CHAPTER 22

I/O and PL/SQL

Many, perhaps most, of the PL/SQL programs you write need to interact only with the underlying Oracle database using SQL. However, there will inevitably be times when you will want to send information from PL/SQL to the external environment or read information from some external source (screen, file, etc.) into PL/SQL. This chapter explores some of the most common mechanisms for I/O in PL/SQL, including the following built-in packages:

DBMS_OUTPUT
: For displaying information on the screen

UTL_FILE
: For reading and writing operating system files

UTL_MAIL and UTL_SMTP
: For sending email from within PL/SQL

UTL_HTTP
: For retrieving data from a web page

It is outside the scope of this book to provide full reference information about the built-in packages introduced in this chapter. Instead, I will demonstrate how to use them to handle the most frequently encountered requirements. Check out Oracle's documentation for more complete coverage. You will also find *Oracle Built-in Packages* (O'Reilly) a helpful source for information on many packages; several chapters from that book are available on this book's website.

Displaying Information

Oracle provides the DBMS_OUTPUT package to enable you to send information from your programs to a buffer. This buffer can then be read and manipulated by another

PL/SQL program or by the host environment. DBMS_OUTPUT is most frequently used as a simple mechanism for displaying information on your screen.

Each user session has a DBMS_OUTPUT buffer of predefined size, which developers commonly set to UNLIMITED. Oracle versions prior to Oracle Database 10*g* Release 2 had a 1-million-byte limit. Once the buffer is filled, you will need to empty it before you can reuse it; you can empty it programmatically, but more commonly you will rely on the host environment (such as SQL*Plus) to empty it and display its contents. This only occurs after the outermost PL/SQL block terminates; you cannot use DBMS_OUTPUT for real-time streaming of messages from your program.

The way to write information to this buffer is by calling the DBMS_OUTPUT.PUT and DBMS_OUTPUT.PUT_LINE programs. If you want to read from the buffer programmatically, you can use DBMS_OUTPUT.GET_LINE or DBMS_OUTPUT.GET_LINES.

Enabling DBMS_OUTPUT

Since the default setting of DBMS_OUTPUT is disabled, calls to the PUT_LINE and PUT programs are ignored and the buffer remains empty. To enable DBMS_OUTPUT, you generally execute a command in the host environment. For example, in SQL*Plus, you can issue this command:

```
SET SERVEROUTPUT ON SIZE UNLIMITED
```

In addition to enabling output to the console, this command has the side effect of issuing the following command to the database server:

```
BEGIN DBMS_OUTPUT.ENABLE (buffer_size => NULL); END;
```

(Null buffer_size equates to an unlimited buffer; otherwise, the buffer_size is expressed in bytes.) SQL*Plus offers a variety of options for the SERVEROUTPUT command; check the documentation for the features for your release.

Developer environments such as Oracle's SQL Developer and Quest's Toad generally display the output from DBMS_OUTPUT in a designated portion of the screen (a *pane*), as long as you have properly enabled the display feature.

Write Lines to the Buffer

There are two built-in procedures to choose from when you want to put information into the buffer. PUT_LINE appends a newline marker after your text; PUT places text in the buffer *without* a newline marker. If you're using PUT alone, the output will remain in the buffer even when the call ends. In this case, call DBMS_OUTPUT.NEW_LINE to flush the buffer.

If the Oracle database knows implicitly how to convert your data to a VARCHAR2 string, then you can pass it in your call to the PUT and PUT_LINE programs. Here are some examples:

```
BEGIN
   DBMS_OUTPUT.put_line ('Steven');
   DBMS_OUTPUT.put_line (100);
   DBMS_OUTPUT.put_line (SYSDATE);
END;
/
```

Unfortunately, DBMS_OUTPUT does not know what to do with a variety of common PL/SQL types, most notably Booleans. You may therefore want to consider writing a small utility to make it easier to display Boolean values, such as the following procedure, which displays a string and then the Boolean:

```
/* File on web: bpl.sp */
PROCEDURE bpl (boolean_in IN BOOLEAN)
IS
BEGIN
   DBMS_OUTPUT.PUT_LINE(
      CASE boolean_in
         WHEN TRUE THEN 'TRUE'
         WHEN FALSE THEN 'FALSE'
         ELSE 'NULL'
      END
   );
END bpl;
/
```

The largest string that you can pass in one call to DBMS_OUTPUT.PUT_LINE is 32,767 bytes in the most recent releases of Oracle. With Oracle Database 10*g* Release 1 or earlier, the limit is 255 bytes. With any version, if you pass a value larger than the maximum allowed, the database will raise an exception (either VALUE_ERROR or *ORU-10028: line length overflow, limit of NNN chars per line*). To avoid this problem, you might want to use an encapsulation of DBMS_OUTPUT.PUT_LINE that automatically wraps long strings. The following files, available on the book's website, offer variations on this theme:

pl.sp
: This standalone procedure allows you to specify the length at which your string will be wrapped.

p.pks/pkb
: The p package is a comprehensive encapsulation of DBMS_OUTPUT.PUT_LINE that offers many different overloadings (for example, you can display an XML document or an operating system file by calling the p.l procedure) and also wraps long lines of text.

Read the Contents of the Buffer

The typical usage of DBMS_OUTPUT is very basic: you call DBMS_OUTPUT.PUT_LINE and view the results on the screen. Behind the scenes, your client environment (e.g., SQL*Plus) calls the appropriate programs in the DBMS_OUTPUT package to extract the contents of the buffer and then display them.

If you need to obtain the contents of the DBMS_OUTPUT buffer, you can call the GET_LINE and/or GET_LINES procedures.

The GET_LINE procedure retrieves one line of information from the buffer in a first-in, first-out fashion, and returns a status value of 0 if successful. Here's an example that uses this program to extract the next line from the buffer into a local PL/SQL variable:

```
FUNCTION next_line RETURN VARCHAR2
IS
   return_value VARCHAR2(32767);
   status INTEGER;
BEGIN
   DBMS_OUTPUT.GET_LINE (return_value, status);
   IF status = 0
   THEN
      RETURN return_value;
   ELSE
      RETURN NULL;
   END IF;
END;
```

The GET_LINES procedure retrieves multiple lines from the buffer with one call. It reads the buffer into a PL/SQL collection of strings (maximum length 255 or 32,767 bytes, depending on your version of Oracle). You specify the number of lines you want to read, and it returns those. Here is a generic program that transfers the contents of the DBMS_OUTPUT buffer into a database log table:

```
/* File on web: move_buffer_to_log.sp */
PROCEDURE move_buffer_to_log
IS
   l_buffer      DBMS_OUTPUT.chararr;
   l_num_lines   PLS_INTEGER;
BEGIN
   LOOP
      l_num_lines := 100;
      DBMS_OUTPUT.get_lines (l_buffer, l_num_lines);

      EXIT WHEN l_buffer.COUNT = 0;

      FORALL indx IN l_buffer.FIRST .. l_buffer.LAST
         INSERT INTO logtab (text) VALUES (l_buffer (indx));
   END LOOP;
END;
```

Reading and Writing Files

The UTL_FILE package 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. 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 that existed prior to Oracle Database 10*g* Release 2.

UTL_FILE lets you read and write files accessible from the server on which your database is running. Sounds dangerous, eh? An ill-intentioned or careless programmer could theoretically use UTL_FILE to write over tablespace datafiles, control files, and so on. Oracle allows the DBA to place restrictions on where you can read and write your files in one of two ways:

- UTL_FILE reads and writes files in directories that are specified by the UTL_FILE_DIR parameter in the database initialization file.
- UTL_FILE also reads/writes files in locations specified by database "Directory" objects.

After explaining how to use these two approaches, I will examine the specific capabilities of the UTL_FILE package. Many of the UTL_FILE programs are demonstrated in a handy encapsulation package found in the *fileIO.pkg* file on the book's website.

The UTL_FILE_DIR Parameter

Although not officially deprecated, the UTL_FILE_DIR approach is rarely used with the latest versions of the Oracle database. Using directories is much easier and more flexible. If you have a choice, don't use UTL_FILE_DIR; just skip this section and jump ahead to "Working with Oracle Directories" on page 931.

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, which you can specify with an entry in the database initialization file such as:

```
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/Linux-like filesystems:

```
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 in production environments. In development environments, this entry certainly makes it easier for developers to get up and running on UTL_FILE, as well as to test their code. However, you should allow access to only a few specific directories when you move the application to production.

Setting up directories

Here are some observations on working with and setting up accessible directories with UTL_FILE:

- Access is not recursive through subdirectories. Suppose that the following lines were in your database initialization file:

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

  You would not be able to open a file in the *c:\group\prod\oe\reports* subdirectory.
- Do not include the following entry on Unix or Linux systems:

  ```
  UTL_FILE_DIR = .
  ```

  This allows you to read/write on the current directory in the operating system.
- Do not enclose the directory names within single or double quotes.
- In a Unix/Linux 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 to have 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/Linux. The following specification of a directory will result in problems when you try to read from or write to the directory:

  ```
  UTL_FILE_DIR = /tmp/orafiles/
  ```

Specifying file locations when opening files

The location of the file is an operating system–specific string that specifies the directory or area in which to open the file. When you pass the location in the call to UTL_FILE.FOPEN, you provide the location specification as it appears in the database 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.

