ORACLE

Users' Guide

Relational Software Incorporated

RSI-100

R S I

O R A C L E

U S E R S - G U I D E

Oracle Users Guide - Version 2.3

R S I

ORACLE

SQL LANGUAGE

USER'S GUIDE

Oracle User's Guide - Version 2.3

S Q L

LANGUAGE USER'S GUIDE

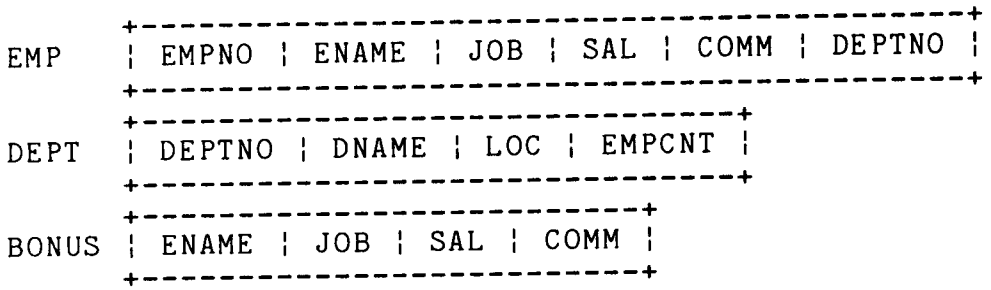
TABLE OF CONTENTS

SQL LANGUAGE - USER'S GUIDE

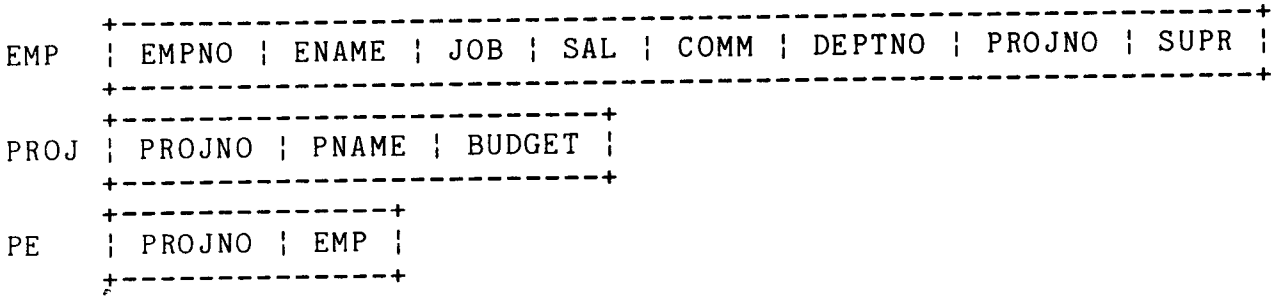
SQL LANGUAGE - EXAMPLES

INTRODUCTION

This section of the "User's Guide" introduces the SQL facilities for query, data manipulation, data definition and data control through a series of examples. The examples are based on the following database:



The EMP table contains information on employees, giving the employee's number, name, job title, salary, commission, and department number. The department table gives the department's number, name, location, and a count of the employees in the department. The BONUS table contains an extract of the information in the EMP table. Through the course of the examples, the EMP table is expanded to contain a project number column and a supervisor column, and two new tables are added. The PROJ table contains columns for the project number, name, and budget. The PE table relates projects to employees where one project can have many employees, and one employee can be working on many projects. The expanded EMP table and the two added tables are as follows:



The examples in this section of the documentation were produced by executing the ORACLE User Friendly Interface (UFI) and capturing the output in hard-copy form. The actual output has been post-processed to add page numbers and to remove the SQL prompts at the beginning of each line. The command file which produces this output is provided with each ORACLE installation. It contains the comments interspersed with the SQL statements. This section of the documentation serves as a self-tutorial in the facilities of SQL as implemented in ORACLE.

ORACLE accepts SQL statements in free format. The arrangement of each SQL clause on a separate line and indentation in the following examples is used for clarity only.

1. DATA BASE CONTENTS

This section of the manual describes the structure and content of the example PERSONNEL data base.

1.1 Dictionary Contents

ORACLE's integrated data dictionary can be queried using standard SQL query facilities.

Example 1-1: List the names of the user tables in the data base.

```
SQL>SELECT  TABLE
SQL>FROM    TAB;
SQL>/

TABLE
----------
EMP
DEPT
BONUS
```

Example 1-2: List the names of the columns of the department and employee tables.

```
SQL>SELECT  *
SQL>FROM    COL
SQL>WHERE   TABLE = 'DEPT';
SQL>/

TABLE       COLUMN
----------- -------
DEPT        DEPTNO
DEPT        DNAME
DEPT        LOC
DEPT        EMPCNT

SQL>SELECT  *
SQL>FROM    COL
SQL>WHERE   TABLE = 'EMP';
SQL>/

TABLE       COLUMN
----------- -------
EMP         EMPNO
EMP         ENAME
EMP         JOB
EMP         SAL
EMP         COMM
EMP         DEPTNO

6 records selected.
```

1.2 **Data Base Contents**

Example 1-3: List all the columns and rows of the DEPT table.

```
SQL>SELECT * FROM DEPT;
SQL>/

DEPTNO DNAME          LOC           EMPCNT
------ -------------- ------------- ------
    10 ADMINISTRATION NEW YORK
    20 RESEARCH       SAN FRANCISCO
    30 SALES          CHICAGO
    40 OPERATIONS     BOSTON
```

Example 1-4: List all the columns and rows of the EMP table.

```
SQL>SELECT * FROM EMP;
SQL>/

EMPNO ENAME  JOB              SAL       COMM DEPTNO
----- ------ ---------- ---------- ---------- ------
 7369 SMITH  CLERK         $800.00                20
 7499 ALLEN  SALESMAN    $1,600.00    $300.00     30
 7521 WARD   SALESMAN    $1,250.00    $500.00     30
 7566 JONES  MANAGER     $2,975.00                20
 7654 MARTIN SALESMAN    $1,250.00  $1,400.00     30
 7698 BLAKE  MANAGER     $2,850.00                30
 7782 CLARK  MANAGER     $2,450.00                10
 7788 SCOTT  ANALYST     $3,000.00                20
 7839 OATES  PRESIDENT   $5,000.00                10
 7844 TURNER SALESMAN    $1,500.00      $0.00     30
 7876 ADAMS  CLERK       $1,100.00                20
 7900 JAMES  CLERK         $950.00                30
 7902 FORD   ANALYST     $3,000.00                20
 7934 MILLER CLERK       $1,300.00                10

14 records selected.
```

2. QUERY FACILITIES

This section of the manual contains a description of the query facilities of ORACLE.

2.1 Query Block

The SELECT clause lists the columns to be returned. The FROM clause lists the tables involved in the query. The WHERE clause specifies the selection criteria.

Example 2-1: Find the name of department 10.

```
SQL>SELECT  DNAME
SQL>FROM    DEPT
SQL>WHERE   DEPTNO=10;
SQL>/

DNAME
--------------
ADMINISTRATION
```

The SELECT clause can contain several columns. Character string constants are enclosed by single quotation marks.

Example 2-2: List the names, numbers, and departments of all clerks.

```
SQL>SELECT  ENAME,EMPNO,DEPTNO
SQL>FROM    EMP
SQL>WHERE   JOB = 'CLERK';
SQL>/

ENAME  EMPNO DEPTNO
------ ----- ------
SMITH   7369     20
ADAMS   7876     20
MILLER  7934     10
JAMES   7900     30
```

If all columns of the row are to be returned, SELECT * is specified.

Example 2-3: List all the columns in the employee table for employees in department 30.

```
SQL>SELECT  *
SQL>FROM    EMP
SQL>WHERE   DEPTNO = 30;
SQL>/
```

EMPNO	ENAME	JOB	SAL	COMM	DEPTNO
7499	ALLEN	SALESMAN	$1,600.00	$300.00	30
7521	WARD	SALESMAN	$1,250.00	$500.00	30
7698	BLAKE	MANAGER	$2,850.00		30
7654	MARTIN	SALESMAN	$1,250.00	$1,400.00	30
7844	TURNER	SALESMAN	$1,500.00	$0.00	30
7900	JAMES	CLERK	$950.00		30

6 records selected.

The WHERE clause can compare two fields of a row with each other.

Example 2-4: List the name, salary, and commission of each employee whose commission is greater than his salary.

```
SQL>SELECT  ENAME,SAL,COMM
SQL>FROM    EMP
SQL>WHERE   COMM > SAL;
SQL>/
```

ENAME	SAL	COMM
MARTIN	$1,250.00	$1,400.00

The absence of a WHERE clause causes all rows to be returned.

Example 2-5: List all columns and all rows in the DEPT table.

```
SQL>SELECT  *
SQL>FROM    DEPT;
SQL>/

DEPTNO DNAME          LOC           EMPCNT
------ -------------- ------------- ------
    10 ADMINISTRATION NEW YORK
    20 RESEARCH       SAN FRANCISCO
    30 SALES          CHICAGO
    40 OPERATIONS     BOSTON
```

2.2 **Logical Expressions**

Predicates within the WHERE clause may be connected by the boolean operators, AND and OR.

Example 2-6: List the name, job title, and salary of all employees in department 20 that make more than $2,000.

```
SQL>SELECT  ENAME,JOB,SAL
SQL>FROM    EMP
SQL>WHERE   DEPTNO = 20
SQL>  AND   SAL > 2000;
SQL>/

ENAME  JOB                 SAL
------ ---------- ----------
JONES  MANAGER     $2,975.00
SCOTT  ANALYST     $3,000.00
FORD   ANALYST     $3,000.00
```

The BETWEEN operator simplifies the syntax for specifying a range. The predicates SAL >= 1200 AND SAL <= 1400 are simplified to SAL BETWEEN 1200 AND 1400.

Example 2-7: List the name, job title, and salary of all employees who earn between $1,200 AND $1,400.

```
SQL>SELECT  ENAME,JOB,SAL
SQL>FROM    EMP
SQL>WHERE   SAL BETWEEN 1200 AND 1400;
SQL>/

ENAME  JOB              SAL
------ ---------- ----------
WARD   SALESMAN    $1,250.00
MARTIN SALESMAN    $1,250.00
MILLER CLERK       $1,300.00
```

Example 2-8: List the name, job, salary, and commission of each employee whose job title begins with MAN or whose salary is greater than 3000 a month.

```
SQL>SELECT  ENAME,JOB,SAL,COMM
SQL>FROM    EMP
SQL>WHERE   JOB = 'MAN...'
SQL>  OR    SAL > 3000;
SQL>/

ENAME  JOB              SAL        COMM
------ ---------- ---------- ----------
JONES  MANAGER     $2,975.00
BLAKE  MANAGER     $2,850.00
CLARK  MANAGER     $2,450.00
OATES  PRESIDENT   $5,000.00
```

Example 2-9: List the department number, name, job title, salary and commission of each employee in department 30 whose salary is greater than his commission.

```
SQL>SELECT  DEPTNO,ENAME,JOB,SAL,COMM
SQL>FROM    EMP
SQL>WHERE   SAL > COMM
SQL>  AND   DEPTNO = 30;
SQL>/

DEPTNO ENAME  JOB              SAL        COMM
------ ------ ---------- ---------- ----------
    30 ALLEN  SALESMAN    $1,600.00    $300.00
    30 WARD   SALESMAN    $1,250.00    $500.00
    30 TURNER SALESMAN    $1,500.00      $0.00
```

Predicates within a WHERE clause form logical expressions with square brackets [ ] used to establish precedence.

Example 2-10: List the department number, name, job title, salary, and commission of all analysts, or all people in department 10 who earn more than $2,500.

```
SQL>SELECT  DEPTNO,ENAME,JOB,SAL,COMM
SQL>FROM    EMP
SQL>WHERE   JOB = 'ANALYST'
SQL>   OR   [SAL > 2500 AND DEPTNO = 10];
SQL>/

DEPTNO ENAME  JOB              SAL        COMM
------ ------ ---------- ---------- ----------
    20 SCOTT  ANALYST     $3,000.00
    20 FORD   ANALYST     $3,000.00
    10 OATES  PRESIDENT   $5,000.00
```

Example 2-11: List the department number, name, job title, salary, and commission of employees in department 10 who are either analysts or earn more than $2,500.

```
SQL>SELECT  DEPTNO,ENAME,JOB,SAL,COMM
SQL>FROM    EMP
SQL>WHERE  [JOB = 'ANALYST'
SQL>   OR   SAL > 2500]  AND  DEPTNO = 10;
SQL>/

DEPTNO ENAME  JOB              SAL        COMM
------ ------ ---------- ---------- ----------
    10 OATES  PRESIDENT   $5,000.00
```

2.3 **Not Conditions**

Predicates within a WHERE clause may be negated.

Example 2-12: List the name, salary, commission, job title, and department number of employees in department 30 who are not a salesman.

```
SQL>SELECT  ENAME,SAL,COMM,JOB,DEPTNO
SQL>FROM    EMP
SQL>WHERE   JOB ^=  'SALESMAN'
SQL>  AND   DEPTNO = 30;
SQL>/
```

ENAME	SAL	COMM	JOB	DEPTNO
BLAKE	$2,850.00		MANAGER	30
JAMES	$950.00		CLERK	30

Entire logical expressions may be negated.

Example 2-13: List the name, salary, commission, job title, and department number of employees in department 30 who are not a salesman or do not earn more than $1,500.

```
SQL>SELECT  ENAME,SAL,COMM,JOB,DEPTNO
SQL>FROM    EMP
SQL>WHERE   NOT [JOB = 'SALESMAN' OR SAL > 1500]
SQL>  AND   DEPTNO = 30;
SQL>/
```

ENAME	SAL	COMM	JOB	DEPTNO
JAMES	$950.00		CLERK	30

2.4 Set Inclusion Operator

A predicate in a WHERE clause may test a field for inclusion in a set of constant literal values. A set of constant literal values is enclosed within angle brackets < >.

Example 2-14: List the name and department number of employees in departments 10 and 30.

```
SQL>SELECT  ENAME,DEPTNO
SQL>FROM    EMP
SQL>WHERE   DEPTNO IN <10,30>;
SQL>/

ENAME  DEPTNO
------ ------
CLARK      10
MILLER     10
OATES      10
ALLEN      30
WARD       30
BLAKE      30
MARTIN     30
TURNER     30
JAMES      30

9 records selected.
```

Example 2-15: List all fields from the department table for departments that are located in either Chicago or New York.

```
SQL>SELECT  *
SQL>FROM    DEPT
SQL>WHERE   LOC IN <'CHICAGO','NEW YORK'>;
SQL>/

DEPTNO DNAME          LOC           EMPCNT
------ -------------- ------------- ------
    30 SALES          CHICAGO
    10 ADMINISTRATION NEW YORK
```

2.5 **Nested Query**

The result of one query may be used in the WHERE clause of another query. The inner query returns one or a set of values. The outer query uses this result as if it were given a set of constant literal values. Query blocks may be nested to any number of levels.

Example 2-16: List the name and job of employees who have the same job as Jones.

```
SQL>SELECT  ENAME,JOB
SQL>FROM    EMP
SQL>WHERE   JOB IN
SQL>        SELECT  JOB
SQL>        FROM    EMP
SQL>        WHERE   ENAME = 'JONES';
SQL>/

ENAME  JOB
------ ----------
JONES  MANAGER
BLAKE  MANAGER
CLARK  MANAGER
```

Example 2-17: List the name, job title, and salary of employees who have the same job and salary as Ford.

```
SQL>SELECT  ENAME,JOB,SAL
SQL>FROM    EMP
SQL>WHERE   <JOB,SAL> =
SQL>        SELECT  JOB,SAL
SQL>        FROM    EMP
SQL>        WHERE   ENAME = 'FORD';
SQL>/

ENAME  JOB              SAL
------ ---------- ----------
SCOTT  ANALYST     $3,000.00
FORD   ANALYST     $3,000.00
```

Inner query blocks may be connected by the boolean operators AND and OR to form compound nested queries.

Example 2-18: List the name, job, and department of employees who have the same job as Jones, or a salary greater than or equal to Ford.

```
SQL>SELECT  ENAME,JOB,DEPTNO,SAL
SQL>FROM    EMP
SQL>WHERE   JOB IN
SQL>        SELECT  JOB
SQL>        FROM    EMP
SQL>        WHERE   ENAME = 'JONES';
SQL>OR SAL >=
SQL>        SELECT  SAL
SQL>        FROM    EMP
SQL>        WHERE   ENAME = 'FORD';
SQL>/
```

ENAME	JOB	DEPTNO	SAL
JONES	MANAGER	20	$2,975.00
BLAKE	MANAGER	30	$2,850.00
CLARK	MANAGER	10	$2,450.00
SCOTT	ANALYST	20	$3,000.00
OATES	PRESIDENT	10	$5,000.00
FORD	ANALYST	20	$3,000.00

6 records selected.

2.6 **ORDER BY**

The ORDER BY clause specifies major and minor sort fields in ascending or descending order. Ascending order is default.

Example 2-19: List the name, job, department, and employee number of employees in a department whose number is greater than or equal to 20, in order of employee name.

```
SQL>SELECT  ENAME,JOB,DEPTNO,EMPNO
SQL>FROM    EMP
SQL>WHERE   DEPTNO >= 20
SQL>ORDER BY ENAME;
SQL>/

ENAME  JOB         DEPTNO EMPNO
------ ---------- ------ -----
ADAMS  CLERK           20  7876
ALLEN  SALESMAN        30  7499
BLAKE  MANAGER         30  7698
FORD   ANALYST         20  7902
JAMES  CLERK           30  7900
JONES  MANAGER         20  7566
MARTIN SALESMAN        30  7654
SCOTT  ANALYST         20  7788
SMITH  CLERK           20  7369
TURNER SALESMAN        30  7844
WARD   SALESMAN        30  7521

11 records selected.
```

Example 2-20: List the department, salary, name, and job of all employees, in descending order by salary, ascending order by job within salary, and ascending order by name within job.

```
SQL>SELECT  DEPTNO,SAL,JOB,ENAME
SQL>FROM    EMP
SQL>ORDER BY SAL DESC,JOB,ENAME;
SQL>/
```

DEPTNO	SAL	JOB	ENAME
10	$5,000.00	PRESIDENT	OATES
20	$3,000.00	ANALYST	FORD
20	$3,000.00	ANALYST	SCOTT
20	$2,975.00	MANAGER	JONES
30	$2,850.00	MANAGER	BLAKE
10	$2,450.00	MANAGER	CLARK
30	$1,600.00	SALESMAN	ALLEN
30	$1,500.00	SALESMAN	TURNER
10	$1,300.00	CLERK	MILLER
30	$1,250.00	SALESMAN	MARTIN
30	$1,250.00	SALESMAN	WARD
20	$1,100.00	CLERK	ADAMS
30	$950.00	CLERK	JAMES
20	$800.00	CLERK	SMITH

14 records selected.

Expressions can be specified within the ORDER BY clause.

Example 2-21: List all salesman in ascending order of the ratio of their salary divided by their commission.

```
SQL>SELECT  ENAME,SAL/COMM,SAL,COMM
SQL>FROM    EMP
SQL>WHERE   JOB = 'SALESMAN'
SQL>ORDER BY SAL/COMM
SQL>/
```

ENAME	SAL/COMM	SAL	COMM
MARTIN	.893	$1,250.00	$1,400.00
WARD	2.500	$1,250.00	$500.00
ALLEN	5.333	$1,600.00	$300.00
TURNER	~.000	$1,500.00	$0.00

2.7 **UNIQUE**

A query returns a set of rows that satisfy the WHERE clause. Duplicate rows are not eliminated unless SELECT UNIQUE is specified.

Example 2-22: List all the different jobs in the job table.

```
SQL>SELECT  UNIQUE JOB
SQL>FROM    EMP;
SQL>/

JOB
----------
CLERK
SALESMAN
MANAGER
ANALYST
PRESIDENT
```

If a WHERE clause has multiple predicates connected by an OR, there exists the possibility that a single row may satisfy both predicates and be returned twice in the query result. UNIQUE specified within the SELECT clause eliminates this duplication.

Example 2-23: List the name and job of employees who are in department 30, or employees who are managers. Sort by employee name.

```
SQL>SELECT  ENAME,JOB,DEPTNO
SQL>FROM    EMP
SQL>WHERE   DEPTNO = 30
SQL>   OR   JOB = 'MANAGER'
SQL>ORDER BY ENAME;
SQL>/

ENAME  JOB         DEPTNO
------ ---------- ------
ALLEN  SALESMAN        30
BLAKE  MANAGER         30
BLAKE  MANAGER         30
CLARK  MANAGER         10
JAMES  CLERK           30
JONES  MANAGER         20
MARTIN SALESMAN        30
TURNER SALESMAN        30
WARD   SALESMAN        30

9 records selected.
```

Example 2-24: List the name and job of employees who are in department 30 or, employees who are managers. Eliminate duplicate rows and sort the result by employee name.

```
SQL>SELECT  UNIQUE ENAME,JOB,DEPTNO
SQL>FROM    EMP
SQL>WHERE   DEPTNO = 30
SQL>  OR    JOB = 'MANAGER'
SQL>ORDER BY ENAME;
SQL>/

ENAME  JOB         DEPTNO
------ ---------- ------
ALLEN  SALESMAN        30
BLAKE  MANAGER         30
CLARK  MANAGER         10
JAMES  CLERK           30
JONES  MANAGER         20
MARTIN SALESMAN        30
TURNER SALESMAN        30
WARD   SALESMAN        30

8 records selected.
```

Example 2-25: List the salary, job title, name, and department number for all employees in departments that have salesmen. Sort the results by salary in descending order.

```
SQL>SELECT  SAL,JOB,ENAME,DEPTNO
SQL>FROM    EMP
SQL>WHERE   DEPTNO IN
SQL>        SELECT  UNIQUE DEPTNO
SQL>        FROM    EMP
SQL>        WHERE   JOB = 'SALESMAN';
SQL>ORDER BY SAL DESC;
SQL>/

       SAL JOB         ENAME  DEPTNO
---------- ---------- ------ ------
 $2,850.00 MANAGER    BLAKE      30
 $1,600.00 SALESMAN   ALLEN      30
 $1,500.00 SALESMAN   TURNER     30
 $1,250.00 SALESMAN   WARD       30
 $1,250.00 SALESMAN   MARTIN     30
   $950.00 CLERK      JAMES      30

6 records selected.
```

2.8 **Arithmetic Expressions**

The SELECT, WHERE, and HAVING clauses may all contain arithmetic expressions containing fields and constants.

Example 2-26: List the name, salary, commission, and sum of salary plus commission of employees in department 30.

```
SQL>SELECT  ENAME,SAL,COMM,SAL + COMM
SQL>FROM    EMP
SQL>WHERE   DEPTNO = 30;
SQL>/
```

ENAME	SAL	COMM	SAL+COMM
ALLEN	$1,600.00	$300.00	$1,900.00
WARD	$1,250.00	$500.00	$1,750.00
BLAKE	$2,850.00		
MARTIN	$1,250.00	$1,400.00	$2,650.00
TURNER	$1,500.00	$0.00	$1,500.00
JAMES	$950.00		

6 records selected.

Example 2-27: List the name, salary, and commission of employees whose commission is greater than or equal to 25% of their salary.

```
SQL>SELECT  ENAME,SAL,COMM
SQL>FROM    EMP
SQL>WHERE   COMM >= 0.25 * SAL;
SQL>/
```

ENAME	SAL	COMM
WARD	$1,250.00	$500.00
MARTIN	$1,250.00	$1,400.00

Parentheses are used to establish precedence within arithmetic expressions.

Example 2-28: List the name, salary, commission, and 1.25 times salary plus two-thirds of the commission of all salesmen.

```
SQL>SELECT  ENAME,SAL,COMM,((SAL * 1.25) + (COMM * ( 2/3 )))
SQL>FROM    EMP
SQL>WHERE   JOB = 'SALES...';
SQL>/
```

ENAME	SAL	COMM	((SAL*1.25)+(COMM*(2/3)))
ALLEN	$1,600.00	$300.00	$2,200.00
WARD	$1,250.00	$500.00	$1,895.83
MARTIN	$1,250.00	$1,400.00	$2,495.83
TURNER	$1,500.00	$0.00	$1,875.00

2.9 **Built-In Functions**

ORACLE provides several built-in functions that may be used in either SELECT or HAVING clauses.

Example 2-29: Find the average salary of all employees who are clerks.

```
SQL>SELECT  AVG(SAL)
SQL>FROM    EMP
SQL>WHERE   JOB = 'CLERK';
SQL>/

  AVG(SAL)
----------
 $1,037.50
```

Example 2-30: Find the maximum, average, and minimum salary of employees in department 10.

```
SQL>SELECT  MAX(SAL),AVG(SAL),MIN(SAL)
SQL>FROM    EMP
SQL>WHERE   DEPTNO = 10;
SQL>/

  MAX(SAL)   AVG(SAL)   MIN(SAL)
---------- ---------- ----------
 $5,000.00  $2,916.67  $1,300.00
```

Example 2-31: Find the sum of all salesmen's commissions.

```
SQL>SELECT  SUM(COMM)
SQL>FROM    EMP
SQL>WHERE   JOB = 'SALESMAN';
SQL>/

 SUM(COMM)
----------
 $2,200.00
```

Example 2-32: Find the number of employees in department 30.

```
SQL>SELECT  COUNT(*)
SQL>FROM    EMP
SQL>WHERE   DEPTNO = 30;
SQL>/

COUNT(*)
--------
       6
```

Built-in functions can be used in arithmetic expressions.

Example 2-33: Compute the average annual salary plus commission for all salesmen.

```
SQL>SELECT  AVG(SAL + COMM) * 12
SQL>FROM    EMP
SQL>WHERE   JOB = 'SALESMAN'
SQL>/

AVG(SAL+COMM)*12
----------------
      $23,400.00
```

ORACLE allows functions to be applied to the results of other built-in functions.

Example 2-34: List the name, job, and salary of the employee who has the largest salary.

```
SQL>SELECT  ENAME,JOB,SAL
SQL>FROM    EMP
SQL>WHERE   SAL =
SQL>        SELECT  MAX(SAL)
SQL>        FROM    EMP;
SQL>/

ENAME  JOB                 SAL
------ ---------- ----------
OATES  PRESIDENT   $5,000.00
```

2.10 **GROUP-BY**

A table may be partitioned into groups according to the values in a column or set of columns. A built-in function may then be applied to each group. When a built-in function is used, each item in the SELECT clause must be a unique property of the group.

Example 2-35: List the department number and average salary of each department.

```
SQL>SELECT  DEPTNO,AVG(SAL)
SQL>FROM    EMP
SQL>GROUP BY DEPTNO;
SQL>/

DEPTNO   AVG(SAL)
------ ----------
    10  $2,916.67
    20  $2,175.00
    30  $1,566.67
```

Example 2-36: List the department number and average annual salary of each departments employees, excluding managers salaries.

```
SQL>SELECT  DEPTNO,AVG(SAL) * 12
SQL>FROM    EMP
SQL>WHERE   NOT JOB = 'MAN...'
SQL>GROUP BY DEPTNO;
SQL>/

DEPTNO AVG(SAL)*12
------ -----------
    10  $37,800.00
    20  $23,700.00
    30  $15,720.00
```

A table can be partitioned into groups based on the values in more than one column.

Example 2-37: Divide all employees into groups by department, and by job within department. Count the employees in each group and compute each group's average salary.

```
SQL>SELECT  DEPTNO,JOB,COUNT(*),AVG(SAL) * 12
SQL>FROM    EMP
SQL>GROUP BY DEPTNO,JOB;
SQL>/

DEPTNO JOB         COUNT(*) AVG(SAL)*12
------ ---------- -------- -----------
    10 CLERK             1  $15,600.00
    10 MANAGER           1  $29,400.00
    10 PRESIDENT         1  $60,000.00
    20 ANALYST           2  $36,000.00
    20 CLERK             2  $11,400.00
    20 MANAGER           1  $35,700.00
    30 CLERK             1  $11,400.00
    30 MANAGER           1  $34,200.00
    30 SALESMAN          4  $16,800.00

9 records selected.
```

Built-in functions can be applied to the results of other group functions to form functions of functions.

Example 2-38: Total the salaries of all the departments and list the department with the maximum total.

```
SQL>SELECT  DEPTNO,MAX(SUM(SAL))
SQL>FROM    EMP
SQL>GROUP BY DEPTNO;
SQL>/

DEPTNO MAX(SUM(SAL))
------ -------------
    30    $10,875.00
```

2.11 **HAVING**

After a table has been partitioned into groups, a predicate or set of predicates in a HAVING clause can be applied to the groups.

Example 2-39: List the average annual salary for all job groups having more than 2 employees in the group.

```
SQL>SELECT  JOB,AVG(SAL) * 12
SQL>FROM    EMP
SQL>GROUP BY JOB
SQL>HAVING  COUNT(*) > 2;
SQL>/

JOB         AVG(SAL)*12
----------  -----------
CLERK        $12,450.00
MANAGER      $33,100.00
SALESMAN     $16,800.00
```

A query block may contain both a WHERE and HAVING clause. First, the WHERE clause is applied to qualify rows; second, the groups are formed and the built-in functions are computed; third, the HAVING clause is applied to qualify groups.

Example 2-40: List all the departments that have more than two clerks.

```
SQL>SELECT  DEPTNO
SQL>FROM    EMP
SQL>WHERE   JOB = 'CLERK'
SQL>GROUP BY DEPTNO
SQL>HAVING COUNT(*) >= 2;
SQL>/

DEPTNO
------
    20
```

HAVING clauses may contain query blocks.

Example 2-41: List the departments and their average salary that have a greater average salary than department 20.

```
SQL>SELECT  DEPTNO,AVG(SAL)
SQL>FROM    EMP
SQL>GROUP BY DEPTNO
SQL>HAVING  AVG(SAL) >
SQL>        SELECT  AVG(SAL)
SQL>        FROM    EMP
SQL>        WHERE   DEPTNO = 20;
SQL>/

DEPTNO   AVG(SAL)
------ ----------
    10  $2,916.67
```

2.12 **NULL Conditions**

Predicates within a WHERE clause can explicitly test for null conditions within a column.

Example 2-42: List the name, salary, commission and job title of all employees who do not receive commissions.

```
SQL>SELECT  ENAME,SAL,COMM,JOB
SQL>FROM    EMP
SQL>WHERE   COMM = NULL;
SQL>/

ENAME          SAL       COMM JOB
------ ---------- ---------- ----------
SMITH     $800.00            CLERK
JONES   $2,975.00            MANAGER
BLAKE   $2,850.00            MANAGER
CLARK   $2,450.00            MANAGER
SCOTT   $3,000.00            ANALYST
OATES   $5,000.00            PRESIDENT
ADAMS   $1,100.00            CLERK
JAMES     $950.00            CLERK
FORD    $3,000.00            ANALYST
MILLER  $1,300.00            CLERK

10 records selected.
```

Example 2-43: List the name, salary, commission, and job title of those employees who receive a commission.

```
SQL>SELECT  ENAME,SAL,COMM,JOB
SQL>FROM    EMP
SQL>WHERE   NOT COMM = NULL;
SQL>/

ENAME          SAL        COMM JOB
------ ---------- ---------- ----------
ALLEN   $1,600.00    $300.00 SALESMAN
WARD    $1,250.00    $500.00 SALESMAN
MARTIN  $1,250.00  $1,400.00 SALESMAN
TURNER  $1,500.00      $0.00 SALESMAN
```

Example 2-44: If a predicate within a WHERE or HAVING clause can be expressed without using NOT or NULL, it is usually more efficient.

```
SQL>SELECT  ENAME,SAL,COMM,JOB
SQL>FROM    EMP
SQL>WHERE   COMM >= 0;
SQL>/

ENAME          SAL        COMM JOB
------ ---------- ---------- ----------
ALLEN   $1,600.00    $300.00 SALESMAN
WARD    $1,250.00    $500.00 SALESMAN
MARTIN  $1,250.00  $1,400.00 SALESMAN
TURNER  $1,500.00      $0.00 SALESMAN
```

Null values in the data base are treated as unknowns in the evaluation of logical expressions. Only those rows that are known to satisfy the WHERE clause are returned as the result of a query.

Example 2-45: List all columns of the employee table for employees in department 30 or employees whose commission is less than or equal to $1,000.

```
SQL>SELECT  *
SQL>FROM    EMP
SQL>WHERE   DEPTNO = 30
SQL>  AND   COMM <= 1000;
SQL>/
```

EMPNO	ENAME	JOB	SAL	COMM	DEPTNO
7499	ALLEN	SALESMAN	$1,600.00	$300.00	30
7521	WARD	SALESMAN	$1,250.00	$500.00	30
7844	TURNER	SALESMAN	$1,500.00	$0.00	30

Example 2-46: List all the columns of the employee table for employees in department 30 or employees whose commission is less than or equal to $1,000.

```
SQL>SELECT  *
SQL>FROM    EMP
SQL>WHERE   DEPTNO = 30
SQL>  OR    COMM <= 1000;
SQL>/
```

EMPNO	ENAME	JOB	SAL	COMM	DEPTNO
7499	ALLEN	SALESMAN	$1,600.00	$300.00	30
7521	WARD	SALESMAN	$1,250.00	$500.00	30
7698	BLAKE	MANAGER	$2,850.00		30
7654	MARTIN	SALESMAN	$1,250.00	$1,400.00	30
7844	TURNER	SALESMAN	$1,500.00	$0.00	30
7900	JAMES	CLERK	$950.00		30
7499	ALLEN	SALESMAN	$1,600.00	$300.00	30
7521	WARD	SALESMAN	$1,250.00	$500.00	30
7844	TURNER	SALESMAN	$1,500.00	$0.00	30

9 records selected.

2.13 **NULL-Function**

When an expression or built-in function references a column of a table that contains one or more null values, the result of the expression or built-in function is null.

Example 2-47: In the following example the expression SAL + COMM returns a null value for all employees that have a null commmission.

```
SQL>SELECT  ENAME,JOB,SAL,COMM,SAL + COMM
SQL>FROM    EMP
SQL>WHERE   DEPTNO = 30
SQL>/
```

ENAME	JOB	SAL	COMM	SAL+COMM
ALLEN	SALESMAN	$1,600.00	$300.00	$1,900.00
WARD	SALESMAN	$1,250.00	$500.00	$1,750.00
BLAKE	MANAGER	$2,850.00		
MARTIN	SALESMAN	$1,250.00	$1,400.00	$2,650.00
TURNER	SALESMAN	$1,500.00	$0.00	$1,500.00
JAMES	CLERK	$950.00		

6 records selected.

The ORACLE Null-Value Function NVL can be used to assign a temporary value to nulls encountered within an expression.

Example 2-48: Assign null commissions a temporary value of zero within the expression SAL + COMM.

```
SQL>SELECT  ENAME,JOB,SAL,COMM,SAL + NVL(COMM,0)
SQL>FROM    EMP
SQL>WHERE   DEPTNO = 30
SQL>/
```

ENAME	JOB	SAL	COMM	SAL+NVL(COMM,0)
ALLEN	SALESMAN	$1,600.00	$300.00	$1,900.00
WARD	SALESMAN	$1,250.00	$500.00	$1,750.00
BLAKE	MANAGER	$2,850.00		$2,850.00
MARTIN	SALESMAN	$1,250.00	$1,400.00	$2,650.00
TURNER	SALESMAN	$1,500.00	$0.00	$1,500.00
JAMES	CLERK	$950.00		$950.00

6 records selected.

The expression SAL + NVL(COMM,0) will return a value equal to SAL when COMM is null.

Example 2-49: Null values do not participate in the computation of built-in functions.

```
SQL>SELECT  SUM(SAL),COUNT(SAL),SUM(COMM),COUNT(COMM)
SQL>FROM    EMP
SQL>WHERE   DEPTNO = 30
SQL>/

  SUM(SAL) COUNT(SAL)  SUM(COMM) COUNT(COMM)
---------- ---------- ---------- -----------
 $9,400.00          6  $2,200.00           4
```

In the above example the count of people who receive a salary, (4), is greater than the number of people that receive a commission, (6), because null commissions were not counted.

Example 2-50: List the average commission of employees who receive a commission, and the average commission of all employees (treating employees who do not receive a commission as receiving a zero commission).

```
SQL>SELECT  AVG(COMM),AVG(NVL(COMM,0))
SQL>FROM    EMP
SQL>WHERE   DEPTNO = 30
SQL>/

 AVG(COMM) AVG(NVL(COMM,0))
---------- ----------------
   $550.00          $366.67
```

Example 2-51: List the average commission of employees who receive a commission, and the average commission of all employees (treating employees who do not receive a commission as receiving a $1000 commission).

```
SQL>SELECT  AVG(COMM),AVG(NVL(COMM,1000))
SQL>FROM    EMP
SQL>WHERE   DEPTNO = 30
SQL>/

 AVG(COMM) AVG(NVL(COMM,1000))
---------- --------------------
   $550.00              $700.00
```

Example 2-52: For department 30, list the average salary of employees that receive a salary, the average commission of employees that receive a commission, the average salary plus commission of only those employees that receive a commission, and average salary plus commission of all employees including those who do not receive a commission.

```
SQL>SELECT  AVG(SAL),AVG(COMM),AVG(SAL+COMM),AVG(SAL+NVL(COMM,0))
SQL>FROM    EMP
SQL>WHERE   DEPTNO = 30
SQL>/

  AVG(SAL)  AVG(COMM) AVG(SAL+COMM) AVG(SAL+NVL(COMM,0))
---------- ---------- ------------- --------------------
 $1,566.67    $550.00     $1,950.00            $1,933.33
```

2.14 **Join Query**

A query may return values from more than one table. The FROM clause may list several tables. The WHERE clause specifies the relationship on which the tables are to be joined.

Example 2-53: List the names of all employees and the locations of their departments.

```
SQL>SELECT  ENAME,LOC
SQL>FROM    EMP,DEPT
SQL>WHERE   EMP.DEPTNO = DEPT.DEPTNO;
SQL>/

ENAME  LOC
------ -------------
CLARK  NEW YORK
MILLER NEW YORK
OATES  NEW YORK
SMITH  SAN FRANCISCO
JONES  SAN FRANCISCO
ADAMS  SAN FRANCISCO
SCOTT  SAN FRANCISCO
FORD   SAN FRANCISCO
ALLEN  CHICAGO
WARD   CHICAGO
BLAKE  CHICAGO
MARTIN CHICAGO
TURNER CHICAGO
JAMES  CHICAGO

14 records selected.
```

Example 2-54: List names of employees and all the fields of the department table for employees in departments located in Chicago.

```
SQL>SELECT  ENAME,DEPT.*
SQL>FROM    EMP,DEPT
SQL>WHERE   EMP.DEPTNO = DEPT.DEPTNO
SQL>  AND   LOC = 'CHICAGO';
SQL>/

ENAME  DEPTNO DNAME          LOC            EMPCNT
------ ------ -------------- ------------- ------
ALLEN      30 SALES          CHICAGO
WARD       30 SALES          CHICAGO
BLAKE      30 SALES          CHICAGO
MARTIN     30 SALES          CHICAGO
TURNER     30 SALES          CHICAGO
JAMES      30 SALES          CHICAGO

6 records selected.
```

Predicates in a WHERE clause may compare data values in columns from any number of tables in a join query.

Example 2-55: List the names and jobs of employees who are not salesmen and work for departments that are located in Chicago.

```
SQL>SELECT  ENAME,JOB,LOC
SQL>FROM    DEPT,EMP
SQL>WHERE   EMP.DEPTNO=DEPT.DEPTNO
SQL>  AND   LOC = 'CHICAGO'
SQL>  AND   JOB ^= 'SALESMAN';
SQL>/

ENAME  JOB        LOC
------ ---------- -------------
BLAKE  MANAGER    CHICAGO
JAMES  CLERK      CHICAGO
```

Example 2-56: List the name, location, salary, job of employees located in Chicago who have the same job as Allen. Sort the results by employee name.

```
SQL>SELECT  ENAME,LOC,SAL,JOB
SQL>FROM    EMP,DEPT
SQL>WHERE   LOC = 'CHICAGO'
SQL>  AND   EMP.DEPTNO = DEPT.DEPTNO
SQL>  AND   JOB =
SQL>        SELECT  JOB
SQL>        FROM    EMP
SQL>        WHERE   ENAME = 'ALLEN';
SQL>ORDER BY ENAME;
SQL>/

ENAME  LOC                  SAL JOB
------ ------------- ---------- ----------
ALLEN  CHICAGO        $1,600.00 SALESMAN
MARTIN CHICAGO        $1,250.00 SALESMAN
TURNER CHICAGO        $1,500.00 SALESMAN
WARD   CHICAGO        $1,250.00 SALESMAN
```

2.15 **Self-Join**

A table may be joined with itself by listing the same table more than once in the FROM clause and associating a temporary label with each table. This label is used in place of the table name in qualifying references to columns within the SELECT and FROM clauses in the query block.

Example 2-57: For each employee whose salary exceeds his department's managers's salary, list the employee's name and salary and the manager's name and salary. Within the context of this query, the EMP table is treated logically as if it were two separate tables named WORKER and MGR. However, the EMP table is "not" physically duplicated.

```
SQL>SELECT  WORKER.ENAME,WORKER.SAL,MGR.ENAME,MGR.SAL
SQL>FROM    EMP WORKER,EMP MGR
SQL>WHERE   WORKER.DEPTNO = MGR.DEPTNO
SQL>  AND  [MGR.JOB = 'MANAGER'  AND  WORKER.SAL > MGR.SAL];
SQL>/
```

ENAME	SAL	ENAME	SAL
SCOTT	$3,000.00	JONES	$2,975.00
FORD	$3,000.00	JONES	$2,975.00
OATES	$5,000.00	CLARK	$2,450.00

2.16 **Outer-Join**

When the DEPT table is joined to the EMP table using the join predicate DEPT.DEPTNO = EMP.DEPTNO, a department without any employees would not satisfy the join and would not be returned as a result of the query.

Example 2-58: List all the departments in the DEPT table.

```
SQL>SELECT  *
SQL>FROM    DEPT
SQL>/

DEPTNO DNAME          LOC            EMPCNT
------ -------------- ------------- ------
    10 ADMINISTRATION NEW YORK
    20 RESEARCH       SAN FRANCISCO
    30 SALES          CHICAGO
    40 OPERATIONS     BOSTON
```

Example 2-59: List all the employees in the EMP table.

```
SQL>SELECT  DEPTNO,ENAME,JOB
SQL>FROM    EMP
SQL>ORDER BY DEPTNO
SQL>/

DEPTNO ENAME  JOB
------ ------ ----------
    10 CLARK  MANAGER
    10 OATES  PRESIDENT
    10 MILLER CLERK
    20 SMITH  CLERK
    20 JONES  MANAGER
    20 SCOTT  ANALYST
    20 ADAMS  CLERK
    20 FORD   ANALYST
    30 ALLEN  SALESMAN
    30 WARD   SALESMAN
    30 MARTIN SALESMAN
    30 BLAKE  MANAGER
    30 TURNER SALESMAN
    30 JAMES  CLERK

14 records selected.
```

Example 2-60: Join the DEPT table to the EMP table.

```
SQL>SELECT  DEPT.DEPTNO,DNAME,LOC,ENAME,JOB
SQL>FROM    DEPT,EMP
SQL>WHERE   DEPT.DEPTNO = EMP.DEPTNO
SQL>/

DEPTNO DNAME          LOC           ENAME  JOB
------ -------------- ------------- ------ ----------
    10 ADMINISTRATION NEW YORK      CLARK  MANAGER
    10 ADMINISTRATION NEW YORK      MILLER CLERK
    10 ADMINISTRATION NEW YORK      OATES  PRESIDENT
    20 RESEARCH       SAN FRANCISCO SMITH  CLERK
    20 RESEARCH       SAN FRANCISCO JONES  MANAGER
    20 RESEARCH       SAN FRANCISCO ADAMS  CLERK
    20 RESEARCH       SAN FRANCISCO SCOTT  ANALYST
    20 RESEARCH       SAN FRANCISCO FORD   ANALYST
    30 SALES          CHICAGO       ALLEN  SALESMAN
    30 SALES          CHICAGO       WARD   SALESMAN
    30 SALES          CHICAGO       BLAKE  MANAGER
    30 SALES          CHICAGO       MARTIN SALESMAN
    30 SALES          CHICAGO       TURNER SALESMAN
    30 SALES          CHICAGO       JAMES  CLERK

14 records selected.
```

The result of this join does not include department 40 because department 40 does not have any employees. An "outer-join", will return those department rows that have no matching employees.

Example 2-61: List all departments that have employees, plus those departments that do not have employees.

```
SQL>SELECT  DEPT.DEPTNO,DNAME,LOC,ENAME,JOB
SQL>FROM    DEPT,EMP*
SQL>WHERE   DEPT.DEPTNO = EMP.DEPTNO
SQL>/

DEPTNO DNAME          LOC            ENAME  JOB
------ -------------- -------------- ------ ----------
    10 ADMINISTRATION NEW YORK       CLARK  MANAGER
    10 ADMINISTRATION NEW YORK       MILLER CLERK
    10 ADMINISTRATION NEW YORK       OATES  PRESIDENT
    20 RESEARCH       SAN FRANCISCO  SMITH  CLERK
    20 RESEARCH       SAN FRANCISCO  JONES  MANAGER
    20 RESEARCH       SAN FRANCISCO  ADAMS  CLERK
    20 RESEARCH       SAN FRANCISCO  SCOTT  ANALYST
    20 RESEARCH       SAN FRANCISCO  FORD   ANALYST
    30 SALES          CHICAGO        ALLEN  SALESMAN
    30 SALES          CHICAGO        WARD   SALESMAN
    30 SALES          CHICAGO        BLAKE  MANAGER
    30 SALES          CHICAGO        MARTIN SALESMAN
    30 SALES          CHICAGO        TURNER SALESMAN
    30 SALES          CHICAGO        JAMES  CLERK
    40 OPERATIONS     BOSTON

15 records selected.
```

The asterisk (*) after the EMP table in the FROM clause indicates that an extra row containing a null value in every column is to be appended to the EMP table when processing this query block. This null row of the EMP table is joined to those DEPT rows that do not have any matching rows in the EMP table.

Example 2-62: List all departments that do not have any employees.

```
SQL>SELECT  UNIQUE DEPT.DEPTNO,DNAME,LOC
SQL>FROM    DEPT,EMP*
SQL>WHERE   DEPT.DEPTNO = EMP.DEPTNO
SQL> AND    EMPNO = NULL
SQL>/

DEPTNO DNAME          LOC
------ -------------- -------------
    40 OPERATIONS     BOSTON
```

The outer-join can be used to join more than two tables, however, at least one table in the join must not be outer-joined. The table that is not outer-joined must be listed first in the FROM clause.

3. DATA MANIPULATION FACILITIES

3.1 INSERT

The INSERT statement specifies the adding of a new row or set of rows into a table.

Example 3-1: Insert a new employee named Carter with an employee number of 7989, a job title of salesman, salary of 1500, and commission of 0, into department 30.

```
SQL>INSERT  INTO EMP(EMPNO,ENAME,JOB,SAL,COMM,DEPTNO):
SQL>        <7989,'CARTER','SALESMAN',1500,0,30>;
SQL>/
1 record created.
```

All fields do not have to be included in the INSERT statement.

Example 3-2: Insert a new employee named Wilson, employee number 7955, in department 20, having all other fields null.

```
SQL>INSERT  INTO EMP(EMPNO,ENAME,DEPTNO):
SQL>        <7955,'WILSON',20>;
SQL>/
1 record created.
```

If all fields are present in the right order, the list of column names may be omitted.

Example 3-3: Insert a new employee into named Jakes into the EMP table.

```
SQL>INSERT  INTO EMP:
SQL>        <7956,'JAKES','CLERK',1000,NULL,20>;
SQL>/
1 record created.

SQL>SELECT  *
SQL>FROM    EMP
SQL>WHERE   DEPTNO = 20;
SQL>/
```

EMPNO	ENAME	JOB	SAL	COMM	DEPTNO
7369	SMITH	CLERK	$800.00		20
7566	JONES	MANAGER	$2,975.00		20
7876	ADAMS	CLERK	$1,100.00		20
7788	SCOTT	ANALYST	$3,000.00		20
7902	FORD	ANALYST	$3,000.00		20
7955	WILSON				20
7956	JAKES	CLERK	$1,000.00		20

```
7 records selected.
```

An INSERT statement may store the result of a query into an existing table.

Example 3-4: Add to the BONUS table all those employees whose commission is greater than 25% of their salary, or those employees who have the job title of president or manager.

```
SQL>INSERT INTO BONUS:
SQL>         SELECT  ENAME,JOB,SAL,COMM
SQL>         FROM    EMP
SQL>         WHERE   COMM > 0.25 * SAL
SQL>           OR    JOB IN <'PRESIDENT','MANAGER'>;
SQL>/
6 records created.
```

Example 3-5: List the BONUS table.

```
SQL>SELECT  *
SQL>FROM    BONUS;
SQL>/

ENAME  JOB                 SAL        COMM
------ ---------- ---------- ----------
BLAKE  MANAGER     $2,850.00
CLARK  MANAGER     $2,450.00
JONES  MANAGER     $2,975.00
MARTIN SALESMAN    $1,250.00  $1,400.00
OATES  PRESIDENT   $5,000.00
WARD   SALESMAN    $1,250.00    $500.00

6 records selected.
```

3.2 **UPDATE**

Update is a process of changing the values of fields within the data base. The rows to be updated are specified by means of a WHERE clause. The updates to be made are specified in a SET clause.

Example 3-6: Set employee number 7782's salary to $2,750.

```
SQL>UPDATE  EMP
SQL>SET     SAL = 2750
SQL>WHERE   EMPNO = 7782;
SQL>/
1 record updated.

SQL>SELECT  * FROM  EMP WHERE EMPNO=7782;
SQL>/

EMPNO ENAME  JOB                SAL       COMM DEPTNO
----- ------ ---------- ---------- ---------- ------
 7782 CLARK  MANAGER     $2,750.00               10
```

A SET clause may update multiple fields within a table.

