session. The buffer size will be set to the largest size passed in any call to ENABLE. In other words, the buffer size is not necessarily set to the size specified in the last call.

If you want to make sure that the DBMS_OUTPUT package is enabled in a program you are testing, add a statement like this one to the start of the program:

```
DECLARE
   ... declarations ...
BEGIN
   DBMS_OUTPUT.ENABLE (1000000);
   ...
END;
```

The DBMS_OUTPUT.DISABLE procedure

The DISABLE procedure disables all calls to the DBMS_OUTPUT package (except for ENABLE). It also purges the buffer of any remaining lines of information. Here's the specification for the procedure:

```
PROCEDURE DBMS_OUTPUT.DISABLE;
```

SQL*Plus and SQL*DBA offer a native command, SET SERVEROUTPUT, with which you can disable the package without having to execute the DISABLE procedure directly. You can use the command as follows:

```
SQL> SET SERVEROUTPUT OFF
```

This command is equivalent to the following PL/SQL statement:

```
DBMS_OUTPUT.DISABLE;
```

After you execute this command, any calls to PUT_LINE and other modules will be ignored, and you will not see any output.

Enabling output in SQL*Plus

Most developers use DBMS_OUTPUT almost exclusively in the SQL*Plus environment. To enable output from calls to PUT_LINE in SQL*Plus, you will use the SET SERVEROUTPUT command,

```
SET SERVEROUTPUT ON SIZE 1000000
```

or:

```
SET SERVEROUTPUT ON
```

Each of these calls the DBMS_OUTPUT.ENABLE procedure.

I have found it useful to add SET SERVEROUTPUT ON SIZE 1000000 to my *login.sql* file, so that the package is automatically enabled whenever I go into SQL*Plus. (I guess that tells you how often I have to

debug my code!)

You should also check the Oracle documentation for SQL*Plus to find out about the latest set of options for the SET SERVEROUTPUT command. As of Oracle8, the documentation shows the following syntax for this SET command:

```
SET SERVEROUT[PUT] {OFF|ON}
   [SIZE n] [FOR[MAT] {WRA[PPED]| WOR[D_WRAPPED]|TRU[NCATED]}]
```

In other words, you have these options when you enable DBMS_OUTPUT in SQL*Plus:

SET SERVEROUTPUT OFF
: Turns off the display of text from DBMS_OUTPUT.

SET SERVEROUTPUT ON
: Turns on the display of text from DBMS_OUTPUT with the default 2000-byte buffer. This is a very small size for the buffer; I recommend that you always specify a size when you call this command.

SET SERVEROUTPUT ON SIZE NNNN
: Turns on the display of text from DBMS_OUTPUT with the specified buffer size (maximum of 1,000,000 bytes).

SET SERVEROUTPUT ON FORMAT WRAPPED
: (Available in Oracle 7.3 and later only.) Specifies that you want the text displayed by DBMS_OUTPUT wrapped at the SQL*Plus line length. The wrapping occurs regardless of word separation. This will also stop SQL*Plus from stripping leading blanks from your text. You can also specify a SIZE value with this variation.

SET SERVEROUTPUT ON FORMAT WORD_WRAPPED
: (Available in Oracle 7.3 and later only.) Specifies that you want the text displayed by DBMS_OUTPUT wrapped at the SQL*Plus line length. This version respects integrity of "words." As a result, lines will be broken in a way that keeps separate tokens intact. This will also stop SQL*Plus from stripping leading blanks from your text. You can also specify a SIZE value with this variation.

SET SERVEROUTPUT ON FORMAT TRUNCATED
: (Available in Oracle 7.3 and later only.) Specifies that you want the text displayed by DBMS_OUTPUT to be truncated at the SQL*Plus line length; the rest of the text will not be displayed. This will also stop SQL*Plus from stripping leading blanks from your text. You can also specify a SIZE value with this variation.

Writing to the DBMS_OUTPUT Buffer

You can write information to the buffer with calls to the PUT, NEW_LINE, and PUT_LINE procedures.

The DBMS_OUTPUT.PUT procedure

The PUT procedure puts information into the buffer, but does not append a newline marker into the buffer. Use PUT if you want to place information in the buffer (usually with more than one call to PUT), but not also automatically issue a newline marker. The specification for PUT is overloaded, so that you

can pass data in its native format to the package without having to perform conversions,

```
PROCEDURE DBMS_OUTPUT.PUT (A VARCHAR2);
PROCEDURE DBMS_OUTPUT.PUT (A NUMBER);
PROCEDURE DBMS_OUTPUT.PUT (A DATE);
```

where A is the data being passed.

Example

In the following example, three simultaneous calls to PUT place the employee name, department ID number, and hire date into a single line in the DBMS_OUTPUT buffer:

```
DBMS_OUTPUT.PUT (:employee.lname || ', ' || :employee.fname);
DBMS_OUTPUT.PUT (:employee.department_id);
DBMS_OUTPUT.PUT (:employee.hiredate);
```

If you follow these PUT calls with a NEW_LINE call, that information can then be retrieved with a single call to GET_LINE.

The DBMS_OUTPUT.PUT_LINE procedure

The PUT_LINE procedure puts information into the buffer and then appends a newline marker into the buffer. The specification for PUT_LINE is overloaded, so that you can pass data in its native format to the package without having to perform conversions:

```
PROCEDURE DBMS_OUTPUT.PUT_LINE (A VARCHAR2);
PROCEDURE DBMS_OUTPUT.PUT_LINE (A NUMBER);
PROCEDURE DBMS_OUTPUT.PUT_LINE (A DATE);
```

The PUT_LINE procedure is the one most commonly used in SQL*Plus to debug PL/SQL programs. When you use PUT_LINE in these situations, you do not need to call GET_LINE to extract the information from the buffer. Instead, SQL*Plus will automatically dump out the DBMS_OUTPUT buffer when your PL/SQL block finishes executing. (You will not see any output until the program ends.)

Of course, you can also call DBMS_OUTPUT programs directly from the SQL*Plus command prompt, and not from inside a PL/SQL block, as shown in the following example.

Example

Suppose that you execute the following three statements in SQL*Plus:

```
SQL> exec DBMS_OUTPUT.PUT ('I am');
SQL> exec DBMS_OUTPUT.PUT (' writing ');
SQL> exec DBMS_OUTPUT.PUT ('a ');
```

You will not see anything, because PUT will place the information in the buffer, but will not append the newline marker. When you issue this next PUT_LINE command,

```
SQL> exec DBMS_OUTPUT.PUT_LINE ('book!');
```

you will then see the following output:

```
I am writing a book!
```

All of the information added to the buffer with the calls to PUT waited patiently to be flushed out with the call to PUT_LINE. This is the behavior you will see when you execute individual calls at the SQL*Plus command prompt to the put programs.

If you place these same commands in a PL/SQL block,

```
BEGIN
   DBMS_OUTPUT.PUT ('I am');
   DBMS_OUTPUT.PUT (' writing ');
   DBMS_OUTPUT.PUT ('a ');
   DBMS_OUTPUT.PUT_LINE ('book');
END;
/
```

the output from this script will be exactly the same as that generated by this single call:

```
SQL> exec DBMS_OUTPUT.PUT_LINE ('I am writing a book!');
```

The DBMS_OUTPUT.NEW_LINE procedure

The NEW_LINE procedure inserts an end-of-line marker in the buffer. Use NEW_LINE after one or more calls to PUT in order to terminate those entries in the buffer with a newline marker. Here's the specification for NEW_LINE:

```
PROCEDURE DBMS_OUTPUT.NEW_LINE;
```

Retrieving Data from the DBMS_OUTPUT Buffer

You can use the GET_LINE and GET_LINES procedures to extract information from the DBMS_OUTPUT buffer. If you are using DBMS_OUTPUT from within SQL*Plus, however, you will never need to call either of these procedures. Instead, SQL*Plus will automatically extract the information and display it on the screen for you.

The DBMS_OUTPUT.GET_LINE procedure

The GET_LINE procedure retrieves one line of information from the buffer. Here's the specification for the procedure:

```
PROCEDURE DBMS_OUTPUT.GET_LINE
   (line OUT VARCHAR2,
    status OUT INTEGER);
```

The parameters are summarized in the following table.

Parameter	Description
line	Retrieved line of text
status	GET request status

The line can have up to 255 bytes in it, which is not very long. If GET_LINE completes successfully, then status is set to 0. Otherwise, GET_LINE returns a status of 1.

Notice that even though the PUT and PUT_LINE procedures allow you to place information into the buffer in their native representations (dates as dates, numbers and numbers, and so forth), GET_LINE always retrieves the information into a character string. The information returned by GET_LINE is everything in the buffer up to the next newline character. This information might be the data from a single PUT_LINE or from multiple calls to PUT.

Example

The following call to GET_LINE extracts the next line of information into a local PL/SQL variable:

```
FUNCTION get_next_line RETURN VARCHAR2
IS
   return_value VARCHAR2(255);
   get_status INTEGER;
BEGIN
   DBMS_OUTPUT.GET_LINE (return_value, get_status);
   IF get_status = 0
   THEN
      RETURN return_value;
   ELSE
      RETURN NULL;
   END IF;
END;
```

The DBMS_OUTPUT.GET_LINES procedure

The GET_LINES procedure retrieves multiple lines from the buffer with one call. It reads the buffer into a PL/SQL string table. Here's the specification for the procedure:

```
PROCEDURE DBMS_OUTPUT.GET_LINES
   (lines OUT DBMS_OUTPUT.CHARARR,
    numlines IN OUT INTEGER);
```

The parameters for this procedure are summarized in the following table.

Parameter	Description
lines	PL/SQL array where retrieved lines are placed
numlines	Number of individual lines retrieved from the buffer and placed into the array

The lines parameter is a PL/SQL table TYPE declared in the specification of the package. It is described at the beginning of this chapter.

The values retrieved by GET_LINES are placed in the first numlines rows in the table, starting from row one. As indicated in the PL/SQL table structure, each line (row in the table) may contain up to 255 bytes.