Following are some examples:

- In Windows:

```
file_id := UTL_FILE.FOPEN ('k:\common\debug', 'trace.lis', 'R');
```

- In Unix/Linux:

```
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/Linux to specify file locations.

Working with Oracle Directories

Prior to Oracle9*i* Database Release 2, whenever you opened a file, you needed to specify the location of the file, as in the preceding examples. Such a hardcoding of values is always to be avoided, however. What if the location of the accounts data changes? How many programs will you have to fix to make sure everyone is looking in the right place? How many times will you have to make such changes?

A better approach is to declare a variable or constant and assign it the value of the location. If you do this in a package, the constant can be referenced by any program in a schema with the EXECUTE privilege on that package. Here is an example, followed by a recoding of the earlier FOPEN call:

```
PACKAGE accts_pkg
IS
   c_data_location
      CONSTANT VARCHAR2(30) := '/accts/data';
   ...
END accts_pkg;

DECLARE
   file_id   UTL_FILE.file_type;
BEGIN
   file_id := UTL_FILE.fopen (accts_pkg.c_data_location, 'trans.dat', 'R');
END;
```

That's great. But even better is to use a schema-level object that you can define in the database: a directory. This particular type of object is also used with BFILEs, so you can in effect "consolidate" file location references in both DBMS_LOB and UTL_FILE by using directories.

To create a directory, you will need the DBA to grant you the CREATE ANY DIRECTORY privilege. You then define a new directory as shown in these examples:

```
CREATE OR REPLACE DIRECTORY development_dir AS '/dev/source';

CREATE OR REPLACE DIRECTORY test_dir AS '/test/source';
```

Here are some things to keep in mind about directories and UTL_FILE:

- The Oracle database does not validate the location you provide when you specify the name of a directory. It simply associates that string with the named database object.
- When you specify the name of a directory in a call to, say, UTL_FILE.FOPEN, it is not treated as the name of an Oracle object; instead, it is treated as a case-sensitive string. In other words, if you do not specify the name as an uppercase string, the operation will fail. This will work:

  ```
  handle := UTL_FILE.FOPEN(
          location => 'TEST_DIR', filename => 'myfile.txt', open_mode => 'r');
  ```

  but this will not:

  ```
  handle := UTL_FILE.FOPEN(
          location => test_dir, filename => 'myfile.txt', open_mode => 'r');
  ```

- Once you've created the directory, you can grant specific users permission to work with it as follows:

  ```
  GRANT READ ON DIRECTORY development_dir TO senior_developer;
  ```

- Finally, you can query the contents of ALL_DIRECTORIES to determine which directories are available in the currently connected schema. You can also leverage this view to build some useful utilities. For example, to print a list of all the directories defined in the database:

  ```
  /* File on web: fileIO.pkg */
  PROCEDURE fileIO.gen_utl_file_dir_entries
  IS
  BEGIN
     FOR rec IN (SELECT * FROM all_directories)
     LOOP
        DBMS_OUTPUT.PUT_LINE ('UTL_FILE_DIR = ' || rec.directory_path);
     END LOOP;
  END gen_utl_file_dir_entries;
  ```

  One advantage of building utilities like those found in *fileIO.pkg* is that you can easily add sophisticated handling of the case of the directory to avoid "formatting errors," such as forgetting to specify the directory name in uppercase.

Open Files

Before you can read or write a file, you must open it. The UTL_FILE.FOPEN function opens the specified file and returns a file handle you can then use to manipulate the file. Here's the header for the function:

```
FUNCTION UTL_FILE.FOPEN (
     location     IN VARCHAR2
```

```
    , filename     IN VARCHAR2
    , open_mode    IN VARCHAR2
    , max_linesize IN BINARY_INTEGER DEFAULT NULL)
 RETURN UTL_FILE.file_type;
```

The parameters are summarized in the following table.

Parameter	Description
location	Location of the file (directory in UTL_FILE_DIR or a database directory).
filename	Name of the file.
open_mode	Mode in which the file is to be opened (see the following options).
max_linesize	Maximum number of characters per line, including the newline character, for this file. Minimum is 1; maximum is 32,767. The default of NULL means that UTL_FILE determines an appropriate value from the operating system (the value has historically been around 1,024 bytes).
UTL_FILE.file_type	Record containing all the information UTL_FILE needs to manage the file.

You can open the file in one of three modes:

R
: Opens the file as read-only. If you use this mode, use UTL_FILE's GET_LINE procedure to read from the file.

W
: Opens 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, you can use any of the following UTL_FILE programs to modify the file: PUT, PUT_LINE, NEW_LINE, PUTF, and FFLUSH.

A
: Opens 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, you can use any of the following UTL_FILE programs to modify the file: PUT, PUT_LINE, NEW_LINE, PUTF, and FFLUSH.

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

- The file location and the filename joined together must represent a legal file path 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.
- 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 and the file does not exist, UTL_FILE will create the file in write mode.

The following example shows how to declare a file handle and then open a 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');
```

Notice that I did not provide a maximum line size when I opened this file. That parameter is, in fact, optional. If you do not provide it, the maximum length of a line you can read from or write to the file is approximately 1,024 bytes. Given this limitation, you'll probably want to include the max_linesize argument, as shown here:

```
DECLARE
    config_file UTL_FILE.FILE_TYPE;
BEGIN
    config_file := UTL_FILE.FOPEN (
     '/maint/admin', 'config.txt', 'R', max_linesize => 32767);
```

Use the FOPEN_NCHAR function to open files written in multibyte character sets. In this case, Oracle recommends limiting max_linesize to 6,400 bytes.

Is the File Already Open?

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, this id field is set to a non-NULL value only when you call FOPEN. It is set back to NULL when you call FCLOSE.

Close Files

Use the UTL_FILE.FCLOSE and UTL_FILE.FCLOSE_ALL procedures to close a specific file and all open files in your session, respectively.

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

```
PROCEDURE UTL_FILE.FCLOSE (file IN OUT UTL_FILE.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.

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

```
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.

You may wish to call FCLOSE_ALL in the exception handlers of programs in which files have been opened, so that if there is an abnormal termination of the program, any open files will still be closed:

```
EXCEPTION
   WHEN OTHERS
   THEN
      UTL_FILE.FCLOSE_ALL;
      ... other cleanup activities ...
END;
```

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).

Read from Files

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

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

where *file* is the file handle returned by a call to FOPEN, and *buffer* is 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.

Oracle offers additional GET programs to read NVARCHAR2 data (GET_LINE_NCHAR) and raw data (GET_RAW).

Here is an example that uses GET_LINE:

```
DECLARE
   l_file UTL_FILE.FILE_TYPE;
   l_line VARCHAR2(32767);
BEGIN
   l_file := UTL_FILE.FOPEN (
          'TEMP_DIR', 'numlist.txt', 'R', max_linesize => 32767);
   UTL_FILE.GET_LINE (l_file, l_line);
   DBMS_OUTPUT.PUT_LINE (l_line);
END;
```

Because 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.

GET_LINE exceptions

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:

- Execute an implicit (SELECT INTO) cursor that returns no rows.
- Reference an undefined row of a PL/SQL collection.
- Read past the end of a BFILE (binary file) with DBMS_LOB.

If you are performing more than one of these operations in the same PL/SQL block, you may need to add extra logic to distinguish between the different sources of this error. See the *who_did_that.sql* file on the book's website for a demonstration of this technique.

Handy encapsulation for 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 any kind of flag but instead raises the NO_DATA_FOUND exception. This design leads to poorly structured code; you might consider using an encapsulation on top of GET_LINE to improve that design, as explained in this section.

Here is a program that reads each line from a file and then processes that line:

```
DECLARE
   l_file   UTL_FILE.file_type;
   l_line   VARCHAR2 (32767);
BEGIN
   l_file := UTL_FILE.FOPEN ('TEMP', 'names.txt', 'R');

   LOOP
      UTL_FILE.get_line (l_file, l_line);
      process_line (l_line);
   END LOOP;
EXCEPTION
   WHEN NO_DATA_FOUND
   THEN
      UTL_FILE.fclose (l_file);
END;
```

Notice that the simple loop does not contain any explicit EXIT statement. The loop terminates *implicitly* and with an exception as soon as UTL_FILE reads past the end of the file. In a small block like this one, the logic is clear. But imagine if my program is hundreds of lines long and much more complex. Suppose further that reading the contents of the file is just one step in the overall algorithm. If an exception terminates my block, I will then need to put the rest of my business logic in the exception section (bad idea), or put an anonymous BEGIN-END block wrapper around my read-file logic.

I am not 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. Why must I be forced into the exception section simply because I want to read a file in its entirety?

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 demonstrates this approach:

```
/* File on web: 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, or FALSE otherwise). Using get_nextline, I can now read through a file with a loop that has an EXIT statement:

```
DECLARE
   l_file   UTL_FILE.file_type;
   l_line   VARCHAR2 (32767);
   l_eof    BOOLEAN;
BEGIN
   l_file := UTL_FILE.FOPEN ('TEMP', 'names.txt', 'R');

   LOOP
      get_nextline (l_file, l_line, l_eof);
      EXIT WHEN l_eof;
      process_line (l_line);
   END LOOP;

   UTL_FILE.fclose (l_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.

Write 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
: 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.

UTL_FILE.NEW_LINE
: Inserts one or more newline characters (the default is 1) 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. This is the program you are most likely to be using with UTL_FILE.

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.

UTL_FILE.FFLUSH
UTL_FILE writes are normally buffered; FFLUSH immediately writes the buffer out to the filesystem.

You can use these procedures only if you have opened your file with modes W or A (see "Open Files" on page 932); if you opened the file as read-only, the runtime engine raises the UTL_FILE.INVALID_OPERATION exception.

Oracle offers additional PUT programs to write NVARCHAR2 data (PUT_LINE_NCHAR, PUT_NCHAR, PUTF_NCHAR) and raw data (PUT_RAW).

Let's take a closer look at UTL_FILE.PUT_LINE. This procedure writes data to a file and then immediately appends a newline character after the text. The header for PUT_LINE is:

```
PROCEDURE UTL_FILE.PUT_LINE (
     file IN UTL_FILE.FILE_TYPE
    ,buffer IN VARCHAR2
    ,autoflush IN BOOLEAN DEFAULT FALSE)
```

The 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 32,767
autoflush	Pass TRUE if you want this line to be flushed out to the operating system immediately

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

Here is an example that uses PUT_LINE to dump the names of all our employees to a file:

```
PROCEDURE names_to_file
IS
   fileid   UTL_FILE.file_type;
BEGIN
   fileid := UTL_FILE.FOPEN ('TEMP', 'names.dat', 'W');

   FOR emprec IN (SELECT * FROM employee)
   LOOP
      UTL_FILE.put_line (fileid, emprec.first_name || ' ' || emprec.last_name);
   END LOOP;

   UTL_FILE.fclose (fileid);
END names_to_file;
```

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).