Example 3-7: Update department 30's location to Paris and increase its employee count by two.

```
SQL>UPDATE  DEPT
SQL>SET     LOC = 'PARIS',EMPCNT=NVL(EMPCNT,0) + 2
SQL>WHERE   DEPTNO = 30;
SQL>/
1 record updated.

SQL>SELECT  *
SQL>FROM    DEPT;
SQL>/

DEPTNO DNAME          LOC           EMPCNT
------ -------------- ------------- ------
    10 ADMINISTRATION NEW YORK
    20 RESEARCH       SAN FRANCISCO
    30 SALES          PARIS              2
    40 OPERATIONS     BOSTON
```

Example 3-8: Update the employee table by giving a 15% raise to all employees whose names appear in the BONUS table.

```
SQL>UPDATE  EMP
SQL>SET     SAL = SAL * 1.15
SQL>WHERE   ENAME IN
SQL>        SELECT  ENAME
SQL>        FROM    BONUS;
SQL>/
6 records updated.

SQL>SELECT * FROM EMP;
SQL>/
```

EMPNO	ENAME	JOB	SAL	COMM	DEPTNO
7369	SMITH	CLERK	$800.00		20
7499	ALLEN	SALESMAN	$1,600.00	$300.00	30
7521	WARD	SALESMAN	$1,437.50	$500.00	30
7566	JONES	MANAGER	$3,421.25		20
7654	MARTIN	SALESMAN	$1,437.50	$1,400.00	30
7698	BLAKE	MANAGER	$3,277.50		30
7782	CLARK	MANAGER	$3,162.50		10
7788	SCOTT	ANALYST	$3,000.00		20
7839	OATES	PRESIDENT	$5,750.00		10
7844	TURNER	SALESMAN	$1,500.00	$0.00	30
7876	ADAMS	CLERK	$1,100.00		20
7900	JAMES	CLERK	$950.00		30
7902	FORD	ANALYST	$3,000.00		20
7934	MILLER	CLERK	$1,300.00		10
7955	WILSON				20
7956	JAKES	CLERK	$1,000.00		20
7989	CARTER	SALESMAN	$1,500.00	$0.00	30

17 records selected.

3.3 **DELETE**

DELETE removes rows from tables in the data base.

Example 3-9: Delete the employee named Oates from the BONUS table.

```
SQL>DELETE  BONUS
SQL>WHERE   ENAME = 'OATES';
SQL>/
1 record deleted.

SQL>SELECT  *  FROM BONUS;
SQL>/

ENAME  JOB              SAL         COMM
------ ---------- ---------- ----------
BLAKE  MANAGER     $2,850.00
CLARK  MANAGER     $2,450.00
JONES  MANAGER     $2,975.00
MARTIN SALESMAN    $1,250.00  $1,400.00
WARD   SALESMAN    $1,250.00    $500.00
```

The WHERE clause in a DELETE statement may contain a query block.

Example 3-10: Delete from the BONUS table all the employees with the same job as Jones.

```
SQL>DELETE  BONUS
SQL>WHERE   JOB IN
SQL>        SELECT  JOB
SQL>        FROM    EMP
SQL>        WHERE   ENAME = 'JONES';
SQL>/
3 records deleted.

SQL>SELECT  * FROM BONUS;
SQL>/

ENAME  JOB              SAL         COMM
------ ---------- ---------- ----------
MARTIN SALESMAN    $1,250.00  $1,400.00
WARD   SALESMAN    $1,250.00    $500.00
```

A DELETE statement without a WHERE clause specifies the removal of all rows in the table.

Example 3-11: Delete all rows from the BONUS table.

```
SQL>DELETE  BONUS;
SQL>/
2 records deleted.

SQL>SELECT  * FROM BONUS;
SQL>/

ENAME  JOB                 SAL        COMM
------ ---------- ---------- ----------

no records selected
```

4. DATA DEFINITION FACILITIES

This section of the manual describes the CREATE TABLE, EXPAND TABLE DEFINE VIEW, and DROP facilities of the system.

4.1 CREATE TABLE

Example 4-1: Display projects table.

```
SQL>SELECT  *
SQL>FROM    PROJ;
SQL>/
FROM    PROJ;
        ^
invalid table name                                    [*** ERROR ***]
```

Example 4-2: Create a new table to contain project number, name, and budget information.

```
SQL>CREATE  TABLE PROJ
SQL>        PROJNO(NUMBER NONULL IMAGE UNIQUE),
SQL>        PNAME(CHAR(10) IMAGE),
SQL>        BUDGET(NUMBER),
SQL>        EMPCNT(NUMBER);
SQL>/
Table created.

SQL>SELECT  *
SQL>FROM    PROJ;
SQL>/

PROJNO PNAME          BUDGET EMPCNT
------ ------- ------------ ------

no records selected
```

Example 4-3: Insert three projects into the project table.

```
SQL>INSERT INTO PROJ(PROJNO,PNAME,BUDGET):
SQL>        <101,'ALPHA',250000>;
SQL>/
1 record created.

SQL>INSERT INTO PROJ(PROJNO,PNAME,BUDGET):
SQL>        <102,'BETA',175000>;
SQL>/
1 record created.

SQL>INSERT INTO PROJ(PROJNO,PNAME,BUDGET):
SQL>        <103,'GAMMA',95000>;
SQL>/
1 record created.

SQL>SELECT  *
SQL>FROM    PROJ;
SQL>/

PROJNO PNAME          BUDGET EMPCNT
------ ------- ------------ ------
   101 ALPHA    $250,000.00
   102 BETA     $175,000.00
   103 GAMMA     $95,000.00
```

4.2 **DROP TABLE**

Tables and views may be dropped dynamically.

Example 4-4: Drop the BONUS table from the data base.

```
SQL>DROP TABLE BONUS;
SQL>/
Table dropped.
```

4.3 **EXPAND TABLE**

Example 4-5: An existing table may be expanded by adding a new column to it.

```
SQL>SELECT  EMPNO,DEPTNO,PROJNO,ENAME
SQL>FROM    EMP
SQL>WHERE   DEPTNO = 10;
SQL>/
SELECT  EMPNO,DEPTNO,PROJNO,ENAME
                     ^
invalid column name                                    [*** ERROR ***]
```

Example 4-6: Add a new project number column to the employee table.

```
SQL>EXPAND  TABLE EMP
SQL>        ADD COLUMN PROJNO(NUMBER IMAGE);
SQL>/
Table expanded.

SQL>SELECT  EMPNO,ENAME,PROJNO,DEPTNO
SQL>FROM    EMP
SQL>WHERE   DEPTNO = 10;
SQL>/

EMPNO ENAME  PROJNO DEPTNO
----- ------ ------ ------
 7782 CLARK             10
 7934 MILLER            10
 7839 OATES             10
```

Example 4-7: Update the employee table by assigning employees to projects.

```
SQL>UPDATE  EMP
SQL>SET     PROJNO = 101
SQL>WHERE   DEPTNO = 20
SQL>    OR  JOB = 'MANAGER';
SQL>/
10 records updated.

SQL>SELECT  *
SQL>FROM    EMP
SQL>WHERE   PROJNO = 101;
SQL>/
```

EMPNO	ENAME	JOB	SAL	COMM	DEPTNO	PROJNO
7369	SMITH	CLERK	$800.00		20	101
7566	JONES	MANAGER	$3,421.25		20	101
7698	BLAKE	MANAGER	$3,277.50		30	101
7782	CLARK	MANAGER	$3,162.50		10	101
7876	ADAMS	CLERK	$1,100.00		20	101
7788	SCOTT	ANALYST	$3,000.00		20	101
7902	FORD	ANALYST	$3,000.00		20	101
7955	WILSON				20	101
7956	JAKES	CLERK	$1,000.00		20	101

```
9 records selected.

SQL>UPDATE  EMP
SQL>SET     PROJNO  = 102
SQL>WHERE   EMPNO > 7700
SQL>  AND   NOT PROJNO = 101;
SQL>/
5 records updated.
```

Example 4-8: The PROJ table may now be joined to the EMP table.

```
SQL>SELECT  ENAME,PNAME
SQL>FROM    EMP,PROJ
SQL>WHERE   EMP.PROJNO = PROJ.PROJNO;
SQL>/

ENAME  PNAME
------ -------
SMITH  ALPHA
JONES  ALPHA
BLAKE  ALPHA
CLARK  ALPHA
ADAMS  ALPHA
SCOTT  ALPHA
FORD   ALPHA
WILSON ALPHA
JAKES  ALPHA
TURNER BETA
MILLER BETA
OATES  BETA
JAMES  BETA
CARTER BETA
ALLEN  GAMMA
WARD   GAMMA
MARTIN GAMMA

17 records selected.
```

4.4 DEFINE VIEW

Alternative views of stored data may be defined. Any vaild query may be used in the definition of a view.

Example 4-9: Define a view called EMP10 containing employee number, name, job and project number information for employees in department 10.

```
SQL>DEFINE  VIEW EMP10 AS
SQL>        SELECT  EMPNO,ENAME,JOB,PROJNO
SQL>        FROM    EMP
SQL>        WHERE   DEPTNO = 10;
SQL>/
View defined.

SQL>SELECT  *
SQL>FROM    EMP10;
SQL>/

EMPNO ENAME  JOB         PROJNO
----- ------ ---------- ------
 7782 CLARK  MANAGER        101
 7934 MILLER CLERK          102
 7839 OATES  PRESIDENT      102
```

Example 4-10: A view can contain more than one table.

```
SQL>DEFINE  VIEW PROJSTAFF(EMPLOYEE,PROJECT) AS
SQL>        SELECT  ENAME,PNAME
SQL>        FROM    EMP,PROJ
SQL>        WHERE   EMP.PROJNO = PROJ.PROJNO;
SQL>/
View defined.

SQL>SELECT  *
SQL>FROM    PROJSTAFF;
SQL>/

EMPLOYEE PROJECT
-------- -------
SMITH    ALPHA
JONES    ALPHA
BLAKE    ALPHA
CLARK    ALPHA
ADAMS    ALPHA
SCOTT    ALPHA
FORD     ALPHA
WILSON   ALPHA
JAKES    ALPHA
TURNER   BETA
MILLER   BETA
OATES    BETA
JAMES    BETA
CARTER   BETA
ALLEN    GAMMA
WARD     GAMMA
MARTIN   GAMMA

17 records selected.
```

When defining a view, ORACLE may draw the column names from the underlying tables or the user may rename the columns as in the example below.

Example 4-11: Define a view containing information from the project, employee, and department table.

```
SQL>DEFINE  VIEW PROJECTS (PROJECT,EMPLOYEE,EMP#,LOCATION) AS
SQL>        SELECT  PNAME,ENAME,EMPNO,LOC
SQL>        FROM    PROJ,EMP,DEPT
SQL>        WHERE   EMP.DEPTNO = DEPT.DEPTNO
SQL>          AND   EMP.PROJNO = PROJ.PROJNO;
SQL>/
View defined.
```

Example 4-12: Views may be selectively queried in the same way as a table.

```
SQL>SELECT  PROJECT,EMPLOYEE,LOCATION
SQL>FROM    PROJECTS
SQL>WHERE   LOCATION = 'NEW YORK';
SQL>/

PROJECT EMPLOYEE LOC
------- -------- -------------
ALPHA   CLARK    NEW YORK
BETA    MILLER   NEW YORK
BETA    OATES    NEW YORK
```

Example 4-13: Views may be joined to tables or other views.

```
SQL>SELECT  ENAME,JOB,PNAME
SQL>FROM    PROJ,EMP10
SQL>WHERE   PROJ.PROJNO=EMP10.PROJNO
SQL>  AND   JOB ^= 'CLERK'
SQL>/

ENAME  JOB         PNAME
------ ---------- -------
CLARK  MANAGER    ALPHA
OATES  PRESIDENT  BETA
```

Views may be defined in terms of other views.

Example 4-14: Define a view containing the name of projects and employees located in Paris.

```
SQL>DEFINE  VIEW PARIS (NAME,PROJ) AS
SQL>        SELECT  EMPLOYEE,PROJECT
SQL>        FROM    PROJECTS
SQL>        WHERE   LOCATION = 'PARIS';
SQL>/
View defined.

SQL>SELECT  *
SQL>FROM    PARIS;
SQL>/

NAME   PROJ
------ ----------
ALLEN  GAMMA
WARD   GAMMA
BLAKE  ALPHA
MARTIN GAMMA
TURNER BETA
JAMES  BETA
CARTER BETA

7 records selected.
```

4.5 **Virtual-Fields**

A view may contain arithmetic expressions or built in functions. These expressions or functions appear to the user of the view as "virtual fields." When expressions or functions are used within a view, column names must be specified for the view.

Example 4-15: Define a view containing employee name, salary, annual salary, and department number.

```
SQL>DEFINE  VIEW PAY (NAME,SAL,COMM,ASAL,DEPTNO) AS
SQL>        SELECT  ENAME,SAL,COMM,SAL * 12,DEPTNO
SQL>        FROM    EMP;
SQL>/
View defined.
```

Example 4-16: List salary information for employees in department 30.

```
SQL>SELECT  *
SQL>FROM    PAY
SQL>WHERE   DEPTNO = 30
SQL>/
```

NAME	SAL	COMM	ASAL	DEPTNO
ALLEN	$1,600.00	$300.00	$19,200.00	30
WARD	$1,437.50	$500.00	$17,250.00	30
BLAKE	$3,277.50		$39,330.00	30
MARTIN	$1,437.50	$1,400.00	$17,250.00	30
TURNER	$1,500.00	$0.00	$18,000.00	30
JAMES	$950.00		$11,400.00	30
CARTER	$1,500.00	$0.00	$18,000.00	30

7 records selected.

Example 4-17: Define a view containing a departments minimum, average, maximum, and total compensation.

```
SQL>DEFINE  VIEW DEPT_SAL (DEPTNO,LOSAL,MEDSAL,HISAL,TOTSAL) AS
SQL>        SELECT  DEPTNO,MIN(SAL),AVG(SAL),MAX(SAL),SUM(SAL)
SQL>        FROM    EMP
SQL>        GROUP BY DEPTNO;
SQL>/
View defined.
```

Example 4-18: List minimum, average, and total salary for each department.

```
SQL>SELECT  DEPTNO,LOSAL,HISAL,TOTSAL
SQL>FROM    DEPT_SAL
SQL>/
```

DEPTNO	LOSAL	HISAL	TOTSAL
10	$1,300.00	$5,750.00	$10,212.50
20	$800.00	$3,421.25	$12,321.25
30	$950.00	$3,277.50	$11,702.50

5. DATA STRUCTURES

Create a data structure that allows one employee to work on many projects, and one project to have many employees.

Example 5-1: Create a two column table relating employees to projects.

```
SQL>CREATE  TABLE PE
SQL>        EMPNO(NUMBER NONULL IMAGE),
SQL>        PROJNO(NUMBER NONULL IMAGE);
SQL>/
Table created.
```

Example 5-2: Move the relationship between employees and projects from the EMP table to the PE table.

```
SQL>INSERT  INTO PE(EMPNO,PROJNO):
SQL>        SELECT EMPNO,PROJNO
SQL>        FROM    EMP;
SQL>/
17 records created.

SQL>SELECT  *
SQL>FROM    PE;
SQL>/

EMPNO PROJNO
----- ------
 7369    101
 7499    103
 7521    103
 7566    101
 7654    103
 7698    101
 7782    101
 7788    101
 7839    102
 7844    102
 7876    101
 7900    102
 7902    101
 7934    102
 7955    101
 7956    101
 7989    102

17 records selected.
```

Example 5-3: Delete the data from the PROJNO column of the EMP table.

```
SQL>UPDATE  EMP
SQL>SET     PROJNO = NULL;
SQL>/
17 records updated.

SQL>SELECT  *
SQL>FROM    EMP;
SQL>/
```

EMPNO	ENAME	JOB	SAL	COMM	DEPTNO	PROJNO
7369	SMITH	CLERK	$800.00		20	
7499	ALLEN	SALESMAN	$1,600.00	$300.00	30	
7521	WARD	SALESMAN	$1,437.50	$500.00	30	
7566	JONES	MANAGER	$3,421.25		20	
7654	MARTIN	SALESMAN	$1,437.50	$1,400.00	30	
7698	BLAKE	MANAGER	$3,277.50		30	
7782	CLARK	MANAGER	$3,162.50		10	
7788	SCOTT	ANALYST	$3,000.00		20	
7839	OATES	PRESIDENT	$5,750.00		10	
7844	TURNER	SALESMAN	$1,500.00	$0.00	30	
7876	ADAMS	CLERK	$1,100.00		20	
7900	JAMES	CLERK	$950.00		30	
7902	FORD	ANALYST	$3,000.00		20	
7934	MILLER	CLERK	$1,300.00		10	
7955	WILSON				20	
7956	JAKES	CLERK	$1,000.00		20	
7989	CARTER	SALESMAN	$1,500.00	$0.00	30	

17 records selected.

Example 5-4: The new structure requires that the EMP table be joined to the PROJ table via the PE table.

```
SQL>SELECT  ENAME,PNAME
SQL>FROM    EMP,PROJ,PE
SQL>WHERE   EMP.EMPNO = PE.EMPNO
SQL>  AND   PROJ.PROJNO = PE.PROJNO;
SQL>/

ENAME  PNAME
------ -------
SMITH  ALPHA
ALLEN  GAMMA
WARD   GAMMA
JONES  ALPHA
MARTIN GAMMA
BLAKE  ALPHA
CLARK  ALPHA
SCOTT  ALPHA
OATES  BETA
TURNER BETA
ADAMS  ALPHA
JAMES  BETA
FORD   ALPHA
MILLER BETA
WILSON ALPHA
JAKES  ALPHA
CARTER BETA

17 records selected.
```

The new data structure with the PE table allows employees to be assigned to more than one project.

Example 5-5: Assign employee 7989 to projects 101 and 103, and employee 7956 to project 102.

```
SQL>INSERT  INTO PE:
SQL>        <7989,101>;
SQL>/
1 record created.

SQL>INSERT  INTO PE:
SQL>        <7989,103>;
SQL>/
1 record created.

SQL>INSERT  INTO PE:
SQL>        <7956,103>;
SQL>/
1 record created.
```

Example 5-6: List the projects for employees 7956 and 7989.

```
SQL>SELECT  ENAME,EMP.EMPNO,PNAME,PROJ.PROJNO
SQL>FROM    EMP,PROJ,PE
SQL>WHERE  [EMP.EMPNO = PE.EMPNO  AND  PROJ.PROJNO = PE.PROJNO]
SQL>  AND   EMP.EMPNO = <7956,7989>;
SQL>/
```

ENAME	EMPNO	PNAME	PROJNO
JAKES	7956	ALPHA	101
JAKES	7956	GAMMA	103
CARTER	7989	BETA	102
CARTER	7989	ALPHA	101
CARTER	7989	GAMMA	103

6. DATA INDEPENDENCE

The view facility within ORACLE in combination with the non-procedural nature of the SQL data language allows the user's programs to be independent to changes in data structure.

A new PROJSTAFF view can be defined on the new data structure. The new view will contain the same information as the old PROJSTAFF view only the definition of the view will be different. Users of the old PROJSTAFF view will be insulated from the change.

Example 6-1: Drop the old PROJSTAFF view.

```
SQL>DROP    VIEW PROJSTAFF;
SQL>/
View dropped.
```

Example 6-2: Define a new PROJSTAFF view that joins the EMP table to the PROJ table via the PE table.

```
SQL>DEFINE VIEW PROJSTAFF(EMPLOYEE,PROJECT) AS
SQL>SELECT  ENAME,PNAME
SQL>FROM    EMP,PROJ,PE
SQL>WHERE   EMP.EMPNO = PE.EMPNO
SQL>  AND   PROJ.PROJNO = PE.PROJNO;
SQL>/
View defined.
```

Example 6-3: Queries that used the old PROJSTAFF view continue to run without modification once the new PROJSTAFF view has been defined even though the structure of the data base has been altered.

```
SQL>SELECT  *
SQL>FROM    PROJSTAFF
SQL>WHERE   PROJECT = 'GAMMA';
SQL>/

EMPLOYEE PNAME
-------- -------
ALLEN    GAMMA
WARD     GAMMA
MARTIN   GAMMA
CARTER   GAMMA
JAKES    GAMMA
```

7. OPERATIONS ON TREE-STRUCTURED TABLES

The EMP table does not contain all the information necessary to define the "reporting structure" between the employees in the EMP table. This is because the EMP table does not identify each employee's direct supervisor. In order to store this reporting structure information, the EMP table has to be expanded.

Example 7-1: Add a new column called SUPR to the EMP table.

```
SQL>EXPAND  TABLE EMP
SQL>        ADD COLUMN SUPR(NUMBER IMAGE);
SQL>/
Table expanded.
```

Example 7-2: Assign a supervisor to each employee except OATES.

```
SQL>UPDATE  EMP
SQL>SET     SUPR=7839
SQL>WHERE   EMPNO=7782
SQL>/
1 record updated.

SQL>UPDATE  EMP
SQL>SET     SUPR=7839
SQL>WHERE   EMPNO=7566
SQL>/
1 record updated.

SQL>UPDATE  EMP
SQL>SET     SUPR=7839
SQL>WHERE   EMPNO=7698
SQL>/
1 record updated.

SQL>UPDATE  EMP
SQL>SET     SUPR=7782
SQL>WHERE   EMPNO=7934
SQL>/
1 record updated.

SQL>UPDATE  EMP
SQL>SET     SUPR=7566
SQL>WHERE   EMPNO=7788
SQL>/
1 record updated.
```

Continue assigning supervisors to employees.

```
SQL>UPDATE  EMP
SQL>SET     SUPR=7566
SQL>WHERE   EMPNO=7902
SQL>/
1 record updated.

SQL>UPDATE  EMP
SQL>SET     SUPR=7566
SQL>WHERE   EMPNO=7955
SQL>/
1 record updated.

SQL>UPDATE  EMP
SQL>SET     SUPR=7788
SQL>WHERE   EMPNO=7876
SQL>/
1 record updated.

SQL>UPDATE  EMP
SQL>SET     SUPR=7902
SQL>WHERE   EMPNO=7369
SQL>/
1 record updated.

SQL>UPDATE  EMP
SQL>SET     SUPR=7955
SQL>WHERE   EMPNO=7956
SQL>/
1 record updated.
```

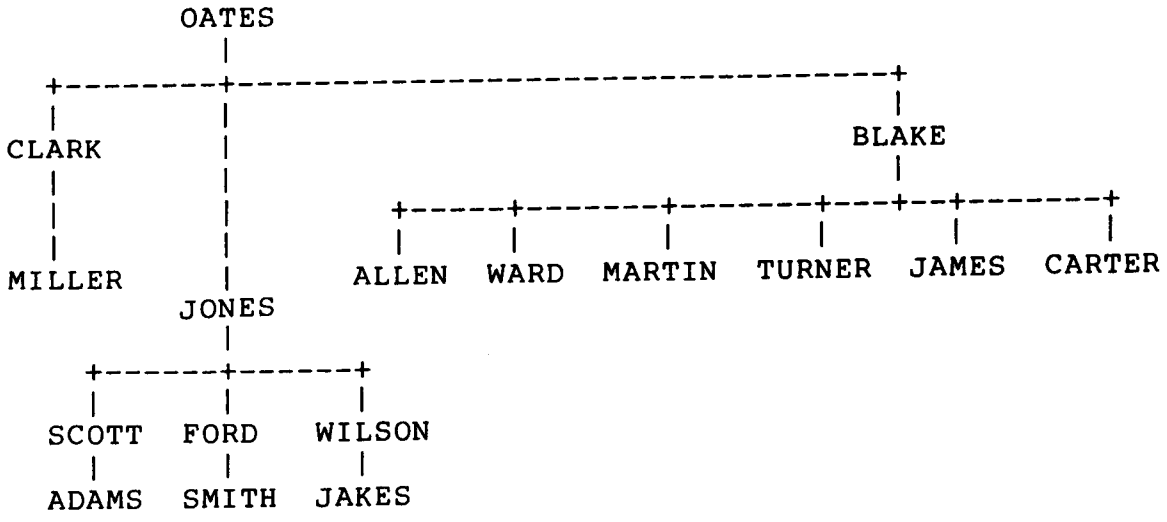
Continue assigning supervisors to employees.

```
SQL>UPDATE  EMP
SQL>SET     SUPR=7698
SQL>WHERE   EMPNO=7499
SQL>/
1 record updated.

SQL>UPDATE  EMP
SQL>SET     SUPR=7698
SQL>WHERE   EMPNO=7521
SQL>/
1 record updated.

SQL>UPDATE  EMP
SQL>SET     SUPR=7698
SQL>WHERE   EMPNO=7654
SQL>/
1 record updated.

SQL>UPDATE  EMP
SQL>SET     SUPR=7698
SQL>WHERE   EMPNO=7844
SQL>/
1 record updated.

SQL>UPDATE  EMP
SQL>SET     SUPR=7698
SQL>WHERE   EMPNO=7900
SQL>/
1 record updated.

SQL>UPDATE  EMP
SQL>SET     SUPR=7698
SQL>WHERE   EMPNO=7989
SQL>/
1 record updated.
```

Example 7-3: List the reporting structure information from the EMP table including each employees name, number, department, and his supervisor's number.

```
SQL>SELECT  ENAME,EMPNO,SUPR,DEPTNO
SQL>FROM    EMP
SQL>/

ENAME  EMPNO  SUPR DEPTNO
------ ----- ----- ------
SMITH   7369  7902     20
ALLEN   7499  7698     30
WARD    7521  7698     30
JONES   7566  7839     20
MARTIN  7654  7698     30
BLAKE   7698  7839     30
CLARK   7782  7839     10
SCOTT   7788  7566     20
OATES   7839           10
TURNER  7844  7698     30
ADAMS   7876  7788     20
JAMES   7900  7698     30
FORD    7902  7566     20
MILLER  7934  7782     10
WILSON  7955  7566     20
JAKES   7956  7955     20
CARTER  7989  7698     30

17 records selected.
```

When considering the reporting structure information contained in the EMP table, it may be useful to think of the EMP table as tree-structured or hierarchical as in the diagram below.



7.1 **CONNECT BY**

ORACLE provides a unique set of operators that allows the user to query these "Tree-Structured" tables. Operations on tree-structured tables include three clauses: START WITH, CONNECT BY, and INCLUDING. CONNECT BY and START WITH are required clauses; INCLUDING is an optional clause.

The CONNECT BY clause indicates the two columns within the table that contain the information necessary to logically specify the structure of the tree. The START WITH clause indicates the leaf within the tree (row within the table) that the query is to start.

Example 7-4: Find all the people who work directly or indirectly for JONES.

```
SQL>SELECT UNIQUE ENAME,EMPNO,JOB,DEPTNO,SUPR
SQL>FROM   EMP
SQL>START  WITH ENAME = 'JONES'
SQL>CONNECT BY PRIOR EMPNO = SUPR
SQL>/

ENAME  EMPNO JOB         DEPTNO  SUPR
------ ----- ---------- ------ -----
JONES   7566 MANAGER         20  7839
SCOTT   7788 ANALYST         20  7566
FORD    7902 ANALYST         20  7566
WILSON  7955                 20  7566
ADAMS   7876 CLERK           20  7788
SMITH   7369 CLERK           20  7902
JAKES   7956 CLERK           20  7955

7 records selected.
```

A query may "walk" the tree in either the UP or DOWN direction. The user specifies the direction the tree is to be walked by means of the PRIOR keyword within the CONNECT BY clause. In this example, if the PRIOR is placed before EMPNO, the tree is walked in the down direction. If the PRIOR is placed before SUPR, the tree is walked in the UP direction.

Example 7-5: List all the people in the reporting structure above SMITH.

```
SQL>SELECT UNIQUE ENAME,EMPNO,JOB,DEPTNO,SUPR
SQL>FROM   EMP
SQL>START WITH ENAME = 'SMITH'
SQL>CONNECT BY   EMPNO = PRIOR  SUPR
SQL>/

ENAME  EMPNO JOB         DEPTNO  SUPR
------ ----- ---------- ------ -----
SMITH   7369 CLERK          20  7902
FORD    7902 ANALYST        20  7566
JONES   7566 MANAGER        20  7839
OATES   7839 PRESIDENT      10
```

Example 7-6: SELECT UNIQUE must be specified when walking the tree in the UP direction.

7.2 START WITH

The START WITH clause can reference more than one starting point within the tree.

Example 7-7: List all the people who work for CLARK or BLAKE.

```
SQL>SELECT  UNIQUE ENAME,EMPNO,JOB,DEPTNO,SUPR
SQL>FROM    EMP
SQL>START   WITH ENAME = 'CLARK'
SQL>  OR    ENAME = 'BLAKE'
SQL>CONNECT BY PRIOR EMPNO = SUPR
SQL>ORDER   BY DEPTNO
SQL>/
```

ENAME	EMPNO	JOB	DEPTNO	SUPR
CLARK	7782	MANAGER	10	7839
MILLER	7934	CLERK	10	7782
BLAKE	7698	MANAGER	30	7839
ALLEN	7499	SALESMAN	30	7698
WARD	7521	SALESMAN	30	7698
MARTIN	7654	SALESMAN	30	7698
TURNER	7844	SALESMAN	30	7698
JAMES	7900	CLERK	30	7698
CARTER	7989	SALESMAN	30	7698

9 records selected.

Example 7-8: List all the people who work for people who have the same job as SCOTT.

```
SQL>SELECT  UNIQUE ENAME,EMPNO,JOB,DEPTNO,SUPR
SQL>FROM    EMP
SQL>START   WITH JOB IN
SQL>        SELECT   JOB
SQL>        FROM     EMP
SQL>        WHERE    ENAME = 'SCOTT';
SQL>CONNECT BY PRIOR EMPNO = SUPR
SQL>/
```

ENAME	EMPNO	JOB	DEPTNO	SUPR
SCOTT	7788	ANALYST	20	7566
FORD	7902	ANALYST	20	7566
ADAMS	7876	CLERK	20	7788
SMITH	7369	CLERK	20	7902

7.3 **INCLUDING**

As a tree is being walked, a predicate or set of predicates can be applied to individual leafs of the tree or entire branches of the tree. The INCLUDING clause is used to qualify or disqualify leafs of the tree. The WHERE clause is used to "prune" entire branches of the tree.

Example 7-9: List all the people who work for JONES except SCOTT.

```
SQL>SELECT      UNIQUE ENAME,EMPNO,JOB,DEPTNO,SUPR
SQL>FROM        EMP
SQL>START       WITH ENAME = 'JONES'
SQL>CONNECT BY PRIOR EMPNO = SUPR
SQL>INCLUDING  ENAME ^= 'SCOTT'
SQL>/

ENAME  EMPNO JOB         DEPTNO  SUPR
------ ----- ---------- ------ -----
JONES   7566 MANAGER         20  7839
FORD    7902 ANALYST         20  7566
WILSON  7955                 20  7566
ADAMS   7876 CLERK           20  7788
SMITH   7369 CLERK           20  7902
JAKES   7956 CLERK           20  7955

6 records selected.
```

Example 7-10: List all the employees that work for JONES except SCOTT and the people who work for SCOTT.

```
SQL>SELECT UNIQUE ENAME,EMPNO,JOB,DEPTNO,SUPR
SQL>FROM   EMP
SQL>WHERE  ENAME ^= 'SCOTT'
SQL>START  WITH ENAME = 'JONES'
SQL>CONNECT BY PRIOR EMPNO = SUPR
SQL>/

ENAME  EMPNO JOB        DEPTNO  SUPR
------ ----- ---------- ------ -----
JONES   7566 MANAGER        20  7839
FORD    7902 ANALYST        20  7566
WILSON  7955                20  7566
SMITH   7369 CLERK          20  7902
JAKES   7956 CLERK          20  7955
```

Note that ADAMS, who works for SCOTT was eliminated when SCOTT was pruned using the WHERE clause. ADAMS was not eliminated when SCOTT was excluded using the INCLUDING clause.

A query on a tree-structured table can contain both a WHERE clause and an INCLUDING clause. First, the tree is logically formed using the CONNECT BY clause; Second, the tree is walked in the direction specified by the PRIOR keyword in the CONNECT BY clause starting with the leaf specified in the START WITH clause; Third, the WHERE clause is applied to each leaf of the tree (row of the table) to prune branches from the tree; Fourth, the INCLUDING clause is applied to each leaf of the tree to qualify or disqualify individual rows.

Example 7-11: List all the employees that work for JONES except SCOTT and the people who work for SCOTT, and FORD.

```
SQL>SELECT UNIQUE ENAME,EMPNO,JOB,DEPTNO,SUPR
SQL>FROM   EMP
SQL>WHERE  ENAME ^= 'SCOTT'
SQL>START  WITH ENAME = 'JONES'
SQL>CONNECT BY PRIOR EMPNO = SUPR
SQL>INCLUDING ENAME ^= 'FORD'
SQL>/

ENAME  EMPNO JOB        DEPTNO  SUPR
------ ----- ---------- ------ -----
JONES   7566 MANAGER        20  7839
WILSON  7955                20  7566
SMITH   7369 CLERK          20  7902
JAKES   7956 CLERK          20  7955
```

Queries on tree-structured tables can include joins.

Example 7-12: List all the employees and the location of their departments that work for JONES except SCOTT and the people who work for SCOTT, and FORD.

```
SQL>SELECT UNIQUE ENAME,LOC,EMPNO,JOB,DEPT.DEPTNO,SUPR
SQL>FROM   EMP,DEPT
SQL>WHERE  EMP.DEPTNO = DEPT.DEPTNO
SQL>AND  ENAME ^= 'SCOTT'
SQL>START  WITH ENAME = 'JONES'
SQL>CONNECT BY PRIOR EMPNO = SUPR
SQL>INCLUDING ENAME ^= 'FORD'
SQL>/

ENAME  LOC           EMPNO JOB        DEPTNO  SUPR
------ ------------- ----- ---------- ------ -----
JONES  SAN FRANCISCO  7566 MANAGER        20  7839
WILSON SAN FRANCISCO  7955                20  7566
SMITH  SAN FRANCISCO  7369 CLERK          20  7902
JAKES  SAN FRANCISCO  7956 CLERK          20  7955
```

8. SECURITY FACILITIES

ORACLE supports both secure and nonsecure data bases. If a data base is defined as secure, that data base's dictionary contains information about the users of the data base in addition to a description of data stored within the data base. This allows ORACLE to control access to the data base on a user by access privilege basis.

An ORACLE data base is created by means of the DBF utility program. At the time a data base is created, the creating user specifies whether the data base is to be secure or nonsecure. If the data base is to be secure, the creating user specifies a USER-NAME and PASSWORD to DBF. The data base used in this manual is a secure data base created with the following DBF command:

```
DBF C PERSONNEL PERSONNEL.DBS 2048 SCOTT/TIGER
```

Initially, only the creating user, SCOTT with a PASSWORD of TIGER is authorized to operate on the PERSONNEL data base. All the SQL example operations prior to this section of the manual have been issued by the fully authorized creating user, SCOTT.

8.1 DEFINE USER

The creator of a secure data base can authorize additional users of the data base by means of the DEFINE USER command. A defined user of the data base is authorized to log on to the data base, create his own tables, define views on his tables, and define new users of the data base.

ORACLE security facilities prevent unauthorized users from logging on the secure PERSONNEL data base.

Have user SCOTT log off of the PERSONNEL data base and attempt to log on as user ADAMS with a password of WOOD.

Example 8-1: Log on to the PERSONNEL data base as user SCOTT.

```
SQL>#DBS PERSONNEL SCOTT/TIGER
Database 'PERSONNEL' opened.
```

The creator of the PERSONNEL data base, in this case SCOTT, can authorize additional users to log on to the data base by means of the DEFINE USER command.

Example 8-2: Define a new user of the PERSONNEL data base with a USER-NAME of ADAMS and a PASSWORD of WOOD.

```
SQL>DEFINE  USER ADAMS/WOOD
SQL>/
User defined.
```

After a user has been defined, that user may log on to the data base.

Example 8-3: Log on as user ADAMS.

```
SQL>#DBS PERSONNEL ADAMS/WOOD
Database 'PERSONNEL' opened.
```

A new user can create his own tables, define views on his tables, and define new users of the data base.

Example 8-4: Have user ADAMS create a new table and insert a record into it.

```
SQL>CREATE  TABLE PARTS
SQL>        PARTNO(NUMBER IMAGE UNIQUE),
SQL>        PART_NAME(CHAR(10));
SQL>/
Table created.

SQL>INSERT INTO PARTS:
SQL>        <1,'WIDGET'>;
SQL>/
1 record created.
```

8.2 GRANT PRIVILEGE

The ORACLE security facilities enable users to control access to their data by other users. It is the responsibility of the user who creates a table or view to control access to that table or view. A user may extend access to his table or view by means of the GRANT command.

Even though user ADAMS can create his own tables he can not access data via tables and views created by other users unless specifically authorized.

Example 8-5: Have user ADAMS attempt to list the DEPT table.

```
SQL>SELECT  *
SQL>FROM    DEPT
SQL>/
FROM    DEPT
        ^
invalid table name                                    [*** ERROR ***]
```

At this point ADAMS is not authorized to access the DEPT table and is told that the table does not exist.

Example 8-6: Log back on as user SCOTT, the creator of the EMP and DEPT tables.

```
SQL>#DBS PERSONNEL SCOTT/TIGER
Database 'PERSONNEL' opened.
```

Example 8-7: Authorize user ADAMS to READ the DEPT table.

```
SQL>GRANT   READ
SQL>ON      DEPT
SQL>TO      ADAMS
SQL>/
Privileges granted.
```

Example 8-8: Log on as user ADAMS.

```
SQL>#DBS PERSONNEL ADAMS/WOOD
Database 'PERSONNEL' opened.
```

Example 8-9: Have ADAMS list the DEPT table.

```
SQL>SELECT  *
SQL>FROM    DEPT
SQL>/

DEPTNO DNAME          LOC            EMPCNT
------ -------------- ------------- ------
    10 ADMINISTRATION NEW YORK
    20 RESEARCH       SAN FRANCISCO
    30 SALES          PARIS              2
    40 OPERATIONS     BOSTON
```

Example 8-10: Have ADAMS list the PARTS table he created.

```
SQL>SELECT  *
SQL>FROM    PARTS
SQL>/

PARTNO PART_N
------ ------
     1 WIDGET
```

Example 8-11: Log on as user SCOTT.

```
SQL>#DBS PERSONNEL SCOTT/TIGER
Database 'PERSONNEL' opened.
```

Example 8-12: Have user SCOTT attempt to list the PARTS table.

```
SQL>SELECT  *
SQL>FROM    PARTS
SQL>/
FROM    PARTS
        ^
invalid table name                                  [*** ERROR ***]
```

Even though user SCOTT defined user ADAMS to the PERSONNEL data base, SCOTT is not allowed to see data stored in tables created by ADAMS unless specifically authorized to do so by means of a GRANT privilege command.

8.3 **PRIVILEGES**

The following privileges may be granted: READ, INSERT, DELETE, UPDATE (by column), and EXPAND. In addition, the grantor may allow the grantee to GRANT the listed privileges to other users.

Example 8-13: Log back on as user SCOTT. #COMMENT #DBS PERSONNEL SCOTT/TIGER #COMMENT *** Give the following privileges on the EMP table to user ADAMS: the right to READ, INSERT, and UPDATE only the JOB, and DEPTNO columns.

```
SQL>GRANT   READ,INSERT,UPDATE(JOB,DEPTNO)
SQL>ON      EMP
SQL>TO      ADAMS
SQL>/
Privileges granted.
```

Example 8-14: Log on as ADAMS.

```
SQL>#DBS PERSONNEL ADAMS/WOOD
Database 'PERSONNEL' opened.
```

Example 8-15: Have ADAMS update the EMP table.

```
SQL>UPDATE  EMP
SQL>SET     JOB='ANALYST'
SQL>WHERE   ENAME='WILSON'
SQL>/
1 record updated.
```

Example 8-16: Have ADAMS attempt to UPDATE both the JOB and SAL column of the EMP table.

```
SQL>UPDATE EMP
SQL>SET JOB='ANALYST',SAL=100000
SQL>WHERE ENAME='WILSON'
SQL>/
SET JOB='ANALYST',SAL=100000
                  ^
security violation                                  [*** ERROR ***]
```

ORACLE security facilities detect a security violation and indicate the column of the EMP table that ADAMS was not authorized to UPDATE.

The phrase ALL RIGHTS may be substituted for the privilege list in the GRANT statement.

Example 8-17: Have ADAMS grant SCOTT all privileges on the PARTS table.

```
SQL>GRANT   ALL RIGHTS
SQL>ON      PARTS
SQL>TO      SCOTT
SQL>/
Privileges granted.
```

The phrase ALL BUT can be specified preceding a privilege list.

Example 8-18: Log on as user SCOTT.

```
SQL>#DBS PERSONNEL SCOTT/TIGER
Database 'PERSONNEL' opened.
```

Example 8-19: Give ADAMS all the privileges on the DEPT table except EXPAND, along with the right to GRANT these privileges to other users.

```
SQL>GRANT   ALL BUT EXPAND
SQL>ON      DEPT
SQL>TO      ADAMS
SQL>WITH GRANT OPTION
SQL>/
Privileges granted.
```

Privileges can be granted to all users by specifying PUBLIC in place of the user list.

Example 8-20: Give all users READ privileges on the PROJSTAFF view.

```
SQL>GRANT   READ
SQL>ON      PROJSTAFF
SQL>TO      PUBLIC
SQL>/
Privileges granted.
```

Note that the only privilege that may be granted on a view is the READ privilege.

8.4 **USER Keyword**

ORACLE requires users to enter their name and password in order to log on to a secure data base. ORACLE maintains the name of the current user as a keyword constant called USER. The keyword USER may be specified in a SQL statement anywhere a constant is allowed. USER will always return the name of the currently logged on user.

The USER keyword can be specified in the WHERE clause of a SQL statement to control access to the data base. This is especially useful when defining views on the data base.
#WORKSIZE 8

Example 8-21: Define a view of the EMP table allowing any employee to see his own department number, name, salary, commission, and job but not any information about any other employee.

```
SQL>DEFINE  VIEW MYSELF AS
SQL>        SELECT  *
SQL>        FROM    EMP
SQL>        WHERE   ENAME = USER;
SQL>/
View defined.
```

Example 8-22: Give all users access to the MYSELF view.

```
SQL>GRANT   READ
SQL>ON      MYSELF
SQL>TO      PUBLIC
SQL>/
Privileges granted.
```

The USER keyword will always return the USER-NAME of the user currently logged on to the data base.

Example 8-23: Have user SCOTT query the MYSELF view.

```
SQL>SELECT  *
SQL>FROM    MYSELF
SQL>/

EMPNO ENAME  JOB              SAL       COMM DEPTNO PROJNO  SUPR
----- ------ ---------- ---------- ---------- ------ ------ -----
 7788 SCOTT  ANALYST     $3,000.00                20         7566
```

Example 8-24: Log on as user ADAMS and query the MYSELF view.

```
SQL>#DBS PERSONNEL ADAMS/WOOD
Database 'PERSONNEL' opened.
SQL>SELECT  *
SQL>FROM    MYSELF
SQL>/

EMPNO ENAME  JOB              SAL       COMM DEPTNO PROJNO  SUPR
----- ------ ---------- ---------- ---------- ------ ------ -----
 7876 ADAMS  CLERK       $1,100.00                20         7788
```

Example 8-25: Define a view giving managers access to employees in their department only if the manager earns the same amount or more that the employee.

```
SQL>DEFINE  VIEW MY_EMPS (NAME,JOB,SAL,COMM,DEPTNO) AS
SQL>SELECT  WORKER.ENAME,WORKER.JOB,WORKER.SAL,WORKER.COMM,
SQL>SELECT  WORKER.DEPTNO
SQL>FROM    EMP WORKER,EMP MGR
SQL>WHERE   WORKER.DEPTNO = MGR.DEPTNO
SQL>  AND   [MGR.JOB = 'MANAGER'  AND MGR.SAL >= WORKER.SAL]
SQL>  AND   MGR.ENAME=USER;
SQL>/
SELECT  WORKER.DEPTNO
        ^
missing 'from' keyword                                  [*** ERROR ***]
```

Example 8-26: Define JONES and CLARK as users of the PERSONNEL data base.

```
SQL>DEFINE  USER JONES/WOOD
SQL>/
User defined.

SQL>DEFINE  USER CLARK/CLOTH
SQL>/
User defined.
```

Example 8-27: Grant all users access to the MY_EMPS view. Note that the view will only return data if the user's JOB = MANAGER.

```
SQL>GRANT   READ
SQL>ON      MY_EMPS
SQL>TO      PUBLIC
SQL>/
TO      PUBLIC
        ^
security violation                                      [*** ERROR ***]
```

Example 8-28: Have user ADAMS,who is not a manager attempt to query the MY_EMPS view.

```
SQL>SELECT  *
SQL>FROM    MY_EMPS
SQL>/
FROM    MY_EMPS
        ^
invalid table name                                    [*** ERROR ***]
```

Example 8-29: Have users JONES and CLARK, who are managers, query the MY_EMPS view.

```
SQL>#DBS PERSONNEL JONES/WOOD
Database 'PERSONNEL' opened.
SQL>SELECT  *
SQL>FROM    MY_EMPS
SQL>/
FROM    MY_EMPS
        ^
invalid table name                                    [*** ERROR ***]
```

Example 8-30: The view only allows the manager to see people in their own department.

```
SQL>#DBS PERSONNEL CLARK/CLOTH
Database 'PERSONNEL' opened.
SQL>SELECT  *
SQL>FROM    MY_EMPS
SQL>/
FROM    MY_EMPS
        ^
invalid table name                                    [*** ERROR ***]
```

Note that CLARK was not able to see employee OATES because the view prevents managers from seeing anyone in their department who earns more than they do.

8.5 **REVOKE Privilege**

Once a privilege has been granted it may be withdrawn by means of the REVOKE command. Privileges are revoked from the named grantee and from all users to whom he has granted them.

Example 8-31: Log on the PERSONNEL data base as user ADAMS.

```
SQL>#DBS PERSONNEL ADAMS/WOOD
Database 'PERSONNEL' opened.
```

Example 8-32: Have ADAMS INSERT a new department into the DEPT table.

```
SQL>INSERT  INTO DEPT:
SQL>        <50,'SERVICE','DETROIT',NULL>
SQL>/
1 record created.
```

Example 8-33: Log on the PERSONNEL data base as user SCOTT.

```
SQL>#DBS PERSONNEL SCOTT/TIGER
Database 'PERSONNEL' opened.
```

Example 8-34: Revoke from ADAMS the right to INSERT into the DEPT table.

```
SQL>REVOKE  INSERT
SQL>ON      DEPT
SQL>FROM    ADAMS
SQL>/
Privileges revoked.
```

Example 8-35: Log on the PERSONNEL data base as user ADAMS.

```
SQL>#DBS PERSONNEL ADAMS/WOOD
Database 'PERSONNEL' opened.
```

Example 8-36: Have ADAMS attempt to INSERT a new department into the DEPT table.

```
SQL>INSERT  INTO DEPT:
SQL>        <60,'DEVELOPMENT','PORTLAND',NULL>
SQL>/
INSERT  INTO DEPT:
             ^
invalid table name                                  [*** ERROR ***]
```

9. DATA DICTIONARY STRUCTURE

This section of the manual contains a description of ORACLE's integrated data dictionary.

The ORACLE data dictionary is made up of several system defined tables and views. These dictionary tables are dynamically updated by ORACLE to contain a current description of all user tables, views, and access privileges. In addition, the ORACLE dictionary is self describing. Therefore, a user may query the dictionary to determine the names of the tables that make up the dictionary.

DTAB contains the names and a description of the dictionary tables. DCOL contains the names of the columns of the dictionary tables.

Example 9-1: List the names and a description of the tables in the dictionary.

```
SQL>SELECT  *
SQL>FROM    DTAB
SQL>/

TABLE      COMMENT
---------- ----------------------------------------
COL        COLUMN NAMES OF USERS TABLES AND VIEWS
COLDEF     DEFINITION OF COLUMNS IN USERS TABLES
DCOL       COLUMN NAMES OF DICTIONARY TABLES
DTAB       COMMENTS ON DICTIONARY TABLES
DTABLES    DESCRIPTION OF DICTIONARY TABLES
EXPDEF     COLUMN DEFINITIONS USED BY EXPORT
GRANTS     ACCESS PRIVILEGES GRANTED BY USER
PRIVS      ACCESS PRIVILEGES HELD BY USER
TAB        NAMES OF USERS TABLES AND VIEWS
USERS      NAME OF USERS YOU DEFINED
VIEWS      DEFINITIONS OF VIEWS
VXREF      CROSS-REFERENCE OF VIEWS OF TABLES

12 records selected.
```

Example 9-2: List the names of the columns of the dictionary tables.

```
SQL>SELECT  *
SQL>FROM    DCOL
SQL>/

TABLE      COLUMN
---------- ---------
TAB        TABLE
TAB        TYPE
TAB        CREATOR
TAB        GRANTEE
COL        TABLE
COL        COLUMN
COLDEF     TABLE
COLDEF     COLUMN
COLDEF     DATATYPE
COLDEF     LENGTH
COLDEF     IMAGE
COLDEF     NONULL
EXPDEF     TABLE
EXPDEF     COLID
EXPDEF     COLUMN
EXPDEF     DATATYPE
EXPDEF     LENGTH
EXPDEF     IMAGE
EXPDEF     NONULL
VIEWS      VIEW
VIEWS      TEXT
VXREF      VIEW
VXREF      TABLE
USERS      USER
USERS      OWNER
DTAB       TABLE
DTAB       COMMENT
DTABLES    TABLE
DTABLES    TYPE
DTABLES    CREATOR
DTABLES    GRANTEE
DCOL       TABLE
DCOL       COLUMN
GRANTS     TABLE
GRANTS     COLUMN
GRANTS     GRANTEE
GRANTS     ACCESS
PRIVS      TABLE
PRIVS      COLUMN
PRIVS      GRANTOR
PRIVS      ACCESS

41 records selected.
```

The dictionary table TAB allows a user to list the names of all the tables and views that that user has access privileges on. TAB also indicates if the table is a view, the name of the user who created the table (or view), and the grantee of the privileges.