Notice that numlines is an IN OUT parameter. The IN aspect of the parameter specifies the number of lines to retrieve. Once GET_LINES is done retrieving data, however, it sets numlines to the number of lines actually placed in the table. If you ask for ten rows and there are only six in the buffer, then you need to know that only the first six rows of the table are defined.

Notice also that even though the PUT and PUT_LINE procedures allow you to place information into the buffer in their native representations (dates as dates, numbers and numbers, and so forth), GET_LINES

always retrieves the information into a character string. The information in each line returned by GET_LINES is everything in the buffer up to the next newline character. This information might be the data from a single PUT_LINE or from multiple calls to PUT.

While GET_LINES is provided with the DBMS_OUTPUT package, it is not needed to retrieve information from the DBMS_OUTPUT buffer--at least when used inside SQL*Plus. In this interactive query tool, you simply execute calls to PUT_LINE, and when the PL/SQL block terminates, SQL*Plus will automatically dump the buffer to the screen.

Example

The following script demonstrates both the kind of code you would write when using the GET_LINES procedure, and also the way in which the PL/SQL table is filled:

```
/* Filename on companion disk: getlines.tst */

DECLARE
  output_table DBMS_OUTPUT.CHARARR;  /* output_buf_tab */
  a_line VARCHAR2(10) := RPAD('*',10,'*');
  status INTEGER;
  max_lines CONSTANT NUMBER := 15;
BEGIN
   output_table (0) := 'ABC';
   output_table (12) := 'DEF';

   /* Output 10 lines */
   FOR linenum IN 1..10
   LOOP
      DBMS_OUTPUT.PUT_LINE (a_line ||  TO_CHAR (linenum);
   END LOOP;
   /* retrieve 15 lines, status will receive the line count */
   status := max_lines;
   DBMS_OUTPUT.GET_LINES ( output_table, status);
   DBMS_OUTPUT.PUT_LINE ('lines retrieved= ' || status));

   FOR linenum in 0..max_lines
   LOOP
       BEGIN
          DBMS_OUTPUT.PUT_LINE
             (linenum || ':' || NVL (output_table(linenum),'<null>') );
       EXCEPTION
          WHEN OTHERS
          THEN
             DBMS_OUTPUT.PUT_LINE (linenum || ':' || sqlerrm );
       END;
    END LOOP;
EXCEPTION
   WHEN OTHERS
   THEN
       DBMS_OUTPUT.PUT_LINE ('Exception, status=' || status);
       DBMS_OUTPUT.PUT_LINE (SQLERRM );
END;
/
```

Here is the output from the execution of this script:

```
lines retrieved= 10
0:ORA-01403: no data found
```

```
1:**********1
2:**********2
3:**********3
4:**********4
5:**********5
6:**********6
7:**********7
8:**********8
9:**********9
10:**********10
11:<null>
12:ORA-01403: no data found
13:ORA-01403: no data found
14:ORA-01403: no data found
15:ORA-01403: no data found
```

You can therefore deduce the following rules:

1. The PL/SQL table is filled starting with row 1.

2. If DBMS_OUTPUT.GET_LINES finds N lines of data to pass to the PL/SQL table, it sets row N+1 in that table to NULL.

3. All other rows in the PL/SQL table are set to "undefined." In other words, any other rows that might have been defined before the call to GET_LINES are deleted.

Tips on Using DBMS_OUTPUT

As noted at the beginning of the chapter, DBMS_OUTPUT comes with several handicaps. The best way to overcome these handicaps is to create your own layer of code over the built-in package. This technique is explored in the "DBMS_OUTPUT Examples" section.

Regardless of the use of an encapsulation package, you should keep the following complications in mind as you work with DBMS_OUTPUT:

1. If your program raises an unhandled exception, you may not see any executed output from PUT_LINE, even if you enabled the package for output **.**

   This can happen because the DBMS_OUTPUT buffer will not be flushed until it is full or until the current PL/SQL block completes its execution. If a raised exception never gets handled, the buffer will not be flushed. As a result, calls to the DBMS_OUTPUT.PUT_LINE module might never show their data. So if you are working with DBMS_OUTPUT.PUT_LINE and are frustrated because you are not seeing the output you would expect, make sure that you have:

   a. Enabled output from the package by calling SET SERVEROUTPUT ON in SQL*Plus.

   b. Placed an exception section with a WHEN OTHERS handler in the outer block of your code (usually some sort of test script) so that your output can be flushed to your terminal by SQL*Plus.

2. When package state has been reinitialized in your session, DBMS_OUTPUT is reset to "not enabled."

Packages can be reset to their initial state with a call to DBMS_SESSION.RESET_PACKAGE. (See Chapter 11, *Managing Session Information*, for more information about this program.) You might call this procedure yourself, but that is unlikely. A more common scenario for resetting package states is when an error is raised in your session that *causes* packages to be reset to their initial state. Here is the error for which you need to beware:

ERROR at line 1:

```
ORA-04068: existing state of packages has been discarded
ORA-04061: existing state of package "PKG.PROC" has been invalidated
ORA-04065: not executed, altered or dropped package "PKG.PROC"
ORA-06508: PL/SQL: could not find program unit being called
```

If you get this error and simply continue with your testing, you may be surprised to find that you are not getting any output. If you remember that DBMS_OUTPUT relies on package variables for its settings, this makes perfect sense. So when you get the preceding error, you should immediately "re-enable" DBMS_OUTPUT with a command such as the following:

```
SQL> set serveroutput on size 1000000 format wrapped
```

I usually just re-execute my *login.sql* script, since I may be initializing several different packages:

```
SQL> @login.sql
```

When will you get this error? I have found that it occurs when I have multiple sessions connected to Oracle. Suppose that I am testing program A in session USER1. I run it and find a bug. I fix the bug and recompile program A in session USER2 (the owner of the code). When I try to execute program A from session USER1 again, it raises the ORA-04068 error.

If you do encounter this error, don't panic. Just reset your package variables and run the program again. It will now work fine; the error is simply the result of a quirk in Oracle's automatic recompilation feature.

DBMS_OUTPUT Examples

This section contains several longer examples of DBMS_OUTPUT operations.

Encapsulating DBMS_OUTPUT

Sure, it was nice of Oracle Corporation to give us the DBMS_OUTPUT package. Without it, as users of PL/SQL 1.0 found, we are running blind when we execute our code. As is the case with many of the developer-oriented utilities from Oracle, however, the DBMS_OUTPUT package is not a polished and well-planned tool. It offers nothing more than the most basic functionality, and even then it is crippled in some important ways. When I started to use it in real life (or whatever you might call the rarified atmosphere of authoring a book on software development), I found DBMS_OUTPUT.PUT_LINE to be cumbersome and limiting in ways.

I hated having to type "DBMS_OUTPUT.PUT_LINE" whenever I simply wanted to display some information. That's a mouthful and a keyboardful. I felt insulted that they hadn't even taken the time to overload for Booleans, requiring me to write silly IF logic just to see the value of a Boolean variable or function. I also found myself growing incensed that DBMS_OUTPUT would actually raise a VALUE_ERROR exception if I tried to pass it a string with more than 255 characters. I had enough errors in my code without having to worry about DBMS_OUTPUT adding to my troubles.

I decided that all this anger and frustration was not good for me. I needed to move past this nonconstructive lashing out at Oracle. I needed, in short, to *fix* my problem. So I did--with a package of my own. I am not going to provide a comprehensive explanation of my replacement package, but you can read about it (there are actually two of them) in my other books as follows:

Oracle PL/SQL Programming
: The Companion Disk section on "Package Examples" introduces you to the do package, which contains the do.pl procedure, a substitute for DBMS_OUTPUT.PUT_LINE. The *do.sps* and *do.spb* files in the book you are reading also contain the source code for this package.

Advanced Oracle PL/SQL Programming with Packages
: Chapter 7, *p: A Powerful Substitute for DMBS_OUTPUT*, presents the p package and the p.l procedure (I told you I didn't like typing those long program names!), a component of the PL/Vision library.[2]

The following section shows you the basic elements involved in constructing an encapsulation around DBMS_OUTPUT.PUT_LINE, which compensates for many of its problems. You can pursue building one of these for yourself, but I would strongly suggest that you check out the PL/Vision p package. That will leave you more time to build your own application-specific code.

Package specification for a DBMS_OUTPUT encapsulator

The absolute minimum you need for such an encapsulator package is an overloading of the "print" procedure for dates, strings, and numbers. Let's at least add Booleans to the mix in this prototype:

```
/* Filename on companion disk: prt.spp */

CREATE OR REPLACE PACKAGE prt
IS
   c_prefix CONSTANT CHAR(1) := '*';
   c_linelen CONSTANT INTEGER := 80;

   PROCEDURE ln (val IN VARCHAR2);
   PROCEDURE ln (val IN DATE);
   PROCEDURE ln (val IN NUMBER);
   PROCEDURE ln (val IN BOOLEAN);
END;
/
```

The prefix constant is concatenated to the beginning of any string to be displayed to avoid the problem of truncated spaces and ignored lines in SQL*Plus. The line length constant is used when the string is longer than 255 bytes. Finally, each of the prt.ln procedures prints a different type of data.

A complete implementation of this package would allow you to change the line length and the prefix, specify a date format for conversion, and so on. Again, check out the p package of PL/Vision for such a package.