Writing formatted text to file

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. The header for the procedure is:

```
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);
```

The 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 because 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 it is written to the file.

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\n',
    '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.

Copy Files

UTL_FILE.FCOPY lets you easily copy the contents of one source file to another destination file. The following snippet, for example, uses UTL_FILE.FCOPY to perform a backup by copying a single file from the development directory to the archive directory:

```
DECLARE
   file_suffix   VARCHAR2 (100)
            := TO_CHAR (SYSDATE, 'YYYYMMDDHH24MISS');
BEGIN
   -- Copy the entire file...
   UTL_FILE.FCOPY (
      src_location       => 'DEVELOPMENT_DIR',
      src_filename       => 'archive.zip',
      dest_location      => 'ARCHIVE_DIR',
      dest_filename      =>    'archive'
                            || file_suffix
                            || '.zip'
   );
END;
```

You can also use FCOPY to copy just a *portion* of a file. The program offers two additional parameters that allow you to specify the starting and ending line numbers of the portion you want to copy from the file. Suppose that I have a text file containing the names of the winners of a monthly PL/SQL quiz that started in January 2008. I would like to transfer all the names in 2009 to another file. I can do that by taking advantage of the fifth and sixth arguments of the FCOPY procedure, as shown here:

```
DECLARE
   c_start_year CONSTANT PLS_INTEGER := 2008;
   c_year_of_interest CONSTANT PLS_INTEGER := 2009;
   l_start PLS_INTEGER;
   l_end PLS_INTEGER;
BEGIN
   l_start := (c_year_of_interest - c_start_year)*12 + 1;
   l_end := l_start + 11;

   UTL_FILE.FCOPY (
      src_location       => 'WINNERS_DIR',
      src_filename       => 'names.txt',
      dest_location      => 'WINNERS_DIR',
      dest_filename      => 'names2008.txt',
      start_line         => l_start,
      end_line           => l_end
   );
END;
```

A useful encapsulation to UTL_FILE.FCOPY allows me to specify start and end strings instead of line numbers. I will leave the implementation of such a utility as an exercise for the reader (see the *infile.sf* file on the book's website for an implementation of an "INSTR for files" that might give you some ideas).

Delete Files

You can remove files using UTL_FILE.FREMOVE, as long as you are using Oracle9*i* Database Release 2 or later. The header for this procedure is:

```
PROCEDURE UTL_FILE.FREMOVE (
   location IN VARCHAR2,
   filename IN VARCHAR2);
```

For example, here I use UTL_FILE.FREMOVE to remove the original archive file shown previously:

```
BEGIN
   UTL_FILE.FREMOVE ('DEVELOPMENT_DIR', 'archive.zip');
END;
```

That's simple enough. You provide the location and name of the file, and UTL_FILE *attempts* to delete it. What if UTL_FILE encounters a problem? It might raise one of the exceptions in the following table.

Exception name	Meaning
UTL_FILE.invalid_path	Not a valid file handle
UTL_FILE.invalid_filename	File not found or filename NULL
UTL_FILE.file_open	File already open for writing/appending
UTL_FILE.access_denied	Access to the directory object is denied
UTL_FILE.remove_failed	Failed to delete file

In other words, UTL_FILE will raise an exception if you try to remove a file that doesn't exist or if you do not have the privileges needed to remove the file. Many file-removal programs in other languages (for example, File.delete in Java) return a status code to inform you of the outcome of the removal attempt. If you prefer this approach, you can use (or copy) the fileIO.FREMOVE program found in the *fileIO.pkg* file on the book's website.

Rename and Move Files

I can combine copy and remove operations into a single step by calling the UTL_FILE.FRENAME procedure. This handy utility allows me to either rename a file in the same directory or rename a file to another name *and* location (in effect, moving that file).

The header for FRENAME is:

```
PROCEDURE UTL_FILE.frename (
   src_location   IN VARCHAR2,
   src_filename   IN VARCHAR2,
   dest_location  IN VARCHAR2,
   dest_filename  IN VARCHAR2,
   overwrite      IN BOOLEAN DEFAULT FALSE);
```

This program may raise one of the exceptions shown in the following table.

Exception name	Meaning
UTL_FILE.invalid_path	Not a valid file handle
UTL_FILE.invalid_filename	File not found or filename NULL
UTL_FILE.rename_failed	Unable to perform the rename as requested
UTL_FILE.access_denied	Insufficient privileges to access directory object

You will find an interesting application of FRENAME in the *fileIO.pkg*—the chgext procedure. This program changes the extension of the specified file.

Retrieve File Attributes

Sometimes you need to get information about a particular file. How big is this file? Does the file even exist? What is the block size of the file? Such questions are no longer mysteries that can be solved only with the help of an operating system command (or, in the case of the file length, the DBMS_LOB package), as they were in early Oracle releases. UTL_FILE.FGETATTR provides all that information in a single native procedure call.

The header for FGETATTR is:

```
PROCEDURE UTL_FILE.FGETATTR (
   location    IN VARCHAR2,
   filename    IN VARCHAR2,
   fexists     OUT BOOLEAN,
   file_length OUT NUMBER,
   block_size  OUT BINARY_INTEGER);
```

Thus, to use this program, you must declare three different variables to hold the Boolean flag (does the file exist?), the length of the file, and the block size. Here is a sample usage:

```
DECLARE
   l_fexists      BOOLEAN;
   l_file_length  PLS_INTEGER;
   l_block_size   PLS_INTEGER;
BEGIN
   UTL_FILE.FGETATTR (
      location    => 'DEVELOPMENT_DIR',
      filename    => 'bigpkg.pkg',
      fexists     => l_fexists,
      file_length => l_file_length,
      block_size  => l_block_size
   );
   ...
END;
```

This interface is a bit awkward. What if you just want to find out the length of this file? You still have to declare all those variables, obtain the length, and then work with that value. Perhaps the best way to take advantage of FGETATTR is to build some of your own functions *on top of* this built-in that answer a single question, such as:

```
FUNCTION fileIO.flength (
   location_in   IN   VARCHAR2,
   file_in       IN   VARCHAR2
   )
   RETURN PLS_INTEGER;
```

or:

```
FUNCTION fileIO.fexists (
   location_in IN VARCHAR2,
   file_in IN VARCHAR2
   )
      RETURN BOOLEAN;
```

As a result, you will not have to declare unneeded variables, and you can write simpler, cleaner code.

Sending Email

Over the years, Oracle has gradually made it easier to send email from within a stored procedure. Here's a short example:

```
/* Requires Oracle Database 10g or later */
BEGIN
   UTL_MAIL.send(
      sender     => 'me@mydomain.com'
     ,recipients => 'you@yourdomain.com'
     ,subject    => 'API for sending email'
     ,message    =>
'Dear Friend:

This is not spam. It is a mail test.
Mailfully Yours,
Bill'
   );

END;
```

When you run this block, the database will attempt to send this message using whatever SMTP[1] host the DBA has configured in the initialization file (see the discussion in the next section).

The header for UTL_MAIL.SEND is:

```
PROCEDURE send(sender     IN VARCHAR2,
               recipients IN VARCHAR2,
               cc         IN VARCHAR2 DEFAULT NULL,
               bcc        IN VARCHAR2 DEFAULT NULL,
               subject    IN VARCHAR2 DEFAULT NULL,
               message    IN VARCHAR2 DEFAULT NULL,
               mime_type  IN VARCHAR2 DEFAULT 'text/plain; charset=us-ascii',
               priority   IN PLS_INTEGER DEFAULT 3);
```

Most of the parameters are self-explanatory. One nonobvious usage hint is that if you want to use more than one recipient (or Cc or Bcc), you should separate the addresses with commas, like this:

```
recipients => 'you@yourdomain.com, him@hisdomain.com'
```

OK, so that's pretty good if you have a recent version of Oracle, but what if you only have access to earlier versions, or what if you just want a little more control? You can still use the UTL_SMTP package, which is a little more complicated but nevertheless workable. Or if you want to code at an even lower level, you can use UTL_TCP, an external procedure, or a Java stored procedure, but I'll leave those as an exercise for anyone who wants to write some entertaining code.

Oracle Prerequisites

Unfortunately, not all versions of Oracle provide support for sending email from PL/SQL out of the box. The built-in UTL_SMTP is part of a default installation, so it generally will work. If you are using Oracle Database 11*g* Release 2, though, there is one security hoop you will have to jump through, as explained in the next section.

Starting with Oracle Database 10*g*, the default Oracle installation does not include the UTL_MAIL package. To set up and use UTL_MAIL, your DBA will have to perform the following tasks:

1. SMTP is one of many Internet acronyms governed by other acronyms. Simple Mail Transfer Protocol is governed by Request for Comment (RFC) 2821, which obsoletes RFC 821.

1. Set a value for the initialization parameter SMTP_OUT_SERVER. In Oracle Database 10*g* Release 2 and later, you can just do something like this:

   ```
   ALTER SYSTEM SET SMTP_OUT_SERVER = 'mailhost';
   ```

   In Oracle Database 10*g* Release 1, you need to edit your pfile by hand to set this parameter. The string you supply will be one or more (comma-delimited) mail hostnames that UTL_MAIL should try one at a time until it finds one it likes.
2. After setting this parameter, you must *bounce the database server* for the change to take effect. Amazing but true.
3. As SYS, run the installation scripts:

   ```
   @$ORACLE_HOME/rdbms/admin/utlmail.sql
   @$ORACLE_HOME/rdbms/admin/prvtmail.plb
   ```
4. Finally, grant execute permission to the "privileged few" who need to use it:

   ```
   GRANT EXECUTE ON UTL_MAIL TO SCOTT;
   ```

Configuring Network Security

As of Oracle Database 11*g* Release 2, your DBA will need to jump through one more security hoop for any package that makes network callouts, including UTL_SMTP and UTL_MAIL. The DBA will need to create an *access control list* (ACL), put your username or role into it, and grant the network-level privilege to that list. Here is a simple cookbook ACL for this purpose:

```
BEGIN
   DBMS_NETWORK_ACL_ADMIN.CREATE_ACL (
       acl          => 'mail-server.xml'
      ,description  => 'Permission to make network connections to mail server'
      ,principal    => 'SCOTT'  /* username or role */
      ,is_grant     => TRUE
      ,privilege    => 'connect'
   );

   DBMS_NETWORK_ACL_ADMIN.ASSIGN_ACL (
       acl         => 'mail-server.xml'
      ,host        => 'my-STMP-servername'
      ,lower_port  => 25   /* The default SMTP network port */
      ,upper_port  => NULL  /* Null here means open only port 25 */
   );
END;
```

These days, your network administrator might also need to configure a firewall to allow port 25 outbound connections from your database server, and your email administrator might also have some permissions to set!