Example 9-3: List user ADAM's tables.

```
SQL>SELECT  *
SQL>FROM    TAB
SQL>/

TABLE      TYPE  CREATOR GRANTEE
---------- ----- ------- -------
PARTS      TABLE ADAMS   ADAMS
DEPT       TABLE SCOTT   ADAMS
EMP        TABLE SCOTT   ADAMS
PROJSTAFF  VIEW  SCOTT   PUBLIC
MYSELF     VIEW  SCOTT   PUBLIC
```

Example 9-4: Log on as user SCOTT.

```
SQL>#DBS PERSONNEL SCOTT/TIGER
Database 'PERSONNEL' opened.
```

Example 9-5: List user SCOTT's table.

```
SQL>SELECT  *
SQL>FROM    TAB
SQL>/

TABLE      TYPE  CREATOR GRANTEE
---------- ----- ------- -------
EMP        TABLE SCOTT   SCOTT
DEPT       TABLE SCOTT   SCOTT
PROJ       TABLE SCOTT   SCOTT
EMP10      VIEW  SCOTT   SCOTT
PROJECTS   VIEW  SCOTT   SCOTT
PARIS      VIEW  SCOTT   SCOTT
PAY        VIEW  SCOTT   SCOTT
DEPT_SAL   VIEW  SCOTT   SCOTT
PE         TABLE SCOTT   SCOTT
PROJSTAFF  VIEW  SCOTT   SCOTT
PARTS      TABLE ADAMS   SCOTT
MYSELF     VIEW  SCOTT   SCOTT
PROJSTAFF  VIEW  SCOTT   PUBLIC
MYSELF     VIEW  SCOTT   PUBLIC

14 records selected.
```

Example 9-6: The dictionary table COL contains the names of the columns of user defined tables.

```
SQL>#DBS PERSONNEL ADAMS/WOOD
Database 'PERSONNEL' opened.
```

Example 9-7: List the names of the columns of user ADAMS' tables.

```
SQL>SELECT  *
SQL>FROM    COL
SQL>/

TABLE       COLUMN
----------- ---------
PARTS       PARTNO
PARTS       PART_NAME
DEPT        DEPTNO
DEPT        DNAME
DEPT        LOC
DEPT        EMPCNT
EMP         EMPNO
EMP         ENAME
EMP         JOB
EMP         SAL
EMP         COMM
EMP         DEPTNO
EMP         PROJNO
EMP         SUPR
PROJSTAFF   EMPLOYEE
PROJSTAFF   PROJECT
MYSELF      EMPNO
MYSELF      ENAME
MYSELF      JOB
MYSELF      SAL
MYSELF      COMM
MYSELF      DEPTNO
MYSELF      PROJNO
MYSELF      SUPR

24 records selected.
```

Example 9-8: Log on as user SCOTT.

```
SQL>#DBS PERSONNEL SCOTT/TIGER
Database 'PERSONNEL' opened.
```

The dictionary table COLDEF contains the definition of columns in user tables.

Example 9-9: List the definition of the columns in the EMP table.

```
SQL>SELECT  COLUMN,DATATYPE,LENGTH,IMAGE,NONULL
SQL>FROM    COLDEF
SQL>WHERE   TABLE = 'EMP'
SQL>/

COLUMN     DATATYPE LENGTH IMAGE       NON
---------- -------- ------ ---------- ---
EMPNO      NUMBER       22 UNIQUE     YES
ENAME      CHAR         10 NON-UNIQUE NO
JOB        CHAR          9 NON-UNIQUE NO
SAL        NUMBER       22            NO
COMM       NUMBER       22            NO
DEPTNO     NUMBER       22 NON-UNIQUE NO
PROJNO     NUMBER       22 NON-UNIQUE NO
SUPR       NUMBER       22 NON-UNIQUE NO

8 records selected.
```

VIEWS is the dictionary table that contains the SQL text of the DEFINE VIEW statement. Comments that were entered as a part of the SQL view definition are also stored in VIEWS.

Example 9-10: List the definition of the PROJECTS view.

```
SQL>SELECT  *
SQL>FROM    VIEWS
SQL>WHERE   VIEW = 'PROJECTS';
SQL>/

VIEW       TEXT
---------- --------------------------------------------------
PROJECTS   DEFINE  VIEW PROJECTS (PROJECT,EMPLOYEE,EMP#,LOC
PROJECTS   ATION) AS
PROJECTS            SELECT  PNAME,ENAME,EMPNO,LOC
PROJECTS            FROM    PROJ,EMP,DEPT
PROJECTS            WHERE   EMP.DEPTNO = DEPT.DEPTNO
PROJECTS              AND   EMP.PROJNO = PROJ.PROJNO;

6 records selected.
```

The VXREF dictionary table defines the relationship of user views to underlying tables and views. The first column contains the name of the view. The second column contains the name of the underlying table or view.

Example 9-11: List a cross reference of views and their base tables.

```
SQL>SELECT  *
SQL>FROM    VXREF
SQL>/

VIEW       TABLE
---------- ----------
EMP10      EMP
PROJECTS   EMP
PAY        EMP
DEPT_SAL   EMP
PROJSTAFF  EMP
MYSELF     EMP
PROJECTS   DEPT
PROJECTS   PROJ
PROJSTAFF  PROJ
PARIS      PROJECTS
PROJSTAFF  PE

11 records selected.
```

The ORACLE dictionary allows users to obtain information about only those tables that the user has access privilges on. The user may also determine: the names of users he has directly or indirectly defined, the access privileges he holds, and the access privileges he has directly or indirectly granted.

The USERS table contains the names of all those users that were defined by the user who is currently logged on to the system. The first column in the USERS table contains the name of the defined user. The second column of the USERS table contains the name of the creating user.

Example 9-12: List the names of the users that originated with SCOTT.

```
SQL>SELECT  *
SQL>FROM    USERS
SQL>/

USER  OWNER
----- ------
ADAMS SCOTT
JONES ADAMS
CLARK ADAMS
```

User ADAMS was defined directly by user SCOTT, but users JONES and CLARK were defined by ADAMS and thereby indirectly by SCOTT.

Example 9-13: Log on as user ADAMS.

```
SQL>#DBS PERSONNEL ADAMS/WOOD
Database 'PERSONNEL' opened.
```

Example 9-14: List the users that originated with ADAMS.

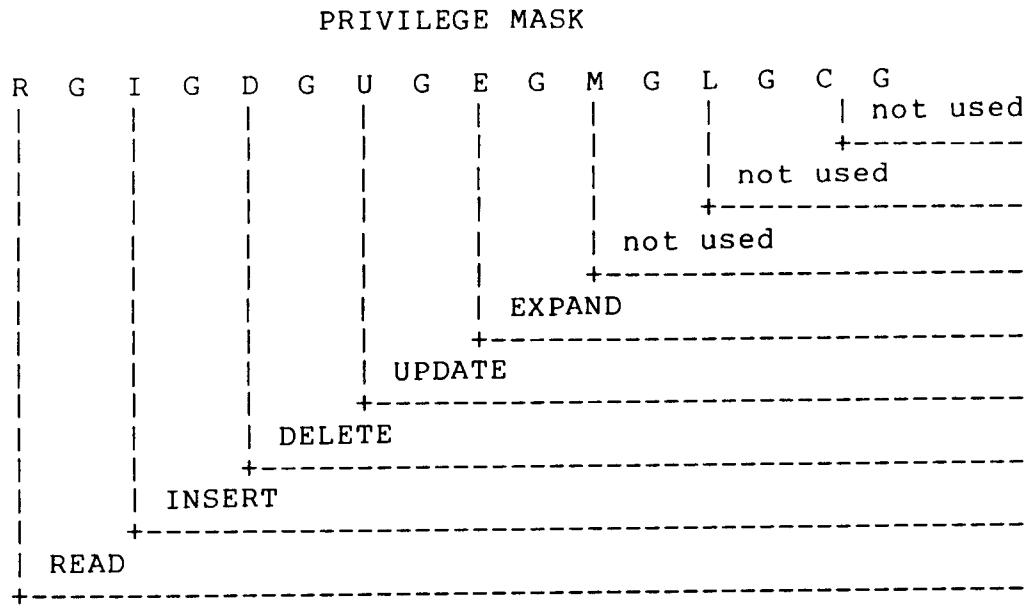
```
SQL>SELECT  *
SQL>FROM    USERS
SQL>/

USER  OWNER
----- ------
JONES ADAMS
CLARK ADAMS
```

The PRIVS table contains access privileges held by users of the data base. Each user can only see the privileges he holds. The PRIVS table contains: the name of the table the privileges are on, the name of the column the privileges apply to, the name user who granted the privileges, and the privilege mask described below. A PRIVS table entry will only contain a column name if update privileges have been granted on a column column rather than a table basis.

Example 9-15: List all the privileges held by ADAMS.

```
SQL>SELECT  *
SQL>FROM    PRIVS
SQL>/

TABLE       COLUMN     GRANTOR ACCESS
----------- ---------- ------- ----------------
PARTS                  ADAMS   RGIGDGUGEGMGLGCG
DEPT                   SCOTT   RGXXDGUGXXMGLGCG
EMP                    SCOTT   RXIXXXXXXXXXXXX
EMP         JOB        SCOTT   XXXXXXUXXXXXXXX
EMP         DEPTNO     SCOTT   XXXXXXUXXXXXXXX
TAB                    ORACLE  RXXXXXXXXXXXXXX
COL                    ORACLE  RXXXXXXXXXXXXXX
COLDEF                 ORACLE  RXXXXXXXXXXXXXX
EXPDEF                 ORACLE  RXXXXXXXXXXXXXX
VIEWS                  ORACLE  RXXXXXXXXXXXXXX
VXREF                  ORACLE  RXXXXXXXXXXXXXX
USERS                  ORACLE  RXXXXXXXXXXXXXX
DTAB                   ORACLE  RXXXXXXXXXXXXXX
DTABLES                ORACLE  RXXXXXXXXXXXXXX
DCOL                   ORACLE  RXXXXXXXXXXXXXX
GRANTS                 ORACLE  RXXXXXXXXXXXXXX
PRIVS                  ORACLE  RXXXXXXXXXXXXXX
PROJSTAFF              SCOTT   RXXXXXXXXXXXXXX
MYSELF                 SCOTT   RXXXXXXXXXXXXXX

19 records selected.
```

Note that ADAMS has all privileges on MY_EMPS and PARTS, the table and view he created. He has all but INSERT and EXPAND on the DEPT table. He has READ and INSERT without the GRANT option, and UPDATE on the JOB and DEPTNO columns of the EMP table. The diagram below describes the entries in the ACCESS column of both the PRIVS and GRANTS dictionary tables.



The G following each privilege flag indicates that the GRANT OPTION is present for that privilege. If an X is present for any privilege or grant flag, that privilege was not granted.

The GRANTS table indicates the privileges granted either directly or indirectly by the current user. The GRANTS table contains: the name of the table the privileges are on, the name of the column the privileges apply to, the name of the user who is the grantee of the privileges, and the privilege mask. A GRANTS table entry will only contain a column name if update privileges have been granted on a column rather than a table basis.

Example 9-16: List all the privileges granted by ADAMS.

```
SQL>SELECT  *
SQL>FROM    GRANTS
SQL>/

TABLE      COLUMN     GRANTEE ACCESS
---------- ---------- ------- ----------------
PARTS                 ADAMS   RGIGDGUGEGMGLGCG
PARTS                 SCOTT   RXIXDXUXEXMXLXCX
```

9.1 **Dictionary Extensions**

The user is free to expand the ORACLE data dictionary by creating additional tables, and defining views of the user tables joined to the system defined dictionary tables and views.

Example 9-17: Create a dictionary extension to contain comments on user defined tables.

```
SQL>CREATE  TABLE TAB_COM
SQL>        TABLE(CHAR(20) IMAGE UNIQUE),
SQL>        COMMENT(CHAR(35))
SQL>/
Table created.
```

Example 9-18: Enter a comment on the EMP table into the TAB_COM table.

```
SQL>INSERT  INTO TAB_COM:
SQL>        <'EMP','Information about company employees'>
SQL>/
1 record created.
```

Example 9-19: Define a view joining TAB from the dictionary to the TAB_COM table.

```
SQL>DEFINE  VIEW TABC AS
SQL>        SELECT  TAB.TABLE,TYPE,CREATOR,COMMENT
SQL>        FROM    TAB,TAB_COM
SQL>        WHERE   TAB.TABLE=TAB_COM.TABLE;
SQL>/
View defined.

SQL>#WORKSIZE 12
```

Example 9-20: List the characteristics of the EMP table.

```
SQL>SELECT  *
SQL>FROM    TABC
SQL>WHERE   TABLE = 'EMP'
SQL>/

------------------------------------------------------------------
TYPE  CREATOR COMMENT
----- ------- ----------------------------------------

EMP
TABLE SCOTT   Information about company employees
```

10. Concurrency Control Facilities

ORACLE allows multiple user to concurrently UPDATE the same table in a data base. Yet, ORACLE does not require a user to issue locking statements or statements of intent to update. The setting and clearing of locks are the responsibility of ORACLE.

No explicit lock requests are required to insure that concurrent UPDATE operations do not read the same row of a table and attempt to write back that row. ORACLE automatically places locks on individual records to synchronize UPDATE operations preventing the updates from overwriting each other.

A user may not wish to operate on data that has been modified by an operation that is still in progress. To accomplish this serialization the user can place his SQL request between BEGIN TRANSACTION and END TRANSACTION statements.

In addition, placing several SQL statements inside a transaction causes ORACLE to execute these SQL statements as an "atomic act" without permitting interference (UPDATES to the same tables) by other users during the transaction.

ORACLE's automatic update synchronization locks a record (row of a table) at a time. USER requested transactions lock a table at a time.

Example 10-1: Log on as user SCOTT

```
SQL>#DBS PERSONNEL SCOTT/TIGER
Database 'PERSONNEL' opened.
```

Example 10-2: Begin a transaction to calculate the average salaries of each job group within the EMP table.

```
SQL>BEGIN TRANSACTION
SQL>ON TABLE EMP READ
SQL>/
Transaction begun.

SQL>SELECT  JOB,AVG(SAL)
SQL>FROM    EMP
SQL>GROUP BY JOB
SQL>/

JOB          AVG(SAL)
---------- ----------
ANALYST     $3,000.00
CLERK       $1,030.00
MANAGER     $3,287.08
PRESIDENT   $5,750.00
SALESMAN    $1,495.00

SQL>END TRANSACTION
SQL>/
Transaction ended.
```

In the above READ TRANSACTION, all UPDATE activity on the EMP table was suspended for the life of the transaction. All READ activity on the EMP table was allowed to continue concurrently.

Example 10-3: Execute an UPDATE TRANSACTION that gives all employees with the job of ANALYST a 10% raise.

```
SQL>BEGIN TRANSACTION
SQL>ON TABLE EMP UPDATE
SQL>/
Transaction begun.

SQL>UPDATE  EMP
SQL>SET     SAL = SAL * 1.10
SQL>WHERE   JOB = 'ANALYST'
SQL>/
3 records updated.

SQL>END TRANSACTION
SQL>/
Transaction ended.
```

In the above UPDATE TRANSACTION, all UPDATE activity on the EMP table was suspended for the life of the transaction. In addition, all READ transactions on the EMP table also waited for the UPDATE transaction to complete. READ activity on the EMP table that was not a part of a TRANSACTION was allowed to continue concurrently.

Placing several SQL statements inside a transaction causes ORACLE to execute these SQL statements as a "atomic act" without permitting interference (UPDATES to the same tables) by other users during the transaction.

Example 10-4: SELECT the count of employees in department 20 from the EMP table, and UPDATE the DEPT table with the result.

```
SQL>BEGIN TRANSACTION
SQL>ON TABLE EMP READ, DEPT UPDATE
SQL>/
Transaction begun.

SQL>SELECT  COUNT(*)
SQL>FROM    EMP
SQL>WHERE   DEPTNO = 20
SQL>/

COUNT(*)
--------
       7

SQL>UPDATE  DEPT
SQL>SET     EMPCNT = 7
SQL>WHERE   DEPTNO = 20
SQL>/
1 record updated.

SQL>END TRANSACTION
SQL>/
Transaction ended.
```

If the above transaction had been done from within a program it could have used from one to three cursors: one for the BEGIN TRANSACTION, one for the query on the EMP table, and one for the UPDATE of the DEPT table.

TRANSACTIONS may be nested. If they are nested, they must be numbered.

Example 10-5: SELECT the count of employees in department 20 from the EMP table and UPDATE the DEPT table with the result as a nested transaction rather than a single transaction.

```
SQL>BEGIN TRANSACTION 1
SQL>ON TABLE EMP READ
SQL>/
Transaction begun.

SQL>SELECT  COUNT(*)
SQL>FROM    EMP
SQL>WHERE   DEPTNO = 20
SQL>/

COUNT(*)
--------
       7

SQL>BEGIN TRANSACTION 2
SQL>ON TABLE DEPT UPDATE
SQL>/
Transaction begun.

SQL>UPDATE  DEPT
SQL>SET     EMPCNT = 7
SQL>WHERE   DEPTNO = 20
SQL>/
1 record updated.

SQL>END TRANSACTION 2
SQL>/
Transaction ended.

SQL>END TRANSACTION 1
SQL>/
Transaction ended.
```

In doing the above operation as a nested transaction rather than a single transaction, the DEPT table was not locked during the query on the EMP table. This allowed update activity to continue on the DEPT table while the EMP table was being queried. This could be significant depending on the length of time it takes the query to execute. However, the nesting of transactions creates the possibility that TRANSACTION 2 will have to wait for access to the DEPT table which may be locked by another ongoing transaction. The nesting of transactions creates the possibility of deadlock. Doing the entire operation as one transaction has no potential for deadlock because once a transaction has begun, it has acquired all the data resources required for completion.

It is important to remember that the above query/update operation could have been done three different ways: 1) without placing them within a transaction; 2) as a single transaction; 3) as a nested transaction. It is up to the user to decide what level of serialization of operations he requires for a particular application.

The following chart indicates what operations on a table wait, and what operations continue when a READ, READ TRANSACTION, UPDATE OR UPDATE TRANSACTION is active on that table.

CONCURRENCY CONTROL TABLE

	READ	READ TRANS	UPDATE	UPDATE TRANS
READ	ALLOWED	ALLOWED	ALLOWED	ALLOWED
READ TRANS	ALLOWED	ALLOWED	SUSPENDED	SUSPENDED
UPDATE	ALLOWED	SUSPENDED	ALLOWED	SUSPENDED
UPDATE TRANS	ALLOWED	SUSPENDED	SUSPENDED	SUSPENDED

R S I

ORACLE

SQL LANGUAGE

REFERENCE GUIDE

Oracle Users Guide - Version 2.3

S Q L

LANGUAGE REFERENCE MANUAL

TABLE OF CONTENTS

SQL LANGUAGE - REFERENCE GUIDE

INTRODUCTION

SQL is a relational data language that provides a unified set of facilities for query, data manipulation, data definition, and data control. SQL is both a terminal interface for nonspecialists in data processing, and a data sublanguage embedded in host programming language for use by application programmers.

SQL was developed by IBM as the main external interface to be supported by System R, IBM's experimental relational database management system. In 1976, a complete BNF syntax for SQL was published in the "IBM Journal of Research and Development." In 1977, RSI began the development of ORACLE incorporating the SQL language.

ORACLE is based on the relational model of data. SQL is a non-procedural language that operates on normalized data. The advantages of the relational model with a non-procedural language are ease of use, maximum data independence, and flexibility. SQL is an easy to learn English-like language that enhances user productivity. It is a high-level non-procedural language offering greater data independence than conventional procedural database languages. SQL allows complete flexibility in the formulation of statements relating data in the database.

FORMAT NOTATION

This manual uses the following notation to describe the syntax of SQL statements.

CAPITALIZED WORDS	identify words that have specific meanings in SQL.
lower-case words	identify words that are names or labels to be specified by the user.
[ ] Square Brackets	are used to indicate that the enclosed word is optional and may be omitted.
| | Vertical Bars	enclosing vertically stacked items indicate that one of the enclosed items may be chosen.
. . . Ellipsis	indicates that the immediately preceding unit may occur once, or any number of times in succession.

QUERY STATEMENTS

SQL Query Statements consist of one or more Query Blocks. A Query Block starts with and must include a SELECT clause and a FROM clause. The SELECT clause specifies what is to be returned as a result of the query block. The FROM clause specifies what tables and/or views are involved in the query.

A Query Block may optionally contain other clauses as follows:

WHERE	to specify selection criteria for the rows.
GROUP BY	for use with built-in functions.
HAVING	for specifying election criteria on groups.
CONNECT BY	for tree-structured access.
START WITH	for tree-structured access.
INCLUDING	for specifying selection criteria for leafs for tree-structured access.

The values resulting from processing a Query Block can be referred to in the WHERE clause of another Query Block. This is accomplished by nesting Query Blocks within a Query Statement.

```
SELECT  ENAME,JOB
FROM    EMP
WHERE   JOB =
        SELECT   JOB
        FROM     EMP
        WHERE    ENAME = 'JONES';
```

Query Blocks can be nested to any level within a Query Statement, and may be combined with other SQL predicates using boolean AND, OR, and NOT. In the syntax specifications of the SQL statements, "SELECT . . ." is used to denote a nested Query Block.

The SELECT Clause

```
SELECT [ UNIQUE ] |*             |,|column       |, . . .
                  |column        | |table.column|
                  |table.column| |table.*       |
                  |table.*       | |expression   |
                  |expression    | |function     |
                  |function      | |USER         |
```

The SELECT clause specifies the columns to be returned as the result of a query. The SELECT clause may request: all columns; a list of specific columns; the results of arithmetic expressions or built-in functions; or any combination of columns, expressions, and functions. ORACLE will return all rows that satisfy the WHERE clause of the query block. Duplicate rows are not eliminated unless SELECT UNIQUE is specified. UNIQUE is an option rather than a default because the process of elimination of duplicate values requires extra processing.

UNIQUE	indicates that duplicate rows which satisfy the WHERE clause are to be eliminated from the query result.
*	returns all columns from all of the table(s) and view(s) specified in the FROM clause of the query block.
column	specifies the name of a column contained in a table or view specified in the FROM clause of the query block.
table.column	specifies the name of a column qualified by the name of the table that contains the column. Qualified column names are used to eliminate ambiguity when the FROM clause lists multiple tables or views that contain duplicate column names.
table.*	returns all the columns in the table or view specified. The * can be qualified with a table name when there are multiple tables and/or views listed in the FROM clause.
expression	specifies an arithmetic expression

made up of columns and constants that are connected by the operators +, -, *, /. Parenthesis ( ) are used to establish precedence. Note that expressions involving a column value of NULL, will result in a null value unless the NULL Function parameter is used.

function — indicates any of the SQL built-in group functions COUNT, SUM, AVG, MAX, MIN. The presence of a built-in function within a SELECT clause implies a GROUP BY. If the GROUP BY is not explicitly stated, the entire query result is treated as one group and each field in the SELECT clause must be a unique property of the group. See the GROUP BY clause and Built-in-Functions sections for a more detailed description.

USER — returns the name of the user (as specified in the DEFINE USER command) who is executing this SQL statement.

The FROM Clause

```
FROM  |table       |, . . .
      |table label|
      |table*     |
```

The FROM clause lists the tables and views that are referred to by the other clauses in the query block. A query block must contain a SELECT and FROM clause, and may optionally contain a WHERE, GROUP BY, and HAVING clause.

table	specifies the name of a table or view that contains columns referenced by SELECT, WHERE, GROUP BY, HAVING, or ORDER BY clauses.
table label	specifies that the table or view is to be renamed within the context of a query block. The renaming of a table with a label is necessary when the same table or view is listed more than once in the same FROM clause. This mechanism is used to join a table to itself. The temporary label is used in place of the table name to qualify columns referenced by the other clauses within the query block.
table*	specifies that the rows of the table listed in the from clause are to participate in the join if the join-column contains a null value. This is referred to as a "Outer-Join". An outer join table can not be the first table listed in the FROM clause.

The WHERE Clause

```
WHERE  [NOT]  |column       |   |=       |   |column          |   |AND| . . .
              |table.column |   |^=      |   |table.column    |   |OR |
              |constant     |   |>       |   |constant        |
              |NULL         |   |>=      |   |generic-constant|
              |expression   |   |<       |   |NULL            |
              |<column,...> |   |<=      |   |expression      |
              |USER         |   |BETWEEN |   |<column,...>    |
                              |   |IN      |   |<expression,...>|
                                              |SELECT...       |
                                              |USER            |
```

The WHERE clause qualifies the rows that are to be returned as the result of a query. The WHERE clause may contain any combination of predicates that compare fields of rows to constant values, compare two fields of a row with each other, compare fields to expressions, etc. Multiple predicates within the same WHERE clause can be combined to form logical expressions connected by AND and OR with square brackets [ ] used to establish precedence. NOT may be specified prior to any predicate to negate a predicate or a boolean expression. The absence of a WHERE clause indicates that all rows in the table or view specified in the FROM clause, are to be returned.

NOT	specifies that the following predicate or boolean expression is to be negated.
column	specifies the name of a column defined in a table or view specified in the FROM clause of the query block. Note that columns specified here, need not be specified in the SELECT clause.
table.column	specifies the name of a column qualified by the name of the table that contains the column. Qualified column names are used to eliminate ambiguity when the FROM clause lists multiple tables or views that contain duplicate column names.

constant	specifies any numeric or character-string constant literal value. Single quotation marks are required around all character-string constants to distinguish them from column names.
generic-constant	specifies the leading character-string of a literal value. The leading string must be followed by the ellipsis notation "..." and the result must be enclosed in single quotation marks. Specification of a generic constant allows for a search on a leading character-string of a value.
NULL	indicates the absence of a value in the database. Null values are ignored in the evaluation of all arithmetic expressions, and the computation of all built-in functions except COUNT. NULL values are treated as unknowns in the evaluation of logical expressions. The evaluation of logical expressions is described in a separate section of this manual which contains the truth tables for three valued logic.
expression	specifies an arithmetic expression made up of columns and constants that are connected by the operators +, -, *, /. Parenthesis ( ) are used to establish precedence. Note that expressions involving a column value of NULL, will result in a null value unless the NULL Function parameter is used.

<column,...>	specifies a set of numeric or character-string literal values. The set is enclosed in angle-brackets < > and items within the set are separated by commas.
<expression,...>	specifies a set of constant values or expressions. The predicate in the WHERE clause tests the field for inclusion in the set. example: WHERE DNO IN <5+2,17,11*3>
SELECT...	specifies the use of the result of one query in the WHERE clause of another query. The inner query returns a set of values to the WHERE clause of the outer query. The outer query proceeds as though it were given a set of constants in place of the inner query. Query blocks may be nested to any number of levels.
USER	returns the name of the user (as specified in the DEFINE USER command) who is executing this SQL statement.
=	indicates the equal comparison operator.
^=	indicates the not equal comparison operator.
>	indicates the greater than comparison operator.
>=	indicates the greater than or equal comparison operator.
<	indicates the less than comparison operator.
<=	indicates the less than or equal comparison operator.

BETWEEN	indicates the range comparison operator. The range is specified as a pair of constants, expressions, or columns connected by an AND.
IN	indicates the set inclusion operator. IN tests a field for inclusion in a set of values. The comparison operator = may be used in place of IN without changing the meaning of the WHERE clause.
AND	indicates the boolean operator AND. The boolian operators are used to connect predicates to form compound logical expressions within the WHERE clause.
OR	indicates the boolean operator OR.

Built-In Functions

```
|COUNT|   |*            |
|SUM  |   |column       |
|AVG  |   |table.column|
|MAX  |
|MIN  |
```

ORACLE provides five built-in functions as a standard part of the system. These functions may be used in the SELECT clause and the HAVING clause. When a built-in function is used in a SELECT clause, and there is no GROUP BY clause in the query block, the entire table is treated as one group. Only unique attributes of the group may be selected. No Built-in function other than COUNT may be applied to columns defined as CHAR in the CREATE TABLE. With the exception of the COUNT function, null values will not be included in a built-in function unless the NULL Function parameter is used.

COUNT	specifies the count of the set of all fields or rows qualified by the WHERE clause. COUNT includes null fields in its total.
SUM	specifies the arithmetic sum of the values of qualifying fields.
AVG	specifies the arithmetic average of the values contained in the set of qualifying fields.
MAX	specifies the maximum numeric value contained in the set of qualifying fields.
MIN	specifies the minimum numeric value contained in the set of qualifying fields.
*	specifies the count of all rows that satisfy the WHERE clause. The * may only by used with the COUNT function in the form: COUNT(*).

column	specifies the name of a column defined in a table or view specified in the FROM clause of the query block.
table.column	specifies the name of a column qualified by the name of the table that contains the column. Qualified column names are used to eliminate ambiguity when the FROM clause lists multiple tables or views that contain duplicate column names.

Null Value Function

NVL (column,value)

The ORACLE Null Value Function is used to assign a temporary value to null value encountered within an expression. The Null Value Function may be used in a SELECT, SET or WHERE clause anywhere a column name may be used including within arithmetic expressions and built-in-functions.

column	specifies the name of a column within a SELECT, SET or WHERE clause. The column must have been defined as NUMBER within the CREATE TABLE.
value	specifies a temporary numeric value to be assigned to null values encountered during processing.

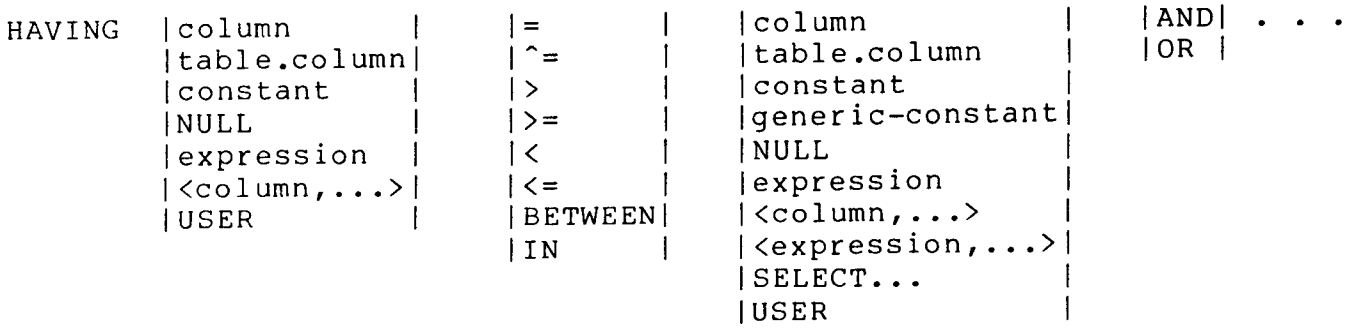
The GROUP BY Clause

```
GROUP BY  |column       |, . . .
          |table.column|
```

The GROUP BY clause is used to partition tables or views into groups according to the values in a column or a list of columns. A built-in set function is then applied to each group. A GROUP BY clause is always used together with a built-in function. When a GROUP BY clause is used, or implied by the presence of a built-in function in the SELECT clause, each field in the SELECT clause must be a unique property of the group.

column	specifies the name of a column defined in a table or view specified in the FROM clause of the query block. Note that columns specified here, need not be specified in the SELECT clause.
table.column	specifies the name of a column qualified by the name of the table that contains the column. Qualified column names are used to eliminate ambiguity when the FROM clause lists multiple tables or views that contain duplicate column names.

The HAVING Clause



The HAVING clause qualifies groups that are to be returned as the result of a query. Each field listed in the HAVING clause must be a unique property of the group. The HAVING clause may contain any combination of predicates to accept certain groups and disqualify others. The predicates can use a built-in function to compare the aggregate property of the group to a constant value or to another aggregate property of the same group. When a query block has both a WHERE clause and a HAVING clause: first the WHERE clause is applied to qualify rows; then the groups are formed; then the HAVING clause is applied to qualify groups. Multiple predicates within the same HAVING clause form logical expressions connected by AND and OR with square brackets [ ] used to establish precedence. The absence of a HAVING clause indicates that all groups formed are to be returned.

column — specifies the name of a column defined in a table or view specified in the FROM clause of the query block. Note that columns specified here, need not be specified in the SELECT clause.

table.column — specifies the name of a column qualified by the name of the table that contains the column. Qualified column names are used to eliminate ambiguity when the FROM clause lists multiple tables or views that contain duplicate column names.

constant	specifies any numeric or character-string constant literal value. Single quotation marks are required around all character-string constants to distinguish them from column names.
generic-constant	specifies the leading character-string of a literal value. The leading string must be followed by the ellipsis notation "..." and the result must be enclosed in single quotation marks. Specification of a generic constant allows for a search on a leading character-string of a value.
NULL	indicates the absence of a value in the database. Null values are ignored in the evaluation of all arithmetic expressions, and the computation of all built-in functions except COUNT. NULL values are treated as unknowns in the evaluation of logical expressions (see Three Values Logic).
expression	specifies an arithmetic expression made up of columns and constants that are connected by the operators +, -, *, /. Parenthesis ( ) are used to establish precedence. Note that expressions involving a column value of NULL, will result in a null value unless the NULL Function parameter is used.
<column,...>	specifies a set of numeric or character-string literal values. The set is enclosed in angle-brackets < > and items within the set are separated by commas.
<expression,...>	specifies a set of constant values or expressions. The predicate in the HAVING clause tests the field for inclusion in the set.

SELECT...	specifies the use of the result of one query in the HAVING clause of another query. The inner query returns a set of values to the HAVING clause of the outer query. The outer query proceeds as though it were given a set of constants in place of the inner query. Query blocks may be nested to any number of levels.
USER	returns the name of the user (as specified in the DEFINE USER command) who is executing this SQL statement.
=	indicates the equal comparison operator.
^=	indicates the not equal comparison operator.
>	indicates the greater than comparison operator.
>=	indicates the greater than or equal comparison operator.
<	indicates the less than comparison operator.
<=	indicates the less than or equal comparison operator.
BETWEEN	indicates the range comparison operator. The range is specified as a pair of constants, expressions, or columns connected by an AND.
IN	indicates the set inclusion operator. IN tests a field for inclusion in a set of values. The comparison operator = may be used in place of IN without changing the meaning of the WHERE clause.
AND	indicates the boolean operator AND.
OR	indicates the boolean operator OR.

The CONNECT BY Clause

CONNECT BY [PRIOR] major-column = [PRIOR] minor-column

In ORACLE, a table may be used to represent tree-structured data. Consider a parts table consisting of assemblies (major) and components (minor). There is one row in the table for each combination of a component and assembly. One assembly can be a component of another assembly, etc. The table contains one column identifying component-number and another column identifying assembly-number. This table then represents a complete bill of materials.

The CONNECT BY clause specifies the selection of rows according to their tree-structure relationship. The clause requires specification of the major and minor columns. The PRIOR keyword is positioned before the major column to indicate that the rows are to be selected going up the tree, or before the minor column to indicate the rows are to be selected going down the tree.

PRIOR	specifies the direction in which rows are to be selected. If the PRIOR keyword is placed before the minor (component) column, the query proceeds down the tree (explosion). If the PRIOR is placed before the major (assembly) column, the query proceeds up the tree (implosion).
major-column	specifies the name of the assembly column.
minor-column	specifies the name of the component column.

The START WITH Clause

```
START WITH [NOT] |column       |   |=       |   |column           |   |AND| . . .
                 |table.column |   |^=      |   |table.column     |   |OR |
                 |constant     |   |>       |   |constant         |
                 |NULL         |   |>=      |   |generic-constant |
                 |expression   |   |<       |   |NULL             |
                 |<column,...> |   |<=      |   |expression       |
                 |USER         |   |BETWEEN |   |<column,...>     |
                                   |IN      |   |<expression,...> |
                                                |SELECT...        |
                                                |USER             |
```

The START WITH clause specifies the rows that are to be used as starting points in queries on tree-structured tables. The START WITH clause may contain any predicate or logical expressions that may be contained within a WHERE clause. The START WITH clause is always used in conjunction with the CONNECT BY clause.

NOT	specifies that the following predicate or boolean expression is to be negated.
column	specifies the name of a column defined in a table or view specified in the FROM clause of the query block.
table.column	specifies the name of a column qualified by the name of the table that contains the column. Qualified column names are used to eliminate ambiguity when the FROM clause lists multiple tables or views that contain duplicate column names.
constant	specifies any numeric or character-string constant literal value. Single quotation marks are required around all character-string constants to distinguish them from column names.

generic-constant	specifies the leading character-string of a literal value. The leading string must be followed by the ellipsis notation "..." and the result must be enclosed in single quotation marks. Specification of a generic constant allows for a search on a leading character-string of a value.
NULL	indicates the absence of a value in the database.
expression	specifies an arithmetic expression made up of columns and constants that are connected by the operators +, -, *, /. Parenthesis ( ) are used to establish precedence. Note that expressions involving a column value of NULL, will result in a null value unless the NULL Function parameter is used.
<column,...>	specifies a set of numeric or character-string literal values. The set is enclosed in angle-brackets < > and items within the set are separated by commas.
<expression,...>	specifies a set of constant values or expressions.
SELECT...	specifies that the result of a query block is to be used in the START WITH clause.
USER	returns the name of the user (as specified in the DEFINE USER command) who is executing this SQL statement.
=	indicates the equal comparison operator.
^=	indicates the not equal comparison operator.

>	indicates the greater than comparison operator.
>=	indicates the greater than or equal comparison operator.
<	indicates the less than comparison operator.
<=	indicates the less than or equal comparison operator.
BETWEEN	indicates the range comparison operator. The range is specified as a pair of constants, expressions, or columns connected by an AND.
IN	indicates the set inclusion operator. IN tests a field for inclusion in a set of values. The comparison operator = may be used in place of IN.
AND	indicates the boolean operator AND. The boolian operators are used to connect predicates to form compound logical expressions.
OR	indicates the boolean operator OR.

The INCLUDING Clause

```
INCLUDING [NOT] |column       |   |=       |   |column           |   |AND| . . .
                |table.column |   |^=      |   |table.column     |   |OR |
                |constant     |   |>       |   |constant         |
                |NULL         |   |>=      |   |generic-constant |
                |expression   |   |<       |   |NULL             |
                |<column,...> |   |<=      |   |expression       |
                |USER         |   |BETWEEN |   |<column,...>     |
                                |   |IN      |   |<expression,...> |
                                                 |SELECT...        |
                                                 |USER             |
```

The INCLUDING clause is used with queries on tree-structured tables. The INCLUDING clause is used in conjunction with the WHERE clause to determine which rows are to be returned as a result of the query. Any rows which are excluded by virtue of not satisfying the WHERE clause, result in exclusion of entire "branches" of the tree structure. Any rows which are excluded by virtue of not satisfying the INCLUDING clause, result only in that row being excluded. In other words, the WHERE clause causes exclusion before the CONNECT BY is applied; the INCLUDING clause causes exclusion after the CONNECT BY is applied.

The INCLUDING clause may contain any predicates or logical expressions that may be contained within the WHERE clause. INCLUDING is an optional clause used in conjunction with the CONNECT BY clause.

NOT	specifies that the following predicate or boolean expression is to be negated.
column	specifies the name of a column defined in a table or view specified in the FROM clause of the query block.
table.column	specifies the name of a column qualified by the name of the table that contains the column. Qualified column names are used to eliminate ambiguity when the FROM clause lists multiple tables or views that contain duplicate column names.
constant	specifies any numeric or

	character-string constant literal value. Single quotation marks are required around all character-string constants to distinguish them from column names.
generic-constant	specifies the leading character-string of a literal value. The leading string must be followed by the ellipsis notation "..." and the result must be enclosed in single quotation marks. Specification of a generic constant allows for a search on a leading character-string of a value.
NULL	indicates the absence of a value in the database.
expression	specifies an arithmetic expression made up of columns and constants that are connected by the operators +, -, *, /. Parenthesis ( ) are used to establish precedence. Note that expressions involving a column value of NULL, will result in a null value unless the NULL Function parameter is used.
<column,...>	specifies a set of numeric or character-string literal values. The set is enclosed in angle-brackets < > and items within the set are separated by commas.
<expression,...>	specifies a set of constant values or expressions.
SELECT...	specifies that the result of a query block is to be used in the INCLUDING clause.

USER	returns the name of the user (as specified in the DEFINE USER command) who is executing this SQL statement.
=	indicates the equal comparison operator.
^=	indicates the not equal comparison operator.
>	indicates the greater than comparison operator.
>=	indicates the greater than or equal comparison operator.
<	indicates the less than comparison operator.
<=	indicates the less than or equal comparison operator.
BETWEEN	indicates the range comparison operator. The range is specified as a pair of constants, expressions, or columns connected by an AND.
IN	indicates the set inclusion operator. IN tests a field for inclusion in a set of values. The comparison operator = may be used in place of IN.
AND	indicates the boolean operator AND. The boolian operators are used to connect predicates to form compound logical expressions.
OR	indicates the boolean operator OR.

The ORDER BY Clause

```
ORDER BY  |column       |  |ASC |, . . .
          |table.column|  |DESC|
          |expression   |
```

The ORDER BY clause indicates the sequence that the query result is to be returned. The ORDER BY clause may contain a major and up to 254 minor sorting fields, with a maximum concatenated sort field of 255 characters. Each sort field may specify ascending or descending order. An ORDER BY clause is not part of a query block, and may only be associated with the first query block of a SQL query statement.

column	specifies the name of a column defined in a table or view specified in the FROM clause of the query block. Note that columns specified here, need not be specified in the SELECT clause.
table.column	specifies the name of a column qualified by the name of the table that contains the column. Qualified column names are used to eliminate ambiguity when the FROM clause lists multiple tables or views that contain duplicate column names.
expression	specifies an arithmetic expression made up of columns and constants that are connected by the operators +, -, *, /. Parenthesis ( ) are used to establish precedence. Note that expressions involving a column value of NULL, will result in a null value unless the NULL Function parameter is used.
ASC	indicates ascending sort order. If no sort direction is specified for a field, ascending is assumed.
DESC	indicates descending sort order.

DATA MANIPULATION STATEMENTS

The INSERT INTO and DELETE clauses provide for addition and deletion of rows of a table. The combination of the UPDATE and SET clauses allows modification of column values within a row or set of rows within a table.

Nested Query Blocks may be used with the INSERT INTO clause to copy data from another table. A WHERE clause is used with the DELETE and UPDATE clauses to specify sets of records to be processed.

Refer to the section on Concurrency Control Statements for information about locking during the execution of data manipulation statements.

The following SQL clauses are provided for Data Manipulation:

INSERT INTO	for adding rows to a table.
DELETE	for deleting rows from a table.
UPDATE	for specifying a table whose rows are to be updated.
SET	for specifying the updates to be made to columns of a row.

The INSERT INTO Clause

```
INSERT INTO  table(column,...): |<constant,...>  |
                                |NULL            |
                                |USER            |
                                |SELECT . . .    |
```

The INSERT statement specifies the adding of a new row or set of rows into a table. Fields that are not present in the insertion statement are given null values. If the row to be inserted has all its fields present in the correct order, the list of column names may be omitted.

table	specifies the name of the table into which the rows are to be inserted.
(column,...)	specifies the names of the columns of the table in the order the values will appear in the INSERT statement. If values are being provided for all columns of a row (with any missing values being indicated by the keyword NULL), and the columns are in the order that they are defined in the CREATE TABLE, then the column list may be omitted.
constant	specifies any numeric or character-string constant literal value that is to be inserted into the database in the specified column. Single quotation marks are required around all character-string constants to distinguish them from column names.
NULL	indicates that the column associated with the NULL is to be null in the database.
USER	returns the name of the user (as specified in the DEFINE USER command) who is executing this SQL statement.

SELECT...	specifies that the result of a query is to be inserted into a table in the database. Query blocks specified within an INSERT statement can be nested to any number of levels.

The DELETE Clause

DELETE table

The DELETE clause specifies the name of the table containing a row or set of rows that are to be removed from the database. The specific rows that are to be deleted are qualified by a WHERE clause. The WHERE clause in a DELETE statement is identical to the WHERE clause in a query and may contain nested query blocks.

table	specifies the name of the table that contains the rows to be removed from the database.

The UPDATE Clause

UPDATE table

The UPDATE clause specifies the name of the table containing a row or set of rows that are to be modified. A SET clause is used to specify the updates which are to be performed on the one or more columns within a row. The specific row or rows to be modified are qualified by means of a WHERE clause. The WHERE clause in a UPDATE statement is identical to the WHERE clause in a query and may contain nested query blocks. Primary keys may not be modified by an UPDATE statement (see CREATE TABLE).

table	specifies the name of a table that is to be modified.

The SET Clause

```
SET    column = |constant  |, . . .
                |expression|
```

The SET clause specifies a column or list of columns to be modified within the table referenced by the UPDATE clause. A SET clause is always used in conjunction with an UPDATE clause. New values for fields that are to be updated may be stated as constants or expressions.

constant	specifies any numeric or character-string constant literal value as the new value for the field. Single quotation marks are required around all character-string constants to distinguish them from column names.
expression	specifies the use of the result of an arithmetic expression as the new value for the field. An arithmetic expression can be made up of columns and constants that are connected by the operators +, -, *, /. Parenthesis ( ) are used to establish precedence.

DATA DEFINITION STATEMENTS

The SQL Data Definition Statements provide for establishing and modifying data definitions within the ORACLE Data Dictionary. The execution of these statements does not require any reorganization activity.

The following statements are provided:

CREATE TABLE	for defining a new TABLE in the database.
EXPAND TABLE	for defining a new COLUMN for an existing TABLE.
DEFINE VIEW	for defining a new VIEW.
DROP	for removing a TABLE or VIEW definition.

The CREATE TABLE Statement

```
CREATE TABLE table

  column(|CHAR(len) [VAR]| [NONULL] [UNIQUE] [UC] [IMAGE]), . . .
         |NUMBER         |
```

The CREATE TABLE statement defines a new table that is to be physically stored in the database. A table may contain from 1 to 255 columns. The CREATE TABLE specifies the name of the table, the names of the columns, and the column data types. The presence of null or duplicate values within a column may be restricted. High-performance access paths may be specified on any columns.

ORACLE automatically maintains an index (IMAGE) for the first column defined in the table. To optimize sequential processing the rows of the table are stored in physical sequence based on this index. This column is automatically treated as a required (NONULL) item.

table	specifies the name of the table that is being defined. The name must be unique within the database. The maximum length of the table name is 30 characters. The first character must be alphabetic.
column	specifies the name of a column defined within the table. Column names must be unique within a table. The column name can have a maximum length of 30 characters. The first character must be alphabetic.
CHAR	indicates the column is to contain alpha-numeric character string values.
len	specifies the maximum length of a value to be stored in a character string field. The length must be a number from 1 to 255.

VAR	indicates that the value stored in a character string field is to be stored in variable length format. Currently, ORACLE stores all character string values in variable length format whether or not VAR is specified.
NUMBER	indicates the column is to contain numeric values. Numeric values are stored internally in base 256 format to maintain maximum precision.
NONULL	indicates that null values are not permitted in the column.
UNIQUE	indicates that no two fields within this column can have the same value. UNIQUE can only be specified if IMAGE is also specified.
UC	indicates that the index to be maintained on this column is to have forward compression only. If UC is not specified, the index will have both forward and backward compression.
IMAGE	indicates that an index is to be maintained for the values in the column. Join operations can be performed only if both columns referenced in the joining predicate are defined as IMAGE.

The EXPAND TABLE Statement

```
EXPAND TABLE table

 ADD COLUMN column(|CHAR(len) [VAR]|[NONULL]|[UNIQUE] [UC] [IMAGE]|)
                  |NUMBER          |
```

The EXPAND TABLE statement adds a new column to an existing table stored in the database. The new column is added to the right side of the table. Existing rows are considered to have null values in the new column until they are updated. Queries and views that were written in terms of the existing table are not affected by the expansion. EXPAND TABLE specifies the name of the table to be enlarged and defines the new column with a syntax identical to that used in the CREATE TABLE statement. The presence of null or duplicate values within the column may be restricted. A high-performance access path (IMAGE) may be specified. EXPAND TABLE is a instantaneous operation. No physical reorganization of any part of the database takes place.

table	specifies the name of the table that is being expanded.
column	specifies the name of a column being added to the table. Column names must be unique within a table. The column name can have a maximum length of 30 characters. The first character must be alphabetic.
CHAR	indicates the column is to contain alpha-numeric character string values.
len	specifies the maximum length of a value to be stored in a character string field. The length must be a number from 1 to 255.

VAR	indicates that the value stored in a character string field is to be stored in variable length format. Currently, ORACLE stores all character string values in variable length format whether or not VAR is specified.
NUMBER	indicates the column is to contain numeric values. Numeric values are stored internally in base 256 format to maintain maximum precision.
NONULL	indicates that null values are not permitted in the column.
UNIQUE	indicates that no two fields within this column can have the same value. UNIQUE can only be specified if IMAGE is also specified.
UC	indicates that the index to be maintained on this column is to have forward compression only. If UC is not specified, the index will have both forward and backward compression.
IMAGE	indicates that an index is to be maintained for the values in the column. Join operations can be performed only if both columns referenced in the joining predicate are defined as IMAGE.

The DEFINE VIEW Statement

DEFINE VIEW view [(column, . . .)] AS SELECT . . .

The DEFINE VIEW statement creates an alternative view of data stored in tables in the database. The definition of a view is similar to the process of stating a query, because the result of any query on one or more tables is itself a table. Therefore, any query formulation can be used in the definition of a view. The DEFINE VIEW statement names the view and optionally names its columns. A view may be defined in terms of other views. Views may be queried in the same way as stored tables; however, DELETE, UPDATE, and INSERT clauses may "not" reference views.

view	specifies the name of the view that is being defined. Table and view names must be unique within the database. The maximum length of the view name is 30 characters. The first character must be alphabetic.
column	specifies the name of a column defined within the view. Column names must be unique within a view. The column name can have a maximum length of 30 characters. The first character must be alphabetic. A view's column names may be drawn from the SELECT clause of the query defining the view if the column names in the SELECT clause are unique.
SELECT...	specifies the use of the result of a query as a view on the database. Any valid query block can be used as a database view. The query blocks may be nested to any number of levels.