Here is the body of the prt package:

```
/* Filename on companion disk: prt.spp */

CREATE OR REPLACE PACKAGE BODY prt
IS
   PROCEDURE ln (val IN VARCHAR2)
```

```
   IS
   BEGIN
      IF LENGTH (val) > 255
      THEN
         PLVprs.display_wrap (val, c_linelen);
      ELSE
         DBMS_OUTPUT.PUT_LINE (c_prefix || val);
      END IF;
   EXCEPTION
      WHEN OTHERS
      THEN
         DBMS_OUTPUT.ENABLE (1000000);
         DBMS_OUTPUT.PUT_LINE (c_prefix || val);
   END;
   PROCEDURE ln (val IN DATE)
   IS
   BEGIN
      ln (TO_CHAR (val));
   END;
   PROCEDURE ln (val IN NUMBER)
   IS
   BEGIN
      ln (TO_CHAR (val));
   END;
   PROCEDURE ln (val IN BOOLEAN)
   IS
   BEGIN
      IF val
      THEN
         ln ('TRUE');
      ELSIF NOT val
      THEN
         ln ('FALSE');
      ELSE
         ln ('NULL BOOLEAN');
      END IF;
   END;
END;
/
```

Here are a few things to notice about the package implementation:

- The string version of prt.ln is the "core" print procedure. The other three programs all call that one, after they have formatted the string appropriately.
- The Boolean version of prt.ln simply performs the same IF logic you would have to write if you were using DBMS_OUTPUT. By hiding it inside the prt procedure, though, nobody else has to write that kind of code again. Plus, it handles NULL values.
- The string version of prt.ln contains all the complex logic. For long strings, it relies on the PL/Vision display wrap procedure of the PLVprs package.[3] For strings with fewer than 256 characters, it calls DBMS_OUTPUT.PUT_LINE.
- As an added feature, if the attempt to display using DBMS_OUTPUT.PUT_LINE raises an exception, prt.ln assumes that the problem might be that the buffer is too small. So it increases the buffer to the maximum possible value and then tries again. I believe that it is very important for developers to make the extra effort to increase the usefulness of our code.

The prt package should give you a solid idea about the way to encapsulate a built-in package inside a package of your own construction.

UTL_FILE: Reading and Writing Server-side Files

UTL_FILE is a package that has been welcomed warmly by PL/SQL developers. It allows PL/SQL programs to both read from and write to any operating system files that are accessible from the server on which your database instance is running. File I/O was a feature long desired in PL/SQL, but available only with PL/SQL Release 2.3 and later (Oracle 7.3 or Oracle 8.0). You can now read *ini* files and interact with the operating system a *little* more easily than has been possible in the past. You can load data from files directly into database tables while applying the full power and flexibility of PL/SQL programming. You can generate reports directly from within PL/SQL without worrying about the maximum buffer restrictions of DBMS_OUTPUT

Getting Started with UTL_FILE

The UTL_FILE package is created when the Oracle database is installed. The utlfile.sql script (found in the built-in packages source code directory, as described in Chapter 1) contains the source code for this package's specification. This script is called by *catproc.sql*, which is normally run immediately after database creation. The script creates the public synonym UTL_FILE for the package and grants EXECUTE privilege on the package to public. All Oracle users can reference and make use of this package.

UTL_FILE programs

Table 6-2 shows the UTL_FILE program names and descriptions.

Table 6-2: UTL_FILE Programs

Name	Description	Use in SQL
FCLOSE	Closes the speci?ed ?les	No
FCLOSE_ALL	Closes all open ?les	No
FFLUSH	Flushes all the data from the UTL_FILE buffer	No
FOPEN	Opens the speci?ed ?le	No
GET_LINE	Gets the next line from the ?le	No
IS_OPEN	Returns TRUE if the ?le is already open	No
NEW_LINE	Inserts a newline mark in the ?le at the end of the current line	No
PUT	Puts text into the buffer	No
PUT_LINE	Puts a line of text into the ?le	No
PUTF	Puts formatted text into the buffer	No

Trying out UTL_FILE

Just getting to the point where your first call to UTL_FILE's FOPEN function works can actually be a pretty frustrating experience. Here's how it usually goes.

You read about UTL_FILE and you are excited. So you dash headlong into writing some code like this,

```
DECLARE
   config_file UTL_FILE.FILE_TYPE;
BEGIN
   config_file := UTL_FILE.FOPEN ('/tmp', 'newdata.txt', 'W');

   ... lots of write operations ...

   ... and no exception section ...
END;
/
```

and then this is all you get from your "quick and dirty script" in SQL*Plus:

```
SQL> @writefile.sql
DECLARE
*
ERROR at line 1:
ORA-06510: PL/SQL: unhandled user-defined exception
ORA-06512: at "SYS.UTL_FILE", line 91
ORA-06512: at "SYS.UTL_FILE", line 146
ORA-06512: at line 4
```

What is going wrong? This error message certainly provides little or no useful information. So you go back to the documentation, thoroughly chastened, and (over time) discover the following:

- You need to modify the INIT.ORA parameter initialization file of your instance. You will have to contact your database administrator and have him or her make the changes (if willing) and then "bounce" the database.
- You need to get the format of the parameter entries correct. That alone used to take me days!
- You need to add exception sections to your programs to give yourself a fighting chance at figuring out what is going on.

I hope that the information in this chapter will help you avoid most, if not all, of these frustrations and gotchas. But don't give up! This package is well worth the effort.

File security

UTL_FILE lets you read and write files accessible from the server on which your database is running. So you could theoretically use UTL_FILE to write right over your tablespace data files, control files, and so on. That is of course a very bad idea. Server security requires the ability to place restrictions on where you can read and write your files.

UTL_FILE implements this security by limiting access to files that reside in one of the directories specified in the INIT.ORA file for the database instance on which UTL_FILE is running.

When you call FOPEN to open a file, you must specify both the location and the name of the file, in separate arguments. This file location is then checked against the list of accessible directories.

Here's the format of the parameter for file access in the INIT.ORA file:

```
utl_file_dir = <directory>
```

Include a parameter for utl_file_dir for each directory you want to make accessible for UTL_FILE operations. The following entries, for example, enable four different directories in UNIX:

```
utl_file_dir = /tmp
utl_file_dir = /ora_apps/hr/time_reporting
utl_file_dir = /ora_apps/hr/time_reporting/log
utl_file_dir = /users/test_area
```

To bypass server security and allow read/write access to all directories, you can use this special syntax:

```
utl_file_dir = *
```

You should not use this option on production systems. In a development system, this entry certainly makes it easier for developers to get up and running on UTL_FILE and test their code. However, you should allow access to only a few specific directories when you move the application to production.

Some observations on working with and setting up accessible directories with UTL_FILE follow:

- Access is not recursive through subdirectories. If the following lines were in your INIT.ORA file, for example,

  ```
  utl_file_dir = c:\group\dev1 utl_file_dir = c:\group\prod\oe utl_file_dir =
  c:\group\prod\ar
  ```

  then you would not be able to open a file in the *c:\group\prod\oe\reports* subdirectory.

- Do not include the following entry in UNIX systems:

  ```
  utl_file_dir = .
  ```

  This would allow you to read/write on the current directory in the operating system.

- Do not enclose the directory names within single or double quotes.

- In the UNIX environment, a file created by FOPEN has as its owner the shadow process running the Oracle instance. This is usually the "oracle" owner. If you try to access these files outside of UTL_FILE, you will need the correct privileges (or be logged in as "oracle") to access or change these files.

- You should not end your directory name with a delimiter, such as the forward slash in UNIX. The following specification of a directory will result in problems when trying to read from or write to the directory:

  ```
  utl_file_dir = /tmp/orafiles/
  ```

Specifying file locations

The location of the file is an operating system-specific string that specifies the directory or area in which to open the file. The location you provide must have been listed as an accessible directory in the INIT.ORA file for the database instance.

The INIT.ORA location is a valid directory or area specification, as shown in these examples:

- In Windows NT:
  `'k:\common\debug'`
- In UNIX:
  `'/usr/od2000/admin'`

Notice that in Windows NT, the backslash character (\) is used as a delimiter. In UNIX, the forward slash (/) is the delimiter. When you pass the location in the call to UTL_FILE.FOPEN, you provide the location specification as it appears in the INIT.ORA file (unless you just provided * for all directories in the initialization file). And remember that in case-sensitive operating systems, the case of the location specification in the initialization file must match that used in the call to UTL_FILE.FOPEN.

Here are some examples:

- In Windows NT:
```
file_id := UTL_FILE.FOPEN ('k:\common\debug', 'trace.lis', 'R');
```
- In UNIX:
```
file_id := UTL_FILE.FOPEN ('/usr/od2000/admin', 'trace.lis', 'W');
```

Your location must be an explicit, complete path to the file. You cannot use operating system-specific parameters such as environment variables in UNIX to specify file locations.

UTL_FILE exceptions

The package specification of UTL_FILE defines seven exceptions. The cause behind a UTL_FILE exception can often be difficult to understand. Here are the explanations Oracle provides for each of the exceptions:

> **NOTE:** As a result of the way these exceptions are declared (as "user-defined exceptions"), there is no error number associated with any of the exceptions. Thus you must include explicit exception handlers in programs that call UTL_FILE if you wish to find out which error was raised. See the section "Handling file I/O errors" for more details on this process.

INVALID_PATH
: The file location or the filename is invalid. Perhaps the directory is not listed as a utl_file_dir parameter in the INIT.ORA file (or doesn't exist as all), or you are trying to read a file and it does not exist.

INVALID_MODE
: The value you provided for the open_mode parameter in UTL_FILE.FOPEN was invalid. It must be "A," "R," or "W."

INVALID_FILEHANDLE
: The file handle you passed to a UTL_FILE program was invalid. You must call UTL_FILE.FOPEN to obtain a valid file handle.

INVALID_OPERATION
: UTL_FILE could not open or operate on the file as requested. For example, if you try to write to a

read-only file, you will raise this exception.

READ_ERROR
: The operating system returned an error when you tried to read from the file. (This does not occur very often.)

WRITE_ERROR
: The operating system returned an error when you tried to write to the file. (This does not occur very often.)

INTERNAL_ERROR
: Uh-oh. Something went wrong and the PL/SQL runtime engine couldn't assign blame to any of the previous exceptions. Better call Oracle Support!

Programs in UTL_FILE may also raise the following standard system exceptions:

NO_DATA_FOUND
: Raised when you read past the end of the file with UTL_FILE.GET_LINE.