Send a Short (32,767 Bytes or Less) Plain-Text Message

The example at the beginning of this section showed how to send a plain-text message if you have UTL_MAIL at your disposal. If, however, you are using UTL_SMTP, your program will have to communicate with the mail server at a lower programmatic level: opening the connection, composing the headers, sending the body of the message, and (ideally) examining the return codes. To give you a sense of what this looks like, Figure 22-1 shows a sample conversation between a mail server and a PL/SQL mail client I've named send_mail_via_utl_smtp.

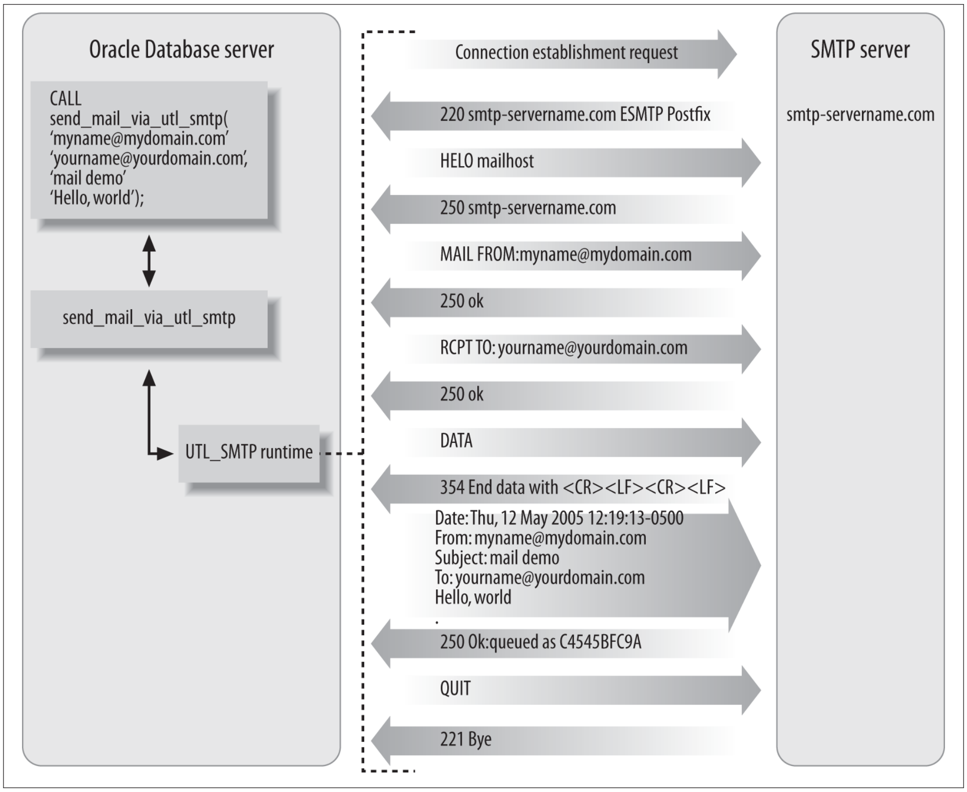

Oracle Database server
CALL
send_mail_via_utl_smtp(
'myname@mydomain.com'
'yourname@yourdomain.com',
'mail demo'
'Hello, world');
send_mail_via_utl_smtp
UTL_SMTP runtime
Connection establishment request
220 smtp-servername.com ESMTP Postfix
HELO mailhost
250 smtp-servername.com
MAIL FROM:myname@mydomain.com
250 ok
RCPT TO: yourname@yourdomain.com
250 ok
DATA
354 End data with <CR><LF><CR><LF>
Date: Thu, 12 May 2005 12:19:13-0500
From: myname@mydomain.com
Subject: mail demo
To: yourname@yourdomain.com
Hello, world
.
250 Ok:queued as C4545BFC9A
QUIT
221 Bye
SMTP server
smtp-servername.com

Figure 22-1. A "conversation" between the PL/SQL mail client and SMTP server

Here is the code for this simple stored procedure:

```
/* File on web: send_mail_via_utl_smtp.sp */
PROCEDURE send_mail_via_utl_smtp
  ( sender IN VARCHAR2
   ,recipient IN VARCHAR2
   ,subject IN VARCHAR2 DEFAULT NULL
   ,message IN VARCHAR2
```

```
 6      ,mailhost IN VARCHAR2 DEFAULT 'mailhost'
 7      )
 8    IS
 9      mail_conn UTL_SMTP.connection;
10      crlf CONSTANT VARCHAR2(2) := CHR(13) || CHR(10);
11      smtp_tcpip_port CONSTANT PLS_INTEGER := 25;
12    BEGIN
13      mail_conn := UTL_SMTP.OPEN_CONNECTION(mailhost, smtp_tcpip_port);
14      UTL_SMTP.HELO(mail_conn, mailhost);
15      UTL_SMTP.MAIL(mail_conn, sender);
16      UTL_SMTP.RCPT(mail_conn, recipient);
17      UTL_SMTP.DATA(mail_conn, SUBSTR(
18         'Date: ' || TO_CHAR(SYSTIMESTAMP, 'Dy, dd Mon YYYY HH24:MI:SS TZHTZM')
19         || crlf || 'From: ' || sender || crlf
20         || 'Subject: ' || subject || crlf
21         || 'To: ' || recipient || crlf
22         || message
23       , 1, 32767));
24
25      UTL_SMTP.QUIT(mail_conn);
26    END;
```

The following table explains a few concepts behind this code.

Line(s)	Description
9	You must define a variable to handle the "connection," which is a record of type UTL_SMTP.connection.
10	According to Internet mail standards, all header lines must end with a carriage return followed by a linefeed, and you are responsible for making this happen (see lines 19–21).
14–25	These lines send specific instructions to the SMTP server in this sequence to ensure Internet compliance.
18	This line uses SYSTIMESTAMP (introduced in Oracle9*i* Database) to gain access to time zone information.

If you look at lines 17–23, you'll see that this procedure cannot send a message whose "DATA" part exceeds 32,767 bytes, which is the limit of PL/SQL variables. It's possible to send longer emails using UTL_SMTP, but you will need to stream the data using multiple calls to UTL_SMTP.WRITE_DATA, as shown later.

By convention, most email programs limit each line of text to 78 characters plus the 2 line-terminating characters. In general, you'll want to keep each line of text to a maximum of 998 characters exclusive of carriage return/linefeed, or CRLF (that is, 1,000 bytes if you count the CRLF). Don't go over 1,000 bytes unless you're sure that your server implements the relevant SMTP "Service Extension."

Include "Friendly" Names in Email Addresses

If I invoke the previous procedure like this:

```
BEGIN
   send_mail_via_utl_smtp('myname@mydomain.com',
      'yourname@yourdomain.com', 'mail demo', NULL);
END;
```

the "normally" visible headers of the email, as generated by lines 17–21, will show up something like this:

```
Date: Wed, 23 Mar 2005 17:14:30 -0600
From: myname@mydomain.com
Subject: mail demo
To: yourname@yourdomain.com
```

Most humans (and many antispam programs) prefer to see real names in the headers, in a form such as:

```
Date: Wed, 23 Mar 2005 17:14:30 -0600
From: Bob Swordfish <myname@mydomain.com>
Subject: mail demo
To: "Scott Tiger, Esq." <yourname@yourdomain.com>
```

There is, of course, more than one way to make this change; perhaps the most elegant would be to add some parsing to the sender and recipient parameters. This is what Oracle has done in UTL_MAIL. So, for example, I can call UTL_MAIL.SEND with addresses of the form:

```
["]Friendly name["] <email_address>
```

as in:

```
BEGIN
   UTL_MAIL.send('Bob Swordfish <myname@mydomain.com>',
      '"Scott Tiger, Esq." <yourname@yourdomain.com>',
      subject=>'mail demo');
END;
```

However, you need to realize that Oracle's package also adds character set information, so the previous code generates an email header that looks something like this:

```
Date: Sat, 24 Jan 2009 17:47:00 -0600 (CST)
From: Bob Swordfish <me@mydomain.com>
To: Scott Tiger, Esq. <you@yourdomain.com>
Subject: =?WINDOWS-1252?Q?mail=20demo?=
```

While that looks odd to most ASCII speakers, it is completely acceptable in Internet-standards-land; an intelligent mail client should interpret (rather than display) the character set information anyway.

One quick and dirty modification of the send_mail_via_utl_smtp procedure would simply be to add parameters for the friendly names (or change the existing parameters to record structures).