The DROP Statement

```
DROP    |TABLE|   name
        |VIEW |
```

The DROP statement removes tables or views from the system. Once a system entity has been dropped, its name may be reused. A table cannot be dropped if the table contains data. A table or view cannot be dropped if another view is defined upon it.

TABLE	indicates the system entity to be dropped is a table. A table may not be dropped until all rows in that table have been deleted.
VIEW	indicates the system entity to be dropped is a view.
name	specifies the name of the table or view to be dropped.

DATA CONTROL STATEMENTS

The SQL Data Control Statements provide for Security and Concurrency Control.

The following SQL statements are provided for Security Control:

DEFINE USER	to define a user of a database and his password.
GRANT	to give privileges on a TABLE or VIEW to a user.
REVOKE	to remove privileges on a TABLE or VIEW from a user.
PASSWORD	to allow a user to change his password.

The following SQL statements are provided for Concurrency Control:

BEGIN TRANSACTION	Lock a resource.
END TRANSACTION	Unlock a resource.

The DEFINE USER Statement

DEFINE USER user-name/password

The DEFINE USER statement adds an authorized user to a secure ORACLE database. Only defined users are permitted to log on to a secure database.

Initially, the user who creates the database is the only authorized user of that database. Thereafter, the creating user can define additional users via the DEFINE USER command. These new users may themselves define additional users, etc.

Users defined by means of the DEFINE USER command are authorized to log on to a secure database and create tables. These users are not allowed any access to any data stored within the data base without data access privileges. Data access privileges are given to a user via the GRANT command.

user-name	specifies the name or identifier of the user being defined. The user must enter this name when logging on to an ORACLE database. The user-name can have a maximum length of 20 characters.
password	specifies the name of the password for the user being defined. The user must enter this password when logging on to an ORACLE database. The password can have a maximum length of 20 characters.

The GRANT Statement

```
GRANT|privilege[,...]          |ON table TO|user-name[,...]|
     |ALL RIGHTS               |           |PUBLIC         |
     |ALL BUT privilege[,...]|

                                                [WITH GRANT OPTION]
```

It is the responsibility of the user who creates a table or view to control access to it. When a user creates a table, only that creating user is privileged to access that table. The creating user may allow other users access privileges on his table via the GRANT command.

The following privileges may be granted:

READ
INSERT
DELETE
UPDATE (by column)
EXPAND

Only the READ privilege may be specified for a view.

Users that have been granted the right to exercise a privilege may or may not have the right to grant the same privilege to other users. The grantor of privileges may permit the grantee to grant the listed privileges to other users by including the clause WITH GRANT OPTION.

privilege	specifies the type of operations that are to be authorized for the table.
ALL RIGHTS	indicates that all privileges are to be granted.
ALL BUT	indicates that all privileges except those listed in the GRANT command are to be granted.
table	specifies the name of the table or view for which the privileges apply. If a view is specified, only the READ privilege may be granted.

user-name	specifies the name of the user or users that are to receive the privileges. User-name is the user-name field specified in the DEFINE USER command.
PUBLIC	indicates that all users of the database are to receive the privileges listed.
WITH GRANT OPTION	specifies that the grantee may grant the privileges listed to other users.

The REVOKE Statement

```
REVOKE |privilege[,...]          | ON table FROM |user-name[,...]|
       |ALL RIGHTS               |               |PUBLIC        |
       |ALL BUT privilege[,...]|
```

Privileges that have been granted by means of the GRANT command may be withdrawn through the use of the REVOKE command. The named privileges are removed from the grantee and from all users to whom he has granted them.

The following privileges may be revoked:

READ
INSERT
DELETE
UPDATE (by column)
EXPAND

Only the READ privilege may be specified for a view.

privilege	specifies the type of operation that is no longer authorized for the table.
ALL RIGHTS	indicates that all privileges are to be revoked.
ALL BUT	indicates that all privileges except those listed in the REVOKE command are to be withdrawn.
table	specifies the name of the table or view for which the privileges are to be revoked. If a view is specified, only the READ privilege may be revoked.

user-name	specifies the name of the user or users whose privileges are to be revoked. User name is the user-name field specified in the DEFINE USER command.
PUBLIC	indicates that all users of the database are to have the listed privileges revoked.

The PASSWORD Statement

PASSWORD password

The PASSWORD statement is used to redefine a user's password. It can only be used by a user to redefine his own password.

password	specifies the name of the new password for the currently logged on user. A password can have a maximum length of 20 characters.

CONCURRENCY CONTROL STATEMENTS

ORACLE allows multiple users to concurrently UPDATE the same table in a database. Yet, ORACLE does not require a user to issue locking statements or statements of intent to update. The setting and clearing of locks are the responsibility of ORACLE. No explicit lock requests are required to insure that concurrent UPDATE operations do not read the same row of a table and attempt to write back that row. ORACLE automatically places locks on individual records (rows) in order to synchronize UPDATE operations, thus preventing the updates from overwriting each other.

A user may not wish to operate on data that has been modified by an operation that is still in progress. To accomplish this serialization the user can place his SQL request between BEGIN TRANSACTION and END TRANSACTION statements. In addition, placing several SQL statements inside a transaction causes ORACLE to execute these SQL statements as a "atomic act" without permitting interference (UPDATES to the same tables) by other users during the transaction.

ORACLE's automatic update synchronization locks a record (row of a table) at a time. USER requested transactions lock a table at a time.

Whereas transaction level control will often be desired in UPDATE operations, it is also useful in READ-only situations when it is required that data being retrieved not be subject to modification during the period of retrieval.

The BEGIN TRANSACTION Statement

```
BEGIN TRANSACTION [tran-id] ON TABLE table | UPDATE |, . . .
                                           | READ   |
```

The BEGIN TRANSACTION statement is used to identify the start of a logical transaction consisting of one or more SQL statements. The BEGIN TRANSACTION must specify those tables (if any) being locked for UPDATE purposes, and those tables (if any) being locked for READ purposes.

Transactions may be nested. When transactions are nested, the BEGIN TRANSACTION statements must be numbered beginning with 1.

tran-id	specifies an integer value. Tran-id must be specified when transactions are nested.
table	specifies the name of a table which will be updated or read.
UPDATE	specifies that the table should be locked for all other update and read transactions.
READ	specifies that the table should be locked to update transactions. Read transactions may concurrently access the table.

The END TRANSACTION Statement

END TRANSACTION [tran-id]

The END TRANSACTION statement is used to terminate a transaction that was started with a BEGIN TRANSACTION statement.

tran-id	specifies an integer value. Tran-id must be specified when transactions are numbered in the BEGIN TRANSACTION statement.

S Q L

PUNCTUATION AND CODING RULES

The following general rules of punctuation apply in writing SQL statements:

Blank Spaces	are used as general purpose delimiters. The number of blank spaces used is optional and will not change the meaning of a SQL statement.
, Comma	is used to separate items in a list.
. Period	is used to separate qualifiers in a qualified name.
; Semicolon	is used to indicate the end of a query block. The semicolon may be omitted in query statements containing only one query block.
: Colon	is used as a general purpose terminator within a SQL statement.
[ ] Square Brackets	are used to establish precedence within logical expressions in WHERE and HAVING clauses.
< > Angle Brackets	are used to enclose sets of literal values.
( ) Parentheses	are used to establish precedence within arithmetic expressions. Parentheses are also used for general purpose enclosure within SQL.

LOGICAL EXPRESSIONS

SQL WHERE and HAVING clauses contain logical expressions. Logical expressions are made up of predicates connected by the boolean operators AND and OR. ORACLE tests fields in a given row in the database to determine if the row satisfies the predicates in the logical expression.

ORACLE allows unknown or null values in the database. Therefore, the evaluation of logical expressions requires the use of three valued logic.

Rows that contain null fields tested by the WHERE clause are assigned an unknown truth value (?). The truth value of a row against the entire logical expression is then evaluated using the three valued logic truth tables depicted below.

AND	T	F	?
T	T	F	?
F	F	F	F
?	?	F	?

OR	T	F	?
T	T	T	T
F	T	F	?
?	T	?	?

NOT	
T	F
F	T
?	?

Only those rows whose overall truth value is true are considered to have satisfied the WHERE clause. If the row's overall truth value is false or unknown the row is disqualified by the WHERE clause.

Null values are ignored in the evaluation of arithmetic expressions.

S Q L

BNF Syntax

```
sql-statement :: = query
                 | dml-statement
                 | ddl-statement
                 | control-statement
dml-statement :: = insertion
                 | deletion
                 | update
query :: = query-block [ ORDER BY ord-spec-list ]
insertion :: = INSERT INTO receiver : insert-spec
receiver :: = table-name [ ( field-name-list ) ]
field-name-list :: = field-name
                   | field-name-list , field name
insert-spec :: = query-block
               | lit-tuple
deletion :: = DELETE table-name [ where-clause ]
update :: = UPDATE table-name [ where-clause ]
          SET set-clause-list [ where-clause ]
where-clause :: = WHERE boolean
set-clause-list :: = set-clause
                   | set-clause-list , set-clause
set-clause :: = field-name = expr
query-block :: = select-clause
                 FROM from-list
                 [ WHERE boolean ]
                 [ GROUP BY field-spec-list ]
                 [ HAVING boolean ]
                 [ CONNECT BY [PRIOR] field-spec = field-spec]
                 [ START WITH boolean ]
                 [ INCLUDING boolean ]
select-clause :: = SELECT [ UNIQUE ] set-expr-list
                 | SELECT [ UNIQUE ] *
sel-expr-list :: = sel-expr
                 | sel-expr-list , sel-expr
sel-expr :: = expr
            | var-name . *
            | table-name . *
from-list :: = table-name [ var-name ]
             | from-list , table-name [ var-name ]
field-spec-list :: = field-spec
                   | field-spec-list , field-spec
ord-spec-list :: = field-spec [ direction ]
                 | expr
                 | ord-spec-list , field-spec [ direction ]
direction :: = ASC
             | DESC
```

```
boolean :: = boolean-term
           | boolean OR boolean-term
boolean-term :: = boolean-factor
                | boolean-term AND boolean factor
boolean-factor :: = [ NOT ] boolean primary
boolean-primary :: = predicate
                   | [ boolean ]
predicate :: = expr comparison expr
             | expr BETWEEN expr AND expr
             | expr comparison table-spec
             | < field-spec-list>  = table spec
             | < field-spec-list > [ IS ]  IN table-spec
table-spec :: = query-block
              | literal
expr :: = arith-term
        | expr add-op arith-term
arith-term :: = arith-factor
              | arith-term mult-op arith-factor
arith-factor :: = [ add-op ] primary
primary :: = field-spec
           | set-fn ( expr )
           | COUNT ( * )
           | NVL ( field-spec , constant )
           | constant
           | ( expr )
field-spec :: = field-name
              | table-name . field-name
              | var-name . field-name
comparison :: = comp-op
              | [ IS ] IN
comp-op :: = =
           | ^=
           | >
           | >=
           | <
           | <=
add-op :: = +
          | -
mult-op :: = *
           | /
set-fn :: = AVG
          | MAX
          | MIN
          | SUM
          | COUNT
```

```
literal :: = < lit-tuple-list >
           | lit-tuple
           | constant
lit-tuple-list :: = lit-tuple
                  | lit-tuple-list , lit-tuple
lit-tuple :: = < entry-list >
entry-list :: = entry
              | entry-list , entry
entry :: = [ constant ]
constant :: = quoted-string
            | number
            | NULL
table-name :: = name
image-name :: = name
name :: = identifier
field-name :: = identifier
var-name :: = identifier
integer :: = number
ddl-statement :: = create-table
                 | expand-table
                 | define-view
                 | drop
create-table :: = CREATE TABLE table-name ( field-defn-list )
field-defn-list :: = field-defn
                   | field-defn-list , field-defn
field-defn :: = field-name ( type [ , type-mod ] )
type :: = CHAR ( integer ) [ VAR ]
        | NUMBER
type-mod :: = NONULL
            | IMAGE [ image-mod ]
image-mod :: = UNIQUE
             | UC
expand-table :: = EXPAND TABLE table-name ADD COLUMN field-defn
define-view :: = DEFINE VIEW table-name
                 [ ( field-name-list ) ] AS query
drop :: = system-entity name
system-entity :: = TABLE
                 | VIEW
control-statement :: = define-user
                     | password-spec
                     | revoke
                     | begin-trans
                     | end-trans
define-user :: = DEFINE USER user-defn
user-defn :: = user-name/password
password-spec :: = PASSWORD password
```

```
grant :: = GRANT [ auth ] table-name TO user-list
         [ WITH GRANT OPTION ]
auth :: = ALL RIGHTS ON
        | operation-list ON
        | ALL BUT operation-list ON
user-list :: = user-name
             | user-list , user-name
             | PUBLIC
operation-list :: = operation
                  | operation-list , operation
operation :: = READ
             | INSERT
             | DELETE
             | UPDATE [ ( field-name-list ) ]
             | EXPAND
revoke :: = REVOKE [ auth ] table-name FROM user-list
begin trans :: = BEGIN TRANSACTION [ tran-number ]
                 ON TABLE  table-name  trans-type
tran-number :: = ( integer )
trans-type :: = UPDATE
              | READ
end trans :: = END TRANSACTION [ tran-number ]
```

R S I

ORACLE

TERMINAL INTERFACE

USER FRIENDLY INTERFACE (UFI)

Oracle Users Guide - Version 2.3

U F I

USER FRIENDLY INTERFACE

TABLE OF CONTENTS

UFI

USER FRIENDLY INTERFACE

I. INTRODUCTION

The User Friendly Interface (UFI) provides the capability for ORACLE users to utilize SQL to access and manipulate a database directly via display terminals.

UFI can be utilized by programmers to try out various SQL commands and table designs interactively. This capability is particularly useful in testing during program development. The reporting facilities can often meet one-time programming requirements without having to write a program.

UFI can be utilized by data administrators in evaluating and implementing design decisions concerning what tables should be created in the database. UFI also provides a way to quickly and easily implement decisions about controlling user access to stored data.

UFI enables users to:

- enter SQL statements and observe the results on the display screen.
- control the format of the display with UFI commands.
- edit the current SQL statement with UFI commands.
- route an output report to a system file and optionally schedule it to be printed.
- create and run stored routines (command files) containing SQL statements and UFI commands.
- specify the SQL Work Area.
- cancel the SQL statement in progress.

UFI is the primary facility provided with ORACLE for interactive processing of SQL statements from a user terminal.

II. OPERATION

The two modes of operation for UFI are processing SQL statements and processing UFI commands. The mode is indicated by the prompt (**SQL>** for SQL statements and **UFI>** for UFI commands). The mode can be changed by simply entering a "CR" (return) following the prompt. This is used to switch from SQL mode to UFI mode or vice versa. UFI commands are used to edit SQL statements, control UFI files, control display formats, and specify UFI control commands.

A convention exists whereby a user can issue UFI commands while in SQL mode. If the UFI command is preceded by a pound sign "#", the command will be interpreted as a UFI command.

A user initiates a UFI session and logs into ORACLE by entering "SQL" and responding to the prompt for database name.

```
> SQL
RSI
Welcomes you to ORACLE
Release 2.3
Enter database-name [user-name/password]: demo
SQL>
```

If the database is secure (created with a user-name), then the optional user-name/password parameter must be specified. In this case the user-name must have been specified with a DEFINE USER statement and all access will be controlled by the GRANT privileges.

When the "SQL" commands is typed the database name, and if appropriate the user-name and password may optionally be included on the same command line.

```
> SQL pers scott/tig
RSI
Welcomes you to ORACLE
Release 2.3
Enter database-name [user-name/password]: pers scott/tig
SQL>
```

When signing on to a secure database, it might be desirable to do so without having the password displayed (or echoed) on the screen. This can be accomplished by specifying only the user-name (and no slash), in which case the user receives a prompt for the password. When responding to this prompt, the password being entered is not displayed on the screen.

```
> SQL pers scott
RSI
Welcomes you to ORACLE
Release 2.3
Enter database-name [user-name/password]: pers scott
Enter password:
SQL>
```

Following log on, UFI is in SQL mode and the user can directly enter SQL statements, or he can switch modes and enter UFI commands. If a user wishes to switch to another database, a UFI command allows him to do so without terminating the UFI session. When the session is finished, the user must switch to the UFI mode and log off with the EXIT command.

III. SQL STATEMENTS

SQL statements are entered in free form utilizing as many lines as desired (to a maximum of 60 lines or a maximum of 2048 characters). Multiple SQL clauses may be combined on a single line, or an individual clause may be spread over several lines. Indentation is optional and is generally used to clarify the meaning. UFI will process SQL statements containing up to a total of 50 columns in the SELECT clauses. The punctuation and coding rules for SQL statements are described in the ORACLE SQL Language Reference Manual.

A SQL statement is made up of one or more lines, followed by a line containing only a slash (/). Note that a SQL Comment (initiated by a "/*" in any position) will not be interpreted as a statement termination. The line containing only a slash causes UFI to combine all lines preceding it into a single SQL statement and send it to ORACLE. If another line containing only a slash is entered immediately following the execution of the SQL statement, that statement will be executed again.

A SQL statement in process may be canceled by depressing the Control Key and entering "C" (i.e., ^C). Under the IAS operating system, it is necessary to abort the task following the ^C. Under the UNIX operating system, the SQL statement may be canceled with the "Delete" key.

The following sequence of lines would be interpreted as a single SQL statement.

```
SQL>/*
SQL>   Find the name and department of employees
SQL>   who have the same job as Jones or a salary
SQL>   greater than Ford's salary.
SQL>*/
SQL>SELECT          ENAME, DEPTNO
SQL>FROM            EMP
SQL>WHERE           JOB IN
SQL>      SELECT    JOB
SQL>      FROM      EMP
SQL>      WHERE     ENAME = 'JONES'
SQL>OR SAL >
SQL>      SELECT    SAL
SQL>      FROM      EMP
SQL>      WHERE     ENAME = 'FORD'
SQL>/
```

IV. UFI COMMANDS

UFI commands are used to edit SQL statements, specify display formats, route UFI output, specify the database, and specify the SQL work area. The format of all UFI commands is:

```
[#]command [argument-string]
```

The space between the command and the argument string is optional. The optional # sign appended to the beginning of the command is used to enter UFI commands when in SQL mode (in response to a SQL prompt).

1. EDIT COMMANDS

For purposes of the Editor, all lines of a SQL statement are considered to be numbered sequentially and must be referenced as such ("1" referring to the first line, "3" referring to the third line, etc.)

Line

This command is used to position to a specified line thus changing which line is the current line. The new current line will be displayed at the terminal.

```
L line-number
```

Add

This command adds a string of characters to the end of the current line. The string is considered to start with the first non-blank character (following the "A") and end with the last non-blank character on the line.

```
A string
```

Delete

This command is used to delete the current line. After the current line is deleted the new current line is the line after the deleted line. If the deleted line was the last line the new current line is the new last line.

```
D
```

Change

This command is used to change the contents of the current line. The changed line is displayed at the terminal.

C /string1/string2[/]

/	is a single character which delimits string1 and string2. Any character not contained in string1 or string2 may be used. The closing delimiter is optional.
string1	is the string to be searched for in the current line. If string1 is not found an error message is printed.
string2	is the string which replaces string1 if string1 is found in the current line.

String1 may contain an ellipsis (...). An ellipsis represents any number of intervening characters. There are are four possible ways to use an ellipsis:

string1...string2	is any string that starts with string1, continues with any number of intervening characters, and ends with the first occurrence of string2.
...string	is any string that starts at the beginning of the current line and ends with the first occurrence of string.
string...	is The first string that starts with string and ends at the end of the current line.
...	is the entire current line.

Insert

This command is used to insert a line after the current line. The inserted line becomes the new current line. The lines following the inserted line are specified with a number one greater than before the insertion (see the L command).

I new-line

new-line	is any string of characters to be inserted after the current line.

List

This command is used to list all lines of the current SQL statement.

LIST

Run

This command is used to cause the previously executed SQL statement, together with any editing performed on that SQL statement, to be executed.

R

2. UFI DISPLAY FORMAT COMMANDS

UFI Display Format commands are utilized to control the format of the display of the results of processing a SQL statement. These commands provide the following functions:

- Control the option of column headers being displayed.
- Control the line size.
- Specify display formats for data.
- Control default size of numeric data columns.
- Control the truncation/wrap-around option for display lines longer than the line size.

Column Headers

This command allows the user to alternately turn on and off the option of having column headers with the displayed data. When being used, column headers will appear once for each command. The column headers will be the ORACLE column names as specified in the table or view definition. When displayed data is the result of an expression or SQL function, the expression or function will be used as the header. The default condition is to display headers.

HEADING

This command will be followed by a message indicating that the heading option has been turned on or off.

Linesize

This command allows the user to override the default line size for UFI output (display of data returned by ORACLE in response to a SQL statement).

LINESIZE number

number	is the line size in bytes. The default size is 80 and the maximum size is 132.

Display Format

This command allows the user to specify display formats for particular columns of data being displayed as a result of a SQL query statement. If no display formats are specified,

- character values will be displayed as character values with the length determined by the maximun size specified in the ORACLE data dictionary (as provided when the column was defined).
- numeric values will be displayed as 10-digit numbers (unless overriden with the NUMWIDTH command). Fractional numbers will be displayed with as many fractional digits as possible within the field size.
- fields which were entered with IAF as "date-type" fields will be displayed as a number representing a julian date.

The Display Format command is specified as follows:

```
FORMAT [ [col_name   ]    format ]
       [ [expression ]           ]
```

where:

col-name — is the column name within a database table.

expression — is the expression used in a SQL query (including arithmetic operators, constants, and SQL functions).

format — is the format definition as described in TABLE 3-1.

If the format command is used without specifying a format, it will have the effect of removing a previously specified format for the column or expression, and replacing it with the default. For example:

```
FORMAT salary
```

has the effect of removing a previously defined format for the salary column.

If the format command consists of only the term "FORMAT", UFI will display all formats which are currently specified as other than default.

TABLE 3-1 DISPLAY FORMAT SPECIFICATIONS

PARM	MEANING
An	defines a character field n bytes in length.
DATE	defines a date format (only for IAF entered data) of the form mm/dd/yy.
EDATE	defines a date format (only for IAF entered data) of the form dd/mm/yy.
9	defines each digit of a numeric variable. Leading zeros are not displayed.
.	defines the decimal point within a numeric variable. The position is used for arithmetic alignment and the period is displayed.
,	causes a comma to be inserted in the display. Omitted if there are no digits to the left of this position.
$	causes a dollar sign to precede the number.
MI	causes a minus sign to be displayed to the right of a negative number. The default is to the left.
PR	causes the variable to be displayed within "< >" brackets when negative.
0	may be used instead of a "9" to designate a digit. Normally leading zeros are suppressed, however a zero in the format will cause every digit position to be filled.
V	defines the position of a decimal point within a numeric variable. The position is used for alignment in arithmetic statements, but the decimal point is not displayed.
B	causes the variable to be displayed as blanks if the value is zero.

The following are examples of various formats:

Format	Value	Displayed
999.99	56.478	56.48
999V99	56.478	5648
9,999	8410	8,410
9,999	639	639
99999	607	607
09999	607	00607
9999	-5609	-5609
9999MI	-5609	5609-
9999PR	-5609	<5609>
B999	564	564
B999	0	
99.99	124.98	##.##
$99.99	45.23	$45.23
DATE	2441453	12/23/80
a20	Customer	Customer
A5	Customer	Custo mer

Notes:

1. If a numeric value is larger than the format allows, it is displayed as "#" signs.

2. If a character value is larger than the format allows, it is either truncated or wrapped around, depending on whether truncation or wrap-around is specified at the time (the default is truncation).

Number Width

This command allows the user to override the default size (10) for numeric fields. If specified, it will apply from that point forward for all columns containing numeric data.

NUMWIDTH number

number	is the column width in bytes. The default width is 10.

Wrap-around

This command specifies that if a line of UFI output is longer than the LINESIZE specification, the excess data will be wrapped around and displayed on successive lines. The wrapping of data will always occur on a field boundary (i.e., values will never be split at the end of a line). The maximum amount of data per row which can be returned by UFI in a SQL statement is 1000 bytes. Wrap-around is the default condition.

WRAP

Truncation

This command specifies that if a line of UFI output is longer than the LINESIZE specification, the excess data is truncated. The data will be wrapped unless this command is issued. If this command is issued, data will be truncated for the remainder of the UFI session, or until a WRAP command is issued.

TRUNC

3. UFI FILE COMMANDS

UFI file commands are used to create command files (files containing SQL statements which can subsequently be input to UFI as an alternative to terminal input). A command file can be executed from within UFI by simply entering a special character followed by the file name. The special character is the "@" sign, except in UNIX where the "!" is used. UFI file commands also are provided for creating output files for subsequent printing.

Save

This command is used to create and open an operating system file to be used as a command file, to write the current SQL statement to that file, and to close that file. Subsequent SQL statements can be added to the file with the APPEND command.

```
SAVE file-name
```

Append

This command is used to add the current SQL statement to an existing operating system file which was originally created with a SAVE command. The APPEND command opens the file, positioning it at the end; writes the current SQL statement to the file; and closes the file.

```
APPEND file-name
```

Spool

This command is used to open an operating system file to which all output generated by UFI will be written, or to cause a previously opened file to be closed and optionally scheduled for printing.

SPOOL option

option — determines which of the three functions will be performed as mentioned above. The options are:

file-name — Open the specified file for spooling. If a file for spooling is already opened, it will be closed.

OUT — Close the file currently opened for printing, and schedule that file for printing. This option assumes that the operating system supports spooling and that it has been generated to do so. Note that once the file has been printed, the file may, depending on the operating system, be deleted.

OFF — Close the file currently opened for spooling.

Page

This command causes a form feed to be placed in the open spool file.

PAGE

3. UFI CONTROL COMMANDS

Change Database

This command allows the user to close the current database and open another database without terminating UFI. If the database being opened is a secure database, the user-name and password parameter must be specified. If there is an error in opening the new database, an error message will appear and the previous database will still be open.

```
DBS database-name [user-name/password]
```

Terminate UFI

This command terminates a UFI session.

```
EXIT
```

Worksize

This command sets the size of the SQL Work Area for this user (terminal) in the ORACLE communication region. This command would be used in response to an ORACLE error message indicating insufficient space in the SWA.

```
WORKSIZE number
```

number	is the size of the SWA in K bytes. The number specified must be from 1 to 16. The default value of 3 has been determined to be adequate for the majority of SQL statements.

R S I

ORACLE

INTERACTIVE APPLICATION FACILITY

USER'S GUIDE

Oracle Users Guide - Version 2.3

I N T E R A C T I V E A P P L I C A T I O N F A C I L I T Y

TERMINAL OPERATOR USER'S GUIDE

TABLE OF CONTENTS

**tbs - means "to be supplied"

I N T R O D U C T I O N

1.0 **Manual Objectives and Use**

This manual describes the operation of an IAF application. It is intended mainly for persons who will be writing IAF applications. Persons reading this should be familiar with SQL. It instructs the terminal operator on the procedures for executing and controlling the application processing.

After reading this manual the user should be able to execute the desired application, cause data to be added, retrieved, updated, and deleted from the database, and control the order or flow of the processing.

EXECUTING AN IAF APPLICATION

2.1 IAP Execution

An IAF application is executed interactively from the operator's terminal. A CRT or video type terminal is required. IAF is distributed with support for the DEC VT52, VT100, and Perkin Elmer OWL. Terminals compatible with this list may also be used. Other terminal types can be used, but their characteristics must be defined to IAF using the procedure outlined in Appendix A.

IAF applications are specified to, and generated by, the Interactive Application Generator (IAG) utility described in the IAF Application Design Guide. Following the generation process, an application is immediately available for execution.

To execute an application the user must invoke the Interactive Application Processor (IAP) by entering the following command from the terminal:

IAP <applname> [<terminal type>]

<applname>	is the name of the application specified to the IAG at generation time.
<terminal type>	Specify the terminal type identifier. For the IAF supplied types the identifiers are: "VT100", "VT52", "OWL". This parameter is optional, and if omitted the installation defined default type will be used. See Appendix A for more information on terminal types.

If the application database is secured the following message will prompt the operator to supply a valid userid and password:

```
SECURE DATABASE:  ENTER NAME AND PASSWORD

     NAME: scott
PASSWORD : xxxxx
```

The entered password will be displayed as x's on the screen. If valid, application processing will continue.

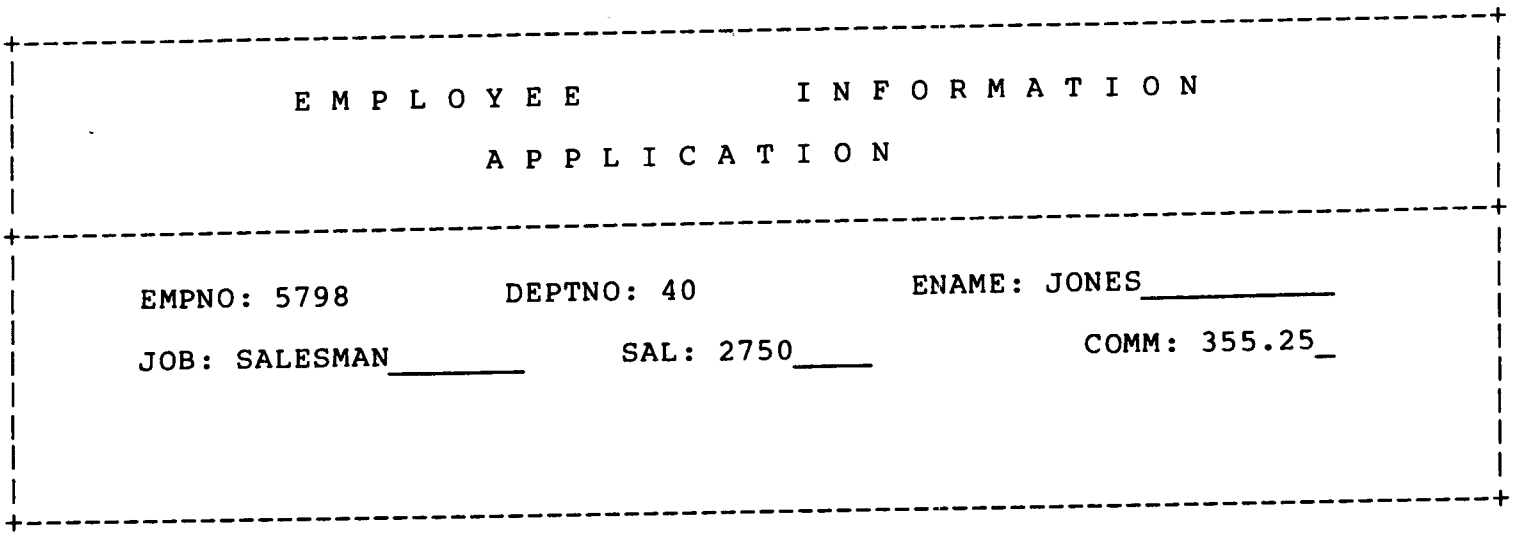
In addition to database access, the operator must have been granted privileges to read and modify the database tables referenced within the application. Access privileges to specific tables and columns are verified during application execution, and violations reported at that time.

2.2 **Definition of Terms**

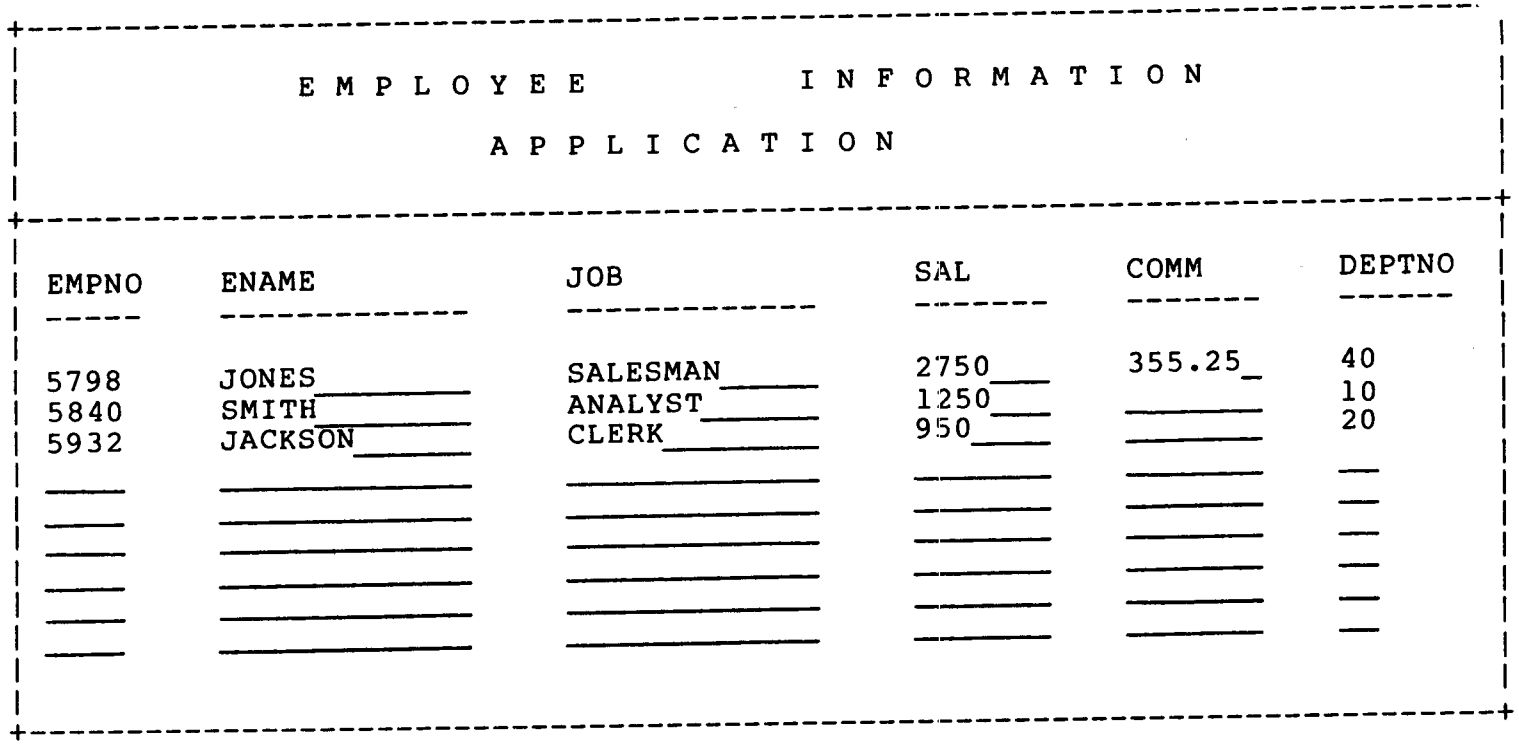
This section will define some terms which will be used in describing the operations of an IAF application. It is assumed that the reader understands basic computer concepts and the operation of the terminal device. An understanding of the ORACLE Data Base Management System, and the SQL data language is essential.

APPLICATION IAF applications allow an operator to maintain his database interactively from a video terminal. In addition to storing and retrieving data, the application helps the operator by checking the input, and indicating when an error has been made. Requiring the operator to enter correct input prevents invalid data from being stored.

BLOCK A block in IAF corresponds closely to a view or a table in SQL. One block corresponds to one SQL table. Blocks are made up of one to twenty records and can hold a corresponding number of rows from its associated table. Operators enter data into records of a block and may request IAF to insert, update, delete or query records associated with that block.



Block 'EMP1' - Single Record Area



Block 'EMP2' - Multiple Record Areas

Figure 2.1

The top diagram in Figure 2.1 shows a block for entering employee information. The information displayed is from Jones' row in the employee table. A block provides the operator with a window to view one or more rows in a single database table.

RECORD - A record corresponds closely to a row of a table or view. A record is composed of fields. Rows are identified on the status line of the display as being stored records or not. A stored record is one that exists in the database. Records that are stored may be updated or deleted. Records that have data entered into them but not yet stored may be subsequently inserted or have queries executed against them. A query executed against a record will attempt to retrieve records with columns that match the data entered in that record.

CURRENT RECORD - Although the operator may view multiple records, only one record may be processed at a time. When adding new records, the data items for one record are entered stored prior to beginning the next one. The current record area is indicated by the location of the cursor on the screen.

FIELD - A field is a data item on the screen. They are identified by labels which are displayed either above or to the left of the data entry area. Reverse video or underlining characters identify a field's data entry and display area. A field may correspond to a column in the table or contain data to aid the operator. In Figure 2.1 the field "EMPNO" corresponds with the column EMPNO in the EMP table.

Fields which are not columns of the block's table are generally used to provide entry aids for the operator. For instance, when entering a part number, it is helpful to display the associated part name for verification. SQL SELECT statements may be associated with any field in a block that are executed when a value is entered into it. These SELECTs may then load values into other fields of the block which are then displayed.

There are other attributes associated with each field that control how a field may be used, what kinds of values it may accept, etc. More information on SELECTs and attributes may be found in the Application Design Guide.

2.2.1 **IAF Screen Format**

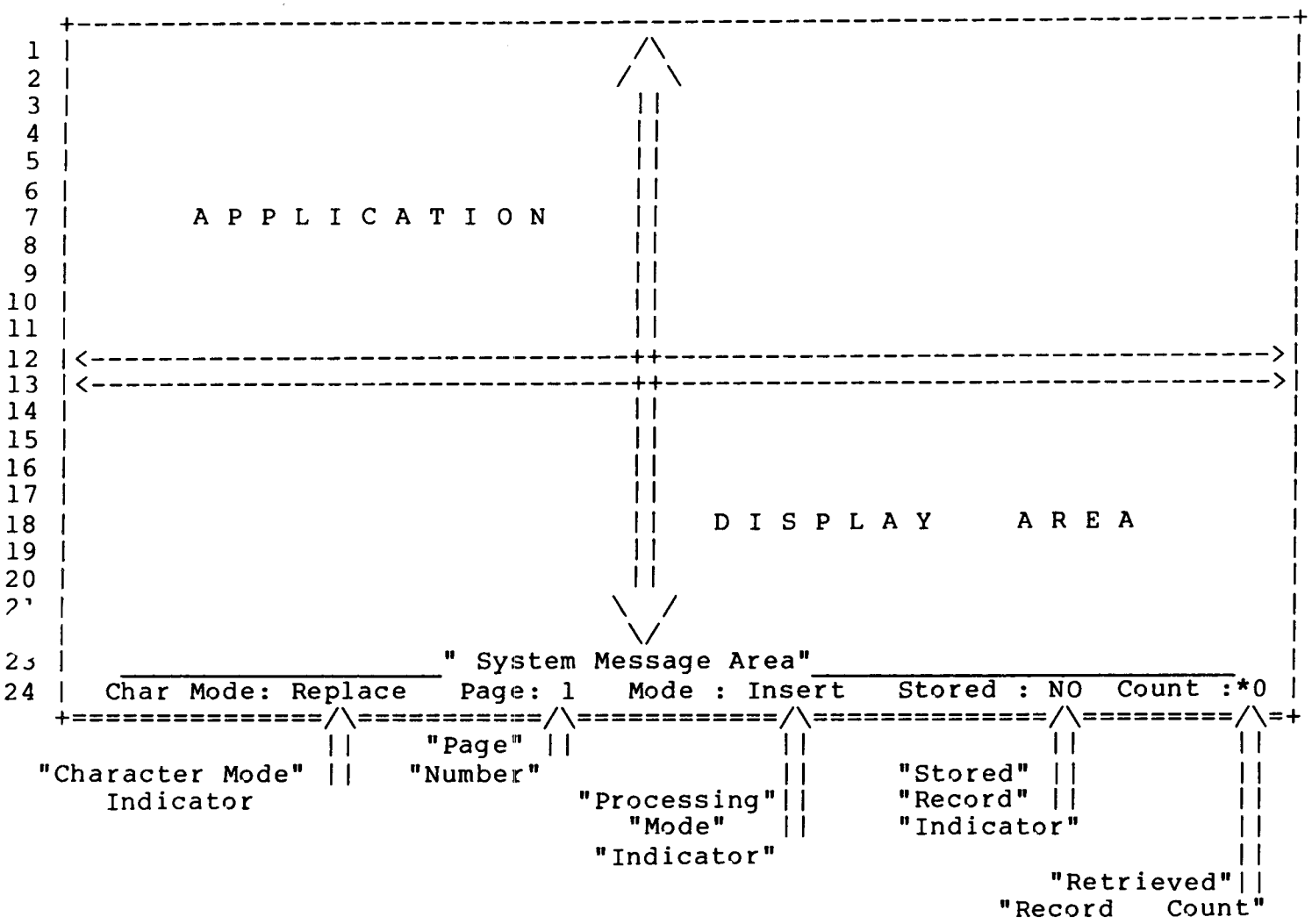
Figure 2.2 is a layout of an IAF screen. All screen pages have the same format. Lines 1 through 22 contain application fields and descriptive text. Lines 23 and 24 are reserved for use by the IAP. Line 23 is the "System Message Area" where status, help, and error messages are displayed. Line 24 contains status indicators and is also used as an operator entry area for the "QUERY" and "PRINT" functions.

The following is a description of each of the indicators appearing on the status line:

"Page x" is the number of the currently displayed application page. "Char Mode:" indicates whether an entered character will either 'Replace' the character in the cursor location, or be inserted('Insert') to its left. "Mode :" indicates that the operator may either query and update existing records (QUERY/UPDATE), or insert new records (INSERT). "Stored :" tells the operator whether the data displayed within the 'current' record area is stored in the database. Following a "QUERY" request, the number of records which the operator has already viewed is accumulated in the "Count:" indicator. The status areas are discussed in more detail in the section on "Keyboard Functions".

2.3 **Keyboard Functions**

As data is entered and retrieved, the operator controls the application processing through a set of predefined keyboard functions. A summary of these functions is provided in Figure 2.3. Each function is invoked by depressing one or more keys from the keyboard of the CRT. Since terminals vary greatly in number of function keys and layout, each terminal will have a unique sequence for invoking a particular function. However, each terminal can display what function keys do what by depressing the "escape" key, followed by the 'k' (lowercase) key. This key layout can subsequently be printed out by the operator with the PRINT function described later. Additionally, the function keys may be customized by the user by modifying the crt definition IAF uses. This is described in Appendix A.



IAF Screen Layout

Figure 2.2.

Next Field	Perform edit checks on field before advancing to next field in record
Previous Field	Advance cursor to previous field in block.
Clear Field	Clear the contents of the current field from cursor to end of field
Next Record	After Query: Move cursor to next record. If at last record, retrieve next record and display. After Insert: Move cursor to next area and initialize for data entry.
Previous Record	Move cursor up one record area on page
Next Set	After Query Only: Display next set of selected records, one in each record area.
Next Block	Terminate processing of current block and begin processing of next block
Previous Block	Terminate processing of current block and begin processing of previous block
Clear Block	Clear all the data in all record areas of block. Insert Mode: Initialize default values and move cursor to first record area. Query Mode: Release selected records and move cursor to first record area.
Clear Form	Clear data in all application blocks. Processing resumes in first block
Insert	Store current record into database. Must be in "Insert" processing mode
Query	Request selection of specified records. Must be in Query/Update processing mode

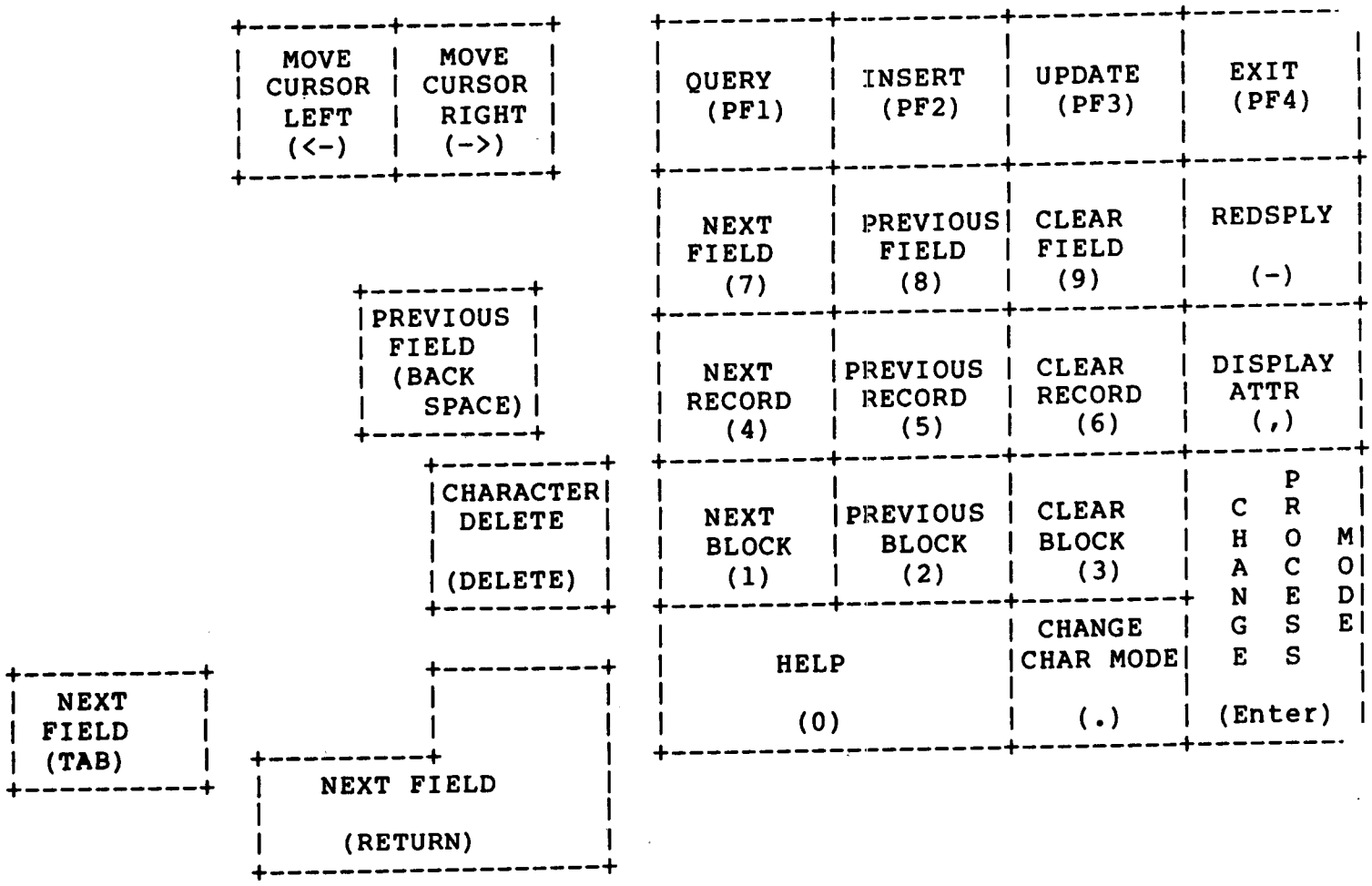
" IAP Keyboard Functions "

Figure 2.3 - Part 1 of 2

Update	Request that the changed data fields in the current record be permanently recorded in the database. Must be in Query/Update processing mode
Delete	Request that the current record be deleted from the database. Must be in Query/Update processing mode
Change Processing Mode	Change from one processing mode (Insert or Query/Update) to the other.
Change Character Mode	Change from one character mode (Replace or Insert) to the other.
Move Cursor Left	Move cursor left one position within current field.
Move Cursor Right	Move cursor right one position within current field.
Help	Display help message for current field
Display Attributes	Display the attributes for the current field.
Redisplay	Redisplay current screen page after communications or terminal failure.
Exit	Terminate IAP application processing
Print Form	Request printing of 1) Current page 2) Entire Form 3) Function key layout

▪ IAP Keyboard Functions ▪

Figure 2.3 - Part 2 of 2



FUNCTIONS	KEYSTROKES
EXIT	CTRL - Z
DELETE	CTRL - D
CLEAR FORM	ESC - C
DISPLAY FUNCTION KEYS	ESC - K
NEXT SET	ESC - S
PRINT FORM	ESC - P

IAF Function Key Layout - VT-100

Figure 2.4

2.3.1 **Processing Modes and Actions**

An IAF application may be processed in either of two modes: "INSERT" or "QUERY/UPDATE". By selecting a processing mode the operator states their intention to either insert new records or retrieve existing records with the possibility of making changes. Therefore, each mode places restrictions on the functions which the operator may perform. When initially executed, the application is in "INSERT" mode.

CHANGE MODE Function:
An application may be switched from one mode to the other by depressing the 'CHANGE MODE' key. The status indicator "Mode:" on the bottom line will alternate between "INSERT" and "QUERY/UPDATE" as the CHANGE MODE key is depressed.

In an empty block, changing from QUERY mode to INSERT/UPDATE mode will cause the first record to be initialized with default INSERT values. Changing back to QUERY mode will clear the default values.

The following functions are available only in the mode indicated.

Insert Mode

INSERT Function:
This action will cause the contents of the current record area to be inserted into the database table. The application must be in 'INSERT' mode. If the current record is already stored, the request will be rejected. When a record has been sucessfully inserted the "Stored: " indicator will change form 'NO' to 'YES'.

Query/Update Mode

QUERY Function:
This action will retrieve those records in the database table who satisfy the defined criteria. There are two ways to state these conditions. Prior to selecting the QUERY function the operator may enter data into any field. When the query is executed only those records that have a corresponding column value equal to the entered field values will be returned.

For instance, after entering data into this sample screen

DEPTNO: 30 JOB: SALESMAN

the following 'where' clause would be generated in the query:

```
WHERE  deptno=30 AND job='SALESMAN'
```

and only those rows where the 'deptno' column is equal to 30 and 'job' column is equal 'SALESMAN' would be returned.