VALUE_ERROR
: Raised when you try to read or write lines in the file which are too long. The current implementation of UTL_FILE limits the size of a line read by UTL_FILE.GET_LINE to 1022 bytes.

INVALID_MAXLINESIZE
: Oracle 8.0 and above: raised when you try to open a file with a maximum linesize outside of the valid range (between 1 through 32767).

In the following descriptions of the UTL_FILE programs, I list the exceptions that can be raised by each individual program.

UTL_FILE nonprogram elements

When you open a file, PL/SQL returns a handle to that file for use within your program. This handle has a datatype of UTL_FILE.FILE_TYPE currently defined as the following:

```
TYPE UTL_FILE.FILE_TYPE IS RECORD (id BINARY_INTEGER);
```

As you can see, UTL_FILE.FILE_TYPE is actually a PL/SQL record whose fields contain all the information about the file needed by UTL_FILE. However, this information is for use only by the UTL_FILE package. You will reference the handle, but not any of the individual fields of the handle. (The fields of this record may expand over time as UTL_FILE becomes more sophisticated.)

Here is an example of how to declare a local file handle based on this type:

```
DECLARE
   file_handle UTL_FILE.FILE_TYPE;
BEGIN
   ...
```

UTL_FILE restrictions and limitations

While UTL_FILE certainly extends the usefulness of PL/SQL, it does have its drawbacks, including:

- Prior to Oracle 8.0, you cannot read or write a line of text with more than 1023 bytes. In Oracle 8.0 and above, you can specify a maximum line size of up to 32767 when you open a file..
- You cannot delete files through UTL_FILE. The best you can do is *empty* a file, but it will still be present on the disk.
- You cannot rename files. The best you can do is copy the contents of the file to another file with that new name.
- You do not have random access to lines in a file. If you want to read the 55th line, you must read through the first 54 lines. If you want to insert a line of text between the 1,267th and 1,268th lines, you will have to (a) read those 1,267 lines, (b) write them to a new file, (c) write the inserted line of text, and (d) read/write the remainder of the file. Ugh.
- You cannot change the security on files through UTL_FILE.
- You cannot access mapped files. Generally, you will need to supply real directory locations for files if you want to read from or write to them.

You are probably getting the idea. UTL_FILE is a basic facility for reading and writing server-side files. Working with UTL_FILE is not always pretty, but you can usually get what you need done with a little or a lot of code.

The UTL_FILE process flow

The following sections describe each of the UTL_FILE programs, following the process flow for working with files. That flow is described for both writing and reading files.

In order to write to a file you will (in most cases) perform the following steps:

1. Declare a file handle. This handle serves as a pointer to the file for subsequent calls to programs in the UTL_FILE package to manipulate the contents of this file.
2. Open the file with a call to FOPEN, which returns a file handle to the file. You can open a file to read, replace, or append text.
3. Write data to the file using the PUT, PUTF, or PUT_LINE procedures.
4. Close the file with a call to FCLOSE. This releases resources associated with the file.

To read data from a file you will (in most cases) perform the following steps:

1. Declare a file handle.
2. Declare a VARCHAR2 string buffer that will receive the line of data from the file. You can also read directly from a file into a numeric or date buffer. In this case, the data in the file will be converted implicitly, and so it must be compatible with the datatype of the buffer.
3. Open the file using FOPEN in read mode.

4. Use the GET_LINE procedure to read data from the file and into the buffer. To read all the lines from a file, you would execute GET_LINE in a loop.

5. Close the file with a call to FCLOSE.

Opening Files

Use the FOPEN and IS_OPEN functions when you open files via UTL_FILE.

> **NOTE:** Using the UTL-FILE package, you can only open a maximum of ten files for each Oracle session.

The UTL_FILE.FOPEN function

The FOPEN function opens the specified file and returns a file handle that you can then use to manipulate the file. Here's the header for the function:

```
All PL/SQL versions:           Oracle 8.0 and above only:
FUNCTION UTL_FILE.FOPEN (      FUNCTION UTL_FILE.FOPEN (
   location     IN VARCHAR2,      location     IN VARCHAR2,
   filename     IN VARCHAR2,      filename     IN VARCHAR2,
   open_mode    IN VARCHAR2)      open_mode    IN VARCHAR2,
RETURN file_type;                 max_linesize IN BINARY_INTEGER)
                                  RETURN file_type;
```

Parameters are summarized in the following table.

Parameter	Description
location	Location of the file
filename	Name of the file
openmode	Mode in which the file is to be opened (see the following modes)
max_linesize	The maximum number of characters per line, including the newline character, for this file. Minimum is 1, maximum is 32767

You can open the file in one of three modes:

R

Open the file read-only. If you use this mode, use UTL_FILE's GET_LINE procedure to read from the file.

W

Open the file to read and write in replace mode. When you open in replace mode, all existing lines in the file are removed. If you use this mode, then you can use any of the following UTL_FILE programs to modify the file: PUT, PUT_LINE, NEW_LINE, PUTF, and FFLUSH.

A

Open the file to read and write in append mode. When you open in append mode, all existing lines

in the file are kept intact. New lines will be appended after the last line in the file. If you use this mode, then you can use any of the following UTL_FILE programs to modify the file: PUT, PUT_LINE, NEW_LINE, PUTF, and fFFLUSH.

Keep the following points in mind as you attempt to open files:

- The file location and the filename joined together must represent a legal filename on your operating system.
- The file location specified must be accessible and must already exist; FOPEN will not create a directory or subdirectory for you in order to write a new file, for example.
- If you want to open a file for read access, the file must already exist. If you want to open a file for write access, the file will either be created, if it does not exist, or emptied of all its contents, if it does exist.
- If you try to open with append, the file must already exist. UTL_FILE will not treat your append request like a write access request. If the file is not present, UTL_FILE will raise the INVALID_OPERATION exception.

Exceptions

FOPEN may raise any of the following exceptions, described earlier:

```
UTL_FILE.INVALID_MODE
UTL_FILE.INVALID_OPERATION
UTL_FILE.INVALID_PATH
UTL_FILE.INVALID_MAXLINESIZE
```

Example

The following example shows how to declare a file handle and then open a configuration file for that handle in read-only mode:

```
DECLARE
        config_file UTL_FILE.FILE_TYPE;
BEGIN
        config_file := UTL_FILE.FOPEN ('/maint/admin', 'config.txt', 'R');
        ...
```

The UTL_FILE.IS_OPEN function

The IS_OPEN function returns TRUE if the specified handle points to a file that is already open. Otherwise, it returns false. The header for the function is,

```
FUNCTION UTL_FILE.IS_OPEN (file IN UTL_FILE.FILE_TYPE) RETURN BOOLEAN;
```

where file is the file to be checked.

Within the context of UTL_FILE, it is important to know what this means. The IS_OPEN function does not perform any operating system checks on the status of the file. In actuality, it merely checks to see if the id field of the file handle record is not NULL. If you don't play around with these records and their contents, then this id field is only set to a non-NULL value when you call FOPEN. It is set back to NULL

when you call FCLOSE.

Reading from Files

UTL_FILE provides only one program to retrieve data from a file: the GET_LINE procedure.

The UTL_FILE.GET_LINE procedure

The GET_LINE procedure reads a line of data from the specified file, if it is open, into the provided line buffer. Here's the header for the procedure:

```
PROCEDURE UTL_FILE.GET_LINE
   (file IN UTL_FILE.FILE_TYPE,
    buffer OUT VARCHAR2);
```

Parameters are summarized in the following table.

Parameter	Description
file	The file handle returned by a call to FOPEN
buffer	The buffer into which the line of data is read

The variable specified for the buffer parameter must be large enough to hold all the data up to the next carriage return or end-of-file condition in the file. If not, PL/SQL will raise the VALUE_ERROR exception. The line terminator character is not included in the string passed into the buffer.

Exceptions

GET_LINE may raise any of the following exceptions:

```
NO_DATA_FOUND
VALUE_ERROR
UTL_FILE.INVALID_FILEHANDLE
UTL_FILE.INVALID_OPERATION
UTL_FILE.READ_ERROR
```

Example

Since GET_LINE reads data only into a string variable, you will have to perform your own conversions to local variables of the appropriate datatype if your file holds numbers or dates. Of course, you could call this procedure and read data directly into string and numeric variables as well. In this case, PL/SQL will be performing a runtime, implicit conversion for you. In many situations, this is fine. I generally recommend that you avoid implicit conversions and perform your own conversion instead. This approach more clearly documents the steps and dependencies. Here is an example:

```
DECLARE
   fileID UTL_FILE.FILE_TYPE;
   strbuffer VARCHAR2(100);
   mynum NUMBER;
BEGIN
   fileID := UTL_FILE.FOPEN ('/tmp', 'numlist.txt', 'R');
   UTL_FILE.GET_LINE (fileID, strbuffer);
   mynum := TO_NUMBER (strbuffer);
```

```
END;
/
```

When GET_LINE attempts to read past the end of the file, the NO_DATA_FOUND exception is raised. This is the same exception that is raised when you (a) execute an implicit (SELECT INTO) cursor that returns no rows or (b) reference an undefined row of a PL/SQL (nested in PL/SQL8) table. If you are performing more than one of these operations in the same PL/SQL block, remember that this same exception can be caused by very different parts of your program.

Writing to Files

In contrast to the simplicity of reading from a file, UTL_FILE offers a number of different procedures you can use to write to a file:

UTL_FILE.PUT
: Puts a piece of data (string, number, or date) into a file in the current line.

UTL_FILE.NEW_LINE
: Puts a newline or line termination character into the file at the current position.

UTL_FILE.PUT_LINE
: Puts a string into a file, followed by a platform-specific line termination character.

UTL_FILE.PUTF
: Puts up to five strings out to the file in a format based on a template string, similar to the printf function in C.

You can use these procedures only if you have opened your file with modes W or A; if you opened the file for read-only, the runtime engine will raise the UTL_FILE.INVALID_OPERATION exception.