Send a Plain-Text Message of Arbitrary Length

UTL_MAIL is pretty handy, but if you want to send a text message larger than 32,767 bytes, it won't help you. One way around this limitation is to modify the send_mail_via_utl_smtp procedure so that the "message" parameter is a CLOB datatype. Take a look at the other changes required:

```
/* File on web: send_clob.sp */
PROCEDURE send_clob_thru_email (
   sender    IN VARCHAR2
 , recipient IN VARCHAR2
 , subject   IN VARCHAR2 DEFAULT NULL
 , MESSAGE   IN CLOB
 , mailhost  IN VARCHAR2 DEFAULT 'mailhost'
)
IS
   mail_conn         UTL_SMTP.connection;
   crlf              CONSTANT VARCHAR2 (2) := CHR (13) || CHR (10);
   smtp_tcpip_port   CONSTANT PLS_INTEGER := 25;
   pos               PLS_INTEGER := 1;
   bytes_o_data      CONSTANT PLS_INTEGER := 32767;
   offset            PLS_INTEGER := bytes_o_data;
   msg_length        CONSTANT PLS_INTEGER := DBMS_LOB.getlength (MESSAGE);
BEGIN
   mail_conn := UTL_SMTP.open_connection (mailhost, smtp_tcpip_port);
   UTL_SMTP.helo (mail_conn, mailhost);
   UTL_SMTP.mail (mail_conn, sender);
   UTL_SMTP.rcpt (mail_conn, recipient);

   UTL_SMTP.open_data (mail_conn);
   UTL_SMTP.write_data (
      mail_conn
    ,    'Date: '
      || TO_CHAR (SYSTIMESTAMP, 'Dy, dd Mon YYYY HH24:MI:SS TZHTZM')
      || crlf
      || 'From: '
      || sender
      || crlf
      || 'Subject: '
      || subject
      || crlf
      || 'To: '
      || recipient
      || crlf
   );

   WHILE pos < msg_length
   LOOP
      UTL_SMTP.write_data (mail_conn, DBMS_LOB.SUBSTR (MESSAGE, offset, pos));
      pos := pos + offset;
      offset := LEAST (bytes_o_data, msg_length - offset);
   END LOOP;
```

```
      UTL_SMTP.close_data (mail_conn);

      UTL_SMTP.quit (mail_conn);
   END send_clob_thru_email;
```

Using open_data, write_data, and close_data allows you to transmit an arbitrary number of bytes to the mail server (up to whatever limit the server imposes on email size). Note the one big assumption that this code is making: that the CLOB has been properly split into lines of the correct length.

Let's next take a look at how to attach a file to an email.

Send a Message with a Short (32,767 Bytes or Less) Attachment

The original email standard required all messages to be composed of 7-bit U.S. ASCII characters.[2] But we all know that emails can include attachments—such as viruses and word-processing documents—and these kinds of files are normally binary, not text. How can an ASCII message transmit a binary file? The answer, in general, is that attachments are transmitted via mail extensions known as MIME[3] in combination with a binary-to-ASCII translation scheme such as base64. To see MIME in action, let's take a look at an email that transmits a tiny binary file:

```
Date: Wed, 01 Apr 2009 10:16:51 -0600
From: Bob Swordfish <my@myname.com>
MIME-Version: 1.0
To: Scott Tiger <you@yourname.com>
Subject: Attachment demo
Content-Type: multipart/mixed;
 boundary="------------060903040208010603090401"

This is a multi-part message in MIME format.
--------------060903040208010603090401
Content-Type: text/plain; charset=us-ascii; format=fixed
Content-Transfer-Encoding: 7bit

Dear Scott:

I'm sending a gzipped file containing the text of the first
paragraph. Hope you like it.

Bob
--------------060903040208010603090401
Content-Type: application/x-gzip; name="hugo.txt.gz"
```

2. Modern mail programs generally support 8-bit character transfer per an SMTP extension known as 8BITMIME. You can discover whether it's supported via SMTP's EHLO directive.

3. Multipurpose Internet Mail Extensions, as set forth in RFCs 2045, 2046, 2047, 2048, and 2049, and updated by 2184, 2231, 2646, and 3023. And then some...

```
Content-Transfer-Encoding: base64
Content-Disposition: inline; filename="hugo.txt.gz"

H4sICDh/TUICA2xlc21pcy50eHQAPY5BDoJAEATvvqI/AJGDxjMaowcesbKNOwmZITsshhf7
DdGD105Vpe+K5tQc0Jm6sGScU8gjvbrmoG8Tr1qhLtSCbs3CEa/gaMWTTbABF3kqa9z42+dE
RXhYmeHcpHmtBlmIoBEpREyZLpERtjB/aUSxns5/Ci7ac/u0P9a7Dw4FECSdAAAA
--------------060903040208010603090401--
```

Although a lot of the text can be boilerplated, there are still a lot of details to handle when you generate the email. Fortunately, if you just want to send a "small" attachment (less than 32,767 bytes), and you have Oracle Database 10*g* or later, UTL_MAIL comes to the rescue. In this next example, I'll use UTL_MAIL.SEND_ATTACH_VARCHAR2, which sends attachments that are expressed as text.

The previous message and file can be sent as follows:

```
DECLARE
   b64 VARCHAR2(512) := 'H4sICDh/TUICA2xlc21...'; -- etc., as above
   txt VARCHAR2(512) := 'Dear Scott: ...'; -- etc., as above
BEGIN
   UTL_MAIL.send_attach_varchar2(
      sender => 'my@myname.com'
     ,recipients => 'you@yourname.com'
     ,message => txt
     ,subject => 'Attachment demo'
     ,att_mime_type => 'application/x-gzip'
     ,attachment => b64
     ,att_inline => TRUE
     ,att_filename => 'hugo.txt.gz'
   );
END;
```

The new parameters are described in the following table.

Parameter	Description
att_mime_type	Indication of the type of media and format of the attachment
att_inline	Directive to the mail-reading program as to whether the attachment should be displayed in the flow of the message body (TRUE) or as a separate thing (FALSE)
att_filename	Sender's designated name for the attached file

The MIME type isn't just something you make up; it's loosely governed, like so many things on the Internet, by the Internet Assigned Numbers Authority (IANA). Common MIME content types include text/plain, multipart/mixed, text/html, application/pdf, and application/msword. For a complete list, visit IANA's MIME Media Types web page.

You may have noticed that there was quite a bit of handwaving earlier to attach a base64-encoded file to an email. Let's take a closer look at the exact steps required to convert a binary file into something you can send to an inbox.

Send a Small File (32,767 Bytes or Less) as an Attachment

To have the Oracle database convert a small binary file to something that can be emailed, you can read the contents of the file into a RAW variable and use UTL_MAIL.SEND_ATTACH_RAW. This causes the database to convert the binary data to base64 and properly construct the MIME directives. If the file you want to send is in */tmp/hugo.txt.gz* (and is less than 32,767 bytes in size), you might specify:

```
/* File on web: send_small_file.sql */
CREATE OR REPLACE DIRECTORY tmpdir AS '/tmp'
/
DECLARE
   the_file BFILE := BFILENAME('TMPDIR', 'hugo.txt.gz');
   rawbuf RAW(32767);
   amt PLS_INTEGER :=  32767;
   offset PLS_INTEGER := 1;
BEGIN
   DBMS_LOB.fileopen(the_file, DBMS_LOB.file_readonly);
   DBMS_LOB.read(the_file, amt, offset, rawbuf);
   UTL_MAIL.send_attach_raw
   (
      sender => 'my@myname.com'
     ,recipients => 'you@yourname.com'
     ,subject => 'Attachment demo'
     ,message => 'Dear Scott...'
     ,att_mime_type => 'application/x-gzip'
     ,attachment => rawbuf
     ,att_inline => TRUE
     ,att_filename => 'hugo.txt.gz'
   );

   DBMS_LOB.close(the_file);
END;
```

If you don't have UTL_MAIL, follow the instructions in the next section.

Attach a File of Arbitrary Size

To send a larger attachment, you can use the trusty UTL_SMTP package; if the attachment is not text, you can perform a base64 conversion with Oracle's built-in UTL_ENCODE package. Here is an example procedure that sends a BFILE along with a short text message:

```
/* File on web: send_bfile.sp */
PROCEDURE send_bfile
  ( sender IN VARCHAR2
   ,recipient IN VARCHAR2
   ,subject IN VARCHAR2 DEFAULT NULL
   ,message IN VARCHAR2 DEFAULT NULL
   ,att_bfile IN OUT BFILE
   ,att_mime_type IN VARCHAR2
   ,mailhost IN VARCHAR2 DEFAULT 'mailhost'
   )
IS
  crlf CONSTANT VARCHAR2(2) := CHR(13) || CHR(10);
  smtp_tcpip_port CONSTANT PLS_INTEGER := 25;
  bytes_per_read CONSTANT PLS_INTEGER := 23829;
  boundary CONSTANT VARCHAR2(78) := '-------5e9i1BxFQrgl9cOgs90-------';
  encapsulation_boundary CONSTANT VARCHAR2(78) := '--' || boundary;
  final_boundary CONSTANT VARCHAR2(78) := '--' || boundary || '--';

  mail_conn  UTL_SMTP.connection;
  pos PLS_INTEGER := 1;
  file_length PLS_INTEGER;

  diralias VARCHAR2(30);
  bfile_filename VARCHAR2(512);
  lines_in_bigbuf PLS_INTEGER := 0;

  PROCEDURE writedata (str IN VARCHAR2, crlfs IN PLS_INTEGER DEFAULT 1)
  IS
  BEGIN
     UTL_SMTP.write_data(mail_conn, str || RPAD(crlf, 2 * crlfs, crlf));
  END;

BEGIN
  DBMS_LOB.fileopen(att_bfile, DBMS_LOB.LOB_READONLY);
  file_length := DBMS_LOB.getlength(att_bfile);

  mail_conn := UTL_SMTP.open_connection(mailhost, smtp_tcpip_port);
  UTL_SMTP.helo(mail_conn, mailhost);
  UTL_SMTP.mail(mail_conn, sender);
  UTL_SMTP.rcpt(mail_conn, recipient);

  UTL_SMTP.open_data(mail_conn);
  writedata('Date: ' || TO_CHAR(SYSTIMESTAMP,
               'Dy, dd Mon YYYY HH24:MI:SS TZHTZM') || crlf
      || 'MIME-Version: 1.0' || crlf
      || 'From: ' || sender || crlf
      || 'Subject: ' || subject || crlf
      || 'To: ' || recipient || crlf
      || 'Content-Type: multipart/mixed; boundary="' || boundary || '"', 2);

  writedata(encapsulation_boundary);
  writedata('Content-Type: text/plain; charset=ISO-8859-1; format=flowed');
  writedata('Content-Transfer-Encoding: 7bit', 2);
  writedata(message, 2);

  DBMS_LOB.filegetname(att_bfile, diralias, bfile_filename);
  writedata(encapsulation_boundary);
  writedata('Content-Type: '
     || att_mime_type || '; name="' || bfile_filename || '"');
  writedata('Content-Transfer-Encoding: base64');
```

```
60    writedata('Content-Disposition: attachment; filename="'
61       || bfile_filename || '"', 2);
62
63    WHILE pos < file_length
64    LOOP
65       writedata(UTL_RAW.cast_to_varchar2(
66                   UTL_ENCODE.base64_encode
67                      DBMS_LOB.substr(att_bfile, bytes_per_read, pos))), 0);
68       pos := pos + bytes_per_read;
69    END LOOP;
70
71    writedata(crlf || crlf || final_boundary);
72
73    UTL_SMTP.close_data(mail_conn);
74    UTL_SMTP.QUIT(mail_conn);
75    DBMS_LOB.CLOSE(att_bfile);
76  END;
```

The following table looks at a few highlights.