The second way a user may conditionally retrieve rows is by entering a specific consdition to be met. After the QUERY key has been entered the operator will be prompted on line 24 of the screen for the additional text. Expanding on the above example, the operator may only want to see those salesman in department 30 with a salary greater than $2000. The following response:

```
QUERY WHERE?: sal > 2000
```

will cause the 'Where' clause:

```
WHERE deptno=30 AND job='SALESMAN' AND sal>2000
```

to be generated. Only those records which satisfies all these conditions will be returned. Essentially, any condition that could be specified in SQL directly could be specified in IAF providing it may be stated on one line. The user however must be aware of what table is being queried and know the names of the columns in that table. Refer to the SQL language User's Guide for additional details on constructing a 'Where' clause.

Following the entry of all conditions, IAF will display rows returned by the query in each of the blocks records. The 'Count :' indicator will reflect the number of records displayed, and a YES in the stored record indicator (Stored :) indicates the current record is in the database. If the query results in more rows than can be initially displayed, the 'NEXT RECORD' and 'NEXT SET' functions are used to view the remaining records.

Each time a new query is executed the results of the previous query are lost.

UPDATE Function:
Following a QUERY request, the retrieved records may be updated. One record is updated at a time. The operator moves the cursor within the 'current' record to the fields to be changed. Only fields which were designated as 'updatable' may be modified. When all the desired fields have been entered, the UPDATE function is invoked to have the modified data written to the database.

DELETE Function:
The DELETE function deletes the 'current' record from the database. The deleted record is scrolled of the screen, and each remaining record is scrolled up on position. The last record will contain the next record returned by the query or will be blank.

2.3.2 Processing in Either Mode: Field Control

NEXT FIELD:
This function will advance the cursor to the next field in the current record area. If data has been entered into the present field it will trigger the field's edit checking. If an error is detected, an error message is displayed, and the cursor position remains unchanged. The error must be corrected before the cursor can be advanced.

If the cursor is positioned to the last field of the record, NEXT FIELD will advance the cursor back to the first field of the current record area. If the next field of a record is defined on a different screen page, the new page is displayed and the cursor positioned to the appropriate field.

A field may have been defined to use the auto skip feature. In this case if data is entered into every character position, the cursor will automatically skip to the next field. This implied NEXT FIELD will also trigger field editing.

Fields which can not be entered by the user will be skipped over.

PREVIOUS FIELD:
This function is similar to NEXT FIELD except the cursor advances in a backward direction to the previous field. If the cursor is positioned to the first field in the record area, PREVIOUS FIELD will move the cursor to the last field in the current record area. No edit chacking takes place when this function is executed.

If the previous field is on another screen page, the new page is displayed, and the cursor positioned to the appropriate field.

CLEAR FIELD: This function will erase the contents of the current field from the current cursor position to the end of the field.

2.3.2.1 Character Control

CHANGE CHAR MODE:
Two modes of character entry are supported. In 'Replace' mode, each entered character replaces the character previously displayed at the cursor location, and the cursor is moved one space to the right. In 'Insert' mode, entered characters are inserted at the cursor location, with all the characters from the cursor position to the end of the field shifted to the right one position.

The CHANGE CHAR MODE function will change from one character mode to the other. The current character mode is displayed in the 'Char Mode:' indicator on line 24.

MOVE CURSOR LEFT:
This function moves the cursor to the left one position.

MOVE CURSOR RIGHT:
This function moves the cursor to the right one position.

DELETE CHARACTER:
This function will delete the character pointed to by the cursor. Remaining characters in the field are shifted one space to the left, overlaying the deleted character.

2.3.3 Record Control

NEXT RECORD:

Insert Mode:
For a single record block, NEXT RECORD will erase the data area and reinitialize the area with the default values.

A multi-record block will advance the cursor into the next record area, and initialize the fields with their default values. If positioned to the last record area, all the areas will scroll upward one record area, with the top area disappearing from view. The cursor will remain in the bottom area which will be cleared and initialized with the default field values.

Query/Update Mode:
For a single record area block, NEXT RECORD will display the next record retrieved.

For multi-record blocks, the cursor will advance to the next record area. If positioned to the last record area, all areas will scroll upward, and the next retrieved record is displayed at the bottom. The top record will disappear from view.

As each new record is displayed the 'Count:' indicator will be incremented by 1.

NEXT SET OF RECORDS
This function allows the operator to display the next 'n' records, where 'n' is the number of records within the block. The 'Count:' indicator is incremented by the number of new records displayed. This function is used in QUERY/UPDATE mode only.

Referring to the sample blocks in Figure 2.2, this function will display the next record in block 'EMP1', whereas the next nine are displayed in 'EMP2'.

For multi-record blocks, PREVIOUS RECORD will move the cursor up one record area. If positioned to the top area, 'At Top of Block' message is returned and the cursor is unchanged.

Once an inserted or retrieved record has scrolled off the top of the viewing area it can not be viewed again without issuing a new query.

Clear Record:
Clear the content of all fields in the current record area. For INSERT mode, the fields are initialized with their default values. If the current record area contains a database record (indicated by Stored : YES) it can not be cleared. CLEAR BLOCK or CLEAR FORM must be used to clear the record area.

2.3.4 Block Control

IAF applications can contain one or more **blocks.** When the application is executed, processing begins with the first defined block.

NEXT BLOCK:
This function will terminate the processing of the current block, causing the first screen page of the next block is displayed. The data within the current block is retained, and available if the block is reprocessed. The cursor is positioned to the first field within the first record area. Blocks are accessed in the order in which they were defined to the IAG. The NEXT BLOCK function in the last block will take the user back to the first block.

PREVIOUS BLOCK:
This function will terminate the processing of the current block and cause the first screen page of the previous block to be displayed The cursor is positioned to the first field within the first record area. Blocks are accessed in the reverse order of definition. If positioned to the first block, PREVIOUS BLOCK will take the user to the last block.

CLEAR BLOCK:
This function will clear the field contents in the current block. All record areas will be cleared. If processing a QUERY, the records which have not been viewed are lost.

CLEAR FORM:
This function clears an entire form or application. All the data within each block is cleared, and processing continues in the first block. The application is in the same state as when initially executed.

2.3.5 Help Functions

HELP Message:
This function will display a help message for the current field. The message is displayed on line 23 in the System Message Area. The content of the message is entered at application generation time.

DISPLAY ATTRIBUTES:
This function will display the attributes of the current field in the message area at the bottom of the screen.

SHOW FUNCTION KEYS
This function will display the function key assignment for the current terminal type.

2.3.5 Control Functions:

REDISPLAY:
In the case of a communication line error, or terminal failure, this function will cause the last screen to be redisplayed.

QUIT:
This function causes normal termination of the IAP.

PRINT:
This function allows the operator to print the current screen page, all the pages of the application, or the function key layout. When PRINT is requested the operator must specify the name of the file where the printed output will be stored. Any valid operating system file name is permitted. Following the file specification the following message will appear on line 24:

```
Select: 0)Abort 1)Current Page 2)Form 3)Funct Key ? _
```

After choosing 1, 2, or 3, the following message will appear:

```
Send File to System Printer ? _
```

A 'Y' response will cause the IAP to issue a system command to have the contents of the created print file printed on the system defined printer.

This page intentionally left blank.

COMMENTS PLEASE

Please assist us in improving this manual and in correcting any documentation errors. Forward your comments to:

Documentation Coordinator
Relational Software, Inc.
3000 Sand Hill Road, Bldg. 3-180
Menlo Park, CA 94025

MANUAL: INTERACTIVE APPLICATION FACILITY -
Terminal Operator's Guide

SUBJECT _____

PAGE _____ **INCORRECT** _____ **UNCLEAR** _____ **INCOMPLETE** _____

COMMENTS _____

_____

_____

_____

_____

_____

_____

_____

_____

Name:

Organization:

Address:

Telephone:

I N T E R A C T I V E - A P P L I C A T I O N

F A C I L I T Y

APPLICATION DESIGN GUIDE

TABLE OF CONTENTS

I N T R O D U C T I O N

1.0 Manual Objectives and Use

This manual presents the Interactive Application Facility. It explains the purpose and features of IAF, and describes the procedures for generating an IAF application. The intended audience includes those persons responsible for application design and generation. The level of presentation assumes a working knowledge of the ORACLE Data Base Management System and the SQL Language.

2.0 Structure of the Document

This manual is divided into four sections:

Section 2 - "IAF Overview" - Describes the purpose of IAF and explains the component utilities; Interactive Application Generator (IAG) and Interactive Application Processor(IAP).

Section 3 - "Application Structure and Design" - Presents the features and structure of an IAF application.

Section 4 - "Application Definition - Interactive Application Generator" - Describes the process of defining and generating an application. The generation of the sample "employee" application is discussed.

Section 5 - "Advanced Application Techniques" - Presents some additional techniques for using the features of IAF.

Appendix A - "CRT Interface Utility" - Describes the IAF utility allowing a user to specify a non-standard CRT for use with IAF.

I A F - O V E R V I E W

2.1 Introduction

IAF is an application facility which provides full screen communication with a CRT terminal device, interpretation of operator requests, data validation, and the necessary database operations to store, retrieve and update the requested data. Basic editing functions are supported which verify data types, check ranges, and test for existence within a predefined table of values.

Most CRT terminal types may be used providing they have basic cursor control features and have been previously defined to IAF. As distributed, IAF supports the VT100, VT52, OWL, or a compatible device. Appendix A describes generating new crt definitions.

An application consists of one or more **screen pages.** Figure 2.1 is an example of a single screen page. Each page may contain one or more data input and display areas. A display field may correspond to a column in a database table. In figure 2.1 each display field corresponds to a column in the "EMP" table in the ORACLE sample "PERSONNEL" database. The page may be enhanced with prompts, help text, and format characters to improve readability.

Operators interact with IAF applications via a predefined set of keyboard functions. These functions allow the operator to move from field to field, screen to screen, initiate insert, retrieval and update operations, control the application processing modes, and request help information.

Data is entered a field at a time. After a field is entered, the cursor is automatically moved to the next field area. Edit criteria may be associated with any input field. A field may be tested for data type (character or number), format (ie. date), value range, or existence in a table of acceptable values. If the input field is incorrect, the user is immediately notified, and required to correct the error before the process can continue. In this manner the user is guided through a display until the input process is completed.

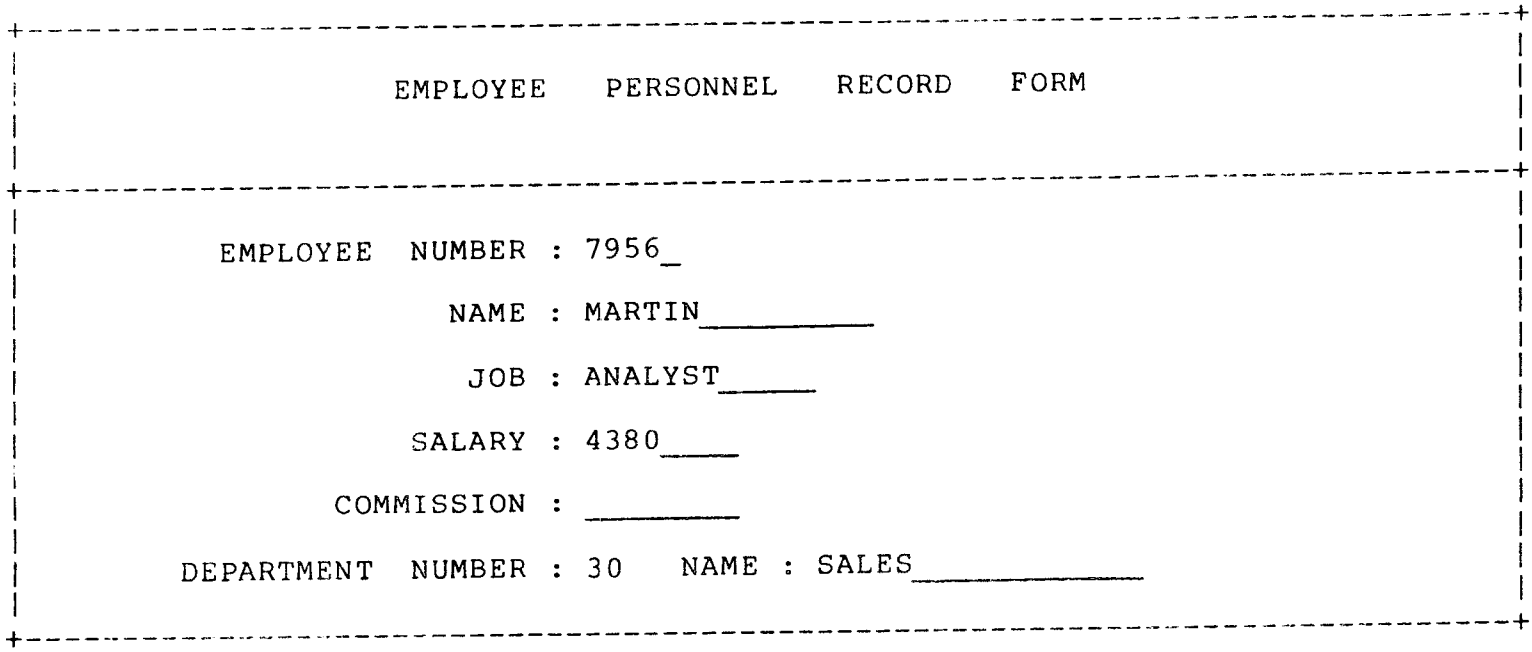


Figure 2.1 - 'Sample Application Screen Layout'

When requisite displays fields have been filled, the data may be inserted into the database via keyboard function. Data is stored a row at a time. Once stored, that row is immediately available to other users.

An IAF application may be used to retrieve previously stored data. Upon retrieval, the data is formatted and displayed on the user's terminal. Data is retrieved one row at a time. Once displayed, the user is free to modify any field which has been designated for update. When all modifications are completed, a keyboard function will initiate the updating of the database. Data which is no longer needed may be deleted.

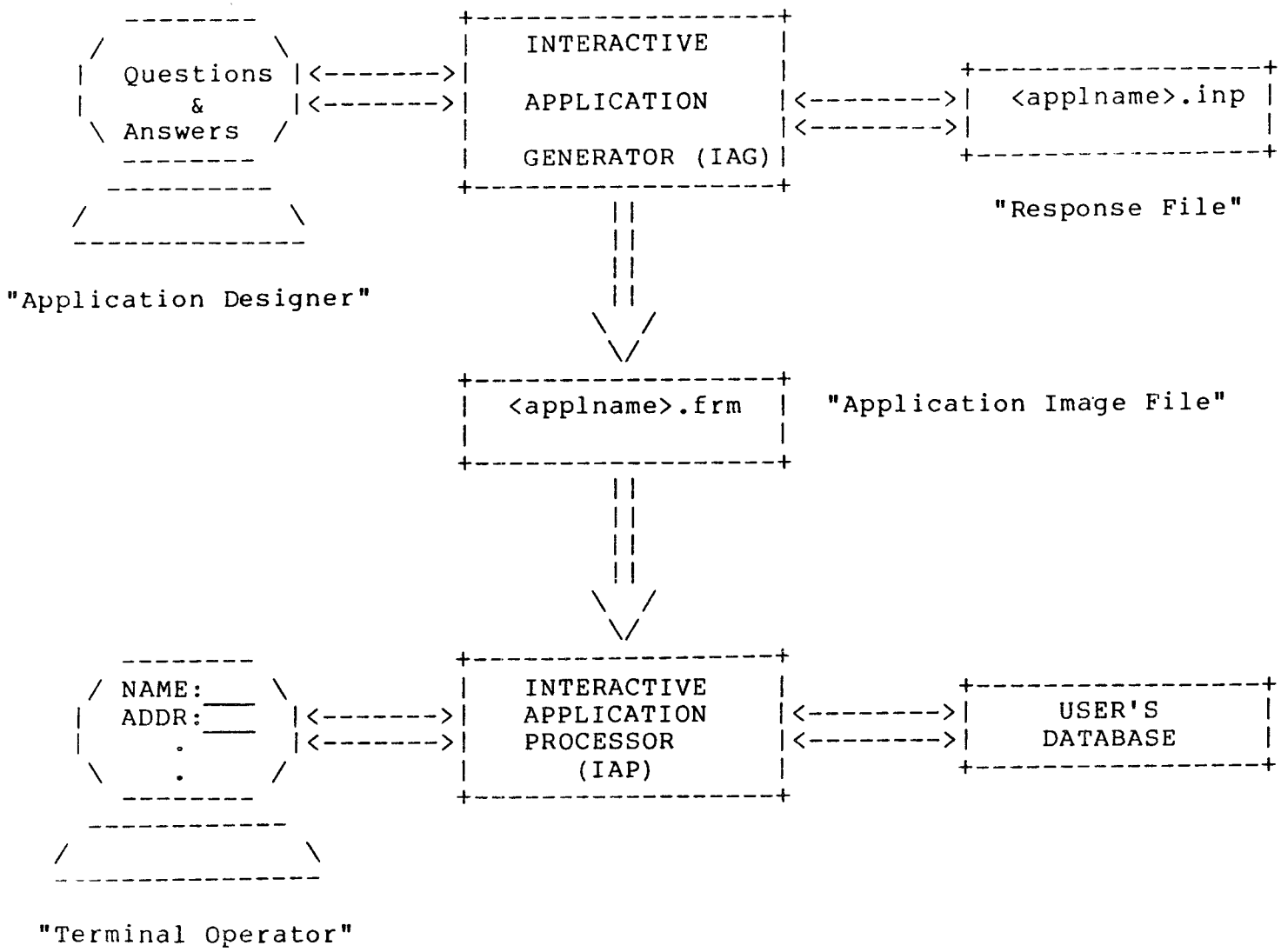
2.2 **Component Description**

The Interactive Application Facility consists of two utility programs. The **Interactive Application Generator (IAG)** interactively communicates with the designer to define the user's application. The **Interactive Application Processor (IAP)** executes the defined application under the operator's control. Figure 2.2 illustrates the relationship between these utilities.

2.2.1 **Interactive Application Generator**

The IAG utility is executed from the application designer's terminal. The designer must specify the name of the application being defined. For new applications IAG will begin the question and answer session. For existing applications a previously created reponse file may be used. The questions address the following areas:

- Database to be referenced within the application.
- Specification of associated database tables and columns.
- For each field (column) within a table:
  - Edit criteria to be applied to input data.
  - Initial value assigned to this field.
  - SQL statement to be executed when the field is entered or retrieved.
  - Placement of field on a screen page
- Placement of prompts, explanatory text, and line drawing characters on each screen page.



Application Development Process

Figure 2.2

These questions will be asked repetitively until the application is defined. When the definition is complete, IAG compiles the responses into an internal format and stores the result on disk. The generation process is now complete and the application can be executed.

As the designer responds to each question, the question text and associated response is saved in an input work file. This file may be used as an alternate input source in subsequent sessions with IAG. Hence, by altering a saved answer file and re-executing IAG, existing applications can be easily modified. A standard text editor may be used to modify the reponse file.

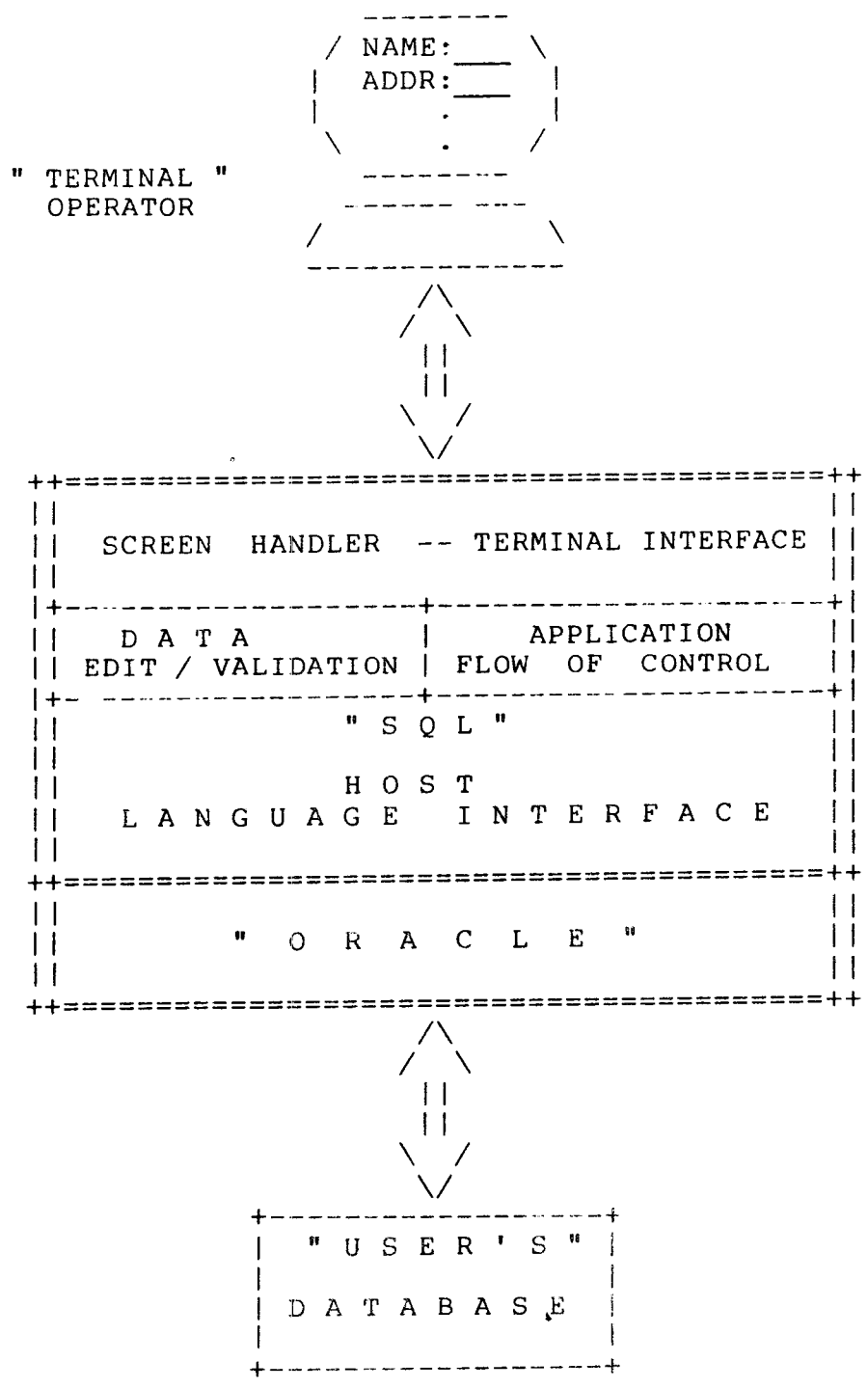
2.2.2 **Interactive Application Processor**

An IAF application is executed by invoking the IAP utility. The operator specifies the name of the 'image file' previously created by IAG. If the referenced database is secure, the user will be prompted for a valid user id and password. In addition, a user must be granted privileges to the desired data.

Figure 2.3 illustrates the relationship of IAP to the terminal operator and ORACLE database. On the front end IAP provides the screen handling functions which communicate with a CRT terminal, interpret the operator's request, control the processing flow, and validate, convert, and format the data. The back-end uses the standard facilities of the Host Language Interface to execute SQL statements which communicate with the user's database.

The 'image' file contains a set of tables which describes the application. The information provides the screen formats, individual field descriptions, and overall structure. IAP uses these tables to guide the application processing.

The IAP utility is viewed by ORACLE as any other application program using the SQL interface. As the operator enters data, the appropriate edits are performed and SQL (SELECT, INSERT, UPDATE, or DELETE) statements are executed against the database.



Interactive Application Processor - IAP

Figure 2.3

The operator uses the IAP function keypad to control the application flow. The operator is free to choose which screens to process and the order of processing. Repetitive processing of a screen allows the operator to enter or retrieve multiple rows of data. When the operator's tasks are completed, the 'Quit' function will terminate IAP. For additional information on executing IAP refer to the "IAF Terminal Operator's Guide".

A P P L I C A T I O N - D E S I G N

3.1 Introduction

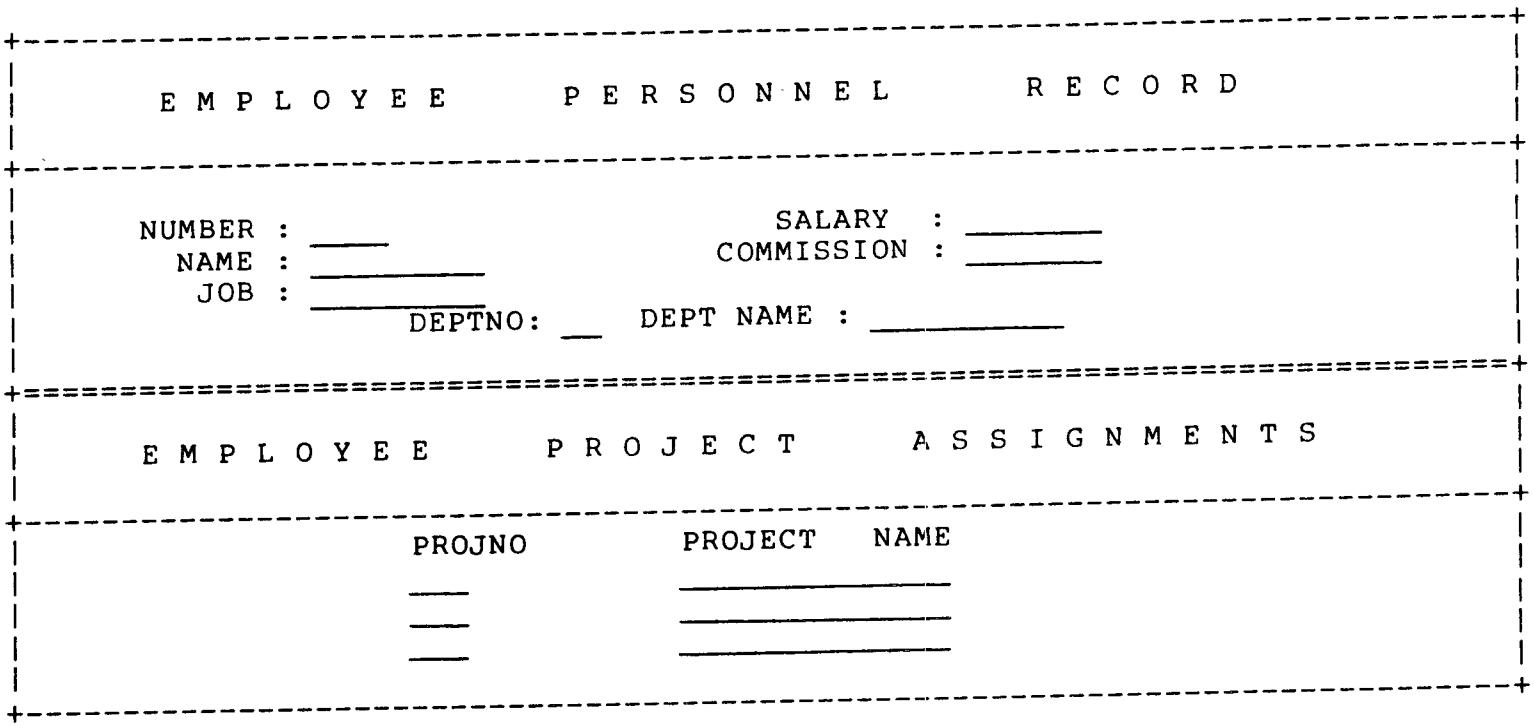
This section will present the basic structure of an IAF application. The objective is to provide the reader with a general understanding of an application's components, providing sufficient information to understand the design and definition of a simple application. The details on defining an application will by deferred to Section 4 : The Interactive Application Generator (IAG)

3.2 Sample Application

Throughout this chapter references will be made to two sample applications. These applications were designed to utilize most of the features of IAF, and will illustrate the concepts being presented.

The first application uses the "EMP", "DEPT", and "PROJ" tables within the ORACLE "PERSONNEL" database. They were chosen because the reader is already familiar with the data in these tables from the SQL Language Examples. A few simple screens will be defined which allow entry and retrieval of data from these tables. Figure 3.1 provides the application's screen layouts which will be referenced in this manual and the "IAF Terminal Operator's Guide". Figure 3.2 shows the columns defined for these database tables.

The second application is an on-line order entry system. It is designed to allow operators to enter orders by storing the information directly into an ORACLE database. Figure 3.3 shows the three screen forms available to the order operator.



"Employe" Application Screen Layout

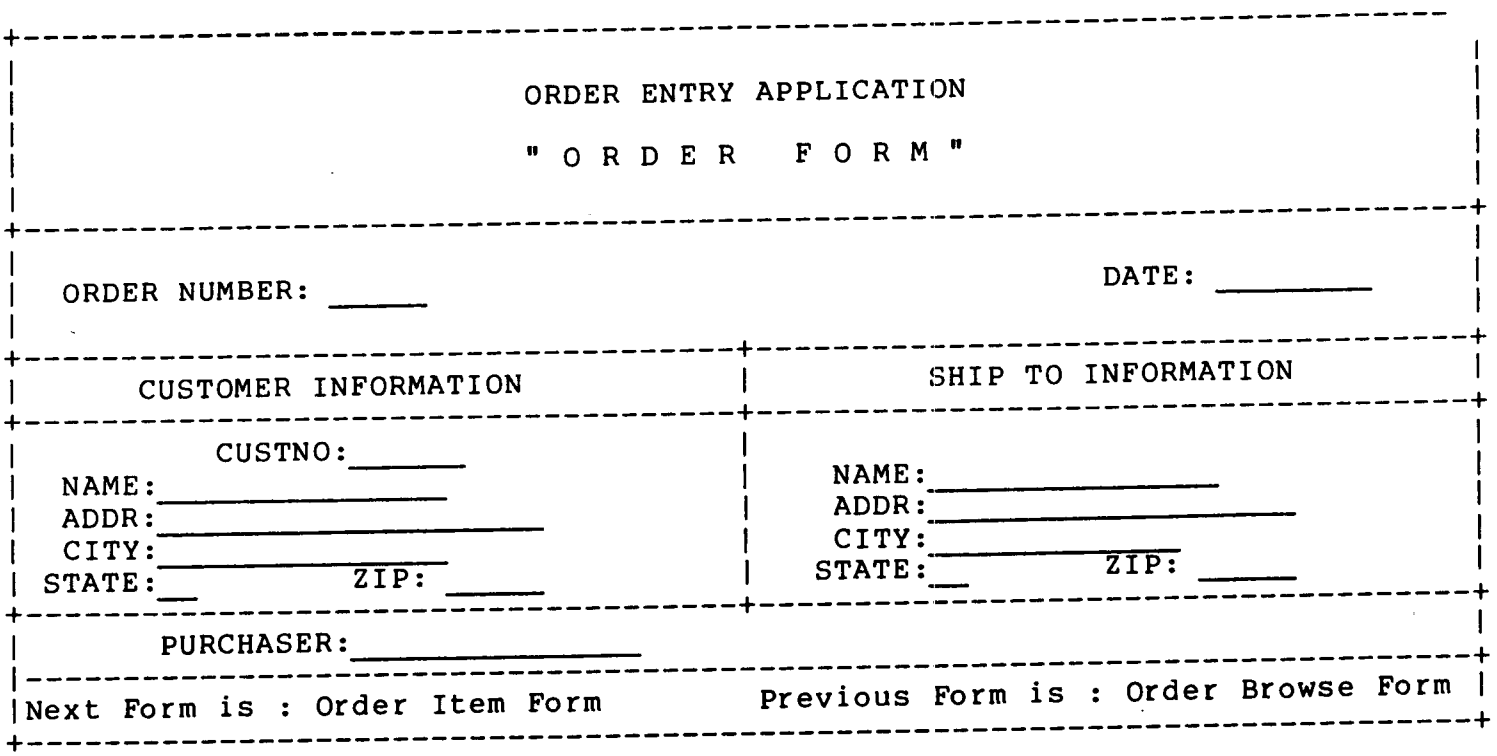
Figure 3.1

Database: 'personnel'

Table Name	Column Name	Data Type	Length	Image
emp	empno	number		unique
	ename	char	10	non-un
	job	char	9	non-un
	sal	number		
	comm	number		
	deptno	number		non-un
dept	deptno	number		unique
	dname	char	14	non-un
	loc	char	13	non-un
	empcnt	number		
pe	empno	number		non-un
	projno	number		non-un
proj	projno	number		unique
	pname	char	10	non-un
	budget	number		
	empcnt	number		

"Employee" Application - Database Tables

Figure 3.2



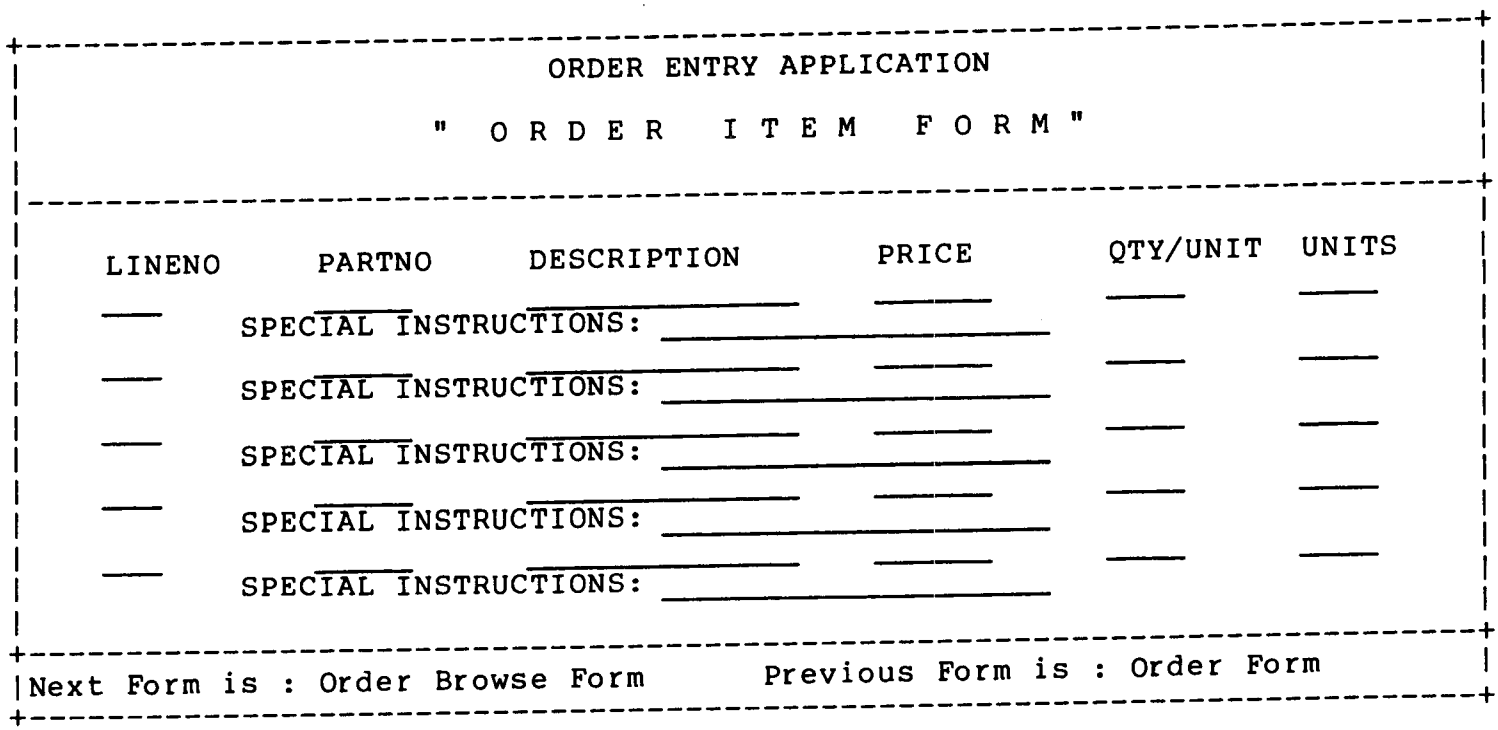


Figure 3.3 - 'Sample Application - Form Layouts'

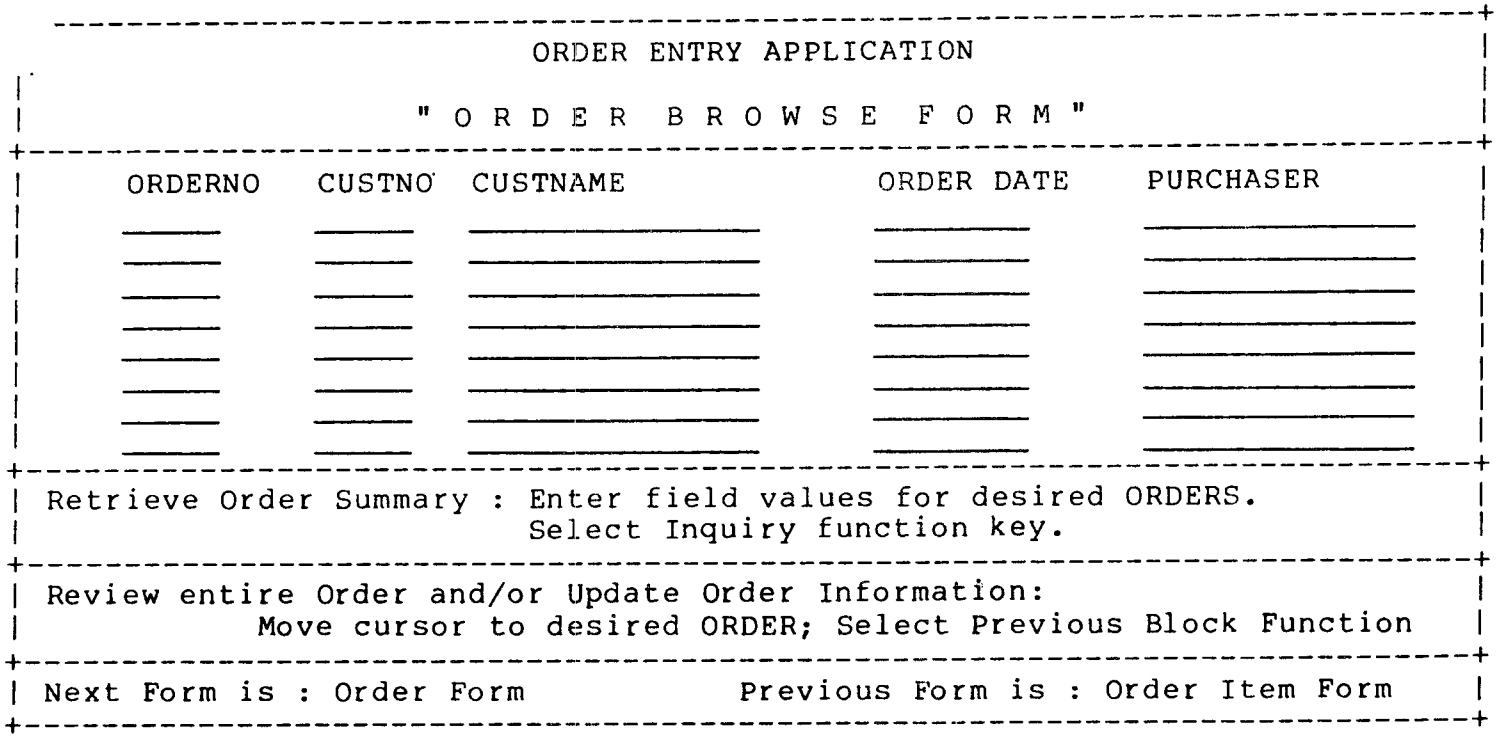


Figure 3.3 - 'Sample Application Form Layouts' (Continued)

To enter an order the operator executes IAP to bring up the "ORDER FORM". This form allows the operator to enter general order information. After this data has been entered and stored, the "Order Item Form" is processed to enter each ordered item. Another form is provided to aid the operator in reponding to customer inquiries. The "Order Browse Form" allows the operator to search the database using one or more search fields to retrieve a list of customer orders. Once located, the operator may use the previous two forms to review and modify the order information.

Figure 3.4 lists the database tables and columns used in this application. The 'order' and 'orderitem' tables are of primary interest, and are used to store the information associated with each order. The other tables are used for reference purposes only. The 'part' table provides a list of valid parts. Later in this section a technique will be presented for restricting part numbers to those in the list. The table also supplies information about the ordered part. In the same manner, the 'state' and 'customer' tables provide a list of valid entries and related information. The state code for the entered city is supplied by the 'citystate' table.

3.3 **Application Structure**

The next section describes the components of display generation. Section 3.3.1 describes a display block, which is a functional unit of a display. Section 3.3.2 describes field specification; a field is a component of a block. Section 3.3.3 describes screen format capabilities; and Section 3.3.4 discusses terminal support details.

3.3.1 **Block Specification**

An IAF **application** consists of one or more application **blocks.** Each block consists of a collection of fields which map to a single database table. All tables referenced in an application must reside in the same ORACLE database. An operator may process in only one block at a time.

Within each block, the collection of fields may be displayed multiple times. This allows multiple records (or rows) from the defined table to be displayed simultaneously. However, only one record may be processed at a time.

SAMPLE 'ORDER ENTRY' APPLICATION

D A T A B A S E - T A B L E - D E F I N I T I O N S

Database: 'order'

Table Name	Column Name	Data Type	Length	Image	Nulls
order	orderno	number		unique	no
	date	number			no
	custno	number		non-uniq	no
	purchaser	char	15		no
	shipname	char	15		no
	shipaddr	char	20		no
	shipcity	char	15		no
	shipstate	char	2		no
	shipzip	number			no
orderitem	orderno	number		non-uniq	no
	lineno	number			no
	partno	number			no
	partprice	number			no
	orderqty	number			no
	instruct	char	20		yes
part	partno	number		unique	no
	desc	char	15		no
	price	number			no
	unitqty	number			no
citystate	city	char	15	non-uniq	no
	state	char	2		no
state	state	char	2	unique	no
customer	custno	number		unique	no
	custname	char	15		no
	custaddr	char	20		no
	custcity	char	15		no
	coststate	char	2		no
	custzip	number			no

Figure 3.4

When thinking about an interactive CRT application, it is common to associate a screen page with the logical unit of work. An operator thinks in terms of processing a display a page at a time. Although IAF allows an application to occupy multiple pages, the unit of work is a block, or more precisely a current record within the current block. The distinction is important in understanding the flow of IAF applications. A loose association exists between a block and the application's screen pages. A block may span multiple pages, or multiple blocks could occupy the same page. The application designer has complete freedom to position a field within a block. The display location of a field will be discussed later in this section.

In the sample order entry application three blocks have been defined. Figure 3.5A defines the relationship between each screen page and the associated block and database tables. The 'Order' block was defined with one record display area, therefore only one record occurrence of the order information may be displayed. In contrast, the 'Orderitem' block contains five record display areas permitting up to five orderitem records to be simultaneously displayed.

Figure 3.5B describes the blocks associated with the "employee" application.

3.3.1.1 **Block Control**

IAF provides the capability for an operator, within a multi-block application, to terminate the processing of one block and begin another. When an application is executed the operator is positioned to the first block which was defined in IAG. The terminal operator can move from block to block using the 'Next Block' and 'Previous Block' function keys. The order of block processing is determined by the order of definition. Requesting the 'Next Block' function while processing the last defined block will jump forward to the first block. Conversely, a 'Previous Block' request in the first block will cause the last block to be processed.

Block Name	Form Name	Table Name	Record Areas
order	Order Form	order	1
orderitem	Order Item Form	orderitem	5
browse	Order Browse Form	order	8

Order Entry Application - Block Specifications

Figure 3.5A

Block Name	Form Name	Table Name	Record Areas
emp	Personnel Record	emp	1
projects	Projects Assignments	pe	3

Employe Application - Block Specifications

Figure 3.5B

The blocks of an application are logically chained in the order in which they were defined. The chain for the order entry application is illustrated in figure 3.6. The 'Order' block is the first to be processed. Normally, the operator would depress the 'Next Block' function key to execute the 'Orderitems' block. When the order is completed, 'Previous Block' brings the operator back to the 'Order' block in preparation to enter another order.

Processing a customer inquiry requires the use of the 'Browse' block. The block is obtained by advancing forward using the 'Next Block' or backwards using the 'Previous Block' keys. From this block the operator can query existing orders. When the desired order has been located, the operator can view the expanded order information by advancing to the 'Order' block.

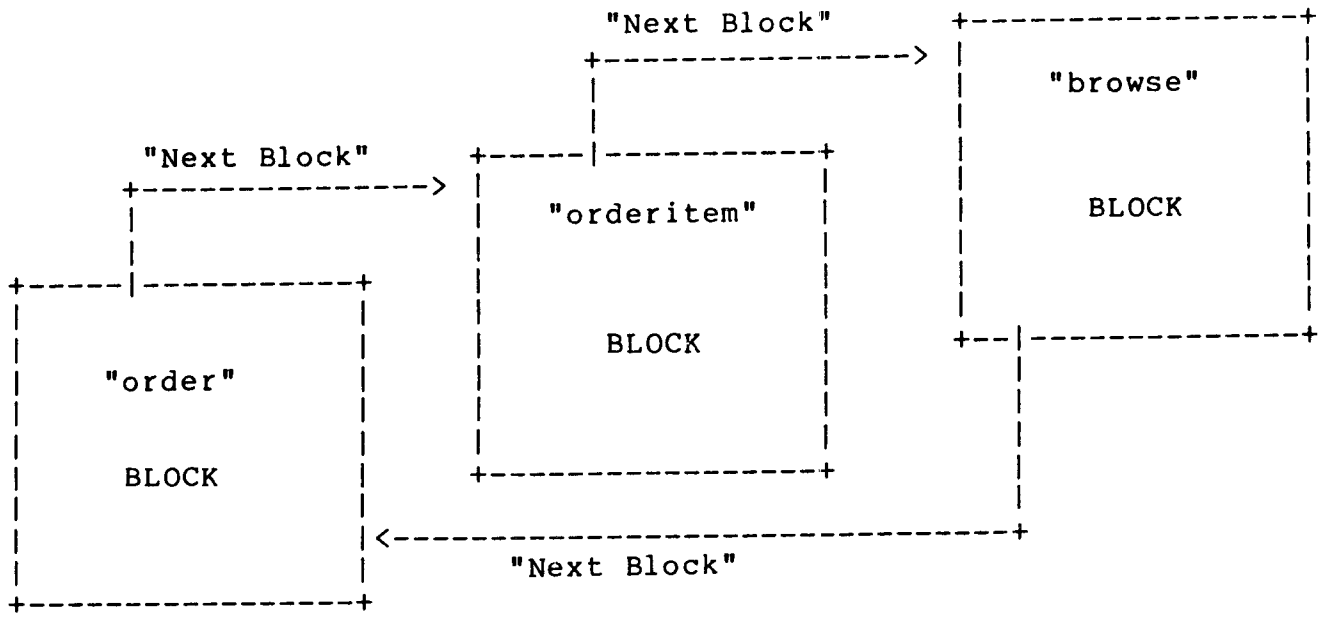
When an operator leaves a block the current data is preserved and will be redisplayed when that block is re-entered.

Advancing blocks in this manner may become tedious, especially if they are processed in a random order. Sequential block processing poses less of a problem. Trial and error will determine an optimal size for an application. Breaking a large application into multiple applications, with fewer blocks, may simplify the operator interface.

3.3.1.2 **Block Processing Modes**

An application block can be processed in either 'Insert' or 'Update' mode. The processing mode states the operator's intentions. In 'Insert' mode, records may only be inserted. In update mode, records may be retrieved; retrieved records can be updated or deleted. Although IAF will permit any user an attempt to retrieve or modify data in a table, ORACLE will reject the request if the appropriate database access has not been granted.

When an application is initially executed it is automatically placed in 'Update' mode. The 'Change Mode' function switches the mode of operation. A mode is retained until explicitly changed by the operator. Advancing from one block to another does not effect the processing mode.



"Order Entry" Application - Block Processing Order

Figure 3.6

3.3.1.2.1 **Insert Mode**

To insert a record the application must be in 'Insert Mode'. 'Clear Block' will clear the block and initialize any default values. The operator can then enter data into any field which was defined as enterable. Fields are entered one at a time until all the fields within a record area are complete. Movement from one field to another is accomplished using the 'Next Field' and 'Previous Field' functions. The 'Insert' function signals the completion of data entry and triggers the insertion of a new row into the table.

3.3.1.2.2 **Update Mode**

Within 'Update' mode, an operator may construct an inquiry to retrieve one or more stored records. To retrieve a record, the operator may supply field values for the associated columns in the target record. The 'Inquiry' function initiates the retrieval of data. IAP will dynamically construct a SQL query with a SELECT clause listing all the database fields in the block, and a WHERE clause specifying an '=' condition for each entered value. The operator may explicitly supply additional retrieval conditions which are added to the generated WHERE clause.

If no field values were entered and no additional conditions are supplied, a WHERE clause will not be generated, and every row in the table will be returned.

For example, using the 'Order' block the operator could request all the orders for customer '18945' . The entered data would be :

```
CUSTNO: 18945
```

Since no additional WHERE clause conditions were specified, the following SQL statement will be created and executed by IAP:

```
SELECT *
FROM   ORDER
WHERE  CUSTNO=18945
```

After depressing the 'Inquiry' key, the operator will be prompted to supply additional WHERE clause text. Up to one line of input is permitted. The entered text is appended to the IAP generated WHERE clause. The 'AND' logical operator is used to connect the generated predicates with those supplied by the operator.

This supplemental text offers the operator greater flexibility in selecting the records to be retrieved. If no fields are entered within the block, only the supplemental text will comprise the WHERE clause. An ORDER BY clause could be specified as part of this text, allowing the resultant rows to be returned in a sorted order.

Following the retrieval request the records satisfying the query will be displayed. The maximum number of records displayed is equal to the number of record areas defined for the block. If more records are retrieved than can be displayed, the 'NEXT SET' function will cause the next page of records to be displayed. The record areas may be scrolled by entering the 'Next Record' function when the cursor is positioned on the bottom line. In this case records are scrolled up , with the next record displayed in the bottom area. The top record will disappear from view. There is no facility to page backward; the retrieval operation must be re-executed to review a passed record.

3.3.1.2.3 **Updating a Retrieved Record**

Retrieved records may be updated one at a time. Only the record in the 'current' record display area can be updated. The 'current' record area is the area in which the cursor is positioned.