Starting with Oracle 8.0.3, the maximum size of a file string is 32K; the limit for earlier versions is 1023 bytes. If you have longer strings, you must break them up into individual lines, perhaps using a special continuation character to notify a post-processor to recombine those lines.

The UTL_FILE.PUT procedure

The PUT procedure puts data out to the specified open file. Here's the header for this procedure:

```
PROCEDURE UTL_FILE.PUT
    (file IN UTL_FILE.FILE_TYPE,
    buffer OUT VARCHAR2);
```

Parameters are summarized in the following table.

Parameter	Description
file	The file handle returned by a call to FOPEN
buffer	The buffer containing the text to be written to the file; maximum size allowed is 32K for Oracle 8.0.3 and above; for earlier versions, it is 1023 bytes

The PUT procedure adds the data to the current line in the opened file, but does not append a line

terminator. You must use the NEW_LINE procedure to terminate the current line or use PUT_LINE to write out a complete line with a line termination character.

Exceptions

PUT may raise any of the following exceptions:

```
UTL_FILE.INVALID_FILEHANDLE
UTL_FILE.INVALID_OPERATION
UTL_FILE.WRITE_ERROR
```

The UTL_FILE.NEW_LINE procedure

The NEW_LINE procedure inserts one or more newline characters in the specified file. Here's the header for the procedure:

```
PROCEDURE UTL_FILE.NEW_LINE
   (file IN UTL_FILE.FILE_TYPE,
    lines IN NATURAL := 1);
```

Parameters are summarized in the following table.

Parameter	Description
file	The file handle returned by a call to FOPEN
lines	Number of lines to be inserted into the file

If you do not specify a number of lines, NEW_LINE uses the default value of 1, which places a newline character (carriage return) at the end of the current line. So if you want to insert a blank line in your file, execute the following call to NEW_LINE:

```
UTL_FILE.NEW_LINE (my_file, 2);
```

If you pass 0 or a negative number for lines, nothing is written into the file.

Exceptions

NEW_LINE may raise any of the following exceptions:

```
VALUE_ERROR
UTL_FILE.INVALID_FILEHANDLE
UTL_FILE.INVALID_OPERATION
UTL_FILE.WRITE_ERROR
```

Example

If you frequently wish to add an end-of-line marker after you PUT data out to the file (see the PUT procedure information), you might bundle two calls to UTL_FILE modules together, as follows:

```
PROCEDURE add_line (file_in IN UTL_FILE.FILE_TYPE, line_in IN VARCHAR2)
IS
BEGIN
   UTL_FILE.PUT (file_in, line_in);
```

```
   UTL_FILE.NEW_LINE (file_in);
END;
```

By using add_line instead of PUT, you will not have to worry about remembering to call NEW_LINE to finish off the line. Of course, you could also simply call the PUT_LINE procedure.

The UTL_FILE.PUT_LINE procedure

This procedure writes data to a file and then immediately appends a newline character after the text. Here's the header for PUT_LINE:

```
PROCEDURE UTL_FILE.PUT_LINE
    (file IN UTL_FILE.FILE_TYPE,
    buffer IN VARCHAR2);
```

Parameters are summarized in the following table.

Parameter	Description
file	The file handle returned by a call to FOPEN
buffer	Text to be written to the file; maximum size allowed is 32K for Oracle 8.0. 3 and above; for earlier versions, it is 1023 bytes

Before you can call UTL_FILE.PUT_LINE, you must have already opened the file.

Exceptions

PUT_LINE may raise any of the following exceptions:

```
UTL_FILE.INVALID_FILEHANDLE
UTL_FILE.INVALID_OPERATION
UTL_FILE.WRITE_ERROR
```

Example

Here is an example of using PUT_LINE to dump the contents of the emp table to a file:

```
PROCEDURE emp2file
IS
   fileID UTL_FILE.FILE_TYPE;
BEGIN
   fileID := UTL_FILE.FOPEN ('/tmp', 'emp.dat', 'W');

   /* Quick and dirty construction here! */
   FOR emprec IN (SELECT * FROM emp)
   LOOP
      UTL_FILE.PUT_LINE
         (TO_CHAR (emprec.empno) || ',' ||
          emprec.ename || ',' ||
          ...
          TO_CHAR (emprec.deptno));
   END LOOP;

   UTL_FILE.FCLOSE (fileID);
END;
```

A call to PUT_LINE is equivalent to a call to PUT followed by a call to NEW_LINE. It is also equivalent to a call to PUTF with a format string of "%s\n" (see the description of PUTF in the next section).

The UTL_FILE.PUTF procedure

Like PUT, PUTF puts data into a file, but it uses a message format (hence, the "F" in "PUTF") to interpret the different elements to be placed in the file. You can pass between one and five different items of data to PUTF. Here's the specification:

```
PROCEDURE UTL_FILE.PUTF
    (file IN FILE_TYPE
    ,format IN VARCHAR2
    ,arg1 IN VARCHAR2 DEFAULT NULL
    ,arg2 IN VARCHAR2 DEFAULT NULL
    ,arg3 IN VARCHAR2 DEFAULT NULL
    ,arg4 IN VARCHAR2 DEFAULT NULL
    ,arg5 IN VARCHAR2 DEFAULT NULL);
```

Parameters are summarized in the following table.

Parameter	Description
file	The file handle returned by a call to FOPEN
format	The string that determines the format of the items in the file; see the following options
argN	An optional argument string; up to five may be specified

The format string allows you to substitute the argN values directly into the text written to the file. In addition to "boilerplate" or literal text, the format string may contain the following patterns:

%s
: Directs PUTF to put the corresponding item in the file. You can have up to five %s patterns in the format string, since PUTF will take up to five items.

\n
: Directs PUTF to put a newline character in the file. There is no limit to the number of \n patterns you may include in a format string.

The %s formatters are replaced by the argument strings in the order provided. If you do not pass in enough values to replace all of the formatters, then the %s is simply removed from the string before writing it to the file.

Exceptions

UTL_FILE.PUTF may raise any of the following exceptions:

```
UTL_FILE.INVALID_FILEHANDLE
UTL_FILE.INVALID_OPERATION
UTL_FILE.WRITE_ERROR
```

Example

The following example illustrates how to use the format string. Suppose you want the contents of the file

to look like this:

```
Employee: Steven Feuerstein
Soc Sec #: 123-45-5678
Salary: $1000
```

This single call to PUTF will accomplish the task:

```
UTL_FILE.PUTF
   (file_handle, 'Employee: %s\nSoc Sec #: %s\nSalary: %s',
    'Steven Feuerstein',
    '123-45-5678',
    TO_CHAR (:employee.salary, '$9999'));
```

If you need to write out more than five items of data, you can simply call PUTF twice consecutively to finish the job, as shown here:

```
UTL_FILE.PUTF
   (file_handle, '%s\n%s\n%s\n%s\n%s\n',
    TO_DATE (SYSDATE, 'MM/DD/YYYY'),
    TO_CHAR (:pet.pet_id),
    :pet.name,
    TO_DATE (:pet.birth_date, 'MM/DD/YYYY'),
    :pet.owner);

UTL_FILE.PUTF
   (file_handle, '%s\n%s\n',
    :pet.bites_mailperson,
    :pet.does_tricks);
```

The UTL_FILE.FFLUSH procedure

This procedure makes sure that all pending data for the specified file is written physically out to a file. The header for FFLUSH is,

```
PROCEDURE UTL_FILE.FFLUSH (file IN UTL_FILE.FILE_TYPE);
```

where file is the file handle.

Your operating system probably buffers physical I/O to improve performance. As a consequence, your program may have called one of the "put" procedures, but when you look at the file, you won't see your data. UTL_FILE.FFLUSH comes in handy when you want to read the contents of a file before you have closed that file. Typical scenarios include analyzing execution trace and debugging logs.

Exceptions

FFLUSH may raise any of the following exceptions:

```
UTL_FILE.INVALID_FILEHANDLE
UTL_FILE.INVALID_OPERATION
UTL_FILE.WRITE_ERROR
```

Closing Files

Use the FCLOSE and FCLOSE_ALL procedures in closing files.

The UTL_FILE.FCLOSE procedure

Use FCLOSE to close an open file. The header for this procedure is,

```
PROCEDURE UTL_FILE.FCLOSE (file IN OUT FILE_TYPE);
```

where file is the file handle.

Notice that the argument to UTL_FILE.FCLOSE is an IN OUT parameter, because the procedure sets the id field of the record to NULL after the file is closed.

If there is buffered data that has not yet been written to the file when you try to close it, UTL_FILE will raise the WRITE_ERROR exception.

Exceptions

FCLOSE may raise any of the following exceptions:

```
UTL_FILE.INVALID_FILEHANDLE
UTL_FILE.WRITE_ERROR
```

The UTL_FILE.FCLOSE_ALL procedure

FCLOSE_ALL closes all of the opened files. The header for this procedure follows:

```
PROCEDURE UTL_FILE.FCLOSE_ALL;
```

This procedure will come in handy when you have opened a variety of files and want to make sure that none of them are left open when your program terminates.

In programs in which files have been opened, you should also call FCLOSE_ALL in exception handlers in programs. If there is an abnormal termination of the program, files will then still be closed.

```
EXCEPTION
   WHEN OTHERS

THEN
      UTL_FILE.FCLOSE_ALL;
      ... other clean up activities ...
END;
```

NOTE: When you close your files with the FCLOSE_ALL procedure, none of your file handles will be marked as closed (the id field, in other words, will still be non-NULL). The result is that any calls to IS_OPEN for those file handles will *still* return TRUE. You will not, however, be able to perform any read or write operations on those files (unless you reopen them).

Exceptions

FCLOSE_ALL may raise the following exception:

```
UTL_FILE.WRITE_ERROR
```

Tips on Using UTL_FILE

This section contains a variety of tips on using UTL_FILE to its full potential.