Line(s)	Description
13	This constant governs how many bytes of the file to attempt to read at a time (see line 67), which should probably be as large as possible for performance reasons. It turns out that UTL_ENCODE.BASE64_ENCODE generates lines that are 64 characters wide. Because of the way base64 works, each 3 bytes of binary data gets translated into 4 bytes of character data. Add in 2 bytes of CRLF per emailed line of base64 text, and you get the largest possible read of 23,829 bytes—obtained from the expression TRUNC((0.75*64)*(32767/(64+2))–1.
14–16	You can reuse the same core boundary string throughout this email. As you can see from the code, MIME standards require that slightly different boundaries be used in different parts of the email. If you want to create an email with nested MIME parts, though, you will need a different boundary string for each level of nesting.
26–30	This is a convenience procedure to make the executable section a little cleaner. The crlfs parameter indicates the number of CRLFs to append to the line (generally 0, 1, or 2).
55	Instead of requiring a filename argument to send_bfile, you can just extract the filename from the BFILE itself.
63–69	This is the real guts of the program. It reads a portion of the file and converts it to base64, sending data out via the mail connection just before hitting the 32,767-byte limit.

I know what you're thinking: I, too, used to think sending email was easy. And this procedure doesn't even provide much flexibility; it lets you send one text part and attach one file. But it provides a starting point that you can extend for your own application's needs.

One more point about crafting well-formed emails: rather than reading yourself to sleep with the RFCs, you may prefer to pull out the email client you use every day, send yourself an email of the form you are trying to generate, and then view the underlying "source text" of the message. It worked for me; I did that many times while writing this section of the book! Note, however, that some mail clients, notably Microsoft Outlook, don't seem to provide a way to examine all of the underlying "source."

Working with Web-Based Data (HTTP)

Let's say you want to acquire some data from the website of one of your business partners. There are lots of ways to retrieve a web page:

- "By hand"—that is, by pointing your web browser to the right location
- Using a scripting language such as Perl, which, incidentally, has lots of available gizmos and gadgets to interpret the data once you retrieve it
- Via a command-line utility such as GNU *wget* (one of my favorite utilities)
- Using Oracle's built-in package UTL_HTTP

Since this is a book about PL/SQL, guess which method I'll be discussing!

If you're running Oracle Database 11*g* Release 2 or later, you will need to set up a network ACL to permit outbound connections to any desired remote hosts, as mentioned in the previous section.

Let's start with a relatively simple means of coding the retrieval of a web page. This first method, which slices up the web page and puts the slices into an array, actually predates Oracle's support of CLOBs.

Retrieve a Web Page in "Pieces"

One of the first procedures that Oracle ever released in the UTL_HTTP package retrieves a web page into consecutive elements of an associative array. Usage can be pretty simple:

```
DECLARE
   page_pieces UTL_HTTP.html_pieces; -- array of VARCHAR2(2000)
BEGIN
   page_pieces := UTL_HTTP.request_pieces(url => 'http://www.oreilly.com/');
END;
```

This format is not terribly fun to work with, because the 2,000-byte boundaries are unrelated to anything you would find on the text of the page. So if you have a parsing algorithm that needs a line-by-line approach, you will have to read and reassemble the lines. Moreover, Oracle says that it may not fill all of the (unending) pieces to 2,000 bytes; Oracle's algorithm does not use end-of-line boundaries as breaking points; and the maximum number of pieces is 32,767.

Even if an array-based retrieval meets your needs, you will likely encounter websites where the preceding code just won't work. For example, some sites may refuse to serve their content to such a script because Oracle's default HTTP "header" looks unfamiliar to the web server. In particular, the User-Agent header is a text string that tells the web server what browser software the client is using (or emulating), and many websites are set up to provide content specific to certain browsers. But by default, Oracle does not send a User-Agent. A commonly used and supported header you might want to use is:

```
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)
```

Sending this header does increase the complexity of the code you must write, though, because doing so means you must code at a lower level of abstraction. In particular, you must initiate a "request," send your header, get the "response," and retrieve the page in a loop:

```
DECLARE
   req UTL_HTTP.req;    -- a "request object" (actually a PL/SQL record)
   resp UTL_HTTP.resp;  -- a "response object" (also a PL/SQL record)
   buf VARCHAR2(32767); -- buffer to hold data from web page
BEGIN
   req := UTL_HTTP.begin_request('http://www.oreilly.com/',
      http_version => UTL_HTTP.http_version_1_1);
   UTL_HTTP.set_header(req, 'User-Agent'
      , 'Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)');
   resp := UTL_HTTP.get_response(req);

   BEGIN
      LOOP
         UTL_HTTP.read_text(resp, buf);
         -- process buf here; e.g., store in array
      END LOOP;
   EXCEPTION
      WHEN UTL_HTTP.end_of_body
      THEN
         NULL;
   END;
   UTL_HTTP.end_response(resp);
END;
```

The heart of the preceding code is this built-in:

```
PROCEDURE UTL_HTTP.read_text(
   r IN OUT NOCOPY UTL_HTTP.resp,
   data OUT NOCOPY VARCHAR2 CHARACTER SET ANY_CS,
   len IN PLS_INTEGER DEFAULT NULL);
```

If *len* is NULL, the Oracle database will fill the buffer up to its maximum size until reaching the end of the page, after which point the read operation raises the UTL_HTTP.end_of_body exception as shown. (Yes, like UTL_FILE.GET_LINE discussed earlier, this goes against a coding practice that normal operations should not raise exceptions.) On each iteration through the loop you will need to process the buffer, perhaps by appending it to a LOB.

You can also use the line-by-line retrieval using READ_LINE rather than READ_TEXT:

```
PROCEDURE UTL_HTTP.read_line(r IN OUT NOCOPY UTL_HTTP.resp,
   data OUT NOCOPY VARCHAR2 CHARACTER SET ANY_CS,
   remove_crlf IN BOOLEAN DEFAULT FALSE);
```

This built-in reads one line of source text at a time, optionally cutting off the end-of-line characters. The caveat with READ_LINE is that each line you fetch from the HTTP server needs to be less than 32,767 bytes in length. Such an assumption is not always a good one, so don't use READ_LINE unless you are sure this limit won't cause a problem.

Retrieve a Web Page into a LOB

Because reading either by "pieces" or by lines can run into various size limits, you may decide that it would make more sense to read into LOBs. Again, Oracle provides a very simple call that may meet your needs. You can retrieve an entire page at once into a single data structure using the HTTPURITYPE built-in object type:

```
DECLARE
   text CLOB;
BEGIN
   text := HTTPURITYPE('http://www.oreilly.com').getclob;
END;
```

If you are retrieving a binary file and you want to put it in a BLOB, you can use getblob:

```
DECLARE
   image BLOB;
BEGIN
   image :=
      HTTPURITYPE('www.oreilly.com/catalog/covers/oraclep4.s.gif').getblob;
END;
```

The HTTPURITYPE constructor assumes HTTP as the transport protocol, and you can either include or omit the "http://"—but, unfortunately, this built-in does not support HTTPS, nor will it let you send a custom User-Agent.

The UTL_HTTP flavor of fetching a LOB looks like this:

```
/* File on web: url_to_clob.sql */
DECLARE
   req UTL_HTTP.req;
   resp UTL_HTTP.resp;
   buf VARCHAR2(32767);
   pagelob CLOB;
BEGIN
   req := UTL_HTTP.begin_request('http://www.oreilly.com/',
      http_version => UTL_HTTP.http_version_1_1);
   UTL_HTTP.set_header(req, 'User-Agent', 'Mozilla/4.0 (compatible;
 MSIE 6.0; Windows NT 5.1)');
   resp := UTL_HTTP.get_response(req);
   DBMS_LOB.createtemporary(pagelob, TRUE);
   BEGIN
      LOOP
```

```
            UTL_HTTP.read_text(resp, buf);
            DBMS_LOB.writeappend(pagelob, LENGTH(buf), buf);
         END LOOP;
      EXCEPTION
         WHEN UTL_HTTP.end_of_body
         THEN
            NULL;
      END;
      UTL_HTTP.end_response(resp);

      ...here is where you parse, store, or otherwise process the LOB...

      DBMS_LOB.freetemporary(pagelob);
   END;
```

Authenticate Using HTTP Username/Password

Although many websites (such as Amazon and eBay) use a custom HTML form for login and authentication, there are still a lot of sites that use *HTTP authentication*, more precisely known as *basic authentication*. You will recognize such sites by your browser client's behavior; it will pop up a modal dialog box requesting your username and password.

It is sometimes possible to bypass the dialog by inserting your username and password in the URL in the following form (although this approach is deprecated in the official standards):

```
http://username:password@some.site.com
```

Both UTL_HTTP and HTTPURITYPE support this syntax, at least since 9.2.0.4. A simple case:

```
DECLARE
   webtext clob;
   user_pass VARCHAR2(64) := 'bob:swordfish'; -- replace with your own
   url VARCHAR2(128) := 'www.encryptedsite.com/cgi-bin/login';
BEGIN
   webtext := HTTPURITYPE(user_pass || '@' || url).getclob;
END;
/
```

If encoding the username and password in the URL doesn't work, try something along these lines:

```
...
   req := UTL_HTTP.begin_request('http://some.site.com/');
   UTL_HTTP.set_authentication(req, 'bob', 'swordfish');
   resp := UTL_HTTP.get_response(req);
...
```

This works as long as the site does not encrypt the login page.