Only fields for which update has been allowed may be modified. When all the desired fields have been modified, the 'Update' function is used to write the updated record to the database.

The operator may move the cursor to another record display area using the 'Next Record' and 'Previous Record' function keys.

3.3.1.2.4 **Deleting a Retrieved Record**

The 'Delete' function deletes the 'current' record from the database.

3.3.2 **Field Specification**

A block can contain one or more fields. Each field within a block is identified by a unique name. Field names do not have to be unique across blocks.

'Database' fields map directly to a column in the block defined table. Fields which are not mapped are called 'Control' fields. Control fields can be entered by the operator, or initialized with database information. They are useful for carrying data forward from block to block, or as reference aids for the operator.

The application designer can specify whether a control or database field can be entered by the operator. Updating a database field may be permitted only if the field is not part of the primary key (see Section 3.3.2.3 for a discussion of primary keys).

In the 'Order' block, order number is an example of a database field. Its value is mapped to the 'orderno' column in the 'order' table. An operator can input a value into this field, but since the field is part of the primary key updating is not permitted.

The customer name is a control field and is used to verify that the correct customer number has been entered. Entering this field is not permitted.

3.3.2.1 **Field Data Types**

Each field has an assigned data type. The type will establish a set of validation rules which will be applied when the field is either entered or modified.

3.3.2.1.1 **Character Data Types**

Two character data types are supported. The column associated with this field type must be specified as 'CHAR' on CREATE TABLE statement.

Alpha - Only alphabetic characters A thru Z and space are permitted

Char - Any printable character is acceptable

3.3.2.1.2 **Numeric Data Types**

There are three explicit forms of the numeric data type.

Number - May contain the digits '0' thru '9', 'E', '+', '-', and '.'. Numbers may be entered or displayed in scientific notation (ie. 3,287 is 3.287E3).

Int - Only integer numbers are accepted. Only digits '0' thru '9', '+' and '-' are accepted.

Money - A special number format for field which contain money values. Excludes scientific notation, and is limited to two digits to the right of the decimal point.

3.3.2.1.3 **Date Data Types**

There are three 'date' data types. Each has a specific entry/display and storage format.

Date - The entry/display format is mm/dd/yy; a length of 8 is required. The values for mm, dd, and yy are validated. mm must be in the range 1 to 12, dd must be in range implied by the month including leap years. The value is converted and stored in a Julian Day number format(Discussed below).

Edate - Similar to 'date' type except entry/display format is in European date format (dd/mm/yy). The same validation is used and the value is stored in Julian Day number format.

YYMMDD - The entry/display format is mm/dd/yy and has the same validation as 'date' type fields. The value is stored as a 6 digit integer in the format YYMMDD.

IAF converts and stores 'date' and 'edate' type fields from a calendar date into its corresponding Julian Day Number. On output, fields of this type are assumed to be in Julian Day format, and are converted back to calendar date. Simple mathematical manipulation (such as addition or subtraction of days) may be performed on date data in this format. There are limitations with 'date' and 'edate' type fields. No format conversion is performed on 'date' and 'edate' type fields when referenced in Host Language program calls. Therefore, when these fields are referenced in a host program, they will appear as a julian date in numeric form. If calendar date format is required in such a program, logic would have to be included to perform the conversion from julian number format to calendar date format. The algorithym for converting to and from julian day is included with the documentation.

3.3.2.1.4 **Time Data Type**

Time - Defines a field type of 'time' with an entry/display format of HH:MM:SS. The field must have a length of 8. Entered values will be verified for a valid time. The stored format is the number of seconds from midnight.

The 'time' format is only recognized by IAF; an operator using UFI or a Host Language program would have to perform their own conversion.

3.3.2.3 **Default Values**

A default value may be assigned for any data or control field. The default values are used only when a block is processed in 'Insert' mode. When the block is 'Cleared' the default values will be displayed. The default value may be a literal, system default, or the value of another field. Literal values must conform to the valid formats for the field's data type. The system defaults are the current date for date type fields, and the current time for 'time' fields.

If a field's default value is to be copied from another

field, both fields must have the same data type. The copied field should be from a different block. .np Assigning a default value and designating a field as "non-enterable" restricts the value to the default. For example, in the "employee" application, a version of the application could be generated which would restrict user's to entering employees only into a specific department.

If a default is not specified and no data is entered, the field is assigned the "NULL" value.

3.3.2.3 **Field Initialization**

3.3.2.3.1 **Copying the Primary Key**

The term primary key is used in ORACLE to refer to the first column defined in a table. However, in IAF, the term is used in its strictest sense to mean that collection of columns which uniquely identifies a row in a table. Some tables may not have a primary key in this sense if there are rows in the table which are duplicated. Currently in IAF, only tables which have a primary key can be updated. This is because IAF qualifies its updates and deletes by the primary key. Updating a row then, will have the effect of updating or deleting all rows where the set of columns comprising the primary key has the same value as the row being updated.

IAF allows the user to specify those fields in a block which comprise the primary key. Any field which is part of the primary key may have its value copied from a field in a different block. The value will be copied in both 'Insert' and 'Update' modes. In a relational system, the relationship between two tables is established by a common domain. For example, employees and departments by department number, and orders and line items by a common order number. This facility allows these records, which are entered in separate blocks, to be related by automatically copying the common data from one block to another.

In the 'Orderitem' block the order number is not displayed. However, order number is necessary to associate a set of line items with a particular order. In the 'Order' block definition, 'orderno' is defined as a non-display field whose value is copied from the 'orderno' field in the 'Order' block. Thereby the common order number is automatically carried forward into each 'Orderitem' record.

3.3.2.3.2 **Selecting Into a Field**

A SQL SELECT statement may be defined with any field. This query is executed each time a field is entered or modified, and also after a query. This SELECT should not be confused with the SELECTs which are implicitly created by IAP to retrieve records into a block. The SELECT may serve three purposes:

- Test for the existence of a field's value in a table of acceptable values.
- Select information to be displayed which will aid the operator in verifying the correctness of the input.
- Select information to initialize the value of one or more fields.

The last two purposes will be discussed in this section. Refer to section 3.3.2.5 for a discussion on "Existence Checking".

Any field can be assigned a value which is returned by a SQL query. The 'INTO' clause identifies the block fields where the data is to be returned. Each field named is postionally associated with a corresponding column name in the 'SELECT' clause. The 'FROM' clause identifies the queried tables.

Only the first row returned from the query is processed. All subsequent rows are ignored. The fields identified in the 'INTO' statement are assigned the corresponding values returned in the first row.

The 'WHERE' clause should normally contain a single predicate of the form:

WHERE <column name> = &<field name>

where <column name> is the database column associated with the specified <field name>. The '&' designates the <field name> as a substitution variable whose value will be substituted into the 'WHERE' prior to execution. The <field name> is commonly the field in which the SQL statement is being defined.

For example, 'partno' in the 'orderitem' block has the following query defined:

```
SELECT  desc, price, unitqty
INTO    orderitem.partdesc, orderitem.partprice,
        orderitem.unitqty
FROM    part
WHERE   partno = &orderitem.partno
```

Field names may be qualified by the block name they reside in and this is generally considered a good idea, especially if the field name appears in more than one block. When a part number is entered the query is executed, and the fields 'partdesc', 'partprice', and 'unitqty' will be assigned the values returned. These values will be immediately displayed and overwrite any previous values. For each new value of 'partno', the SQL query is re-executed, and the fields in the 'INTO' clause re-initialized.

In the above example, 'partdesc' is the description of the entered 'partno'. This information aids the operator in determining whether the correct part number was entered. 'partprice' and 'unitqty' initialize the part and quantity fields and which will be stored with the related line item.

NOTE: When SELECTS are executed, not all fields appearing in the INTO list may be set. In INSERT mode, all items will be set. However in QUERY/UPDATE mode, only those fields which are not columns in the table associated with the block will be set. The reason for this being that when records are retrieved, the stored value should not be overwritten by a field SELECT which executes after a row is returned.

3.3.2.4 **Record Uniqueness**

IAF has an opton that allows a user to verify that a record is unique prior to inserting a record. Unique in this case means that no two records will have the same primary key (as defined above). This is because ORACLE can prevent duplicates only in the case where one column defines the primary key. The uniqueness check is performed when all the fields in the primary key have been entered. If the record is found to exist already, the entire record as it is stored in the database will be retrieved and the message "Record already exists" will be displayed, otherwise the message "End of Query" will be displayed. This option only occurs if in INSERT mode.

IAF will not allow fields within the primary key to be updated. If these fields need to be modified, the record must be deleted and re-inserted with the new values.

3.3.2.5 **Field Editing**

The IAP utility will perform edit checks on fields as they are being entered for either update or insertion. If a field fails an edit check the operator is immediately notified and requested to enter correct information.

The following is a list of the supported edit checks:

Data Type Check
The field will be verified based on the defined data type (Alpha, Char, Number, Date, etc.).

Length Check
The defined length of the field determines the maximum number of characters allowed. Less than the maximum is allowed unless the field is defined as fixed length, requiring each position to be entered (ie. zip code, telephone number, etc.).

Range Check
A value range may be specified for a field of any data type. A high value, low value or both may be specified.

Required Field Check
A value can be required for a field. The value can be entered by the operator or assigned as a result of a copy, SELECT or default value.

Note that if a value is not required for a field, but the associated database column is defined as NO-NULL, the update or insert will be rejected by ORACLE.

Existence Check
As discussed in section 3.3.2.3.2 "Selecting Into a Field", a SQL statement may be defined which checks the existence of the entered value in a database table. The single predicate in the 'WHERE' clause compares the column with the entered field value. If no rows are returned , the field value does not exist in the table, and the entry is rejected.

For example, in the 'Order' block, the entered value for state code is compared with a list of valid state codes. The following SQL statement was defined within the 'shipstate' field:

```
SELECT state
FROM state
WHERE state = &shipstate
```

Note that no INTO clause was specified; therefore, no fields were initialized as a result of this query.

3.3.2.6 **Operator Aids**

A help message can be defined to aid the operator in entering correct information. This message is displayed at the bottom of the screen page when the 'Help' function is requested.

A 'Display Attributes' function allows the operator to request the information about the current field. The message is displayed in the system message area at the bottom of the screen. The attributes include the field's data type, whether is is updatable, and whether it is mandatory.

In addition to this requested information, the current screen page, block processing mode, character insert or replace mode, and number of records retrieved are continuously displayed at the bottom of the screen.

Error and status messages are also displayed when necessary in the system message area.

3.3.2.7 **Field Display**

A field can be defined as either display or non-display. For display fields, the page, line, and column location on the screen must be specified. Each displayed field can have a prompt message which is displayed above or immediately to the left of the field's display location.

More than one record area can be defined for a block. The application designer can define the number of lines in each record display area, the number of display areas, and the base or starting line of the first record area.

For example, the 'Browse' block can contain 8 order records, with each record displayed on a single line. If a query results in more than eight rows, the record display area will scroll up, with new records appearing at the bottom, and earlier records disappearing at the top. The scrolling operation is controlled with the 'Next Record' or 'Next Page' functions. Once a record has scrolled off the screen, the query must be reissued to be viewed again.

Field prompts will always be displayed within the first record area. If multiple areas are defined, the prompts can optionally be repeated in every area.

The 'Orderitem' block allows input and display of multiple order items. The 'SPECIAL INSTRUCTIONS:' prompt is repeated in each record as an operator aid. As records are inserted, the 'Next Record' key will advance the cursor to the next record area down the screen. When the screen is full, the 'Next Record' key will scroll the records up, allowing a new record to be entered in the bottom area.

3.3.3 **Screen Formatting**

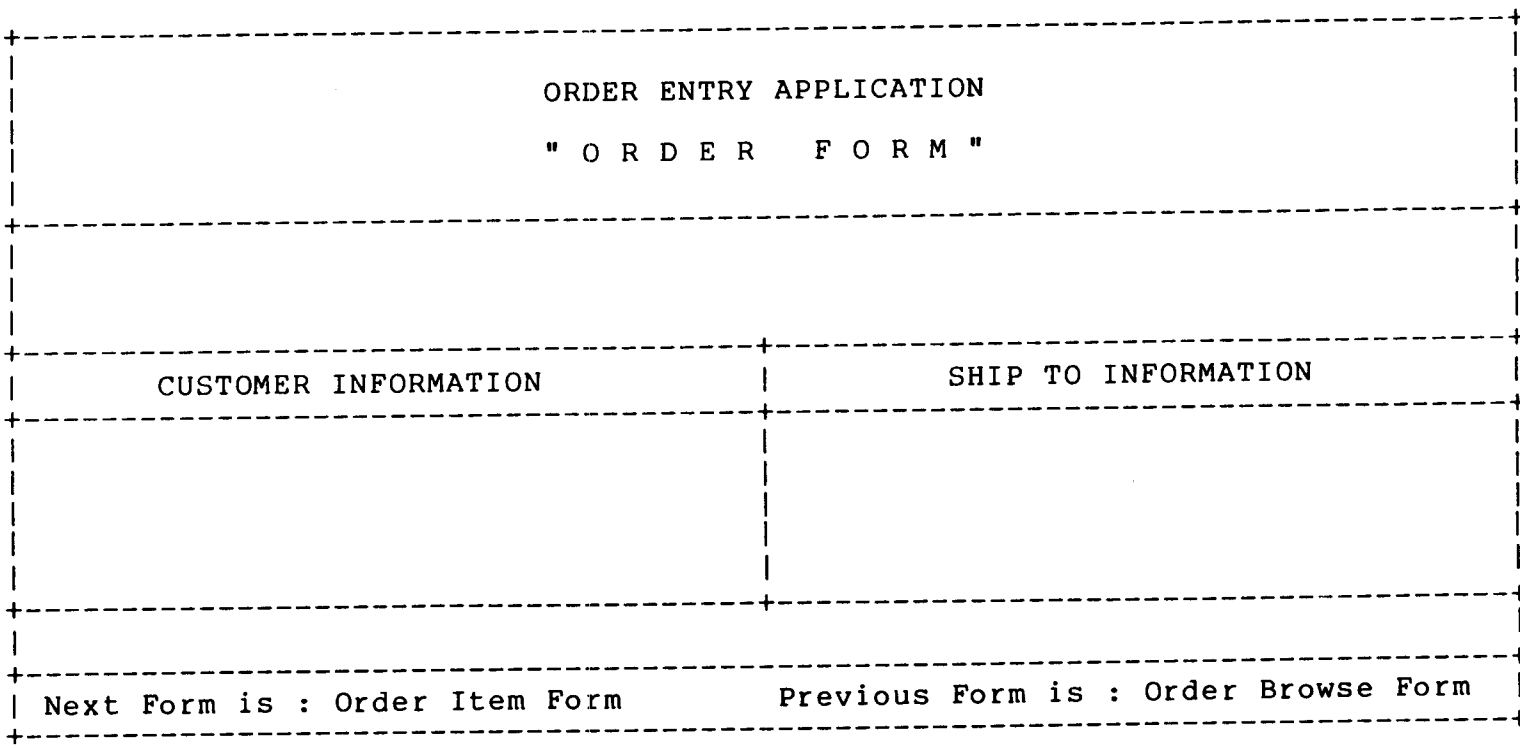
Each application block can occupy one or more screen pages; more than one block can reside on the same page. Page numbers are assigned by the designer. The numbers define specific pages but do not imply a processing order.

A screen overlay of descriptive text may be defined for any page. The text is specified separately from the block and field definitions. IAP will merge the prompts and field values with the screen image format, forming the composite page.

Figure 3.7 shows the line drawing and explanatory text used to create the screen display for the 'Order' block.

3.4 **Terminal Support**

An IAF application can be executed from any CRT terminal device which has basic cursor control, character by character transmission, and has been defined using the procedure outlined in the "IAF Terminal Operator's Guide". The user has complete flexibility in function key and control sequence definition. A terminal identifier is associated with each definition. This identifier is specified when IAP is executed.



%page

'Screen Formatting Text'

Figure 3.7

Pages 4-54 through 4-55 have been omitted.

Pages 4-54 through 4-55 have been omitted.

APPLICATION-DEFINITION

INTERACTIVE-APPLICATION GENERATOR

4.1 Introduction

The Interactive Application Generator (IAG) is used to define an IAF application. Interactively executed from the user's terminal, IAG will enter into a dialogue whereby questions will be asked about the application. These questions fall into the following categories:

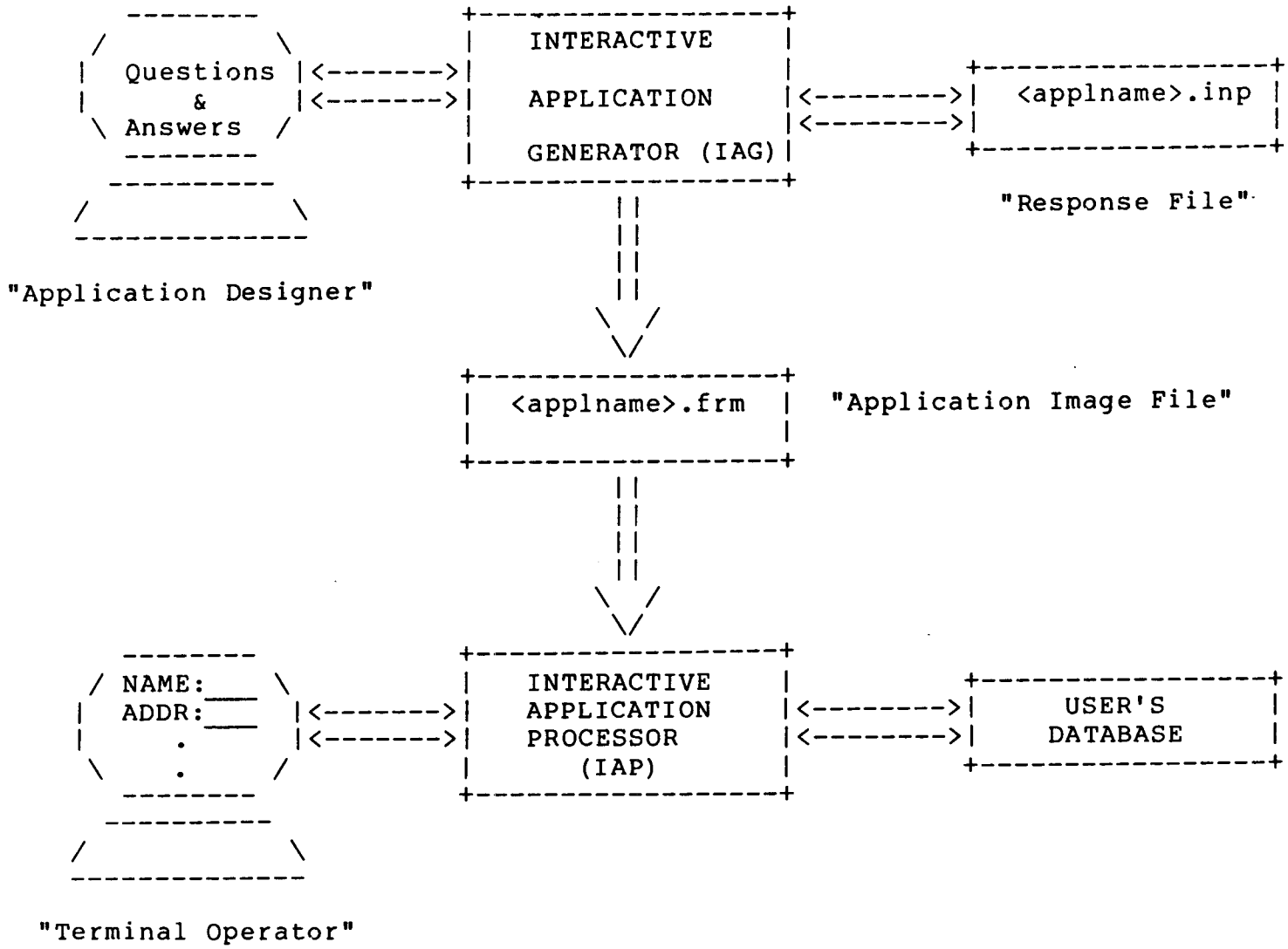
+ General questions about application execution
+ Block specification questions
+ Field specification questions
+ Screen layout - Descriptive text specification

As each response is entered, IAG will save the question text and associated response in a user file. IAG can later be directed to use the response file as an alternate input source to regenerate an application.

For simple application changes the response file may be edited using a standard text editor. For more extensive changes the response file can be combined with additional terminal entered input. Commands are provided which direct IAG to alternate between these input sources.

When an application has been completely and correctly defined, IAG will compile the responses into an IAP executable module. To execute the application, the user must invoke the IAP Utility as described in the IAF Terminal Operators Guide.

This section will present the application definition process. Figure 4.1 describes the components of the application development process. A conceptual overview of IAF, and a description of its features was provided in sections 2 and 3. That material should be referenced to provide the overall structure and design of an application, and explain the implications of the IAG questions and responses.



Application Development Process

Figure 4.1

4.2 **Executing the IAG Utility**

IAG is invoked from the user's terminal by entering the following command:

IAG <applname> [-<options>]

Where:

<applname> Name of the application being defined. Any character string which is acceptable as an operating system file name is allowed.

When IAG is invoked, it will automatically search for an existing response file with the name '<applname>.inp'. If present, this file will be used as the input source. Additionally, for each execution a new version of the response file will be created. When the host operating system is RSX11M, VMS or IAS, the new file will be assigned the next highest version number.

The <options> parameter permits the user to control the creation and use of the response file. This parameter is optional.

T - Direct IAG to use the response file for input; suppress terminal output of the question and answer text. Only error messages will be displayed.

S - Suppress question text when creating the new response file. Only a list of the responses will be saved.

O - Suppress creation of a new version of the response file.

4.3 **Defining the Application**

An application is defined by responding to a series of questions. These questions fall within the general category identified by the question number:

G-x	-General Questions about the application
B-x	-Block Questions
F-x	-Field Questions - field specification information
D-x	-Field Display Questions - field display information
E-x	-Field Edit Questions - field editing information

Figure 4.2 provides a list of these questions with their associated numbers. The question numbers are used to aid in the discussion, but will not be displayed during IAG execution.

In addition free-format descriptive text can be specified for each screen within an application.

Figure 4.3 provides an outline of the order in which the categories of questions are asked. First, the general questions about the application are presented. Then the first block is specified. Within that block, each field is defined. When all the fields in one block have been defined, that block specification is complete. The block questions will be reasked for each successive block in the application. Lastly, the user can define additional descriptive text for any screen page.

Certain questions are conditionally asked depending on previous responses. The column labeled 'A/C' in Figure 4.2 indicates whether the question is 'A'-Always or 'C'-Conditionally asked. For conditional questions, the last two columns identify the responses to other questions which will cause the indicated question to be asked. If more than one condition is listed, each condition must be met, unless an 'or' is specified.

I N T E R A C T I V E - A P P L I C A T I O N - G E N E R A T O R

Question List

?-no.	Question Text	A/C	Response To	?no.
G-1	Database :	A		
G-2	Sequel Workspace size ?	A		
B-1	Block name:	A		
B-2	Table name:	A		
B-3	Check for uniqueness before inserting Y/N	A		
B-4	Buffer how many records ?	A		
B-5	Base CRT line ?	C	> 1	B-4
B-6	How many physical lines per record ?	C	> 1	B-4
F-1	Field name :	A		
F-2	Type of field :	A		
F-3	Length of Field :	A		
F-4	Is this field in the base table Y/N:	A		
F-5	Is this field part of the primary key Y/N :	C	Y	F-4
F-6	Field to copy primary key from :	C	Y	F-5
F-7	Default value :	A		
F-8	Allow field to be entered Y/N :	C	non-blank	D-1
F-9	Allow field to be updated Y/N :	C	N	F-5
			Y	F-8
F-10	SQL >	A		
F-11	Message if value not found :	C	non-blank	F-10
D-1	Page :	A		
D-2	Line :	C	non-blank	D-1
D-3	Column :	C	non-blank	D-1
D-4	Prompt :	C	non-blank	D-1
D-5	Display prompt above field Y/N :	C	non-blank	D-4
D-6	Display prompt once for block Y/N	C	non-blank	D-4
			> 1	B-4

Figure 4.2

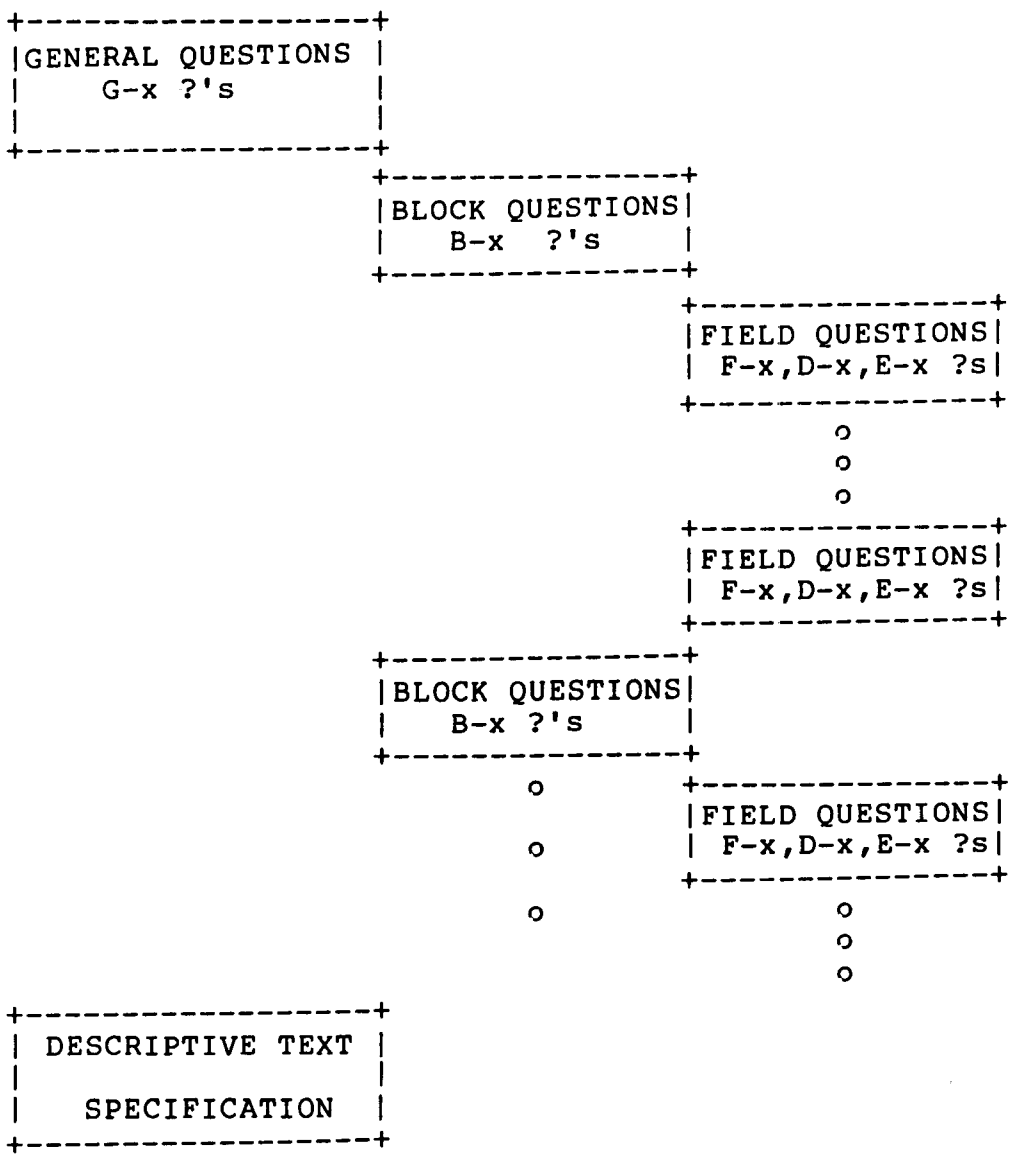
?-no.	Question Text	A/C	Response To	?no.
E-1	Is field mandatory Y/N :	C	non-blank	D-1
			Y	F-8
E-2	Is field fixed length Y/N :	C	non-blank	D-1
			Y	F-8
E-3	Auto jump to next field Y/N :	C	non-blank	D-1
			Y	F-8
E-4	Convert Field to upper case Y/N :	C	non-blank	D-1
			Y	F-8
E-5	Help Message :	C	non-blank	D-1
			Y	F-8
E-6	Lowest value :	C	non-blank	D-1
			Y	F-8
E-7	Highest value :	C	non-blank	D-1
			Y	F-8
E-8	Must value exist Y/N :	C	non-blank	F-10

A: Question is 'ALWAYS' asked

C: Question is 'CONDITIONALLY' asked

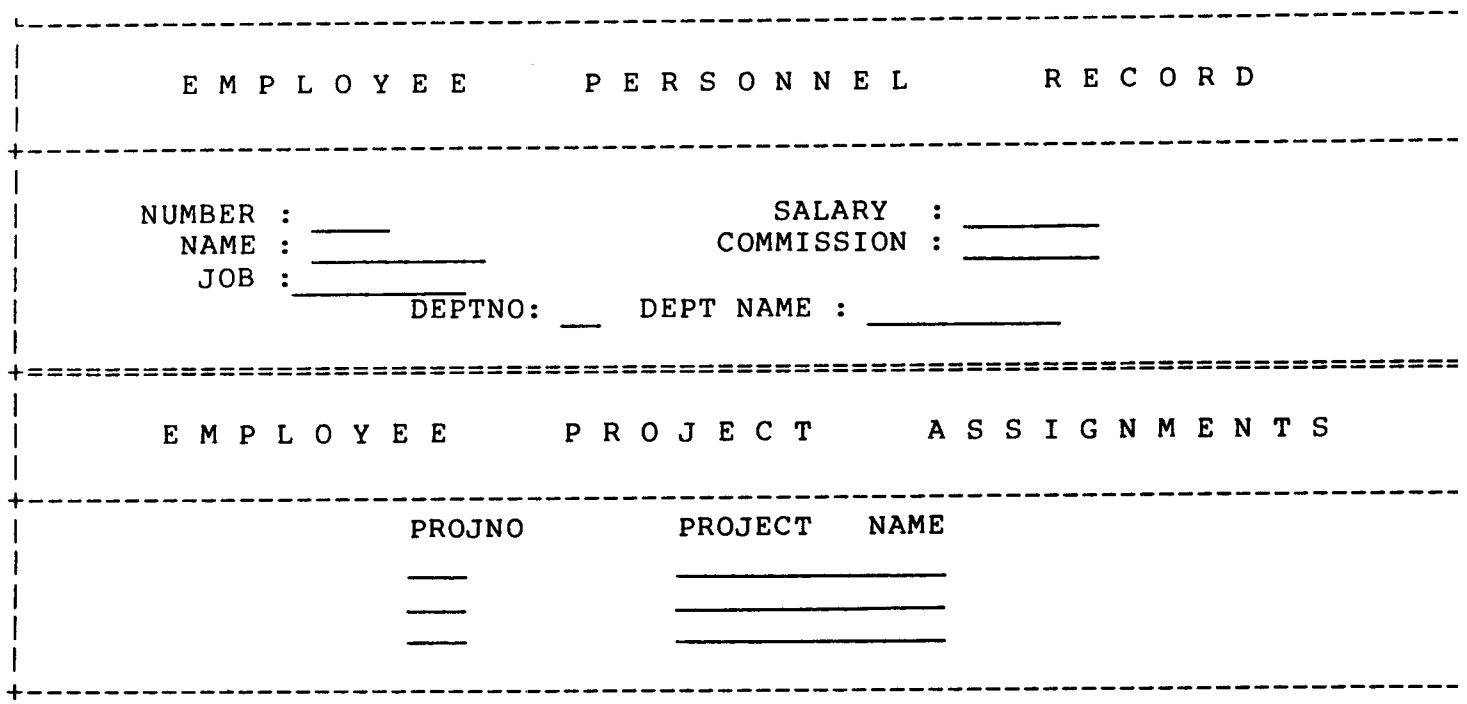
Response to: For conditionally asked questions, question is triggered by the indicated response to the question in the '?-no.' column.

Figure 4.2 (Continued)



Application Definition Outline

Figure 4.3



Sample "Employee" Screen Layout

Figure 4.4

Each question requires either a 'YES/NO' or literal value response. A "Y/N" in the question text indicates that a 'YES' or 'NO' is required. All other questions require a character or numeric value. Character values can contain any printable character. If an incorrect response is entered, IAG will display an error message and reask the question. A correct response to each question is required before proceeding to the next question. Each correct response will be added to the <applname>.inp response file.

Questions will continue until the entire application is defined. An application is considered complete when IAG encounters a '%end' response to a descriptive text prompt. The user may prematurely terminate the session by entering a ^Z (Control Z). This will cause a normal exit from IAG with the <applname>.inp response file containing all the valid responses entered prior to termination. No IAP module will be created for incomplete applications.

When an application is executed by IAP, blocks are processed in the order in which they were defined. This order is followed regardless of the order of the actual display pages. For example, if the first block defined is on page 1, the second on page 3, and the third on page 2, the order of page display is 1,3,2.

For fields within a block, the cursor will advance from field to field in the order of definition. To simplify the operator interface, fields should be defined in their order of display (left to right, top to bottom, etc.)

The fields within a block may be displayed on one or more pages. Advancing to a field on another page will cause the new page to be displayed. Unnecessary page switches may be distracting to the operator, and will cause additional delays in transmitting the screen images.

Many of the questions relate to ORACLE database, table and column names and definitions. During definition and compilation, IAG does not access the ORACLE dictionary to verify their existence or validate their data characteristics. Any errors of this type will be detected by IAP during application execution.

4.3.1 General Questions

The questions in the General (G-x) Category are asked only once for an application. Only one application can be defined within a single execution of IAG.

G-1: Database :
Specify the ORACLE database name to be accessed by this application. Only one database can be accessed per application.

G-2: Sequel Workspace size :
Specify the size of the SQL workarea in 1K increments, which is required for this application. If no value is specified, the ORACLE default value will be used.

4.3.2 Block Questions

The block related questions are asked once for each block definition. An application can contain one or more blocks. Following each block definition, the fields within that block must be defined.

B-1: Block Name :
Specify the name of the block being defined. A blank or null response indicates that no more blocks are to be defined, and IAG will skip to the 'Screen Layout Questions' described in section 4.3.4.

B-2: Table Name :
The database table name referenced by this block. Only one table may be referenced within a block. If no value is specified, the table name will default to the block name.

B-3: Check for uniqueness before inserting Y/N :
Specify whether you want IAP to verify, prior to insertion, that a row does not exist in the table with the same primary key value. Refer to section 3.3.2.3 for a discussion on 'Primary Key Specification'.

B-4: Buffer how many records ?
Specify the maximum number of database records which can be displayed within this block. An integer value between 1 and 22 is required. All the records must be contained within one screen page. If only one record occurrence is to be displayed, enter a value of 1.

B-5: Base CRT line ?
Specify the screen line number where the first line of data for the first record of a multi-record block is to be displayed. An integer between 1 and 22 is required.

B-6: How many physical lines per record ?
Specify the number of display lines for a single occurrence of a record. An integer value between 1 and 7 is required.

Note: The following rules apply to the layout of a multi-record block:

- Line 1 through 22 is available to the user. Lines 23 and 24 are reserved for system information.

- (Base line + (Number of Records * Lines per Record)) <= 22

- If the prompts are displayed above the field display area base line value of 1 is invalid.

For a detailed discussion of Multi-Record Block Layout refer to sections 3.3.1 and 3.3.2.4.

4.3.3 **Field Questions**

For each application block, one or more fields may be defined. Field questions fall into three categories: specification, display and edit. In the following sections the questions will be grouped by category. The order in which they are discussed may vary from the order in which they are actually asked. Additionally, questions from the different categories may be mixed.

4.3.3.1 **Field Specification Questions**

F-1: Field Name:
Specify the name of the field being defined. Any charcter string will be accepted. Field names must be unique within a block, but may be repeated in different blocks. For database fields, it must be a column name from the block's associated table. A blank or null response indicates the end of field specification for the current block.

F-2: Type of Field:
Specify the data type for this field. Valid data types are:

Alpha - Only alphabetic characters A-Z are permitted. Upper and lower case is supported.

Char - Any printable character is permitted

Number - A number which can contain the digits 0-9, '.','+','-'. Numbers may be specified in scientific notation (2.3E2 = 230).

Int - Only integer numbers are accepted.

Money - A special number format which excludes scientific notation (ie. 3.7E2), and is limited to two digits to the right of the decimal point.

Date - Only a valid date of the format mm/dd/yy is permitted. mm must be within the range 1 through 12. dd must be within the range implied by the specified month (including leap years). Dates entered in this format will be stored within the database in the internal Julian Day Number format.

1 Edate - Similar to 'Date' type except the date is entered and displayed using the European format of dd/mm/yy. The same validation is used and the value is stored in Julian Day Number format.

YYMMDD - A date field which is entered and displayed in the format mm/dd/yy, but stored as a numeric value in the format YYMMDD. This date is not converted to Julian Day Number format.

Time - Allows the entry and display of a time value in the format HH:MM:SS. The value is converted to the number of seconds since midnight.

F-3: Length of Field:
Specify the length of the field. An integer value between 1 and 79 is required. For database fields, this value should be consistent with the table column length. The column value will be truncated if its length exceeds the field length.

'Time', 'dae''edate', and 'YYMMDD' type fields must be defined with a length of 8.

F-4: Is this field in the base table Y/N :
Specify whether this field is to be mapped to a column in the table defined for this block. If Y, the field name must be the same as a table column name.

F-5: Is this field part of the primary key Y/N:
Specify whether this field is part of the primary key. Refer to section 3.3.2.3 for a discussion of primary keys. At least one primary key field must be defined in each block.

F-6: Field to copy from :
Specify the name of a field in this or another block, whose value will be copied into this field when the block is initialized. Only fields which are part of the primary key may have their value copied from another field. The format of this response is [<blockname>.]<fieldname>. <blockname> is required if <fieldname> is from another block and is not unique within the application. See Section 3.3.2.3.1. for a discussion on copying fields.

F-7: Default value :
Specify the value to be assigned to this field when the block is cleared or initially entered when in 'INSERT' mode. Three forms of default values may be specified:

- A default value may be either a character or numeric literal. Character literals must be enclosed in single quotes (ie. 'CA'). The value must conform to the field's data type. The correct data format must be specified for date and time type fields (03/31/81 , 12:13:46)

- A default value may be copied from any other field in this of another block. The field name is specified as [<blockname>.]<fieldname>; <blockname> is required only if <fieldname> is not unique within the application.

- Date and time fields may be assigned the value of the current date or time. "$$date$$" is used for field types of 'date', 'edate', and 'YYMMDD'; "$$time$$ is used for 'time' field types.

Primary key fields whose value is copied from another field may not have a default value.

F-8: Allow field to be entered Y/N :
Specify, for the insertion of a new row in the database, whether the operator may enter a value. At least one field in each block must be enterable.

F-9: Allow field to be updated Y/N :
Specify, for the updating of an existing database record, whether the operator may modify the current value of this field. If the field is a part of the primary key, the reponse defaults to 'N' and the question is not asked.

F-10: SQL >
Specify the text of a SQL query to be executed when this field is entered. The use of this query is explained in section 3.3.2.3.2. The SQL statement is free format and must conform to the same rules required by UFI or a Host Language Program. The statement is not validated upon entry. The INTO statement, if provided, will be validated when the application is compiled. The remainder of the statement will not be verified until application execution. A blank or null response indicates the end of statement input. If no SQL statement is to be provided, enter a null response (<cr>) to the first prompt.

F-11: Message if value not found :
Specify the message to be displayed to the operator if no rows are returned from the above SQL statement. If the SQL query is used to test for existence of the entered value, this message informs the operator that the value was not found in the order entry example, if a part number is not found in the 'parts' table the message "Invalid Part Number" is displayed.

4.3.3.2 Field Display Questions

D-1: Page:
Specify the page number where this field will be displayed. An integer between 0 and 31 is required. A null, blank, or 0 response indicates that the field should not be displayed. All the fields within a block do not have to be displayed on the same page.

D-2: Line:
Specify the line number where this field will be displayed. For blocks with a single record display area an integer between 1 and 22 is required. For multi-record blocks this is the relative line number within the record display area. It must be an integer which is less than or equal to the response to question B-6: "How many physical lines per record?".

D-3: Column:
Specify the column number where the left-most character of this field will be displayed. An integer between 1 and 78 is required. This value must take into consideration the prompt message extending to the left or the field extending to the right. If either the prompt or field extends beyond the 80 character screen size an error will be reported. Overlaid fields will not be detected.

D-4: Prompt:
Specify a label or prompt message to be visually associated with this field. Enter any valid character string.

D-5: Display prompt above field Y/N :
Specify whether the prompt identified in question D-4 should be displayed above the field display location. If Y, the prompt is displayed on the line above the field, starting in the same column position as the field. If N, the prompt is displayed on the same line immediately to the left of the field.

D-6: Display prompt once for block Y/N :
For a multi-record block, specify whether the prompt should be repeated in every record area. If N, the field prompt will only be displayed in the first record area.

4.3.3.3 Field Edit Questions

E-1: Is field mandatory Y/N :
For 'Insert' mode, specify whether a value must be provided for this field. If Y, a value can be entered by the operator, assigned as a result of a SQL select, copied from another field, or assigned a default value. For database fields, if the associated column has been defined as 'NONULL' in the 'CREATE TABLE' statement, Y should be specified.

E-2: Is the field fixed length Y/N:
Specify whether the number of characters entered for this field must equal the field length. (ie. Zip Code requires all 5 digits)

E-3: Auto jump to next field Y/N:
Specify whether cursor should automatically skip to the next field after the maximum number of characters have been entered.
If Y, entry of the last character triggers an automatic 'Next Field', which causes the field to be edited and the cursor advanced to the next field.
If N, after the last character has been entered the cursor will remain in the last postion of the field. Once in the last position, an attempt to enter additional characters will be rejected, and the 'alarm' will sound. The 'Next Field' key must be depressed to initiate field editing and advancing of the cursor.

E-4: Convert Field to upper case Y/N :
Specify whether alphabetic characters should be automatically converted to upper case. Characters will be converted as they are keyed and displayed in upper case. This is equivalent to placing the keyboard into shift lock mode.

E-5: Help Message :
Specify a free format help message to aid operator in entering field data. Message will be displayed on the bottom of the screen when the operator requests the 'Help' function. A maximum of 80 characters is permitted.

E-6: Lowest value :
Specify that the field is to be range checked by providing the minimum value in the range. To pass the range check the field value must be greater than or equal to the specified value. Character literals ('Alpha' or 'Char' type) must be enclosed within single quotes (ie. 'CA'). All data types may be ranged checked. The value for 'date', 'edate', 'YYMMDD', and 'time' field types must be specified in the correct format (3/31/81, 12:12:46, etc.).

E-7: Highest value :
Specify the high value in the range check. The field value must be less than or equal to the high value to be accepted. All data types may be ranged checked. The value for 'date', 'edate', 'YYMMDD', and 'time' field types must be specified in the correct format (3/31/81, 12:12:46, etc.)

E-8: Must value exist Y/N :
Following a SQL select, a Y indicates that at least one row must be returned as a result of the query. Using the SQL query facility in this manner permits the verification that the entered value is contained in a table of all allowable values. For example, to verify that an entered part number is valid, a query of the parts table with the WHERE clause:

partno = <entered part number>

must return a row for that part. If Y was specified, and a row was not returned, the entered value will be rejected.

4.3.4 Screen Layout Questions

Each page of an application can be enhanced with additional descriptive text. Following the prompt message:

Enter text for form:

A ':' will be displayed in the first column of the next line. Following the ':', up to 79 columns of descriptive text may be entered representing screen positions 1 through 79. The first line of text corresponds with page 1, line 1. As each line is entered the ':' prompt is redisplayed.

A maximum of 22 lines may be entered for each page. After each 22 lines the page counter is automatically incremented and the line counter reset to 1.

The line counter can be advanced to a specific line by entering the command:

%line

On the next line following this command the new line counter value is entered.

The

%page

will advance the page counter by 1 , and reset the line counter to 1.

The

%end

command will terminate the entry of descriptive text and signal IAG to compile the application. If descriptive text is not included this command can be entered immediately following the "Enter text for form:" message.

4.4 Using the Response File

Each time IAG is executed a new version of the <applname>.inp response file will be created. The format is one line listing the text of the question followed by a line listing the entered reponse. All question text lines are preceded by a ';'. Any line beginning with a ';' in column 1 is treated as a comment and will be ignored. Additional comments lines may be inserted to aid in documentation. Lines without a ';' are treated as responses and are processed by IAG as if they were entered from a terminal. For this reason the order and number of responses must be exactly as originally entered. Figure 4.5 is a listing of the response file associated with the 'employee' application.

An application can be changed and regenerated using the response file, eliminating the need to manually re-enter the original responses. Simple changes, which do not alter the order or number of responses, can be made directly within the response file using a standard text editor.

If an error is detected while processing this modified response file, the error message will be displayed on the user's terminal. The terminal then becomes the source of the input. The failing question will be asked again, and the user will be able to enter a new response. When a valid response is entered, questioning will continue from the terminal. A '%sw' reply will instruct IAG to resume the reading of responses from the response file. In this manner the source of responses can be alternated between the response file and user's terminal.

For example, assume that the field type for 'empno' was to be changed from 'int' to 'number'. However, when the response file was edited the word 'number' was misspelled. While processing this field IAG detected this error and displayed on the designer's terminal the "Invalid data type" error message. The question "Type of field : " is displayed on the terminal and IAG pauses pending a response. The designer now correctly enters the word 'number', and the next question ("Length of field :") is issued to the terminal. The designer enters a reply of '%sw' to resume the use of the response file.

```
;Database :
personnel
;Sequel workspace size ?
3
;Block name :
emp
;Table name :
emp
;Check for uniqueness before inserting Y/N :
y
;Buffer how many records ?
1
;Field name :
empno
;Type of field :
int
;Length of field :
4
;Is this field in the base table Y/N :
y
;Is this field part of the primary key Y/N :
y
;Field to copy primary key from :

;Default value :

;Page :
1
;Line :
7
;Column :
17
;Prompt :
NUMBER :
;Display prompt above field Y/N :
n
;Allow field to be entered Y/N :
y
;SQL>

;Is field fixed length Y/N :
y
;Auto jump to next field Y/N :
n
;Convert field to upper case Y/N :
n
;Help message :
Enter 4 digit employee number -
;Lowest value :
1000
;Highest value :
8000
```

"Employee" Application Response File

Figure 4.5 - Part 1 of 9

```
;Field name :
ename
;Type of field :
alpha
;Length of field :
10
;Is this field in the base table Y/N :
y
;Is this field part of the primary key Y/N :
n
;Default value :

;Page :
1
;Line :
8
;Column :
17
;Prompt :
NAME :
;Display prompt above field Y/N :
n
;Allow field to be entered Y/N :
y
;Allow field to be updated Y/N :
y
;SQL>

;Is field mandatory Y/N :
y
;Is field fixed length Y/N :
n
;Auto jump to next field Y/N :
n
;Convert field to upper case Y/N :
y
;Help message :
Enter employee name -
;Lowest value :

;Highest value :

;Field name :
job
;Type of field :
alpha
;Length of field :
9
;Is this field in the base table Y/N :
y
```

"Employee" Application Response File

Figure 4.5 - Part 2 of 9

```
;Is this field part of the primary key Y/N :
n
;Default value :

;Page :
1
;Line :
9
;Column :
17
;Prompt :
JOB :
;Display prompt above field Y/N :
n
;Allow field to be entered Y/N :
y
;Allow field to be updated Y/N :
y
;SQL>

;Is field mandatory Y/N :
n
;Is field fixed length Y/N :
n
;Auto jump to next field Y/N :
n
;Convert field to upper case Y/N :
y
;Help message :
Enter employee's job title -
;Lowest value :

;Highest value :

;Field name :
salary
;Type of field :
money
;Length of field :
7
;Is this field in the base table Y/N :
y
;Is this field part of the primary key Y/N :
n
;Default value :
1000.00
;Page :
1
;Line :
7
```

"Employee" Application Response File

Figure 4.5 - Part 3 of 9

```
;Column :
51
;Prompt :
SALARY :
;Display prompt above field Y/N :
n
;Allow field to be entered Y/N :
y
;Allow field to be updated Y/N :
y
;SQL>

;Is field mandatory Y/N :
y
;Is field fixed length Y/N :
n
;Auto jump to next field Y/N :
n
;Convert field to upper case Y/N :
n
;Help message :
Enter employee salary
;Lowest value :

;Highest value :
6000.00
;Field name :
comm
;Type of field :
money
;Length of field :
7
;Is this field in the base table Y/N :
y
;Is this field part of the primary key Y/N :
n
;Default value :

;Page :
1
;Line :
8
;Column :
51
;Prompt :
COMMISSION :
;Display prompt above field Y/N :
```

"Employee" Application Response File

Figure 4.5 - Part 4 of 9

```
n
;Allow field to be entered Y/N :
y
;Allow field to be updated Y/N :
y
;SQL>

;Is field mandatory Y/N :
n
;Is field fixed length Y/N :
n
;Auto jump to next field Y/N :
n
;Convert field to upper case Y/N :
n
;Help message :
Enter employee's commission -
;Lowest value :

;Highest value :
3000.00
;Field name :
deptno
;Type of field :
int
;Length of field :
2
;Is this field in the base table Y/N :
y
;Is this field part of the primary key Y/N :
n
;Default value :

;Page :
1
;Line :
10
;Column :
30
;Prompt :
DEPTNO :
;Display prompt above field Y/N :
n
;Allow field to be entered Y/N :
```

"Employee" Application Response File

Figure 4.5 - Part 5 of 9

```
y
;Allow field to be updated Y/N :
y
;SQL>
select dname
into dname
from dept
where deptno = &deptno

;Message if value not found :
Invalid department number
;Must value exist Y/N :
y
;Is field mandatory Y/N :
n
;Is field fixed length Y/N :
y
;Auto jump to next field Y/N :
n
;Convert field to upper case Y/N :
n
;Help message :
Enter employee's department num0er
;Lowest value :

;Highest value :

;Field name :
dname
;Type of field :
char
;Length of field :
10
;Is this field in the base table Y/N :
n
;Default value :

;Page :
1
;Line :
10
;Column :
46
;Prompt :
NAME :
;Display prompt above field Y/N :
n
;Allow field to be entered Y/N :
```

"Employee" Application Response File

Figure 4.5 - Part 6 of 9

```
n
;SQL>

;Field name :

;Block name :
projects
;Table name :
pe
;Check for uniqueness before inserting Y/N :
y
;Buffer how many records ?
3
;Base crt line ?
19
;How many physical lines per record ?
1
;Field name :
empno
;Type of field :
int
;Length of field :
4
;Is this field in the base table Y/N :
y
;Is this field part of the primary key Y/N :
y
;Field to copy primary key from :
emp.empno
;Page :

;SQL>

;Field name :
projno
;Type of field :
int
;Length of field :
3
;Is this field in the base table Y/N :
y
;Is this field part of the primary key Y/N :
y
;Field to copy primary key from :

;Default value :
```

"Employee" Application Response File

Figure 4.5 - Part 7 of 9

```
;Page :
1
;Line :
1
;Column :
22
;Prompt :
PROJNO :
;Display prompt above field Y/N :
y
;Display prompt once for block Y/N :
y
;Allow field to be entered Y/N :
y
;SQL>
select pname
into pname
from proj
where projno = &projno

;Message if value not found :
Invalid project number
;Must value exist Y/N :
y
;Is field fixed length Y/N :
y
;Auto jump to next field Y/N :
n
;Convert field to upper case Y/N :
n
;Help message :
Enter employee's assigned project
;Lowest value :

;Highest value :

;Field name :
pname
;Type of field :
char
;Length of field :
10
;Is this field in the base table Y/N :
n
```

"Employee" Application Response File

Figure 4.5 - Part 8 of 9

```
;Default value :

;Page :
1
;Line :
1
;Column :
36
;Prompt :
PROJECT    NAME
;Display prompt above field Y/N :
y
;Display prompt once for block Y/N :
y
;Allow field to be entered Y/N :
n
;SQL>

;Field name :

;Block name :
```

```
+-----------------------------------------------------------------------+
|                                                                       |
|       E M P L O Y E E      P E R S O N N E L      R E C O R D         |
|                                                                       |
+-----------------------------------------------------------------------+
|                                                                       |
|                                                                       |
|                                                                       |
|                                                                       |
|                                                                       |
+=======================================================================+
|                                                                       |
|    E M P L O Y E E      P R O J E C T      A S S I G N M E N T S      |
|                                                                       |
+-----------------------------------------------------------------------+
|                                                                       |
|                                                                       |
|                                                                       |
|                                                                       |
|                                                                       |
+-----------------------------------------------------------------------+
```

```
%end
```

"Employee" Application Response File

Figure 4.5 - Part 9 of 9

Some changes will cause the order or number of questions IAG asks to be altered. This will occur if the new response triggers a different set of conditional questions. The response file will no longer be synchronized with the list of questions asked by IAG. To compensate, the user can anticipate the new question list and insert new responses in their proper place. If previously asked questions will no longer be asked, their associated responses must be deleted.