Handling file I/O errors

You may encounter a number of difficulties (and therefore raise exceptions) when working with operating system files. The good news is that Oracle has predefined a set of exceptions specific to the UTL_FILE package, such as UTL_FILE.INVALID_FILEHANDLE. The bad news is that these are all "user-defined exceptions," meaning that if you call SQLCODE to see what the error is, you get a value of 1, regardless of the exception. And a call to SQLERRM returns the less-than-useful string "User-Defined Exception."

To understand the problems this causes, consider the following program:

```
PROCEDURE file_action
IS
   fileID UTL_FILE.FILE_TYPE;
BEGIN
   fileID := UTL_FILE.FOPEN ('c:/tmp', 'lotsa.stf', 'R');
   UTL_FILE.PUT_LINE (fileID, 'just the beginning');
   UTL_FILE.FCLOSE (fileID);
END;
```

It is filled with errors, as you can see when I try to execute the program:

```
SQL> exec file_action
declare
*
ERROR at line 1:
ORA-06510: PL/SQL: unhandled user-defined exception
ORA-06512: at "SYS.UTL_FILE", line 91
ORA-06512: at "SYS.UTL_FILE", line 146
ORA-06512: at line 4
```

But what error or errors? Notice that the only information you get is that it was an "unhandled user-defined exception"--even though Oracle defined the exception!

The bottom line is that if you want to get more information out of the UTL_FILE-related errors in your code, you need to add exception handlers designed explicitly to trap UTL_FILE exceptions and *tell you* which one was raised. The following template exception section offers that capability. It includes an exception handler for each UTL_FILE exception. The handler writes out the name of the exception and then reraises the exception.

```
/* Filename on companion disk: fileexc.sql */

EXCEPTION
   WHEN UTL_FILE.INVALID_PATH
   THEN
       DBMS_OUTPUT.PUT_LINE ('invalid_path'); RAISE;

   WHEN UTL_FILE.INVALID_MODE
   THEN
       DBMS_OUTPUT.PUT_LINE ('invalid_mode'); RAISE;

   WHEN UTL_FILE.INVALID_FILEHANDLE
```

```
   THEN
       DBMS_OUTPUT.PUT_LINE ('invalid_filehandle'); RAISE;

   WHEN UTL_FILE.INVALID_OPERATION
   THEN
       DBMS_OUTPUT.PUT_LINE ('invalid_operation'); RAISE;

   WHEN UTL_FILE.READ_ERROR
   THEN
       DBMS_OUTPUT.PUT_LINE ('read_error'); RAISE;

   WHEN UTL_FILE.WRITE_ERROR
   THEN
      DBMS_OUTPUT.PUT_LINE ('write_error'); RAISE;

   WHEN UTL_FILE.INTERNAL_ERROR
   THEN
      DBMS_OUTPUT.PUT_LINE ('internal_error'); RAISE;
END;
```

If I add this exception section to my file_action procedure, I get this message,

```
SQL> @temp
invalid_operation
declare
*
ERROR at line 1:
ORA-06510: PL/SQL: unhandled user-defined exception
```

which helps me realize that I am trying to write to a read-only file. So I change the file mode to "W" and try again, only to receive the same error again! Additional analysis reveals that my file location is not valid. It should be "C:\temp" instead of "C:/tmp". So why didn't I get a UTL_FILE.INVALID_PATH exception? Who is to say? With those two changes made, file_action then ran without error.

I suggest that whenever you work with UTL_FILE programs, you include either all or the relevant part of *fileexc.sql*. (See each program description earlier in this chapter to find out which exceptions each program might raise.) Of course, you might want to change my template. You may not want to reraise the exception. You may want to display other information. Change whatever you need to change--just remember the basic rule that if you don't handle the UTL_FILE exception by name in the block in which the error was raised, you won't be able to tell what went wrong.

Closing unclosed files

As a corollary to the last section on handling I/O errors, you must be very careful to close files when you are done working with them, or when errors occur in your program. If not, you may sometimes have to resort to UTL_FILE.FCLOSE_ALL to close *all* your files before you can get your programs to work properly.

Suppose you open a file (and get a handle to that file) and then your program hits an error and fails. Suppose further that you do *not* have an exception section, so the program simply fails. So let's say that you fix the bug and rerun the program. Now it fails with UTL_FILE.INVALID_OPERATION. The problem is that your file is still open--and you have lost the handle to the file, so you cannot explicitly close just that one file.

Instead, you must now issue this command (here, from SQL*Plus):

```
SQL> exec UTL_FILE.FCLOSE_ALL
```

With any luck, you won't close files that you wanted to be left open in your session. As a consequence, I recommend that you always include calls to UTL_FILE.FCLOSE in each of your exception sections to avoid the need to call FCLOSE_ALL and to minimize extraneous INVALID_OPERATION exceptions.

Here is the kind of exception section you should consider including in your programs. (I use the PLVexc.recNstop handler from PL/Vision as an example of a high-level program to handle exceptions, in this case requesting that the program "record and then stop.")

```
EXCEPTION
   WHEN OTHRES
   THEN
      UTL_FILE.FCLOSE (ini_fileID);
      UTL_FILE.FCLOSE (new_fileID);
      PLVexc.recNstop;
END;
```

In other words, I close the two files I've been working with, and then handle the exception.

Combining locations and filenames

I wonder if anyone else out there in the PL/SQL world finds UTL_FILE as frustrating as I do. I am happy that Oracle built the package, but I sure wish they'd given us more to work with. I am bothered by these things:

- The need to separate my filename from the location. Most of the time when I work with files, those two pieces are stuck together. With UTL_FILE, I have to split them apart.
- The lack of support for paths. It would be nice to not have to provide a file location and just let UTL_FILE *find* my file for me.

This section shows you how to enhance UTL_FILE to allow you to pass in a "combo" filename: location and name joined together, as we so often encounter them. The next section explains the steps for adding path support to your manipulation of files with UTL_FILE.

If you are going to specify your file specification (location and name) in one string, what is the minimum information needed in order to separate these two elements to pass to FOPEN? The delimiter used to separate directories from filenames. In DOS (and Windows) that delimiter is "\". In UNIX it is "/". In VAX/VMS it is "]". Seems to me that I just have to find the *last* occurrence of this delimiter in your string and that will tell me where to break apart the string.

So to allow you to get around splitting up your file specification in your call to FOPEN, I can do the following:

- Give you a way to tell me in advance the operating system delimiter for directories--and store that value for use in future attempts to open files.
- Offer you a substitute FOPEN procedure that uses that delimiter.

Since I want to store that value for your entire session, I will need a package. (You can also use a database table so that you do not have to specify this value each time you start up your application.) Here is the

specification:

```
/* Filename on companion disk: onestring.spp */

CREATE OR REPLACE PACKAGE fileIO
IS
   PROCEDURE setsepchar (str IN VARCHAR2);
   FUNCTION sepchar RETURN VARCHAR2;

   FUNCTION open (file IN VARCHAR2, filemode IN VARCHAR2)
      RETURN UTL_FILE.FILE_TYPE;
END;
/
```

In other words, I set the separation character or delimiter with a call to fileIO.setsepchar, and I can retrieve the current value with a call to the fileIO.sepchar function. Once I have that value, I can call fileIO.open to open a file without having to split apart the location and name. I show an example of this program in use here:

```
DECLARE
   fid UTL_FILE.FILE_TYPE;
BEGIN
   fileIO.setsepchar ('\');
   fid := fileio.open ('c:\temp\newone.txt', 'w'));
END;
/
```

The body of this package is quite straightforward:

```
CREATE OR REPLACE PACKAGE BODY fileIO
IS
   g_sepchar CHAR(1) := '/'; /* Unix is, after all, dominant. */

   PROCEDURE setsepchar (str IN VARCHAR2)
   IS
   BEGIN
      g_sepchar := NVL (str, '/');
   END;

   FUNCTION sepchar RETURN VARCHAR2
   IS
   BEGIN
      RETURN g_sepchar;
   END;

   FUNCTION open (file IN VARCHAR2, filemode IN VARCHAR2)
      RETURN UTL_FILE.FILE_TYPE
   IS
      v_loc PLS_INTEGER := INSTR (file, g_sepchar, -1);
      retval UTL_FILE.FILE_TYPE;
   BEGIN
      RETURN UTL_FILE.FOPEN
         (SUBSTR (file, 1, v_loc-1),
          SUBSTR (file, v_loc+1),
          filemode);
   END;
END;
/
```

Notice that when I call INSTR I pass -1 for the third argument. This negative value tells the built-in to scan from the end of string backwards to the *first* occurrence of the specified character.

Adding support for paths

Why should I have to provide the directory name for my file each time I call FOPEN to read that file? It would be so much easier to specify a path, a list of possible directories, and then just let UTL_FILE scan the different directories in the specified order until the file is found.

Even though the notion of a path is not built into UTL_FILE, it is easy to add this feature. The structure of the implementation is very similar to the package built to combine file locations and names. I will need a package to receive and store the path, or list of directories. I will need an alternative open procedure that uses the path instead of a provided location. Here is the package specification:

```
/* Filename on companion disk: filepath.spp */

CREATE OR REPLACE PACKAGE fileIO
IS
   c_delim CHAR(1) := ';';

   PROCEDURE setpath (str IN VARCHAR2);
   FUNCTION path RETURN VARCHAR2;

   FUNCTION open (file IN VARCHAR2, filemode IN VARCHAR2)
      RETURN UTL_FILE.FILE_TYPE;
END;
/
```

I define the path delimiter as a constant so that a user of the package can see what he should use to separate different directories in his path. I provide a procedure to set the path and a function to get the path--but the variable containing the path is hidden away in the package body to protect its integrity.