Retrieve an SSL-Encrypted Web Page (via HTTPS)

Although HTTPURITYPE does not support SSL-encrypted retrievals, UTL_HTTP will do the job if you set up an *Oracle wallet*. An Oracle wallet is just a catchy name for a file that contains security certificates and, optionally, public/private key pairs. It's the former (the certificates) that you need for HTTPS retrievals. You can store one or more wallets as files in the database server's filesystem or in an LDAP directory service; Oracle does not install any wallets by default. See Chapter 23 for more information on wallets and other Oracle security features.

To set up one of these wallet things, you'll want to fire up Oracle's GUI utility known as Oracle Wallet Manager, which is probably named *owm* on Unix/Linux hosts or will appear on your Start→Oracle... menu on Microsoft Windows. Once you have Oracle Wallet Manager running, the basic steps you need to follow[4] are:

1. Click on the New icon or select Wallet→New from the pull-down menu.
2. Give the wallet a password. In my example, the password will be password1. Use the default wallet type ("standard").
3. If it asks, "Do you want to create a certificate request at this time?" the correct response is almost certainly "No." You don't need your own certificate to make an HTTPS retrieval.
4. Click on the Save icon or choose Wallet→Save As from the menu to designate the directory. Oracle will name the file for you (on my machine, *owm* named it ewallet.p12).
5. Upload or copy the wallet file to some location on the Oracle server to which the Oracle processes have read access. In the next example, the directory is */oracle/wallets*.

Now try something like this:

```
DECLARE
   req UTL_HTTP.req;
   resp UTL_HTTP.resp;
BEGIN
   UTL_HTTP.set_wallet('file:/oracle/wallets', 'password1');
   req := UTL_HTTP.begin_request('https://www.entrust.com/');
   UTL_HTTP.set_header(req, 'User-Agent', 'Mozilla/4.0');
   resp := UTL_HTTP.get_response(req);
   UTL_HTTP.end_response(resp);
END;
```

4. Thanks to Tom Kyte for spelling this out in plain English on his blog.

If you don't get an error message, you can reward yourself with a small jump for joy. This ought to work, because Entrust is one of the few authorities whose certificates Oracle includes by default when you create a wallet.

If you want to retrieve data from another HTTPS site whose public certificate doesn't happen to be on Oracle's list, you can fire up Oracle Wallet Manager again and "import" the certificate into your file, and again put it on the server. To download a certificate in a usable format, you can use Microsoft Internet Explorer and follow these steps:

1. Point your browser to the HTTPS site.
2. Double-click on the yellow lock icon in the lower-right corner of the window.
3. Click on Details→Copy to File.
4. Follow the prompts to export a base64-encoded certificate.

Or, if you have the OpenSSL package installed (typically on a Unix/Linux-based box), you could do this:

```
echo '' | openssl s_client -connect host:port
```

which will spew all kinds of information to stdout; just save the text between the BEGIN CERTIFICATE and END CERTIFICATE lines (inclusive) to a file. And, by the way, the normal port for HTTPS is 443.

Now that you have your certificate, you can do this:

1. Open Oracle Wallet Manager.
2. Open your "wallet" file.
3. Import the certificate from the file you just created.
4. Save your wallet file, and upload it to the database server as before.

Remember that those certificates are not in an Oracle wallet until you import them via Oracle Wallet Manager. And in case you're wondering, a wallet can have more than one certificate, and a wallet directory can hold one or more wallets.

Submit Data to a Web Page via GET or POST

Sometimes, you'll want to retrieve results from a website as if you had filled out a form in your browser and pressed the Submit button. This section will show a few examples that use UTL_HTTP for this purpose, but many websites are quirky and require quite a bit of fiddling about to get things working right. Some of the tools you may find useful while analyzing the behavior of your target site include:

- Familiarity with HTML source code (especially as it relates to HTML forms) and possibly with JavaScript

- A browser's "view source" feature, which lets you examine the source code of the site you're trying to use from PL/SQL
- A tool such as GNU *wget* that easily lets you try out different URLs and has an ability to show the normally hidden conversation between web client and server (use the -d switch)
- Browser plug-ins such as Chris Pederick's Web Developer and Adam Judson's Tamper Data for Mozilla-based browsers

First, let's look at some simple code you can use to query Google. As it turns out, Google's main page uses a single HTML form:

```
<form action=/search name=f>
```

Because the method tag is omitted, it defaults to GET. The single text box on the form is named *q* and includes the following properties (among others):

```
<input autocomplete="off" maxLength=2048 size=55 name=q value="">
```

You can encode the GET request directly in the URL as follows:

```
http://www.google.com/search?q=query
```

Given this information, here is the programmatic equivalent of searching for "oracle pl/sql programming" (including the double quotes) using Google:

```
DECLARE
   url VARCHAR2(64)
      := 'http://www.google.com/search?q=';
   qry VARCHAR2(128) := UTL_URL.escape('"oracle pl/sql programming"', TRUE);
   result CLOB;
BEGIN
   result := HTTPURITYPE(url || qry).getclob;
END;
```

Oracle's handy UTL_URL.ESCAPE function transforms the query by translating special characters into their hex equivalents. If you're curious, the escaped text from the example is:

```
%22oracle%20pl%2Fsql%20programming%22
```

Let's take a look at using POST in a slightly more complicated example. When I looked at the source HTML for *http://www.apache.org*, I found that the search form's "action" is *http://search.apache.org* that the form uses the POST method, and that the search box is named "query". With POST, you cannot simply append the data to the URL as with GET; instead, you send it to the web server in a particular form. Here is some code that POSTs a search for the string "oracle pl/sql" (relevant additions highlighted):

```
DECLARE
   req UTL_HTTP.req;
   resp UTL_HTTP.resp;
   qry VARCHAR2(512) := UTL_URL.escape('query=oracle pl/sql');
```

```
BEGIN
   req :=
       UTL_HTTP.begin_request('http://search.apache.org/', 'POST', 'HTTP  /1.0');
   UTL_HTTP.set_header(req, 'User-Agent', 'Mozilla/4.0');
   UTL_HTTP.set_header(req, 'Host', 'search.apache.org');
   UTL_HTTP.set_header(req, 'Content-Type', 'application/x-www-form-urlencoded');
   UTL_HTTP.set_header(req, 'Content-Length', TO_CHAR(LENGTH(qry)));
   UTL_HTTP.write_text(req, qry);
   resp := UTL_HTTP.get_response(req);

  ...now we can retrieve the results as before (e.g., line by line)...

   UTL_HTTP.end_response(resp);
END;
```

In a nutshell, the BEGIN_REQUEST includes the POST directive, and the code uses write_text to transmit the form data. While POST does not allow the name/value pairs to be appended to the end of the URL (as with GET queries), this site allows the x-www-form-urlencoded content type, allowing name/value pairs in the qry variable that you send to the server.

The earlier Apache example shows one other additional header that my other examples don't use:

```
UTL_HTTP.set_header(req, 'Host', 'search.apache.org');
```

Without this header, Apache's site was responding with its main page, rather than its search page. The Host header is required for websites that use the "virtual host" feature—that is, where one IP address serves two or more hostnames—so the web server knows what site you're looking for. The good thing is that you can always include the Host header, even if the remote site does not happen to serve virtual hosts.

By the way, if you have more than one item in the form to fill out, URL encoding says that each name/value pair must be separated with an ampersand:

```
name1=value1&name2=value2&name3= ...
```

OK, you've got all your GETs and POSTs working now, so you are all set to go forth and fetch... right? Possibly. It's likely your code will sooner or later run afoul of the HTTP "redirect." This is a special return code that web servers send that means, "Sorry, you need to go *over there* to find what you are looking for." We are accustomed to letting our browsers handle redirects for us silently and automatically, but it turns out that the underlying implementation can be tricky: there are at least five different kinds of redirect, each with slightly different rules about what it's "legal" for the browser to do. You may encounter redirects with any web page, but for many of them you should be able to use a feature in UTL_HTTP to follow redirects. That is:

```
UTL_HTTP.set_follow_redirect (max_redirects IN PLS_INTEGER DEFAULT 3);
```

Unfortunately, while testing code to retrieve a weather forecast page from the U.S. National Weather Service, I discovered that its server responds to a POST with a 302 Found, redirect. This is an odd case in the HTTP standard, which holds that clients should *not* follow the redirect... and Oracle's UTL_HTTP adheres to the letter of the standard, at least in this case.

So, I have to ignore the standard to get something useful from the weather page. My final program to retrieve the weather in Sebastopol, California, appears here:

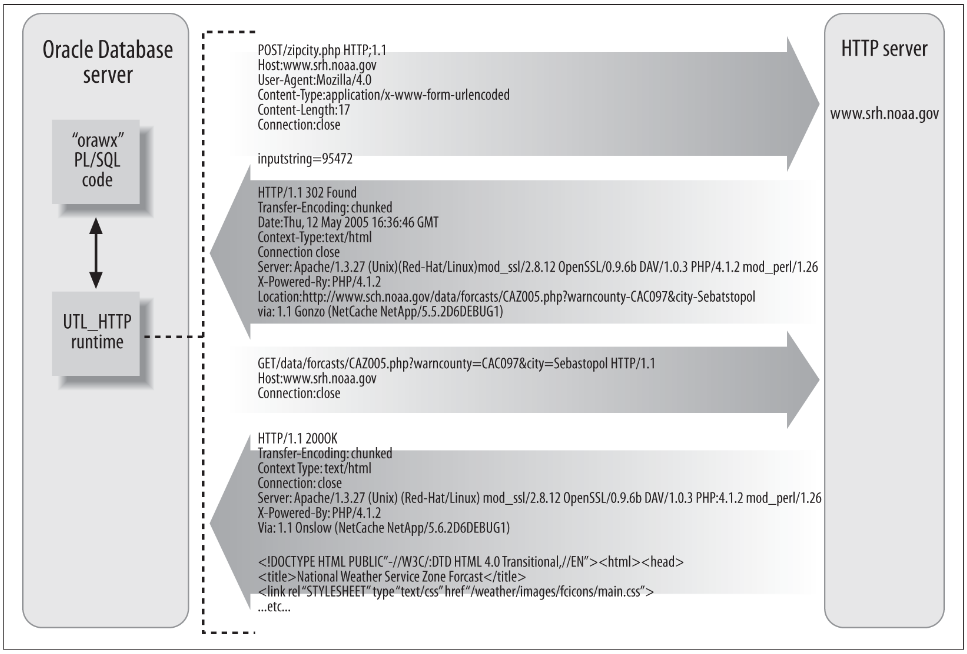
```
/* File on web: orawx.sp */
PROCEDURE orawx
AS
   req UTL_HTTP.req;
   resp UTL_HTTP.resp;
   line VARCHAR2(32767);
   formdata VARCHAR2(512) := 'inputstring=95472';   -- zip code
   newlocation VARCHAR2(1024);
BEGIN
   req := UTL_HTTP.begin_request('http://www.srh.noaa.gov/zipcity.php',
            'POST', UTL_HTTP.http_version_1_0);
   UTL_HTTP.set_header(req, 'User-Agent', 'Mozilla/4.0');
   UTL_HTTP.set_header(req, 'Content-Type', 'application/x-www-form-urlencoded');
   UTL_HTTP.set_header(req, 'Content-Length', TO_CHAR(LENGTH(formdata)));
   UTL_HTTP.write_text(req, formdata);
   resp := UTL_HTTP.get_response(req);

   IF resp.status_code = UTL_HTTP.http_found
   THEN
     UTL_HTTP.get_header_by_name(resp, 'Location', newlocation);
     req := UTL_HTTP.begin_request(newlocation);
     resp := UTL_HTTP.get_response(req);    END IF;

   ...process the resulting page here, as before...

   UTL_HTTP.end_response(resp);
END;
```

Figure 22-2 shows the basic interaction between this code and the web server.



```
POST/zipcity.php HTTP;1.1
Host:www.srh.noaa.gov
User-Agent:Mozilla/4.0
Content-Type:application/x-www-form-urlencoded
Content-Length:17
Connection:close

inputstring=95472
```

```
HTTP/1.1 302 Found
Transfer-Encoding: chunked
Date:Thu, 12 May 2005 16:36:46 GMT
Context-Type:text/html
Connection close
Server: Apache/1.3.27 (Unix)(Red-Hat/Linux)mod_ssl/2.8.12 OpenSSL/0.9.6b DAV/1.0.3 PHP/4.1.2 mod_perl/1.26
X-Powered-Ry: PHP/4.1.2
Location:http://www.sch.noaa.gov/data/forcasts/CAZ005.php?warncounty-CAC097&city-Sebatstopol
via: 1.1 Gonzo (NetCache NetApp/5.5.2D6DEBUG1)
```

```
GET/data/forcasts/CAZ005.php?warncounty=CAC097&city=Sebastopol HTTP/1.1
Host:www.srh.noaa.gov
Connection:close
```

```
HTTP/1.1 2000K
Transfer-Encoding: chunked
Context Type: text/html
Connection: close
Server: Apache/1.3.27 (Unix) (Red-Hat/Linux) mod_ssl/2.8.12 OpenSSL/0.9.6b DAV/1.0.3 PHP:4.1.2 mod_perl/1.26
X-Powered-By: PHP/4.1.2
Via: 1.1 Onslow (NetCache NetApp/5.6.2D6DEBUG1)

<!DOCTYPE HTML PUBLIC"-//W3C/:DTD HTML 4.0 Transitional,//EN"><html><head>
<title>National Weather Service Zone Forcast</title>
<link rel "STYLESHEET" type "text/css" href "/weather/images/fcicons/main.css">
...etc...
```

Figure 22-2. Getting the Sebastopol weather from NOAA involves following a 302 Found, redirection

I don't know how common a problem that is, and my "fix" is not really a general-purpose solution for all redirects, but it gives you an idea of the kinds of quirks you may run into when writing this sort of code.

Disable Cookies or Make Cookies Persistent

For better or for worse, session-level cookie support is *enabled by default* in recent versions of UTL_HTTP. Oracle has set a default of 20 cookies allowed per site and a total of 300 per session. To check whether this is true for your version of UTL_HTTP, use the following:

```
DECLARE
   enabled BOOLEAN;
   max_total PLS_INTEGER;
   max_per_site PLS_INTEGER;
BEGIN
   UTL_HTTP.get_cookie_support(enabled, max_total, max_per_site);
   IF enabled
   THEN
      DBMS_OUTPUT.PUT('Allowing ' || max_per_site || ' per site');
      DBMS_OUTPUT.PUT_LINE(' for total of ' || max_total || ' cookies. ');
   ELSE
      DBMS_OUTPUT.PUT_LINE('Cookie support currently disabled.');
      END IF;
   END;
```

Cookie support is transparent; Oracle automatically stores cookies in memory and sends them back to the server when requested.

Cookies disappear when the session ends. If you'd like to make cookies persistent, you can save them into Oracle tables and then restore them when you start a new session. To do this, have a look at the sample code that Oracle provides in the UTL_HTTP section of the *Packages and Types* manual.

To completely disable cookie support for all your UTL_HTTP requests for the remainder of your session, use this code:

```
UTL_HTTP.set_cookie_support (FALSE);
```

To disable cookies for a particular request, specify this:

```
UTL_HTTP.set_cookie_support(req, FALSE);
```

To change the number of cookies from Oracle's default values, specify this:

```
UTL_HTTP.set_cookie_support(TRUE,
   max_cookies => n,
   max_cookies_per_site => m);
```

Retrieve Data from an FTP Server

Oracle does not provide out-of-the-box support for retrieving data from FTP sites via PL/SQL. However, if you need to send or receive files via FTP, there are several PL/SQL solutions available on the Internet. I've seen at least three different packages, authored, respectively, by Barry Chase, Tim Hall, and Chris Poole. These implementations typically use UTL_TCP and UTL_FILE (and possibly Java), and support most of the commonly used FTP operations. You can find a link to some of these implementations by visiting PLNet.org.

In addition, some proxy servers support the retrieval of FTP via HTTP requests from the client, so you may be able to live without a true FTP package.

Use a Proxy Server

For a variety of reasons, it is common in the corporate world for the network to force all web traffic through a proxy server. Fortunately, Oracle includes support for this kind of arrangement in UTL_HTTP. For example, if your proxy is running on port 8888 at 10.2.1.250, you can use the following:

```
DECLARE
   req UTL_HTTP.req;
   resp UTL_HTTP.resp;
BEGIN
```

```
UTL_HTTP.set_proxy(proxy => '10.2.1.250:8888',
    no_proxy_domains => 'mycompany.com, hr.mycompany.com');

req := UTL_HTTP.begin_request('http://some-remote-site.com');

/* If your proxy requires authentication, use this: */
UTL_HTTP.set_authentication(r => req,
   username => 'username',
   password => 'password',
   for_proxy => TRUE);

resp := UTL_HTTP.get_response(req);... etc.
```

I happened to test this code on a proxy server that uses Microsoft NTLM–based authentication. After an embarrassing amount of trial and error, I discovered that I had to prefix my username with the Microsoft server "domain name" plus a backslash. That is, if I normally log into the NTLM domain "mis" as user *bill* with password *swordfish*, I must specify:

```
username => 'mis\bill', password => 'swordfish'
```

Other Types of I/O Available in PL/SQL

This chapter has focused on some of the types of I/O that I think are most useful in the real world and that aren't well covered elsewhere. But what about these other types of I/O?

- Database pipes, queues, and alerts
- TCP sockets
- Oracle's built-in web server

Database Pipes, Queues, and Alerts

The DBMS_PIPE built-in package was originally designed as an efficient means of sending small bits of data between separate Oracle database sessions. With the introduction of autonomous transactions, database pipes are no longer needed if they are simply being used to isolate transactions from each other. Database pipes can also be used to manually parallelize operations.

Database queuing is a way to pass messages asynchronously among Oracle sessions. There are many variations on queuing: single versus multiple producers, single versus multiple consumers, limited-life messages, priorities, and more. The latest incarnation of Oracle's queuing features is covered in the Oracle manual called *Oracle Streams Advanced Queuing User's Guide*.

The DBMS_ALERT package allows synchronous notification to multiple sessions that various database events have occurred. My impression is that this feature is rarely used today; Oracle provides other products that fill a similar need but with more features.

You can read more about pipes and alerts in Chapter 3 of *Oracle Built-in Packages*, "Intersession Communication." For your convenience, that chapter is posted on this book's website.

TCP Sockets

As interesting a subject as low-level network programming may be to the geeks among us (including yours truly), it's just not a widely used feature. In addition to the UTL_TCP built-in package, Oracle also supports invocation of the networking features in Java stored procedures, which you can invoke from PL/SQL.

Oracle's Built-in Web Server

Even if you haven't licensed the Oracle Application Server product, you still have access to an HTTP server built into the Oracle database. Configuration of the built-in server varies according to Oracle version, but the PL/SQL programming side of it—including the OWA_UTIL, HTP, and HTF packages—has remained relatively unchanged.

These packages let you generate database-driven web pages directly from PL/SQL. This is a fairly extensive topic, particularly if you want to generate and process HTML forms in your web page—not to mention the fact that HTTP is a stateless protocol, so you don't really get to set and use package-level variables from one call to the next. O'Reilly's *Learning Oracle PL/SQL* provides an introduction to PL/SQL that makes heavy use of the built-in web server and provides a number of code samples. The PL/SQL coding techniques are also applicable if you happen to be using Oracle's separate, full-blown application server product; for more information about this product, see *Oracle Application Server 10g Essentials* by Rick Greenwald, Robert Stackowiak, and Donald Bales.

Although not an I/O method per se, Oracle Application Express, also known as Oracle APEX, deserves one final mention. This is a free add-on to the Oracle Database that lets you build full-blown web-based applications that connect to an Oracle database. PL/SQL programmers can write their own stored programs that integrate into the GUI framework that Oracle APEX includes, which provides many convenient tools for exchanging data via a visible user interface.