Adjusting the response file may become extremely complicated. An alternative approach would be to temporarily switch the input source to the user's terminal. This way the designer could answer each question individually, under the control of IAG, without having to correctly anticipate the new set and order of questions. The IAG will switch from the reponse file to the designer's terminal when a '%sw' response is encountered. The designer must change the original response to a '%sw' in the first question to be answered from the terminal. This question will then be reasked from the terminal.

The designer must decide where to resume the use of the response file. A convenient approach is to resume at the beginning of the next field. In this manner all the questions from the '%sw' to the next field will be answered from the terminal. To accomplish this, the designer must delete all the unwanted responses from the response file.

For example, changing the 'Page:' response from blank to a page number will cause the associated field to be displayed. This will trigger additional questions concerning the field's display position, prompt and data entry attributes. In this case the simplest approach would be to replace the 'Page :' response with a '%sw' and delete all the remaining responses for the field. When executing IAG the 'Page :' and subsequent responses for this field would be entered from the terminal. When the 'Field name :' question is asked again a '%sw' will resume the use of the response file.

This approach may be expanded to add new fields or blocks. For example if a new field was to be added after 'ename' the "Field name : " response for 'job' would be changed to '%sw'. When the input source is switched to the terminal this question will be reasked. The new field would then be defined. When the "Field name : " question is asked again, a response of 'job' is entered. A '%sw' response to the next question, "Type of field : ", will cause the resumption of the response file.

When making this type of modification to an application the user should not suppress the creation of a new response file. The new file will contain the combined responses from both the old response file and user's terminal.

Some text editors will not permit blank lines within the edited file. To support these editors IAG will interpret a '//' in columns 1 and 2 as a blank line.

4.5 **Generating the Sample Application**

This section will examine the IAG question and answer dialogue which defined the application shown in figure 4.4. Two processing blocks were defined for this single screen application. The first deals with employee information which is inserted into or retrieved from the "EMP" table in the sample "PERSONNEL" database. The second allows an existing employee to be assigned to one or more projects. This block references the "PE" table, which was created to allow employees to be assigned to multiple projects. Refer to the SQL Language examples in the ORACLE User's Guide for more details.

The name of the application is "employee" and was generated with the command string:

```
IAG  employee
```

Figure 4.5 is a listing of the "employee.inp" response file which was created from this terminal session. This file is included within the ORACLE distribution system, which allows this application to be generated using the command above. The first question identifies the sample 'personnel' database. An initial value of 3K bytes is specified for the SQL workarea. If that proves insufficient the value could be changed within the 'employee.inp' response file and the application regenerated.

The 'emp' block is the first defined and hence will be the first processed when the application is executed. Since the table and block names are the same, the table name could have been omitted and the block name would have been used. Prior to inserting a new row IAP will check that the primary key is unique within the table. The primary key may consist of one or more columns within the block. For a multi-column key, the combined columns will uniquely identify a row. One record will be buffered for this block which means that only one record will be displayed at a time. With the block questions completed the set of field questions will be asked for each field within this block.

'empno' is a four digit integer which maps to 'empno' column in the base table, 'emp'. Fields which are not in the base table, and do not map directly to a column will be discussed later. 'empno' is the only column in the primary key, and will be used to qualify rows for update and delete operations. Although it is the primary key, its value will not be copied from another field in this or another block.

The display area for this field will begin in the 17th position of the seventh line of page 1. Since the prompt is not displayed above the field, it will be placed immediately to the left. This field can be entered, but cannot be updated. Since fields within the primary key cannot be updated, the question "Allow field to be updated" is not asked, and a response of "N" is assumed. Primary key fields are also mandatory, hence a reply of "Y" is assumed for the question "Is field mandatory".

No SQL statement was defined for this field. This feature will be discussed later for the 'deptno' and 'projno' fields. Since the field is defined as 'fixed length' all four digits of an employee number are required for a valid entry. By selecting 'Auto jump', an automatic 'next field' is generated after the last character is entered. If the entry passes validation, the cursor will be moved to the next field position.

Converting to upper case has no meaning for numeric fields, and either reply is ignored. The 'Help message' will be displayed when the operator enters the 'HELP' function key. The valid range for employee numbers is between 1000 and 8000 inclusively. This range will be checked for both data input or update.

The 'ename' field maps to the 'ename' column in the base table; 'EMP'. Since it is defined as 'alpha' only the letters 'A' - 'Z' will be accepted. Responding "Y" to 'upper case' will force lower case letters to be displayed, and stored in upper case.

When an application is executed, the processing order of fields within a block is determined by the order the fields were defined to IAG. In this block 'empno' will be the first field processed, followed by 'ename', 'salary', etc. This order is independent of the page and display position of the field. If the next field is on a different page, the appropriate page will be displayed automatically.

'salary' is a 'money' type field and is restricted to a number with exactly two positions to the right of the decimal point. Since only a 'Highest value' was specified any value less than or equal to 6000 will be accepted. The 'comm' is not mandatory, allowing the operator to omit this field on insertion or update. Fields which are not entered are stored as nulls within the database.

A SQL SELECT statement was defined for the 'deptno' field. This statement will be executed each time the value of 'deptno' is modified. The purpose of this feature is to :

- initialize other fields with a value from the database.

- display information related to the entered field.

- check the existence of the entered field in a table of acceptable values.

In this case the 'deptno' SELECT will serve to verify that the entered department number exists in the department table, and to aid the operator in verifying, by department name, that the desired number has been entered. After executing the SELECT, the INTO clause causes the 'dname' field in the block to be initialized with the value of the 'dname' column. This value will be immediately displayed for the operator. The row returned is determined by the WHERE clause; the 'deptno' column must be equal to the entered value of the field 'deptno'. The '&' signifies a literal substitution of the field value 'deptno'.

A "Y" response to the "Must value exist" question requires that at least one row be returned. This implies that the entered department number must be within the 'dept' table. If not, the value will be rejected. A special message is defined to inform the operator that the value does not exist. "Invalid department number" will be displayed when a non-existent value is entered.

The 'dname' field is not in the base table. Even though it contains database information, that data is not mapped to or stored in the "EMP" table. In this case it is used to display the department name associated with the entered 'deptno'. The field is used strictly for operator reference and may not be entered. Entering data would serve no purpose, since the entered value would only be displayed, and not retained or used elsewhere.

A blank response to the "Field name" question (following definition of 'dname') signifies the end of field specification for the 'emp' block. The next block defined is 'projects', which refers to the 'pe' database table. In this case up to three records may be simultaneously displayed as illustrated in figure 4.4. A 'base' or starting screen line must be defined for the first record display area. Additionally, each record area may occupy one or more physical lines. For this block, three records may be displayed, beginning on line 19, containing one line for each record area.

'empno' is the first field defined. This field comprises half of the primary key for the 'pe' table. Its value will always be equal to the value of 'empno' in the 'emp' block (see "Field to copy primary key from"). Furthermore, the blank response to "Page:" designates the field as non-display. The value can be neither entered nor updated within this block.

Copying the primary key in this manner establishes a logical connection between the two application blocks. Entered or retrieved 'projects' information is implicitly associated with the last referenced employee number within the 'emp' block. The blocks could have been explicitly associated by requiring the operator to repeat the 'empno' within the 'projects' block. However, this requires additional keystrokes and is error prone. Making the field non-display simplifies the display area, especially since the field is already displayed on this screen page.

'projno' is a three digit integer which comprises the second half of the primary key. Unlike 'empno', its value will be entered by the operator. It will be displayed on the first (and only) line of each record display area. The prompt "PROJNO" will be displayed once on the line above the display or 'scrolling' area (line 18). Prompts may be repeated within each record display area of a multi-record block. However, this is most useful for prompts which are displayed to the left rather than above the field display area.

A SELECT statement has been defined for the 'projno' field. Its purpose is the same as described for 'deptno' in the 'emp' block, requiring the operator to enter a project number which already exists in the 'proj' table. If it does exist, the associated project name will be displayed in the 'pname' field. An appropriate message will be displayed if the value is not found. 'pname' is not in the base table and is provided only as an aid to the operator. Data may neither be entered nor updated in this field.

A blank response to "Field name:" terminates the field definitions for the 'projects' block. The blank response to "Block name:" terminates block specification. The input which follows is a line by line specification of the text used to enhance the screen page. This text is combined with the field defined prompts to create the display screen layout. The text must be supplied a page at a time in the order of the physical page number. The '%page' command is used to indicate the end of one page and the beginning of the next. '%line' commands establish positioning to a specific line on the currently defined page. The '%end' terminates the text input process and signals IAG to compile the application.

Compilation errors will be reported at the designer's terminal. Since IAG does not access the database dictionary during this process, errors relating to invalid 'column' or 'table' specifications will not be detected. These errors will be detected and reported during IAP execution. Once compiled, an application may be immediately executed. Section 3 of the "IAF Terminal Operator's Guide" discusses the operation of this 'employee' application.

ADVANCED APPLICATION TECHNIQUES

5.1 Introduction

This section will present some addtional techniques for using IAF. These techniques include consecutive sequence number genertion, setting up field defaults to eliminate setting up field defaults to elminate redundant data entry, using views, and a more sophisticated use of SQL within an application.

5.2 Assigning Sequence Numbers

In many applications it is necessary to uniquely identify new entries by assigning a consecutive sequence number. Upon entry the new record would be assigned a value one greater than the current/maximum value. An example is the 'orderno' in the sample order entry application.

Within an IAF application, sequence numbers can be automatically retrieved. This is accomplished with a SQL SELECT statement which selects the maximum value of the sequence field plus 1. The SELECT statement in the case of the order entry application is:

```
SELECT max(orderno)+1
INTO   orderno
FROM   order
```

This statement should be defined within any mandatory field. When a value is established for that field, either by operator entry, default value, or initialized via an INTO statement, this SQL statement will be executed and the 'orderno' initialized. Using a field which is not mandatory will not insure that a value will be assigned prior to insert.

The sequence field may optionally be displayed if the operator needs the power to override the derived value. In most cases the sequence field is a part of the IAF primary key and thereby not updatable.

An assigned value is not stored within the database until the record is inserted. Thus, it is possible that another operator may be assigned the same number. To minimize the interval between the time the value is computed and stored, the field used to trigger the SELECT should be at the end of the block. This assumes that the operator will perform the insert shortly after entering the triggering field. Duplicate record insertions can be prevented by including the sequence number in the primary key and requesting a check for uniqueness prior to executing the insert.

5.3 **Operator Defined Defaults**

In many applications the operator will repetitively process a block to insert multiple records. For this type of data entry application it is convenient to assign default values. Literal default values can be assigned when the application is defined, but they remain static over the life of the application. It may be more convenient to permit each operator to define his own set of defaults depending on his talk data entry tasks.

Operator defined defaults can be achieved by creating an additional block expressly for default specification. The fields defined in this block would be referenced in the "Default :" responses of fields in other blocks. This form of default is described in Sections 3 and 4.

When the application is initially executed, the operator would first process this form to enter the desired defaults. When other blocks are cleared or initialized in "INSERT" mode, the referenced default values would be copied. The values could be changed by the operator at any time by reprocessing this default block. The defaults then remain in effect until the application is terminated.

This approach can be extended to permit the operator to retain these defaults across executions of the application. This is accomplished by creating a "default" table within the application's database. Each operator would have their own set of default records. Instead of entering the values within the "defaults" block, the appropriate record could be retrieved. The fields within the "defaults" block would map to columns within this table.

The defaults block could also be initialized from some other database table. In this case the "default" fields would not be from the base table. At least one field from the base table must be defined, and at least one of these base table fields must be designated as part of the primary key. These fields are necessary to fulfill an IAF requirement, but will have no effect on the referenced table since this block would not be used for database modification. A dummy table could be defined for this purpose.

5.4 **Using Views**

As discussed in earlier sections an IAF application allows fields on a screen to be mapped directly to the columns of a database table. There is a one to one correspondence between a field and a single column. The association is implicit, requiring that the field and column names be the same. This precludes the possibility that a screen field could map to the average salary, maximum order number, or computation of (price x qty). However, there may be a need to have application blocks which report this summary information.

An effective way to provide this type of report is through SQL defined views. A view may be defined which simply projects a set of columns from a qualified set of rows for a single table. Although IAF could map directly to the table in this case, it cannot restrict the set of rows nor project a derived column (an arithmetic expression consisting of one or more columns - ie. (qty*price)). A view defined column name would be associated with each item in the select list.

An application block could reference this view as if it were a table. The fields would refer to the view defined columns. ORACLE restricts the use of views to retrieval operations. Therefore, if an operator attempted to insert, update, or delete a record within this block an ORACLE error would be returned. Since there is no mechanism to disable these functions by application block, the designer should define these fields so they can be neither entered nor updated. (Note that IAF requires at least one field in a block to be enterable.) Any view may be referenced by an IAF block. This includes views of joined tables or views which use the built-in functions to create virtual columns (ex: avg(sal) or max(qty)).

In some applications it may be desirable to have a summary report which assembles data from multiple tables. Since multiple tables are involved a different block would have to be generated to retrieve data fromeach table. The operator would have to process each block separately to assemble the desired information. A simpler approach is to create a view which joins the referenced tables. A single block is defined which could present all the data in a single operation. Again, this block could be used for retrieval purposes only.

5.5 **Using the SELECT Statement**

Any field may have an associated SQL SELECT statement. The statement is executed each time the field is modified. A field can be modified as the result of an operator entry, assignment of a default value, record retrieval, copied primary key value, or the object of another field's SELECT statement. Recall that, an INTO clause can direct the output of the first row returned to other fields within the block. On each execution, only the first row will be processed, with the remaining rows ignored.

The main purpose of this feature is to test for the existence of a field's value within a table of acceptable values, and to initialize other fields within the block. The second use offers a great deal of application flexibility. The full power of the SQL select can be utilized. No restrictions are placed on the content of this statement which may include arithmetic expressions, multi-table joins, nested selects, and GROUP BY, ORDER BY, and CONNECT BY clauses. A block field value may be substituted within the SELECT anywhere a character or numeric literal can be used.

5.5.1 **Cross Field Totaling**

For certain applications it is desirable to display a field whose value is computed from other fields. In the order entry example, the 'cost' of an item is equal to the 'quantity' times the 'price'. Although 'cost' would not normally be stored in the database (it could be derived from the 'quantity' and 'price' columns) it may be helpful for the operator to see this value as the data is entered.

The present version of IAF does not explicitly support the equating of a field value to the result of an arithmetic expression of other fields within the block. However, the same result can be achieved with an arithmetic expression in the select list of a SQL query. In this case the expression includes any fields within the block; database columns are not involved. The values of the fields are substituted by the IAP prior to statement execution. SQL will evaluate the expression and pass the result back to the IAP. The IAP places this result into the field specified on the INTO statement.

For example, assume the following fields are part of the 'orderitem' block:

PARTNO	QTY	X	PRICE	=	COST
A2B451	20		2.50		50.00

If the operator entered the 'partno', 'qty', and 'price', the 'cost' will be computed and displayed. If the operator changes either the 'qty' or 'price' a new value of 'cost' is automatically computed. This value will be recomputed each time either the 'qty' or 'price' is modified (including record retrieval). Note that if any field in the expression has not been assigned a value, or the value is null, the value of the expression is null. For example, if the annual salary (annsal = sal*12+comm) was computed using the "EMP" table, and either 'sal' or 'comm' were null, a value for 'annsal' would not be displayed.

The technique employed is to specify the following SQL statement within the 'qty' and 'price' field definitions:

```
SQL>select &qty*&price
SQL>into cost
SQL>from orderitem
```

The select list contains the expression to be computed. '&qty' and '&price' are variables whose associated field values will be substituted prior to statement execution. The INTO clause causes the result of the expression as returned by SQL to be stored and displayed in the 'cost' field. Any table name within the database may be specified in the FROM clause. Although a single row is processed by the query, no column data is returned. A WHERE clause is unnecessary.

Since the cost is not stored within the database record, the 'cost' field is defined as "not from the base table". If the computed value was to be retained the field could be mapped to a database column. In either case the field should not be entered into.

5.5.2 **Cross Field Totaling - Manual Request**

In the previous discussion, the computation was automatically triggered by modifying any field within the expression. This was accomplished by defining the SQL statement in every field participating in the computation. In this manner the SQL statement is triggered for execution each time any of the associated fields are modified. Omitting the statement from any field would inhibit the recompilation when that field was changed. This omission may be desirable if another SQL query must be triggered by that field. If omitted from all the associated fields another triggering mechanism is required.

As an alternative, the computational SQL statement may be defined with the field which is receiving the result. In the order example, the same statement as described in 5.5.1 would be defined within the 'cost' field. When the 'cost' field is modified by the operator, the statement will be triggered, and the newly computed value displayed. With this approach, the operator must manually request recompilation. The field may be modified by entering at least one valid character.

5.5.3 **Combining Detail and Summary Information**

A SQL query can be used to add summary information to the detail field entries within a block. Referring back to the order application, the operator may want to review the order following the entry of the individual line items. This 'review' block would contain the summary of the items ordered in addition to the general order information. A sample layout might be:

```
" O R D E R   S U M M A R Y   S C R E E N "

ORDERNO: 5674                              DATE: 04/01/81
CUSTNO : 87653     CUST NAME: Leisure Time Products

NO. ITEMS ORDERED : 6              SUB TOTAL :$ 349.52
                                   SALES TAX :   27.50
                                              --------
                                       TOTAL :$ 377.02
```

The orderno, date, customer number and customer name fields map to the base table 'order'. 'orderno' is the primary key and is copied from the 'order' block. Number of items ordered and sub total are computed by the following SQL query:

```
SQL>select count(*),sum(qty*price)
SQL>into no_items,subtotal
SQL>from orderitem
SQL>where orderno = &orderno
```

This statement may be included with any of the four base table fields. The result is that the summary of all the items associated with the order is computed from the 'orderitem' table and displayed in the 'review' block. Within this block, only the base table fields could be modified. The other fields should not be entered into.

The 'no_items' and 'subtotal' fields contain the count of items, and the sum of the cost of each item. The 'salestax' and 'total' fields are computed with the following SQL query defined with the 'subtotal' field :

```
SQL>select &taxrate*&subtotal,(1+&taxrate)*&subtotal
SQL>into   salestax,total
SQL>from   order
```

The 'taxrate' field was derived from the 'state' field in the 'order' block and is stored in the 'order' row.

5.5.4 **Summary Reports**

In section 5.3 a technique was discussed for creating a summary block containing fields from multiple tables. That approach was to create a view which joined all the participating tables. Each column in the view is mapped to a field in the block.

An alternative approach is to use SQL queries to assemble the desired data. The block would contain both 'database' fields from the base table, and 'control' fields to receive the data from the other tables. After retrieving the base table record, the retrieved field values would trigger the defined SQL statements. Each field could have an associated query. These SELECTs would retrieve the data for the 'control' fields. Each of the 'control' fields could have its own SQL statement which would also be triggered.

APPENDIX A

CRT INTERFACE UTILITY

Introduction

This section describes the IAF utility allowing a user to specify a non-standard crt for use with IAF. This utility, called CRT, compiles information from a user database, containing crt-descriptive parameters. The compiled information is then available to the IAP processor as needed.

A common limitation of programs which use the advanced features of a CRT is that the program can only run on the CRT for which it was originally intended. The reason for this is that for every maker of CRT devices, there is a different way of invoking these features. However, most manufacturers have chosen a fairly compatible scheme for controlling CRTs. This compatibility amongst CRTs lets IAP view all CRTs as being equivalent functionally and uses variables (which are read by IAP at program startup) to characterize their differences. These variables or parameters are stored in source form in a user-provided database. After a CRT's parameters have been entered into the database, the CRT utility compiles this source description into a file which IAP can directly access.

Required CRT Features

The following characteristics are required for IAF to interface to a particular CRT:

(1) Transmits each character as its entered
(2) Use the ASCII character set

Besides the above characteristics, the CRT must be able to perform certain control functions. Most CRTs support these functions, however not all of them will employ a compatible method of invoking them. A CRT must support the following features to be compatible:

(1) Move cursor left one position
(2) Move cursor right one position
(3) Absolute cursor positioning to column and row
(4) Clear to end of screen

In addition, the following restrictions apply to (3):

(3a) Row and column numbers must be able to be expressed either as ASCII strings (ie. Row 23 would be expressed as "23") or as a single character whose ASCII value is equal to the number plus an offset (ie. If the offset was 31 then row 23 would be the ASCII character "6").
(3b) Row and column numbers must begin with 0 or 1 and increase monotonically.

If a given CRT meets these requirements then it should be able to be supported by IAF. A possible exception to this might be that the CRT has certain timing characteristics that limit how fast data/command strings can be sent. IAF assumes that the CRT can handle continous transmission of data/commands up to the baud rate of the terminal. Since most terminals meet this requirement for the required functions, this should not generally be a problem.

Optional Features Utilized By IAF

Besides the required functions, IAF can take advantage of some specialized functions that are provided by more sophisticated CRTs. These include:

(5) Underlining (or equivalent attribute)
(6) Reverse video (or equivalent attribute)
(7) Split screen scrolling

The features of (5) and (6) are attributes of the characters being displayed and as such make the following assumptions:

a) A given attribute may be turned on and will stay on until it is explicitly turned off. That is, all characters transmitted after that will have the attributes assigned until another attribute sequence is detected.
b) Turning attributes on or off does not move the cursor.
c) Turning an attribute on or off does not "mark" a certain portion of the display area as having that attribute (ie. the PE OWL has this feature).

The feature (7) refers to is the ability to define a subset of the lines on the CRT which can act as an independent scrolling region from the rest of the lines on the CRT. For example if lines 1 through 7 are defined as a scrolling region, then these lines can be scrolled up or down without affecting lines 8 through 24. Top and bottom line numbers must meet the same requirements as (3a) and (3b).

CRT Definition - Preparation

Defining a CRT can be as simple as entering data into a form. A form is provided for this purpose and can be used as soon as at least one CRT definition exists. Before defining a CRT the following steps must have been performed. Normally these steps would have been carrried out as part of the ORACLE installation procedure.

(1) Create a database called CRT (If space is a premium, any existing database can be used, in which case this step may be skipped). This is where the CRT definitions will reside. This database may be created with DBF as follows:

```
DBF C CRT CRT.DBS 1K
```

A 1k database should be able to hold quite a few CRT definitions. The database may of course be a secure database so that only certain users may define or look at CRT definitions, as in the following example:

```
DBF C CRT CRT.DBS 1K JON/DOE
```

(2) Initialize the database. This consists of creating the tables and inserting data into them. This can easily be accomplished by logging on to UFI, as follows:

```
SQL CRT
```

substituting the particular database name for CRT if not using the CRT database. The SQL statements for initializing the database are stored on a command file provided with the ORACLE system. The following command will cause UFI to process that command file:

```
@CRT.SQL
```

The last thing the command file will do is select out the names of CRTs that have been pre-loaded into the database. If the CRT in question is among these, the CRT may be directly compiled as described in the "Compiling CRT Descriptions" section of this document. Exit from UFI by typing:

```
#EXIT
```

Parameters Used to Define a CRT

Once the CRT database has been created and initialized, the process of CRT definition may begin. The parameters for defining a CRT are usually available in the CRT manual.

Values will be required (possibly NULL) for the following parameters. These parameters are stored in a table called crt. To enter these parameters into the database the crt form provided with ORACLE may be used, or SQL INSERT statements may be used to bootstrap IAP for the first CRT. Some CRT definitions will already exist in the CRT database as a result of running the CRT.SQL command file. If the terminal is already defined, the crt form may be used to add additional CRT definitions. To find out what CRT definitions exist and a description of them, type SELECT NAME FROM CRT while in UFI.

PARAMETER	**TYPE**	**DESCRIPTION**
NAME	CHAR(20)	The name by which the CRT will be referenced. It should conform to the naming conventions for files in the operating system.
LINES	NUMBER	The number of lines of text that will fit on one screen. This value is 24 for most CRTs.
COLUMNS	NUMBER	The number of characters that will fit on on line of text. This is generally 80 or 132 for most terminals.
MSGL	NUMBER	The line to display help messages and diagnostics on. This is normally line 23. It can be no greater than LINES.
MODL	NUMBER	The line to display mode and other status information. This is normally line 24. It can be no greater than LINES.

BASE	NUMBER	Should be 1 or 0 depending on how the CRT numbers its lines. For example a 24 line CRT may number its lines from 0 to 23 or alternativly 1 to 24. Generally numbering starts with 1.
OFFSET	NUMBER	If row and column values are represented as single characters (for instance when postioning the cursor) then an offset value must be supplied. This number is usually 31 (decimal) and when added to a row or column value determines the character used to represent that value to the CRT. Leave the value NULL if the terminal represents these values as ASCII strings. See the section on CRT Requirements for further details concerning offsets.
CLEARSCREEN	CHAR(20)	The escape sequence to clear the screen or clear to end of screen. This is usually a 2 or 3 character sequence beginning with the escape character (033 octal). Later it will be shown how these sequences can be easily represented in the database.
BACKSPACE	CHAR(20)	The escape sequence (or control character) that will move the cursor left one position. This is a non-destructive backspace, that is it should not erase any characters. Control H works on most CRTs.

FORESPACE	CHAR(20)	The escape sequence for moving the cursor right one position. This must also be a non-destructive sequence. A space character is not an acceptable FORESPACE sequence
GOTOXY	CHAR(20)	The model escape sequence for postioning to an absolute column and row on the screen. The word model is used because the actual escape sequence has the actual row and column values put into the sequence when its used as opposed to the model esacpe sequence which has placeholders for the row and column values. More on that later.
TSET	CHAR(20)	A set of escape sequences used to configure the terminal into a particular mode. This sequence is executed once at program startup prior to any other sequnces.
TRSET	CHAR(20)	This sequence is executed prior to leaving the program to reconfigure the terminal back to a starting state. The definition of CRTs does not provide for storing the initial state of the terminal and restoring it to that starting state.
BOLDON	CHAR(20)	This sequence turns the inverse video attribute on. Any other attribute could be substitued for reverse video as well (such as highlighting).

UNDERON	CHAR(20)	This sequence turns the underline attribute on. Like the former, any other attribute could be substituted for underlining.
ALLON	CHAR(20)	This sequnce turns both underlining and reverse video attributes on simultaneously.
ATTOFF	CHAR(20)	This sequence turns all attributes off.
WINDOW	CHAR(20)	This model sequence creates a new scrolling region which is a subset of the lines on the display. It is a model sequence for the same reasons as GOTOXY is. The actual top and bottom lines are substituted at the time the window is created. The model sequence supplies placeholders for the actual values.
SCRUP	CHAR(20)	This sequence causes a window to scroll up one line causing the top line to disappear and a new line to appear on the bottom line of the window.

Control Sequences for IAP

Besides defining the parameters of a CRT, control sequences for invoking IAP functions (ie. NEXTFIELD, PREVFIELD, etc.) must also be defined. These control sequences map a key or keys to a particular IAP function. When that key(s) is pressed IAP will recognize it as a control sequence and execute the appropriate function. To avoid ambiguity, control sequences must begin with a non-printing character. More than one sequence can be defined to invoke a particular function and not all functions have to be defined allowing a subset of functions to be available.

Some CRTs will provide a group of keys (called function keys) which when pressed will send out an escape character followed by one or more characters. Defining these keys as control sequences makes invoking functions easier for the IAP operator. If the CRT does not provide these keys, single control characters make good alternatives. Its also helpful if certain common functions such as NEXTFIELD are mapped to appropriate keys like TAB or RETURN. The BACKSPACE function should be defined to be the same as the operating system's backspace key for consistency.

Since control sequences must use non-printing characters, it would be helpful if these sequences could be entered as printing sequences so that later it would be possible to see what was entered. For that reason the following conventions are used when defining CRT control parameters and control sequences:

\ooo	Where ooo is a 1 to 3 digit octal number representing some ASCII character.
\e	Represents the escape character (\033).
\\	Represents the \ character.

These sequences will be translated to their defined representations prior to being sent to the CRT. Additionally there are similar conventions for representing the placeholders for inserting the x and y screen coordinates used in GOTOXY and the top and bottom lines used in the WINDOW sequence. These formal parameters tell IAP where to insert the actual values when it wants to execute that function. The following conventions are used:

\x	The x coordinate for GOTOXY. This corresponds to the column to postion to.
\y	The y coordinate for GOTOXY. This corresponds to the line to position to
\t	The top line in the WINDOW function.
\b	The bottom line in the WINDOW function.

Function to Control Sequence Mapping

Defining the function to control sequence mapping consists of entering rows into the esc table in the CRT database. This may be done by INSERT statements when bootstrapping IAP or as part of the form crt. The esc table has the following structure:

NAME	CHAR(10)	The CRT the mapping is to be associated with.
FUNC	CHAR(2)	A one or two character abbreviation of the IAP function name.
ESCSEQ	CHAR(20)	The control sequence used to invoke the function.
COMMENTS	CHAR(20)	A description of what key(s) to press to cause ESCSEQ to be generated. This description will be used to drive the IAP function HELPKEYS which describes to the user how to invoke IAP functions.

The allowable values FUNC may take on are contained in the table functions. To list the allowable values while in SQL type SELECT * FROM FUNCTIONS. They are listed here for convienence:

NF	Next Field
NR	Next Record
NB	Next Block
PF	Previous Field
PR	Previous Record
PB	Previous Block
CF	Clear to end of Field
CR	Clear Record
CB	Clear Block
I	Insert record
Q	Query record
U	Update record
D	Delete record
X	Exit IAP
DC	Delete Character (backspace)
H	Help
DA	Display field Attributes
R	Redisplay screen
MR	Move cursor Right
ML	Move cursor Left
CM	Change character Mode (Replace or Insert)
FM	change Functional Mode (Insert or Query/Update)
CA	Clear All
DK	Display function Keys
SB	Scroll forward one Block
P	Print form

Compiling CRT Descriptions

At this point it is assumed that a CRT description has been entered into the CRT database as outlined above. Now it is necessary to compile the CRT description into a format that IAP can read. The output of CRT is a system library file which is read by IAP at startup. The CRT description is compiled by executing the CRT utility as follows:

```
CRT crtname [user/id]
  or
CRT crtname database [user/id] -u
```

The first case applies when the crt information has been stored on the crt database and the second case applies when a user database has been used. The user/id parameter is optional depending on whether its a secure database. Crtname is the name entered into the NAME column of the crt table. If no diagnostics come out, then the description was compiled successfully. An output file called crtname.crt will be placed in the system library.

After a description has been compiled successfully, IAP may use that CRT as follows:

```
IAP formname crtname
   or
IAP formname
```

The first format is used to specify a particular CRT. The second format is used to specify the default CRT (explained below).

It may be desired to have a particular CRT act as the default. This can be accomplished copying the crt file into a new file called default.crt (which should reside in the system library). In this case the crtname parameter may be omitted when invoking IAP and that CRT will be the default.

R S I

ORACLE

REPORT WRITER/TEXT FORMATTER

USER'S GUIDE

Oracle Users Guide - Version 2.3

"ORACLE"

REPORT-WRITER

USER'S GUIDE

TABLE OF CONTENTS

R E P O R T - W R I T E R

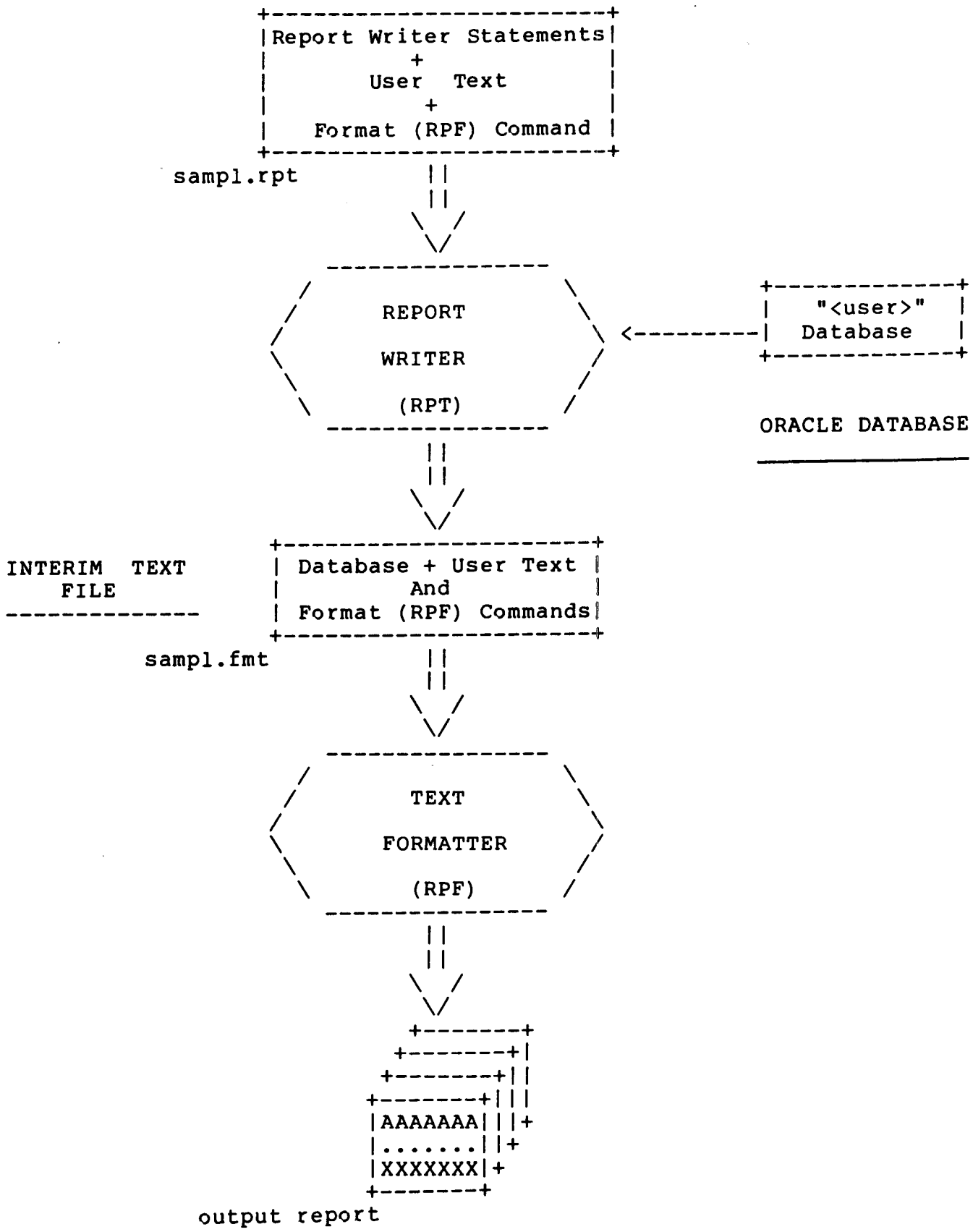
O V E R V I E W

The ORACLE **Report Writer** provides SQL's query capabilities, in addition to text formatting capabilities, to enable ORACLE to produce reports that combine information derived from the database with additional textual information such as headings, explanatory text and any other material desired. Potential applications for the Report Writer are quite varied. In the simplest case, it can be used to produce a report derived from a single ORACLE table, with column headings, columns of database information, and totals as desired. Examples of more complex applications are the production of reports with many levels of nesting, with multiple breaks in columns, and with a variety of subtotals and totals presented. Other applications that emphasize the text processing capabilities could include the production of computer-generated correspondence, with name and address information, as well as information included in the correspondence derived from the database, and the production of periodic budget and cost reports used in the management of a business. Of course, the Report Writer can also be used to print on preprinted forms.

The Report Writer is comprised of two programs that must both be used to derive information from the database and produce a report which presents the derived information in the desired fashion, as illustrated by Figure 1.1. The first of these, RPT, derives database information through SQL statements. The second, RPF, is a text formatter that formats the information based on format commands included in the text.

Generation of a report is controlled by a single file, the report control file, which contains report writer statements (for RPT), text formatter commands (for RPF) and any other text material that is to be included in the report.

This file can be created using any standard text editor. Once the report control file is complete, RPT is executed. RPT reads the report control file, scanning for report writer statements. Text formatter commands and user text are merged with the database-derived information that is produced by the RPT run.



" Report Generation Process "

Figure 1.1

Report Writer statements cause RPT to open the user's ORACLE database to derive information for the report. Incorporated within a report program are SQL queries to derive the desired data. Other statements cause RPT to include report heading and footing information, use specific data output formats, and conditionally branch to and execute other SQL or RPT statements. RPT can be directed to intersperse the database information within the RPF commands and user supplied text.

When RPT processing is completed, the interim file it produces is processed by the Report Formatter (RPF) to generate the desired report. Before processing by RPF, at the end of the RPT run, the interim file contains text supplied by the user, information extracted from the database and RPF commands specifying how that information should be placed on the report. RPF commands can specify:

- horizontal and vertical margins
- centering and underlining
- tabulation
- page numbering
- spacing and actual placement of text

RPF output can be directed to a line printer, typewriter terminal, or CRT. Figures 1.2, 1.3, and 1.4 are examples of Report Writer-generated reports. A detailed explanation of these report programs is provided in Section 5.

This manual serves two purposes. It is a detailed reference manual on the RPT Report Writer and the RPF Report Formatter. It also serves as a User's Guide, informing the reader by example how the various features of these two programs interact to construct a complete report. The reader is encouraged to first become familiar with the RPF text formatting language before tackling the RPT programming statements.

T E C H N O L O G Y S Y S T E M S , I N C

PERSONNEL REPORT FOR SEPTEMBER, 1980

EMPNO	NAME	JOB	SALARY	COMMISSION	DNO
7369	SMITH	CLERK	$800.00		20
7499	ALLEN	SALESMAN	$1,600.00	$300.00	30
7521	WARD	SALESMAN	$1,250.00	$500.00	30
7566	JONES	MANAGER	$2,975.00		20
7654	MARTIN	SALESMAN	$1,250.00	$1,400.00	30
7698	BLAKE	MANAGER	$2,850.00		30
7782	CLARK	MANAGER	$2,450.00		10
7788	SCOTT	ANALYST	$3,000.00		20
7839	OATES	PRESIDENT	$5,000.00		10
7844	TURNER	SALESMAN	$1,500.00		30
7876	ADAMS	CLERK	$1,100.00		20
7900	JAMES	CLERK	$950.00		30
7902	FORD	ANALYST	$3,000.00		20
7934	MILLER	CLERK	$1,300.00		10

END OF REPORT

"Example 1 - Tabular Report "

Figure 1.2

D i v i s i o n P a y r o l l R e p o r t

by Department

****

DEPTNO: 010 - - DEPTNAME: ADMINISTRATION

****

EMPNO	NAME	MONTHLY SALARY	COMM	ANNUAL COMPENSATION
7782	CLARK	$3162.50		$37,950.00
7934	MILLER	$1300.00		$15,600.00
7839	OATES	$5750.00		$69,000.00
	Department Summary			
AVG = $3,404.17	MIN = $1,300.00	MAX = $5,750.00		$122,550.00

****

DEPTNO: 020 - - DEPTNAME: RESEARCH

****

EMPNO	NAME	MONTHLY SALARY	COMM	ANNUAL COMPENSATION
7369	SMITH	$800.00		$9,600.00
7566	JONES	$3421.25		$41,055.00
7876	ADAMS	$1100.00		$13,200.00
7788	SCOTT	$3000.00		$36,000.00
7902	FORD	$3000.00		$36,000.00
7955	WILSON			
7956	JAKES	$1000.00		$12,000.00
	Department Summary			

" Example 2 - Nested Report "

Figure 1.3

Date : 03/25/81

To: BLAKE

Department : 30 - SALES

Location : PARIS

From : Bill James
Director of Personnel

Subj : 1982 Employee Compensation Plan

Its budget time again! To aid you in completing the salary portion of your budget I have computed your department's current salary expenses. For planning purposes we are presently estimating an across the board increase of 10% in the 1982 salary pool.

Your department's 1981 figure is: $140,430.00

Estimated 1982 figure is: $154,473.00

In completing your salary plan you should break this total down by individual employees. Please let me know if you have any questions on this or related issues.

Bill

Example 3 - Sample Letter

Figure 1.4

U S I N G

T H E R E P O R T F O R M A T T E R

(R P F)

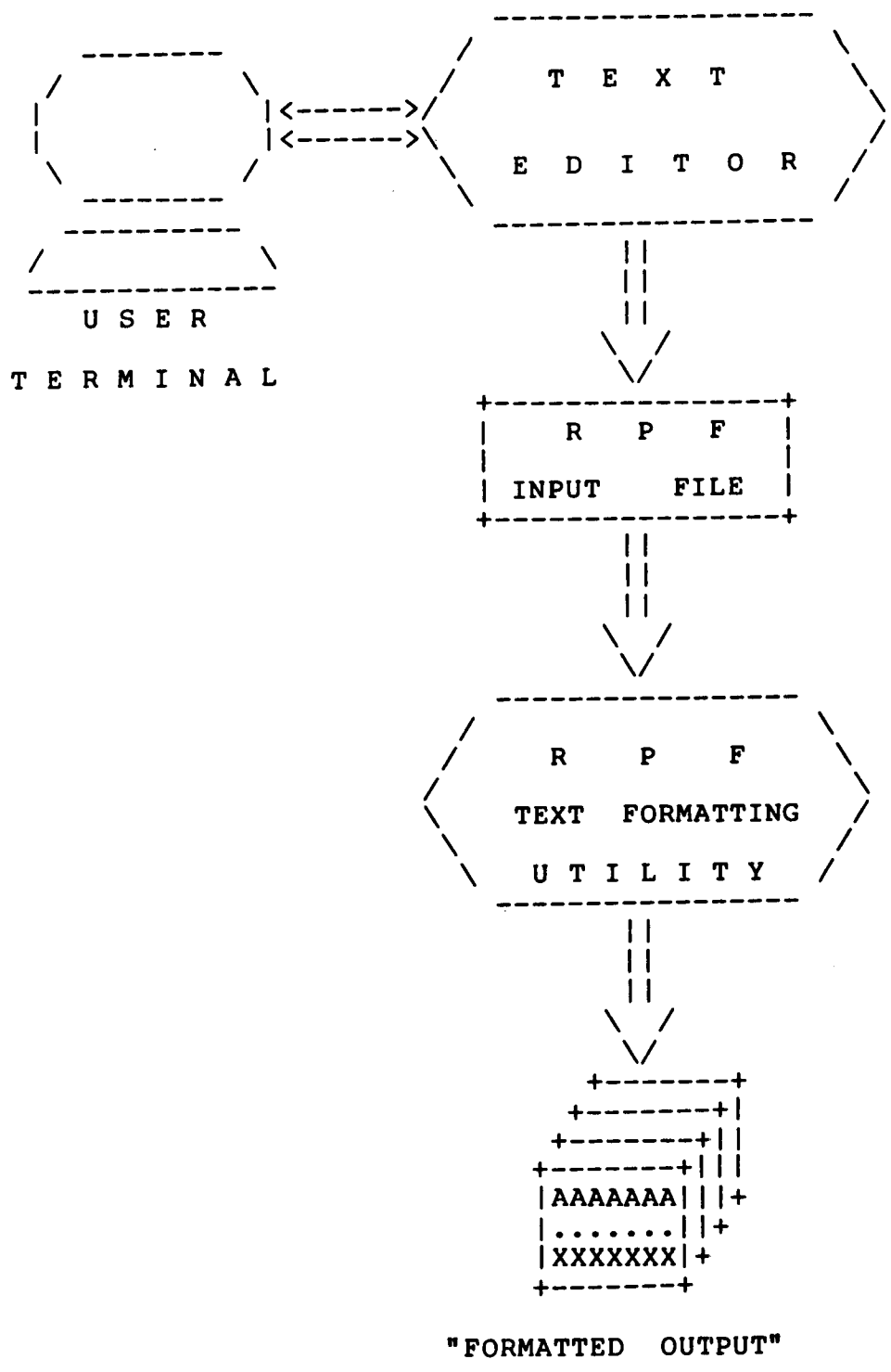
2.1 **Overview**

This section discusses methods for using RPF commands to create a complete report. First, a simple example is presented, and methods for producing it are described; then a sequence of increasingly complex examples is presented, each with its own discussion of methods for producing it.

RPF receives both the input it processes and the commands that direct the processing in the same file. Thus, commands imbedded within the file to be formatted are used to control the formatting operation. The RPF user presents commands to RPF by inserting them into the file to be formatted, usually using a text editor (of course, if the file is constructed by a program, RPF commands can also be inserted by the program). This process is illustrated by the diagram of Figure 2.1.

The imbedded commands control a variety of options, such as what margins to use, when to skip to the next page, how far to indent a paragraph, where to skip a line, how to number the pages, etc.

RPF reads an input file and formats it into output pages as directed by the imbedded commands. The text is viewed as a series of **words.** A word is a collection of one or more characters which is terminated by at least one space or an end of line character (An end of line character is inserted into the text file by the editor when the user enters a carriage return). RPF places each word, one after another into the output line. One space will separate each word on output, regardless of the number of spaces the user may have entered between the words on input.


T E X T
E D I T O R
U S E R
T E R M I N A L
R P F
INPUT FILE
R P F
TEXT FORMATTING
U T I L I T Y
"FORMATTED OUTPUT"

" Figure 2.1 "

When the line is full and the next word will not entirely fit within the current line, the last word is shifted to the right so that the last character is aligned with the right margin. A word is indivisable and is never split between lines. RPF puts spaces between words so that the right and left margins are justified. RPF will continue to place words in the output pages until the end of the input file is encountered.

RPF allows the user to define the starting and ending positions of the output line. The area on a line where text is placed is called a **column.** If the user does not specify the boundaries of a column, RPF will place the text in a default column which begins in position one and extends to position 132.

2.2 **Example 1 - A Simple Letter**

Now that the basics have been reviewed, let's look at the input file which is required to produce a simple letter. Figure 2.2A is a listing of the file which created the letter in figure 2.2B. The input file was created using a standard text editor. The line numbers and titles were added after RPF was executed.

RPF differentiates its commands from the user's text by the leading '.'(period) or '#'(pound sign). A user may begin words with these characters providing the word is not exactly like an RPF command. If it is, a second '.' or '#' will tell RPF that this is text. On output only one '.' or '#' will be printed. Commands may be specified with either upper or lower case letters.

The letter was printed with left and right margins of '13' and '73'. This was done by defining a single column beginning in position '13' and extending to position '73'. The command to do this is shown in Line 1.