Before exploring the implementation of this package, let's see how you would use these programs. The following test script sets a path with two directories and then displays the first line of code in the file containing the previous package:

```
/* Filename on companion disk: filepath.tst */

DECLARE
   fID UTL_FILE.FILE_TYPE;
   v_line VARCHAR2(2000);
BEGIN
   fileio.setpath ('c:\temp;d:\oreilly\builtins\code');
   fID := fileIO.open ('filepath.spp');
   UTL_FILE.GET_LINE (fID, v_line);
   DBMS_OUTPUT.PUT_LINE (v_line);
   UTL_FILE.FCLOSE (fID);
END;
/
```

I include a trace message in the package (commented out on the companion disk) so that we can watch the path-based open doing its work:

```
SQL> @filepath.tst
...looking in c:\temp
```

```
...looking in d:\oreilly\builtins\code
CREATE OR REPLACE PACKAGE fileIO
```

It's nice having programs do your work for you, isn't it? Here is the implementation of the fileIO package with path usage:

```
/* Filename on companion disk: filepath.spp */

CREATE OR REPLACE PACKAGE BODY fileIO
IS
   g_path VARCHAR2(2000);

   PROCEDURE setpath (str IN VARCHAR2)
   IS
   BEGIN
      g_path := str;
   END;

   FUNCTION path RETURN VARCHAR2
   IS
   BEGIN
      RETURN g_path;
   END;

   FUNCTION open (file IN VARCHAR2, filemode IN VARCHAR2)
      RETURN UTL_FILE.FILE_TYPE
   IS
      /* Location of next path separator */
      v_lastsep PLS_INTEGER := 1;
      v_sep PLS_INTEGER := INSTR (g_path, c_delim);
      v_dir VARCHAR2(500);
      retval UTL_FILE.FILE_TYPE;
   BEGIN
      /* For each directory in the path, attempt to open the file. */
      LOOP
         BEGIN
            IF v_sep = 0
            THEN
               v_dir := SUBSTR (g_path, v_lastsep);
            ELSE
               v_dir := SUBSTR (g_path, v_lastsep, v_sep - v_lastsep);
            END IF;
            retval := UTL_FILE.FOPEN (v_dir, file, 'R');
            EXIT;
         EXCEPTION
            WHEN OTHERS
            THEN
               IF v_sep = 0
               THEN
                  RAISE;
               ELSE
                  v_lastsep := v_sep + 1;
                  v_sep := INSTR (g_path, c_delim, v_sep+1);
               END IF;
         END;
      END LOOP;
      RETURN retval;
   END;
END;
/
```

The logic in this fileio.open is a little bit complicated, because I need to parse the semicolon-delimited list. The v_sep variable contains the location in the path of the next delimiter. The v_lastsep variable contains the location of the last delimiter. I have to include special handling for recognizing when I am at the last directory in the path (v_sep equals 0). Notice that I do not hard-code the semi-colon into this program. Instead, I reference the c_delim constant.

The most important implementation detail is that I place the call to FOPEN inside a *loop*. With each iteration of the loop body, I extract a directory from the path. Once I have the next directory to search, I call the FOPEN function to see if I can read the file. If I am able to do so successfully, I will reach the next line of code inside my loop, which is an EXIT statement: I am done and can leave. This drops me down to the RETURN statement to send back the handle to the file.

If I am unable to read the file in that directory, UTL_FILE raises an exception. Notice that I have placed the entire body of my loop inside its own anonymous block. This allows me to trap the open failure and process it. If I am on my last directory (no more delimiters, as in v_sep equals 0), I will simply reraise the exception from UTL_FILE. This will cause the loop to terminate, and then end the function execution as well. Since the fileIO.open does not have its own exception section, the error will be propagated out of the function unhandled. Even with a path, I was unable to locate the file. If, however, there are more directories, I set my start and end points for the next SUBSTR from the path and go back to the top of the loop so that FOPEN can try again.

If you do decide to use utilities like the path-based open shown previously, you should consider the following:

- Combine the logic in *filepath.spp* with *onestring.spp* (a version of open that lets you pass the location and name in a single string). I should be able to *override* the path by providing a location; the version shown in this section assumes that the filename never has a location in it.

- Allow users to add a directory to the path without having to concatenate it to a string with a semicolon between them. Why not build a procedure called fileIO.adddir that does the work for the user and allows an application to modify the path at runtime?

You closed what?

You might run into some interesting behavior with the IS_OPEN function if you treat your file handles as variables. You are not likely to do this, but I did, so I thought I would pass on my findings to you.

In the following script, I define two file handles. I then open a file, assigning the handle record generated by FOPEN to fileID1. I immediately assign that record to fileID2. They now both have the same record contents. I then close the file by passing fileID2 to FCLOSE and check the status of the file afterwards. Finally, I assign a value of NULL to the id field of fileID1 and call IS_OPEN again.

```
DECLARE
   fileID1 UTL_FILE.FILE_TYPE;
   fileID2 UTL_FILE.FILE_TYPE;
BEGIN
   fileID1 := UTL_FILE.FOPEN ('c:\temp', 'newdata.txt', 'W');
   fileID2 := fileID1;
   UTL_FILE.FCLOSE (fileID2);

   IF UTL_FILE.IS_OPEN (fileid1)
   THEN
      DBMS_OUTPUT.PUT_LINE ('still open');
```

```
   END IF;

   fileid1.id := NULL;
   IF NOT UTL_FILE.IS_OPEN (fileid1)
   THEN
      DBMS_OUTPUT.PUT_LINE ('now closed');
   END IF;
END;
/
```

Let's run the script and check out the results:

```
SQL> @temp
still open
now closed
```

We can conclude from this test that the IS_OPEN function returns TRUE if the id field of a UTL_FILE.FILE_TYPE record is NULL. It doesn't check the status of the file with the operating system. It is a check totally internal to UTL_FILE.

This will not cause any problems as long as (a) you don't muck around with the id field of your file handle records and (b) you are consistent with your use of file handles. In other words, if you assign one file record to another, use that new record for all operations. Don't go back to using the original.

UTL_FILE Examples

So you've got a file (or a dozen files) out on disk, filled with all sorts of good information you want to access from your PL/SQL-based application. You will find yourself performing the same kinds of operations against those files over and over again.

After you work your way through this book, I hope that you will recognize almost without conscious thought that you do not want to repeatedly build the open, read, and close operations for each of these files, for each of the various recurring operations. Instead, you will instantly say to yourself, "Hot diggity! This is an opportunity to build a set of standard, generic modules that will help manage my files."

This section contains a few of my candidates for the first contributions to a UTL_FILE toolbox of utilities. I recommend that you consider building a single package to contain all of these utilities.[4]

Enhancing UTL_FILE.GET_LINE

The GET_LINE procedure is simple and straightforward. It gets the next line from the file. If the pointer to the file is already located at the last line of the file, UTL_FILE.GET_LINE does not return data, but instead raises the NO_DATA_FOUND exception. Whenever you write programs using GET_LINE, you will therefore need to handle this exception. Let's explore the different ways you can do this.

The following example uses a loop to read the contents of a file into a PL/SQL table (whose type definition, tabpkg.names_tabtype, has been declared previously):

```
/* Filename on companion disk: file2tab.sp */

CREATE OR REPLACE PACKAGE tabpkg
IS
   TYPE names_tabtype IS TABLE OF VARCHAR2(100)
      INDEX BY BINARY_INTEGER;
```

```
END;
/
CREATE OR REPLACE PROCEDURE file_to_table
   (loc_in IN VARCHAR2, file_in IN VARCHAR2,
    table_in IN OUT tabpkg.names_tabtype)
IS
        /* Open file and get handle right in declaration */
        names_file UTL_FILE.FILE_TYPE := UTL_FILE.FOPEN (loc_in, file_in, 'R');
        /* Counter used to store the Nth name. */
        line_counter INTEGER := 1;
BEGIN
        LOOP
                UTL_FILE.GET_LINE (names_file, table_in(line_counter));
                line_counter := line_counter + 1;
        END LOOP;
EXCEPTION
        WHEN NO_DATA_FOUND
        THEN
                UTL_FILE.FCLOSE (names_file);
END;
/
```

The file_to_table procedure uses an infinite loop to read through the contents of the file. Notice that there is no EXIT statement within the loop to cause the loop to terminate. Instead I rely on the fact that the UTL_FILE package raises a NO_DATA_FOUND exception once it goes past the end-of-file marker and short-circuits the loop by transferring control to the exception section. The exception handler then traps that exception and closes the file.

I am not entirely comfortable with this approach. I don't like to code infinite loops without an EXIT statement; the termination condition is not structured into the loop itself. Furthermore, the end-of-file condition is not really an exception; every file, after all, must end at some point.

I believe that a better approach to handling the end-of-file condition is to build a layer of code around GET_LINE that immediately checks for end-of-file and returns a Boolean value (TRUE or FALSE). The get_nextline procedure shown here embodies this principle.

```
/* Filename on companion disk: getnext.sp */

PROCEDURE get_nextline
   (file_in IN UTL_FILE.FILE_TYPE,
    line_out OUT VARCHAR2,
    eof_out OUT BOOLEAN)
IS
BEGIN
   UTL_FILE.GET_LINE (file_in, line_out);
   eof_out := FALSE;
EXCEPTION
   WHEN NO_DATA_FOUND
   THEN
      line_out := NULL;
      eof_out  := TRUE;
END;
```

The get_nextline procedure accepts an already assigned file handle and returns two pieces of information: the line of text (if there is one) and a Boolean flag (set to TRUE if the end-of-file is reached, FALSE otherwise). Using get_nextline, I can now read through a file with a loop that has an EXIT statement.

My file_to_table procedure will look like the following after adding get_nextline:

```
/* Filename on companion disk: fil2tab2.sp */

PROCEDURE file_to_table
   (loc_in IN VARCHAR2, file_in IN VARCHAR2,
   table_in IN OUT names_tabtype)
IS
   /* Open file and get handle right in declaration */
   names_file CONSTANT UTL_FILE.FILE_TYPE :=
      UTL_FILE.FOPEN (loc_in, file_in, 'R');

   /* counter used to create the Nth name. */
   line_counter INTEGER := 1;

   end_of_file BOOLEAN := FALSE;
BEGIN
   WHILE NOT end_of_file
   LOOP
      get_nextline (names_file, table_in(line_counter), end_of_file);
      line_counter := line_counter + 1;
   END LOOP;
   UTL_FILE.FCLOSE (names_file);
END;
```

With get_nextline, I no longer treat end-of-file as an exception. I read a line from the file until I am done, and then I close the file and exit. This is, I believe, a more straightforward and easily understood program.

Creating a file

A common way to use files does not involve the contents of the file as much as a confirmation that the file does in fact exist. You can use the two modules defined next to create a file and then check to see if that file exists. Notice that when I create a file in this type of situation, I do not even bother to return the handle to the file. The purpose of the first program, create_file, is simply to make sure that a file with the specified name (and optional line of text) is out there on disk.

```
/* Filename on companion disk: crefile.sp */

PROCEDURE create_file
   (loc_in IN VARCHAR2, file_in IN VARCHAR2, line_in IN VARCHAR2 := NULL)
IS
   file_handle UTL_FILE.FILE_TYPE;
BEGIN
   /*
   || Open the file, write a single line and close the file.
   */
   file_handle := UTL_FILE.FOPEN (loc_in, file_in, 'W');
   IF line_in IS NOT NULL
   THEN
      UTL_FILE.PUT_LINE (file_handle, line_in);
   ELSE
      UTL_FILE.PUT_LINE
         (file_handle, 'I make my disk light blink, therefore I am.');
   END IF;
   UTL_FILE.FCLOSE (file_handle);
END;
```

Testing for a file's existence

The second program checks to see if a file exists. Notice that it creates a local procedure to handle the close logic (which is called both in the body of the function and in the exception section).

```
/* Filename on companon disk: filexist.sf */

CCREATE OR REPLACE FUNCTION file_exists
   (loc_in IN VARCHAR2,
    file_in IN VARCHAR2,
    close_in IN BOOLEAN := FALSE)
   RETURN BOOLEAN
IS
   file_handle UTL_FILE.FILE_TYPE;
   retval BOOLEAN;

   PROCEDURE closeif IS
   BEGIN
      IF close_in AND UTL_FILE.IS_OPEN (file_handle)
      THEN
         UTL_FILE.FCLOSE (file_handle);
      END IF;
   END;
BEGIN
   /* Open the file. */
   file_handle := UTL_FILE.FOPEN (loc_in, file_in, 'R');

   /* Return the result of a check with IS_OPEN. */
   retval := UTL_FILE.IS_OPEN (file_handle);

   closeif;

   RETURN retval;
EXCEPTION
   WHEN OTHERS
   THEN
      closeif;
      RETURN FALSE;
 END;
/
```

Searching a file for a string

Because I found the INSTR function to be so useful, I figured that this same kind of operation would also really come in handy with operating system files. The line_with_text function coming up shortly returns the line number in a file containing the specified text. The simplest version of such a function would have a specification like this:

```
FUNCTION line_with_text
   (loc_in IN VARCHAR2, file_in IN VARCHAR2, text_in IN VARCHAR2)
RETURN INTEGER
```

In other words, given a location, a filename, and a chunk of text, find the first line in the file that contains the text. You could call this function as follows:

```
IF line_with_text ('h:\pers', 'names.vp', 'Hanubi') > 0
THEN
   MESSAGE ('Josephine Hanubi is a vice president!');
END IF;
```

The problem with this version of line_with_text is its total lack of vision. What if I want to find the second occurrence in the file? What if I need to start my search from the tenth line? What if I want to perform a case-insensitive search? None of these variations are supported.

I urge you strongly to think through all the different ways a utility like line_with_text might be used before you build it. Don't just build for today's requirement. Anticipate what you will need tomorrow and next week as well.

For line_with_text, a broader vision would yield a specification like this:

```
FUNCTION line_with_text
   (loc_in IN VARCHAR2,
    file_in IN VARCHAR2,
    text_in IN VARCHAR2,
    occurrence_in IN INTEGER := 1,
    start_line_in IN INTEGER := 1,
    end_line_in IN INTEGER := 0,
    ignore_case_in IN BOOLEAN := TRUE)
RETURN INTEGER
```

Wow! That's a lot more parameter passing. Let's take a look at the kind of flexibility we gain from these additional arguments. First, the following table provides a description of each parameter.

Parameter	Description
loc_in	The location of the file on the operating system
file_in	The name of the file to be opened
text_in	The chunk of text to be searched for in each line of the file
occurrence_in	The number of times the text should be found in distinct lines in the file before the function returns the line number
srart_line_in	The first line in the file from which the function should start its search
end_line_in	The last line in the file to which the function should continue its search; if zero, then search through end of file
ignore_case_in	Indicates whether the case of the file contents and text_in should be ignored when checking for its presence in the line

Notice that all the new parameters, occurrence_in through ignore_case_in, have default values, so I can call this function in precisely the same way and with the same results as the first, limited version:

```
IF line_with_text ('names.vp', 'Hanubi') > 0
THEN
   MESSAGE ('Josephine Hanubi is a vice president!');
END IF;
```

Now, however, I can also do so much more:

- Confirm that the role assigned to this user is SUPERVISOR:

  ```
  line_with_text ('c:\temp', 'config.usr', 'ROLE=SUPERVISOR')
  ```

- Find the second occurrence of DELETE starting with the fifth line:

```
line_with_text ('/tmp', 'commands.dat', 'delete', 2, 5)
```

- Verify that the third line contains a terminal type specification:

```
line_with_text ('g:\apps\user\', 'setup.cfg', 'termtype=', 1, 3, 3)
```

Here is the code for the line_with_text function:

```
/* Filename on companion disk: linetext.sf */

CREATE OR REPLACE FUNCTION line_with_text
   (loc_in IN VARCHAR2,
    file_in IN VARCHAR2,
    text_in IN VARCHAR2,
    occurrence_in IN INTEGER := 1,
    start_line_in IN INTEGER := 1,
    end_line_in IN INTEGER := 0,
    ignore_case_in IN BOOLEAN := TRUE)
RETURN INTEGER
/*
|| An "INSTR" for operating system files. Returns the line number of
|| a file in which a text string was found.
*/
IS
   /* Handle to the file. Only will open if arguments are valid. */
   file_handle UTL_FILE.FILE_TYPE;

   /* Holds a line of text from the file. */
   line_of_text VARCHAR2(1000);

   text_loc INTEGER;
   found_count INTEGER := 0;

   /* Boolean to determine if there are more values to read */
   no_more_lines BOOLEAN := FALSE;

   /* Function return value */
   return_value INTEGER := 0;
BEGIN
   /* Assert valid arguments. If any fail, return NULL. */
   IF loc_in IS NULL OR
      file_in IS NULL OR
      text_in IS NULL OR
      occurrence_in <= 0 OR
      start_line_in < 1 OR
      end_line_in < 0
   THEN
      return_value := NULL;
   ELSE
      /* All arguments are fine. Open and read through the file. */
      file_handle := UTL_FILE.FOPEN (loc_in, file_in, 'R');
      LOOP
         /* Get next line and exit if at end of file. */
         get_nextline (file_handle, line_of_text, no_more_lines);
         EXIT WHEN no_more_lines;

         /* Have another line from file. */
         return_value := return_value + 1;
```

```
            /* If this line is between the search range... */
            IF (return_value BETWEEN start_line_in AND end_line_in) OR
               (return_value >= start_line_in AND end_line_in = 0)
            THEN
               /* Use INSTR to see if text is present. */
               IF NOT ignore_case_in
               THEN
                  text_loc := INSTR (line_of_text, text_in);
               ELSE
                  text_loc := INSTR (UPPER (line_of_text), UPPER (text_in));
               END IF;

               /* If text location is positive, have a match. */
               IF text_loc > 0
               THEN
                  /* Increment found counter. Exit if matches request. */
                  found_count := found_count + 1;
                  EXIT WHEN found_count = occurrence_in;
               END IF;
            END IF;
         END LOOP;
         UTL_FILE.FCLOSE (file_handle);
      END IF;

      IF no_more_lines
      THEN
         /* read through whole file without success. */
         return_value := NULL;
      END IF;

      RETURN return_value;
   END;
```

Getting the nth line from a file

What if you want to get a specific line from a file? The following function takes a filename and a line number and returns the text found on that line:

```
/* Filename on companion disk: nthline.sf */

CREATE OR REPLACE FUNCTION get_nth_line
   (loc_in IN VARCHAR2, file_in IN VARCHAR2, line_num_in IN INTEGER)
IS
   /* Handle to the file. Only will open if arguments are valid. */
   file_handle UTL_FILE.FILE_TYPE;

   /* Count of lines read from the file. */
   line_count INTEGER := 0;

   /* Boolean to determine if there are more values to read */
   no_more_lines BOOLEAN := FALSE;

   /* Function return value */
   return_value VARCHAR2(1000) := NULL;
BEGIN
   /* Need a file name and a positive line number. */
   IF file_in IS NOT NULL AND line_num_in > 0
   THEN
      /* All arguments are fine. Open and read through the file. */
      file_handle := UTL_FILE.FOPEN (loc_in, file_in, 'R');
```

```
      LOOP
         /* Get next line from file. */
         get_nextline (file_handle, return_value, no_more_lines);

         /* Done if no more lines or if at the requested line. */
         EXIT WHEN no_more_lines OR line_count = line_num_in - 1;

         /* Otherwise, increment counter and read another line. */
         line_count := line_count + 1;
      END LOOP;
      UTL_FILE.FCLOSE (file_handle);
   END IF;

   /* Either NULL or contains last line read from file. */
   RETURN return_value;
END;
```

---

1. As this book is going to press, the following PL/SQL debuggers are now available: SQL-Station Debugger from Platinum Technology; SQL Navigator from Quest; Xpediter/SQL from Compuware; and Procedure Builder from Oracle Corporation.

2. A version of PL/Vision is available through a free download from the www.revealnet.com site.

3. Available through a free download from the www.revealnet.com site.

4. You will find an example of such a package in Chapter 13 of *Advanced Oracle PL/SQL Programming with Packages*.

Back to: Oracle Built-in Packages