The '.dt' command on Line 1 defines a table with an identifier of '1' and contains the column beginning in position '13' and extending through and including position '73'. Notice the period following the '73'. It is required to tell RPF that there are no more columns in this table.

```
 1: .dt 1 13 73 .
 2: #page 6 58
 3: .sp 1
 4: .t 1
 5: .s 3
 6: January 10, 1981
 7: .s 3
 8: Mr. William     O.       Smith .n
 9: 3752 Oakwood Drive .n Seaport, Me.  96142
10: .s 4
11: Dear Bill:
12: .s 4  .p
13: I hope you have received the medical and dental insurance forms that I se
14: .s 1
15: .p The form you submitted for your November 1, 1980 visit to Dr. George
16: Smedley did not indicate the nature of the treatment. Please have the
17: doctor write me a note describing the reason for the visit and the
18: treatment which was prescribed.
19: .b   .p
20: If you have any questions please give me a call at my office.
21: .s 2
22: Sincerely,
23: .s 4
24: Sharon Brown
25: .te
```

Example 1 - RPF Input Text

*** Figure 2.2 A ***

January 10, 1981

Mr. William O. Smith
3752 Oakwood Drive
Seaport, Me. 96142

Dear Bill:

I hope you have received the medical and dental insurance forms that I sent.

The form you submitted for your November 1, 1980 visit to Dr. George Smedley did not indicate the nature of the treatment. Please have the doctor write me a note describing the reason for the visit and the treatment which was prescribed.

If you have any questions please give me a call at my office.

Sincerely,

Sharon Brown

Example 1 - RPF Output

▪ Figure 2.2 B ▪

Defining a table does not automatically cause the text to be placed within the specified column boundaries. A table must be invoked before it will take effect. The 't' command in Line 4 invokes table '1'. If user text was included prior to Line 4 it would have been formatted according to the definition of the default table; a single column beginning in position '1' and extending through position '132'.

The 'page' command on Line 2 defines the top and bottom margins of a page. RPF will begin placing text on the sixth line of every page and automatically skip to the next page after completing line 58. All pages are assumed to have 66 lines. If the print spacing is six lines per inch, the 66 lines will cover a standard 11 inch page.

The 'sp' command defines the spacing between lines. Since the default is single spacing this command could have been omitted. The '.s 3' command on lines 5 & 7 will cause three lines to be skipped. Printing on stationery will require skipping enough lines to have the first line printed below the letterhead.

Line 6 contains the first line of user text. Three words are specified: 'January', '10,', and '1981'. Remember that the number of spaces between words on input has no effect on the spacing on output. These words could have been input on separate lines and still produced the same results.

Line 8 contains the name of the addressee followed by a '.n' (new line) command. This is the same as the 'return' key on a typewriter. The 'n' command on Line 9 causes the city/state to be printed on the line following the street address.

The paragraph command ('.p') on Line 12 causes the text which follows to begin on the next line and be indented five spaces. The text within the body of the letter is printed within the column boundaries. As noted above as many words as possible will be placed on a line. The word 'insurance' would not completely fit on the same line as 'dental'. The word 'dental' was aligned with the right hand margin, and the additional spaces were evenly distributed throughout that line.

The blank line ('.b') command inserts one blank line and has the same effect as '.s 1' (skip one line).

The table end ('.te') command on Line 25 instructs RPF that table '1' is no longer active. Column boundaries revert back to the default table. If the active table is not terminated an 'UNEXPECTED END OF COLUMN' message will be issued.

The sample letter was printed on a typewriter quality printer with the following system command:

```
RPF tt1:/pa=letter.abc
```

where tt1: is the system address for the desired printer. The /pa instructs RPF to pause and sound the alarm before printing each page. This will permit the user to individually load each sheet of paper. When ready, any character may be entered to start the printing. The entered character will not be printed.

'letter.abc' is the name of the input text file. If the file qualifier is omitted (in this case 'abc'), 'rpf' is assumed.

2.3 **Example 2 - A Tabular Report**

The second example is the tabular report shown in figure 2.3B. Figure 2.3A is a listing of the input text file which produced this report. The line numbers and titles in 2.3A were added after RPF was executed.

The '.dt' command on line 1 defines table '1' which contains a single column extending from position 13 to 73. A second table, table '2', is defined on line 2. This table contains 5 columns; the first extending from position 1 to 5, the second from 9 to 28, the third from 32 to 39, and so on. Notice that the fifth column is defined by the last pair of numbers; 52 and 0. The zero signifies that this column ends on the right hand margin of the column in which this table is invoked. We will see a little later how table '2' is used to create this tabular report.

```
 1: .dt 1 13 73 .
 2: .dt 2 1 5 9 28 32 39 42 49 52 0 .
 3: .page 6 58
 4: .t 1
 5: .s 4
 6: .cul " ALL SEASONS SPORTING GOODS " .
 7: .s 2
 8: .cul MONTHLY SALES REPORT .
 9: .s 4
10: .t 2
11: ITEM .n NO. .nc
12: .cen ITEM . .n .cen DESCRIPTION . .nc
13: .r PREVIOUS .n MONTH .nc
14: .r CURRENT .n MONTH .nc
15: .r Y-T-D .nc
16: .s 2
17: 2354 .nc NFL Football .nc $175.34 .nc $202.45 .nc $564.89 .nc
18: .s 1
19: 6734 .nc Chicago Cubs  Baseball Uniform .nc $56.10 .nc $162.38 .nc $287.01
20: .s 1
21: 8940 .nc Alpine Skis  .nc $941.84 .nc  $1005.93 .nc  $3582.57 .nc
22: .s 1
23: .nc .nc -------- .n $1173.28  .nc -------- .n $1307.76 .nc -------- .n
24: $4434.47
25: .te .s 3
26: .cul END \ OF \ REPORT .
27: .te
```

" Example 2 - A Tabular Report "

Figure 2.3 A

" ALL SEASONS SPORTING GOODS "

MONTHLY SALES REPORT

ITEM NO.	ITEM DESCRIPTION	PREVIOUS MONTH	CURRENT MONTH	Y-T-D
2354	NFL Football	$175.34	$202.45	$564.89
6734	Chicago Cubs Baseball Uniform	$56.10	$162.38	$287.01
8940	Alpine Skis	$941.84	$1005.93	$3582.57
		$1173.28	$1307.76	$4434.47

END OF REPORT

"Example 2 - A Tabular Report"

Figure 2.3 B

Line 3 defines the top and bottom margins of the output pages, and line 4 invokes table '1'. Since table '1' has only one column, the text that follows will be formatted within that column's boundaries. The '.cul' command on line 6 indicates that the text between the command and the terminating period should be centered within the current column and underlined. Since the current column extends from position 13 through 73, this text is centered on the page.

Figure 2.3A was printed on a special printer which allows bold face type to be substituted for underlined characters. If this option was not selected or a standard line printer was used, the specified text would have been underlined. If the output device is a video terminal (CRT) the user can specify at RPF execution time that all underlined text be displayed in reverse video. See Section 2.6 - 'Executing RPF' for additional information.

The '.cul' command on line 8 produces a second line of centered and underlined text.

At this point we are ready to create our tabular report. The report will contain five columns across the page. If this report was being typed the user would probably set tab stops for the first position of each column. With RPF a new table is invoked to establish these column boundaries. The command on line 10 invokes table '2', which was previously defined on line 2. Unlike table '1' which had only one column, table '2' provides five columns in which to output text.

When a table is invoked it subdivides the 'current' column. In this case the 'current' column is the single column of table '1'. Therefore, invoking table '2' subdivides the 61 print positions from position 13 through 73 into five separate areas or columns. The column boundaries for a table are interpreted relative to the current column. For example the first column in table '2' begins in relative position 1 and extends to relative position 5. Since the current column begins in position 13, the absolute boundaries of this column are 13 through 18. The last column begins in relative position 52 (absolute 65) and extends to the end of the current column (absolute position 73).

A table must completely fit within the column in which it is invoked. The column in which table '2' was invoked contained 61 print positions. If any columns in table '2' extended beyond relative position 61 an error would be indicated. For example, if the last column was defined as '52 65', when the command on line 10 was executed RPF would have returned the error "COLUMN TOO SMALL".

Now that table '2' has been defined and invoked, output text may be placed in each of the five columns. The text on line 11 is placed within the first column. The '.n' command causes the text 'NO.' to be placed on a new line within this column. The '.nc' command causes the placement of text in the current column to stop. The text which follows will be place on the first line of the next column in table '2'.

Line 12 requests that the words 'ITEM' and 'DESCRIPTION' be centered on the first two lines of the second column. The '.r' (right justify) command on lines 13, 14, and 15 tells RPF that all subsequent text lines for columns 3, 4, and 5 should be right justified. Notice that the column headings are aligned with the right hand boundary of these columns.

Advancing past the last column of a table ('.nc' on line 15) causes the first column to become current again. The '.s 2' command on line 16 would have the same effect in addition to skipping two lines. Line 17 places the first line of data into each column of the report. Notice that the dollar values in the last three columns are right justified. Lines 19 and 21 define the output for the second and third lines of the report.

The text 'Chicago Cubs Baseball Uniform' would not fit on one line of the 'ITEM DESCRIPTION' column. As explained above, RPF will place as many words as possible on a line of a column. When a line is full, the last word is right aligned and the remaining text is placed on the next line. Since 'ITEM DESCRIPTION' required two lines, RPF adjusted the other columns so that all the entries for the next item ('ITEM NO.' 8940) are on the same line. RPF will cause the appropriate number of lines to be skipped within each column so that text placement will begin on the line after the longest column.

The summary information is specified on lines 23 and 24. Since no text was displayed in the first two columns, two '.nc' commands were included to position to the third column. The output text for the last three columns contains a line of dashes followed by the column total on the next line.

The '.te' on line 25 terminates the current table; Table '2'. Terminating a table causes the previously invoked table to become active again; Table '1'. The text on line 26 will be centered and underlined within the column boundaries of 13 and 73. The back slash '\' requests RPF to insert a second blank character between each of the words. Normally only one blank separates words on output.

The '.te' on line 27 terminates Table '1' causing the default table to become active.

R E P O R T F O R M A T T E R

(R P F)

3.1 **Overview**

The Report Formatter (RPF) reads a file which contains text for the report, database-derived information produced by RPT, and RPF commands. RPF then writes a file that contains the text of the original file, arranged as directed by the RPF commands. In addition to the imbedded RPF commands, the final format is also controlled by the choice of RPF options, which are selected at the start of RPF execution. The file read by RPF is usually constructed by the use of a text editor; then RPT is executed first, to process commands to derive database information for the report, then RPF is executed to format all of the report information for printing.

In addition to the use described above to generate an ORACLE report, RPF can also be used alone as a general-purpose formatting program for a variety of word processing applications, such as correspondence, memoranda and reports such as this manual, which is itself produced using RPF.

3.2 **RPF Input**

Input to RPF consists of text and RPF commands, intermixed in the input file. The basic unit of text data is a **word** which is a string of one or more characters terminated by a blank, tab, carrier return, or form feed (newline) character. The input string:

The boy went<tab>to the<cr> store.

contains the following six words: "The", "boy", "went", "to", "the", and "store". Blanks, tab, carrier return, form feed, and characters serve only as delimiters and have no effect on the placement of words in the RPF output.

3.2.1 **RPF Command Format**

RPF commands control the placement of words in an output line, horizontal and vertical margins, page numbering and control, horizontal and vertical spacing, line skipping, etc.

All RPF commands start with either a pound sign (#) or period (.). The "#" or "." may be used interchangeably without affecting the meaning of the command. Words which begin with a "#" or "." but are not valid commands will be treated as text. To produce output that is identical to a valid command, the valid command is preceded with an additional "." or "#". For example, #b is a command to insert a blank line, but ##b will be treated as text and output as #b.

Commands may be specified in upper or lower case characters. Each command may be followed by one or more parameters. Commands and their associated parameters are separated by one or more blank, tab or form feed character. A single "." or "#" is used to terminate commands which have a variable number of parameters, or operate on a group of words (e.g. centering, underlining, etc.).

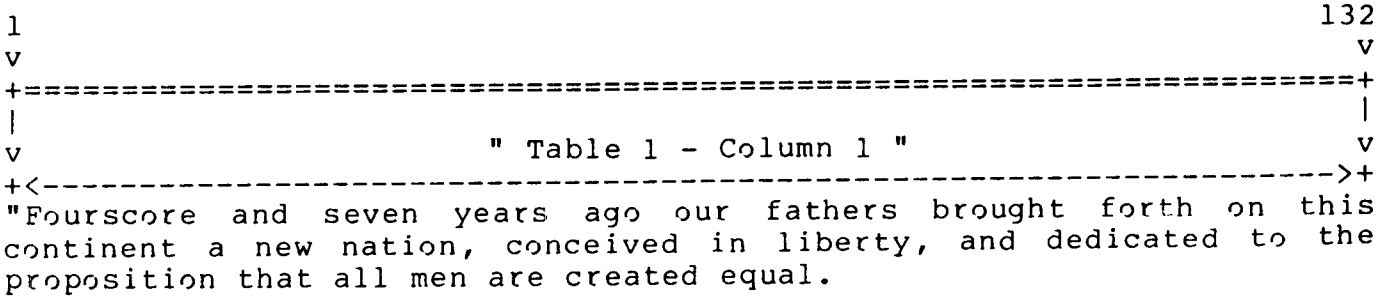
3.3 **Tables and Columns**

RPF processes text one word at a time, placing each word into the output line within the boundaries of the current column. A **column** is defined by its starting and ending character position. Words are placed into columns separated by at least one blank character, beginning with the first character position, and extending to the end of the column. Words are indivisable, and are not split across the lines of a column. Therefore, a word's length must be less than or equal to the width of the current column, or an error indication will be given by RPF.

The width of a column is equal to the last character position minus the first character position plus 1. The last word is aligned with its last character placed in the last column position. Any additional spaces introduced by alignment are evenly distributed throughout the line.

Figure 3.1, Example 1 shows the text of Lincoln's Gettysburg Address placed in a column beginning in position 1 and extending to position 132. Note that all lines except the last are right justified.

Example 1: **Default Table**



Example 2: **Define Table Within Current Column of Previous Table**

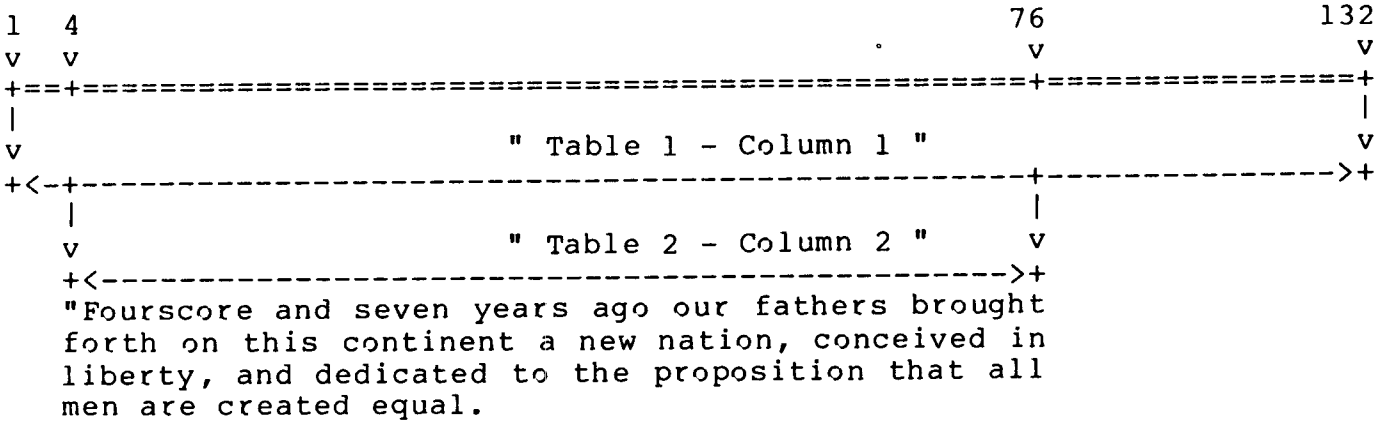
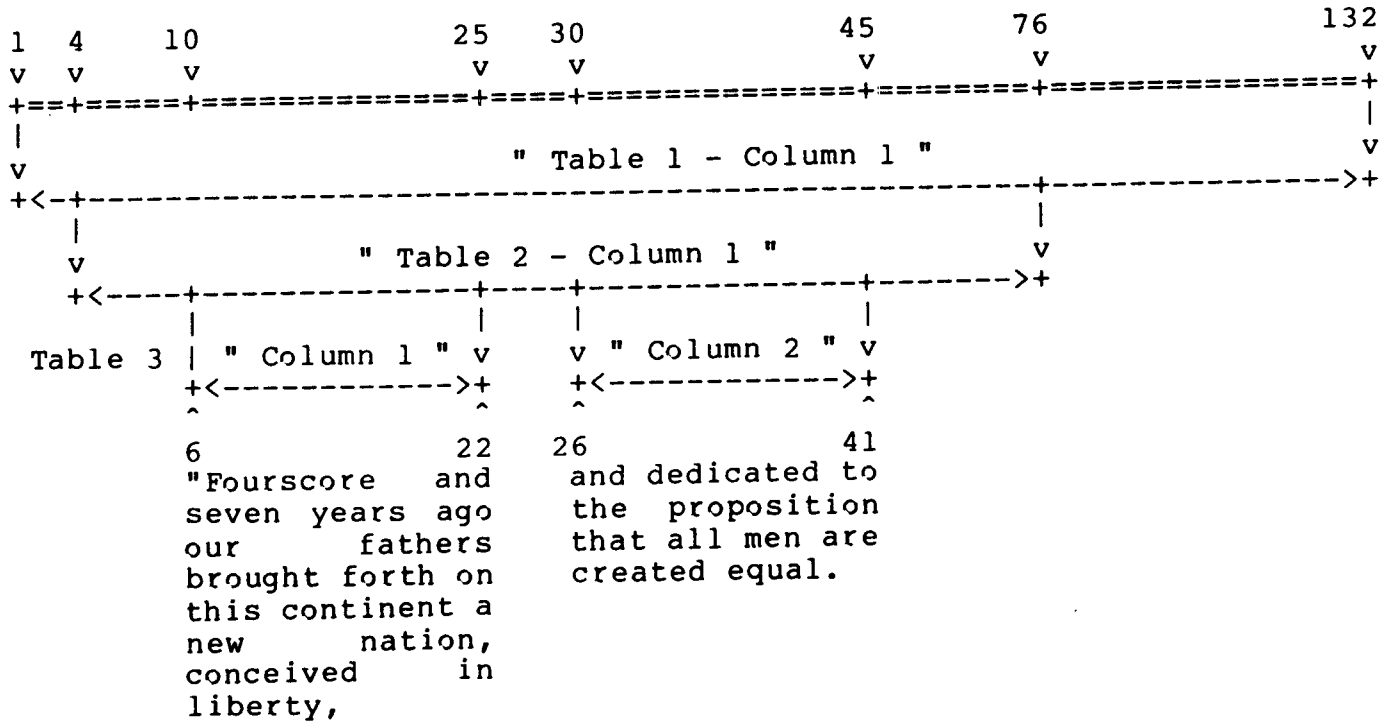


Figure 3.1 - Part 1 of 2

Example 3: **Define Two Column Table within Current Column**



TABLES AND COLUMNS

Figure 3.1 - Part 2 of 2

A **table** consists of from 1 to 20 columns, and defines the boundaries for word placement in the current line. The default table shown in Example 1 contains one column, and serves as the initial table definition. Other tables may be defined, and invoked within the boundaries of the current column.

Example 2 of figure 3.1 shows a second table which has been invoked within the single column of the default table. This table has one column which extends from the 4th position to the 76th position. The text of Example 1 is reformatted within this new table and column definition.

A table is always invoked within a column, and its column's character positions are counted relative to the beginning of that column. Example 3 shows a third table which contains two columns; the first begins in relative position 6 and ends in position 22 and the second begins in relative position 26 and ends in position 41. This table has been invoked within the single column of table 2. Since the column in Table 2 begins relative position 4 of the single column in the default table, the columns of Table 3 begin in absolute locations 10 and 30.

A table must fit within the column in which it is being invoked; that is, the length of the table must be less than or equal to the length of the column. The length of a table is equal to the last character position of the last column defined in that table. For example, Table 3 has a length of 41 (the second and last column extends from position 26 through 41). Therefore, this table can fit within the current column, whose length is 73 (76-4+1).

In Example 3, the sample text has been partially displayed in both columns of the table. RPF will place words into the current column until a command to advance to the next column is encountered. In this example, a "next column" command after the word "liberty," causes RPF to place the remaining text into column 2

Commands to define and invoke tables and advance the column position are presented in detail in section 00.

3.4 **Imbedded Blanks**

Sometimes it is necessary to preserve a specified number of blank spaces between two words. Normally, the number of blanks between words in the input file is ignored, and RPF will separate words by single blanks. A back slash character followed by a blank is used to include additional blanks in the output, as illustrated by this example:

	Input Text	Ouput Text
1-	Abraham Lincoln	Abraham Lincoln
2-	Abraham Lincoln	Abraham Lincoln
3-	Abraham \ Lincoln	Abraham Lincoln
4-	Abraham \ \ Lincoln	Abraham Lincoln
5-	Abraham\\Lincoln	Abraham\Lincoln

Lines 1 and 2 show normal RPF operation. The combined "\ " causes a second blank to be included in line 3 and a third in line 4. The double back slash on line 5 informs RPF that it is desired to have a "\" printed in the output text, as opposed to reserving blanks. A single "\" in the text will be ignored.

3.5 **RPF Commands**

This section will discuss each RPF command in detail. The commands will be presented in alphabetical order, and no attempt will be made to demonstrate their interrelationship. Figure 3.2 provides a summary of these commands.

Command	Description
.APN	Alternate Page Number: placement for even numbered pages.
.B	Blank: Insert one blank line in output text.
.CEN	Center: Center following text in current column.
.CL	Column Literal: Suspend formatting for the following lines in the current column
.CS	Column Skip: Skip 'n' lines in current column
.CUL	Center With Underline: Center and underline the following text within the current column
.DT	Define Table: Define the column boundaries for the specified table.
.F	Figures: Reserve specified page numbers in output document for figures, charts, etc.
.HS	Horizontal Spacing: for characters on Diablo type printer
.I	Indent: Ident the following text in current column.
.L	Literal: Suspend formating for the following text lines; column definitions are ignored.
.N	New Line: in the current column.
.NC	New Column: Advance to next column
.NP	New Page: End current and start new page.
.PAGE	Page: Define top and bottom page boundaries

RPF Commands

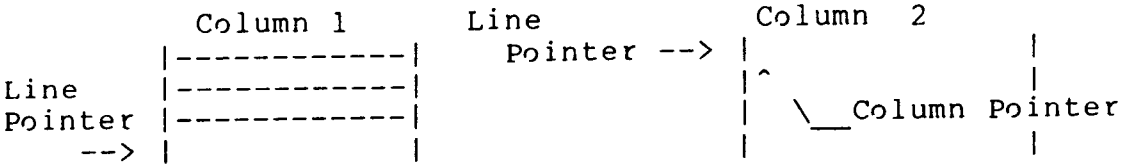
Figure 3.2

Command	Description
.P	Paragraph: New line to be started within the current column, indent 5 spaces at the beginning of the line.
.PAUSE	Pause: Pause outputting to terminal until signal from operator
.R	Right Justify: Set/reset switch to right justify all text placed in current column
.S	Skip: Skip specified number of lines
.SP	Spacing: Define spacing for current column
.SPN	Start Page Numbering: specify page numbering
.T	Table: Invoke specified table within the current column
.TE	Table End: Terminate table and revert to previous column definition
.UL	Underline: Underline the following text.
.VS	Vertical Spacing: Define vertical spacing for Diablo type terminal.

RPF Commands

Figure 3.2 (Continued)

For the purpose of the following command discussions two pointers will be introduced. The **column pointer** identifies the current column within the active table. The **line pointer** identifies where the next line of text for a column will be outputted. Each column has its own line pointer. For some commands, diagrams will be provided to exemplify the movement of these pointers. The following diagram shows a two column table with the **column pointer** positioned to column 2. Column 2's **line pointer** points to line 1, while Column 1's line pointer points to line 4.



3.5.1 **Alternate Page Number**

This command is used to define the character position where the page number will be printed for even number pages. If both sides of the paper are used during reproduction, this command will allow even numbered page numbers to appear on the outside edge of the page. See #SPN command for details on page numbering.

.APN <position>

<position> is the absolute position that the first character of the page number will be placed on even numbered pages.

3.5.2 **Blank**

This command causes one blank line to be inserted, and is equivalent to #S 1. This command affects all the columns in a multi-column table. Refer to #S command for details on the advancing of column and line pointers.

.b

3.5.3 **Center**

This command causes the specified text to be centered within the current column. The centering takes place on the next line and includes all the words between the command and the first unattached "." or "#". The centered text must fit within the current column.

CEN <text> .

<text> .	is the collection of words to be centered. Note that standard formatting is performed; multiple blanks, and tab and form feed characters are ignored.

3.5.4 **Column Literal**

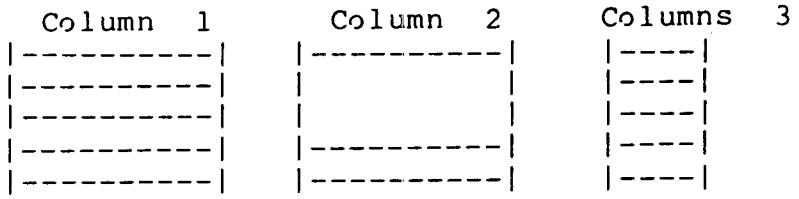
This command defines one or more lines of text which will be outputted in the current column without formatting. Each line will be sent to the output terminal exactly as entered, including multiple blanks, tab characters and form feed characters. The length of each line must not exceed the width of the current column.

.cl
...
...
...
<text lines>
...
...
...
.

<text lines>	lines of text sent unformatted to output terminal

3.5.5 **Column Skip**

This command causes the specified number of blank lines to be inserted into current column. This command has no effect on any other column. The following diagram shows two blank lines inserted into column 2 whereas these lines need not be blank for columns 1 and 3.



.CS <no. lines>

<no. lines> is the number of blank lines inserted.

3.5.6 **Center With Underline**

This command causes the specified text to be centered within the current column and underlined. Centering takes place on the next line and includes all words between the command and the first unattached "." or "#". The centered text must fit within the current column.

.CUL <text> .

<text> is the collection of words to be centered and underlined. Note that standard formatting is performed; multiple blanks, and tab and form feed characters are ignored.

3.5.7 **Define Table**

This command defines a table and its associated columns. The command does not cause the table to be invoked. For information on invoking a table, refer to section 3.5.21 (Table Command).

.DT <table id> <sp1> <ep1> <sp2> <ep2> ... <spn> <epn> .

<table id> is a number from one to ten which identifies the table.

Specifying a previously used **<table.id>** will cause the old definition to be replaced with the new definition.

<spn> <epn> Each pair of numbers defines the boundaries of a column.

<spn> is the starting position of the nth column relative to the beginning of the table.

<epn> is the ending position of the nth column relative to the beginning of the table.

The boundary of a column includes the starting and ending column positions.

The starting position of a column must be at least one greater than the ending position of the previous column.

At least one column must be defined in a table.

If the ending position of the last column is less than or equal to zero, the ending position will default to the end of the column in which the table is being invoked. Extending the last defined column to the end of the invoking column causes the right margin to be right justified.

The **.DT** command is terminated by a single unattached "." or "#".

Example:

.DT 1 12 24 32 48 .	This command defines a table with an id of 1 which contains two columns. The first extends from position 12 to position 24, and the second from position 32 to position 48.
.DT 1 13 73 . **.DT 2 5 0 .**	Two one column tables are defined. The single column of table 2 extends from position 5 to the end of the column in which it is invoked. If table 2 was invoked within table 1, the output text would be right aligned.

3.5.7 **Figures**

This command reserves the specified page numbers for figures, charts, diagrams, etc. If page numbering is used, the specified page numbers will be omitted from the output document.

.F <pgno1> <pgno2> ... .

<pgno1>	Is the page number to be skipped in the output document.

This command is terminated by a single unattached "." or "#".

3.5.8 **Horizontal Spacing**

This command is used to set the horizontal spacing on the Diablo printer terminal.

.HS <spacing>

<spacing>	is the number, when multiplied by 1/60 of an inch, will determine the spacing from one character to the next. For example, a value of 6 will provide spacing of 1/10 of an inch, yielding 10 characters per inch. Note: The spacing between words is variable and beyond the user's control, but is as close to the defined spacing as possible.

The default value is 6, or 10 lines per inch.

3.5.9 **Indent**

This command is used to indent the following text within the current column. It is a shorthand way to define and invoke a table with one column, which begins the specified number of spaces within the current column and extends to the end. For example, **.I 5** is equivalent to invoking the table defined by **.DT 1 5 0 . .** The indenting is terminated when a **.TE** command is encountered. This command follows the same rules used in invoking tables.

.I <indent-number>

<indent-number> is the number of spaces to indent the following text

3.5.10 **Literal**

This command defines one or more lines of text which will be written exactly as entered without formatting. Commands embedded within these lines are ignored and treated as text. Table and column definitions are suspended, with each output line beginning in position 1. Literals are the only case where multiple blanks, tab characters and form feed characters can be explicitly included in the output document.

The command is terminated when a "." or "#" is encountered in the first character position of a line. Any other text on this terminating line is ignored.

.L
...
...
...
<text lines>
...
...
...
.

<text lines> are the lines which are sent directly as entered to the output terminal.

3.5.11 **New Line**

This command causes a new line to be started in the current column. Note that an automatic **.N** command is issued whenever any other command is encountered (e.g. #T, #B, etc.).

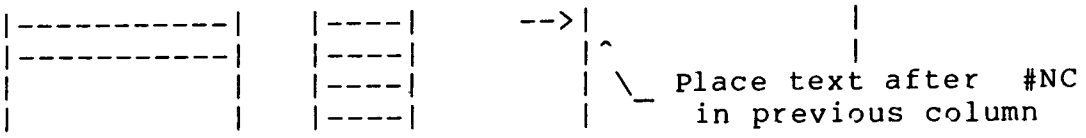
.n

3.5.12 **New Column**

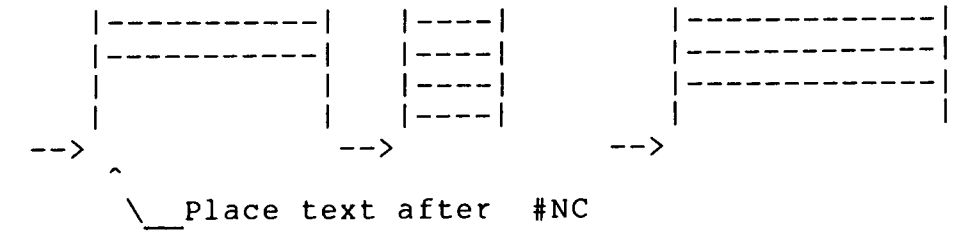
This command causes the current column to end and the next column to be started. If the current column is the last column in the table, the first column of the table is started.

.nc

If the current column contains multiple lines, **.NC** will cause the text to be placed in the top line of the new column. This is shown in the following diagram:



Advancing to the first column will cause each column's line pointer to be set to the line following the longest column.



3.5.13 **New Page**

This command causes a new page to be started.

.NP

3.5.14 **Paragraph**

This command causes a new line to be started within the current column, and five spaces to be inserted at the beginning of the line.

.P

3.5.15 **Page**

This command defines the top and bottom margins of a page.

.PAGE <top-line> <bottom-line>

<top-line>	is the number of lines skipped from the top of the page before outputting text.
<bottom-line>	specifies the last line on the page where text will be outputted.

Line number one is the first line of the page. When a page is full RPF will automatically skip to the next page. See Section (3.6.3) for a discussion of page advancing and printer form feed.

3.5.16 **Pause**

This command causes RPF to pause and wait for a single character to be input from the output terminal. Any character is acceptable, and will cause the printing to resume. The entered character will not be printed. The purpose of the pause is to allow the operator to adjust the form or change the paper before printing will continue.

The **.PAUSE** command is ignored if the PA switch is not selected when RPF is invoked. See section 3.6.4 for details on the PA switch.

.Pause

3.5.17 **Right Justify**

This command causes the text within the current column to be right justified. Right justification is most useful for aligning a column of numbers. The default for any column is left justification.

.R

This command sets a switch which indicates that all text placed in this column will be right justified. Right justification will remain in effect for this column until another **.R** command is issued or the table is terminated.

3.5.18 **Skip**

This command causes the specified number of lines to be skipped. This command should be specified while positioned in the first column of a line; skipping will effect all columns across the line.

.S <no. of lines>

<no. of lines> is the number of lines to be skipped

Following a **.S** , text output will resume at the beginning of the current column (automatic new line - **.N**).

3.5.19 **Space**

This command sets the spacing in the current column. The default is single spacing.

.SP <spacing>

<spacing>	is a number indicating the spacing for the current column. For example, **.SP 3** will cause the spacing to be set to triple spacing.

The spacing value will remain in effect until another **.SP** command is issued When a table is invoked, all the columns in that table will take on the spacing value of the column in which it was invoked. When a table is terminated, a column's spacing value will revert back to the value when the table was invoked.

Each column within a table may have a different spacing value.

3.5.20 **Start Page Numbering**

This command defines page numbering. Parameters specify the starting page number, type of numbering, and placement of the number on the page.

.SPN <type> <pos> <skip-lines> <start-number> [<sect-number>]

<type>	is the number 1, 2, or 3 indicating the type of numbering:

1- Section Page Numbering:
The page number is printed as m-n, where "m" is the section number and "n" is the page number.

2 - Letter Page Numbering:
The page number is printed as -n-, where "n" is the page number. If the page number is one, the page number is not printed.

3 - Period Page Numbering:
The page number is printed as n. , where "n" is the page number.

<pos> is the starting character position where the page number will be printed.

Note: Page numbering is always printed on the first printable line on a page. See section 3.2.15, #PAGE command for defining page margins.

<skip-lines> is the number of lines to skip after printing the page number.

<start-number> is the number of the first output page.

<sect-number> is the section number "m", if type 1 page numbering has been selected. Invalid for the other numbering types.

For example:

.cen .SPN 1 70 4 1 2 .

will cause Section Page Numbering to be used, beginning at page 2-1 of section 2 . The page number will be printed on the first line, beginning in position 70 , and 4 lines will be skipped before text will be printed.

3.5.21 **Table**

This command invokes the specified table within the current column. The column boundaries defined in the invoked table are interpreted relative to the start of the current column.

Text following the **.T** command is placed in the first of the invoked table. Placement of text in other columns is accomplished by using the **.nc** command. A table is terminated with a **.TE** command.

.T <table id>

<table id>	is the table number specified when the table was defined (See **.DT** command, Section 3.5.7).

Refer to Section 3.3 for a detailed discussion of Tables and Columns.

3.5.22 **Table End**

This command indicates the end of a table. When a table is ended, the column in which the table was invoked is in force, and text output will continue at the beginning of that column.

.TE

3.5.23 **Underline**

This command causes the specified text to be underlined.

.ul <text>

<text>	is the text to be underlined.

The underlined text is terminated by the first unattached "." or "#".

3.5.24 **Vertical Spacing**

This command is used to set the vertical spacing on the Diablo printer terminal.

.VS <spacing>

<spacing>	is the number, when multiplied by 1/48 of an inch, will determine the spacing from one line to the next. For example, a value of 8 will provide spacing of 1/6 of an inch, yielding 6 lines per inch.

The default value is 8, or 6 lines per inch.

3.6 **Executing RPF**

The Report Formatter (RPF) is invoked from the user's terminal. output can be directed back to the invoking terminal, or to any other terminal or line printer device including the system spool printer

To execute RPF enter the following command:

RSX-11M,VMS

RPF <output-dev>[/SW.../SW]=<input-file>

UNIX

RPF <input-file> <output-dev> [-SW .. -SW]

<input -file> is the name of the input text file. For RSX-11M and VMS, if the file name extension is not specified, the format <input-file>.RPF is assumed.

<output -dev> is the name of the output device. For RSX-11M and VMS, valid names are TT4:, TI:, LP:, or SPOOL for the spool printer.

For UNIX, valid device names are /dev/ttye, /dev/lp, /di etc.

One or more switches (SW) can be specified to control the execution of RPF. The order of specification is not significant, and switches can be expressed in upper or lower case characters. The following sections describe each switch in detail.

3.6.1 **All Bold -- AB**

This switch will cause the entire output document to be printed in bold face. Bold face printing is supported only on the DIABLO terminal. For other terminal types the switch is ignored. The BF (Bold Face) switch must also be specified.

3.6.2 **Bold Face -- BF**

This switch will cause all underlined text to be printed in bold face. Bold face printing is supported only on the DIABLO terminal. For other terminal types the switch is ignored.

3.6.3 **Form Feed -- FF**

This switch will cause a form feed character to be sent to the terminal prior to printing a page. If not used, the appropriate number of blank lines are printed to position the top of the next page. RPF assumes a 66 line page when inserting blank lines. Forms of other lengths requires the use of FF for positioning.

The switch is ignored for terminals which do not support a form feed character.

3.6.4 **Pause -- PA**

This switch will cause RPF to suspend printing at the end of each page and when a **.PAUSE** command is encountered. Printing is resumed after any input character is entered. The entered character is not printed.

Pause allows documents to be printed on cut forms, and for special setup during the printing process. The command will be ignored if PA was not specified.

3.6.5 **Page -- PG:n[:m]**

This switch will cause a range of pages to be printed, from n to m. For example:

```
PG:5    --- Print the fifth through the last page.
PG:5:7  --- Print pages 5, 6, and 7.
PG:5:5  --- Print only page 5.
```

3.6.6 **Upper Case -- UC**

This switch will cause all alphabetic characters to be printed in upper case.

3.6.7 **VT100 -- VT**

This switch will cause all underlined text to be output in reverse video. If not specified for a video display terminal, underline text will be displayed then overwritten with underline characters.

U S I N G T H E

R E P O R T W R I T E R U T I L I T Y

4.1 **Overview**

This section will demonstrate by example how Report Writer statements are combined with RPF commands and user text to generate a complete report program. The approach will be to present the output report along with the generating program. The discussion will center around the report structure and the Report Writer statements. A working knowledge of RPF is assumed, and individual commands will not be discussed. It is recommended that the reader refer to sections 2 and 3 to obtain the necessary RPF background.

4.2 **Example 1 - Tabular Report**

Figure 4.1A is a listing of a simple tablular report. The data for this report was obtained from the 'EMP' table within the ORACLE demonstration database ('PERSONNEL'). The report program which generated this report is listed in Figure 4.1B. The line numbers and titles were added after execution.

To aid the reader in distinguishing between RPF commands and RPT statements, all RPF commands will use the '#' instead of the '.'. Either symbol is allowed, but this convention simplifies the reading and debugging of report programs. Other conventions have also been adopted which simplify the report writing process.

T E C H N O L O G Y S Y S T E M S , I N C

PERSONNEL REPORT FOR SEPTEMBER, 1980

EMPNO	NAME	JOB	SALARY	COMMISSION	DNO
7369	SMITH	CLERK	$800.00		20
7499	ALLEN	SALESMAN	$1,600.00	$300.00	30
7521	WARD	SALESMAN	$1,250.00	$500.00	30
7566	JONES	MANAGER	$2,975.00		20
7654	MARTIN	SALESMAN	$1,250.00	$1,400.00	30
7698	BLAKE	MANAGER	$2,850.00		30
7782	CLARK	MANAGER	$2,450.00		10
7788	SCOTT	ANALYST	$3,000.00		20
7839	OATES	PRESIDENT	$5,000.00		10
7844	TURNER	SALESMAN	$1,500.00		30
7876	ADAMS	CLERK	$1,100.00		20
7900	JAMES	CLERK	$950.00		30
7902	FORD	ANALYST	$3,000.00		20
7934	MILLER	CLERK	$1,300.00		10

END OF REPORT

"Example 1 - Tabular Report "

Figure 4.1-A

```
 1: .REM ************************************************************
 2: .REM *******  SAMPLE REPORT 1 -----  SIMPLE TABULAR REPORT  ********
 3: .REM ************************************************************
 4: .REM
 5: .REM *******   Define RPF Tables   ---  Print   Title        ********
 6: .REM
 7: .REM                    " T a b l e    1 "
 8: .REM +-----------------------------------------------------------+
 9: .REM 4                                                          76
10: .REM                    " T a b l e    2 "
11: .REM +---+  +---------+  +---------+  +----------+  +--------+ +---+
12: .REM 1   5 10        20  25        35  40         50  55       65 67  0
13: .REM
14: #dt 1 4 76 #
15: #dt 2 1 5 10 20 25 35 40 50 55 65 67 0  #
16: #t 1
17: #page 6 58
18: #s 3
19: #cul T E C H N O L O G Y \ \ S Y S T E M S , \ \  I N C #
20: #s 3
21: #cul PERSONNEL REPORT FOR SEPTEMBER, 1980 #
22: #s 3
23: .REM
24: .REM *******     Declare    Program    Variables              ********
25: .REM
26: .DATABASE personnel
27: .REM *******  Print all positions of empno - ie. 0682        ********
28: .DECLARE empno 0999
29: .DECLARE ename al0
30: .DECLARE job al0
31: .DECLARE sal $99,999.99
32: .REM *******  Print 'comm' as blank if value is zero  **************
33: .DECLARE comm $B99,999.99
34: .DECLARE deptno 99
35: .REM
36: .REM *********          Define    SELECT      Macro           ********
37: .REM
38: .DEFINE selemp
39:         SELECT empno,ename,job,sal,comm,deptno
40:         INTO empno,ename,job,sal,comm,deptno
41:         FROM emp
42: ..
```

"Example 1 - Tabular Report"

Figure 4.1B - Part 1 of 2

```
43: .REM
44: .REM ********          Define    BODY       Macro            ********
45: .REM
46: .DEFINE body
47: .REM ****** Print each column variable  -- Advance to next column *****
48:          .PRINT empno
49:          #nc
50:          .PRINT ename
51:          #nc
52:          .PRINT job
53:          #nc
54:          .PRINT sal
55:          #nc
56:          .PRINT comm
57:          #nc
58:         .PRINT deptno
59:          #nc
60: ..
61: .REM
62: .REM ********          Define      HEAD      Macro              ******
63: .REM
64: .DEFINE head
65:          .REM  **** Invoke "Table 2" - Report Column Layout *********
66:          #t 2
67:          .REM  ***********   Print Column Headings **********
68:          #r EMPNO #n ----- #nc
69:          NAME #n ---- #nc
70:          JOB #n --- #nc
71:          #r SALARY #n ------ #nc
72:          #r COMMISSION #n ---------- #nc
73:          #r DNO #n --- #nc
74:          #b
75:          .REM  ********* Execute body macro to print first row *******
76:          .body
77: ..
78: .REM
79: .REM  **********   Define      Foot           Macro       *******
80: .REM
81: .DEFINE foot
82:          #te
83:          #s 4
84:          #cul   END OF REPORT    #
85: ..
86: .REM
87: .REM  ******** Procedure Section  --- Generate Report    ********
88: .REM
89: .REPORT selemp  body head foot
90: #te
91: .STOP
```

"Example 1 - Tabular Report"

Figure 4.1B - Part 2 of 2

The reader should be reminded of the basic rules which apply to RPT:

- RPT's main function is to copy RPF commands and user text, as encountered, to the interim file which will subsequently be processed by RPF. Database information can be included within this output file using the PRINT command.

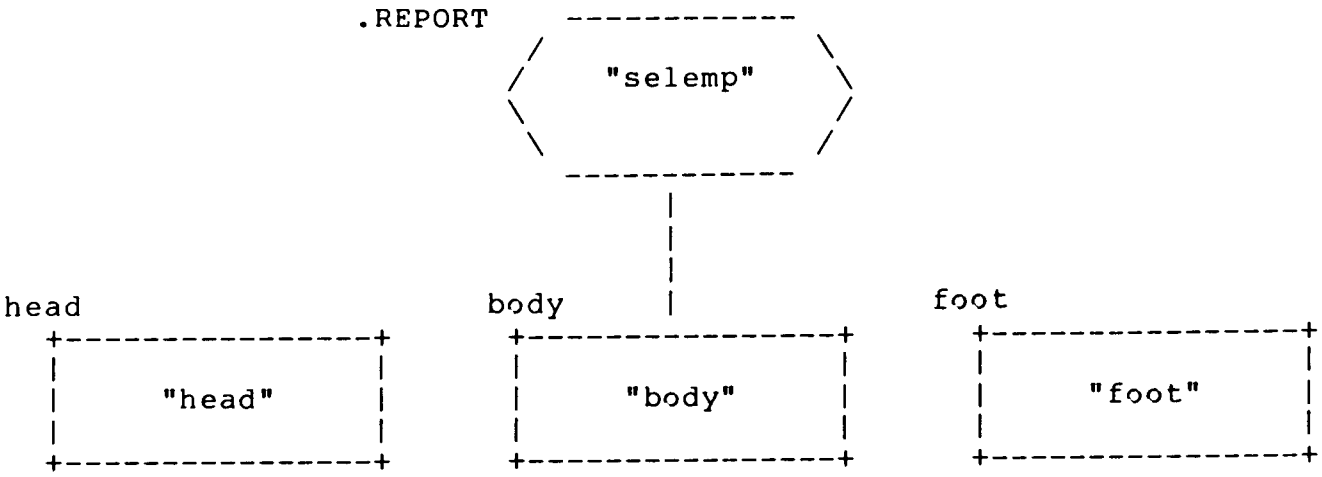
- Program variables and macros (Procedural and SELECT) must be defined prior to being referenced. Macro definitions are stored as encountered, and not executed until explicitly or implicitly requested.

- RPF commands and user text may be defined anywhere within a program.

The basic structure for this report is depicted in figure 4.1C. The **.REPORT** statement drives the retrieval of data and causes the appropriate head, body, and foot macros to be executed. The SQL query defined within the 'selemp' macro (lines 38 through 42 of figure 4.1B) selects all the columns from all the rows in the 'EMP' table. The procedural macros used to output and format this data will be discussed later.

The RPF table definitions used to format the output are pictorially displayed on lines 8 through 12. The data fields returned for each row will be formatted in the six columns defined by Table '2'. Lines 14 through 22 contain the RPF commands to define these tables and the report titles. These commands could also have been included after the program DEFINE and DECLARE statements.

Line 26 identifies the single database which will be referenced within this report. Since the 'personnel' database is secure, a valid 'userid/password' must be specified when the report is executed. Following the 'DATABASE' statement the program variables are declared. The declarative statements may be specified in any order, providing the definition precedes its use.



.REPORT selemp body head foot

"Example 1 - Tabular Report"

Figure 4.1C

By specifying a '0' in the format definition of 'empno', all leading zero digits will be printed. Remember that the declared format defines the maximum value of a variable in addition to the printing format. For example, 'sal' (line 31) can hold a maximum value of 99,999.99; 100,000.00 will overflow this variable. Overflowed variables will be print with a '#' in all digit positions. The 'B' on line 33 will cause zero commission value to be printed as blanks. Any variable which has the 'NULL' value will be printed as blanks.

The 'selemp' SELECT macro defines the SQL query which drives this report. The INTO statement identifies the program variables which will receive the column values defined in the select list. There is a one-to-one correspondence between each column returned and its associated program variable. Column and variable names do not have to be the same.

Lines 43 through 88 define the head, body, and foot procedural macros which will be implicitly executed as a result of executing the '.REPORT' statement on line 89. The head macro called 'head' will be executed once for the first row returned from the query. Table '2' is invoked to establish the six column print layout. RPF commands and user text is included to print the column headings (lines 67 - 74). Notice the '#r' command to establish right justification for the empno, salary, commission, and department number columns. After inserting a blank line the body macro is explicitly executed (Line 76). This will cause the first row to be printed. If omitted, the first row of data would be lost.

The body macro ('body') is implicitly executed for the second through last row returned. A '.PRINT' statement is used to cause the current value of the program variable to be included in the output interim file. Figure 4.1D is a partial listing of the generated 'interim file'. The '#nc' commands causes the row information to be printed in the correct report column. Notice the database information which has been interspersed between the RPF commands.

After the last row of the query has been processed, the foot macro (Lines 81-85) is executed. In this example the foot simply terminates Table '2' and prints 'END OF REPORT'.

```
#dt 1 4 76 #
#dt 2 1 5 10 20 25 35 40 50 55 65 67 0  #
#t 1
#page 6 58
#s 3
#cul T E C H N O L O G Y \ \ S Y T E M S , \ \  I N C #
#s 3
#cul PERSONNEL REPORT FOR SEPTEMBER, 1980 #
#s 3
#t 2
#r EMPNO #n ----- #nc
NAME #n ---- #nc
JOB #n --- #nc
#r SALARY #n ------ #nc
#r COMMISSION #n ---------- #nc
#r DNO #n --- #nc
#b
7369
#nc
SMITH
#nc
CLERK
#nc
$800.00
#nc

#nc
20
#nc
7499
#nc
ALLEN
#nc
SALESMAN
#nc
$1,600.00
#nc
$300.00
#nc
30
#nc
7521
#nc
WARD
#nc
SALESMAN
#nc
$1,250.00
#nc
```

" Example 1 - Tabular Report - 'Interim File' "

Figure 4.1D

Following the '.REPORT' statement in the procedure section is a '.te' to terminate Table '1' and a '.STOP' statement to terminate the report program. The 'STOP' is optional, and if omitted the program would end after the last statement. When included, a stop message is displayed on the executing terminal.

Report generation is a two step process. This report was generated using the following system commands:

Step 1 - RPT Execution:

```
>RPT sampl.rpt sampl.rpf SCOTT/TIGER
```

Where 'sampl.rpt' is the input report text file, and 'sampl.rpf' is the interim file created by RPT. Since the 'PERSONNEL' database is secure, a valid userid and password must be provided. Note: The '>' is an RSX11M system prompt, and will vary by operating environment, however the command text is the same across the supported systems.

Step 2 - RPF Execution:

```
>RPF LP:=sampl
```

Where 'LP:' is the system name for the output device. The output may be directed to either a system device or another file. The input file is the file created by RPT in the previous step. Since no file extension was specified, 'rpf' was assumed, and the fully qualified name 'sampl.rpf' is used.

4.3 **Example 2 - Nested Report**

Figure 4.2A is a listing of a **nested** report. A nested report is one in which a second report is executed within the head, body, or foot macro of the primary report. In this example, the primary or 'outer' report, provides a listing of the department names and numbers for each department within the division. Then for each department selected, a second report is executed which lists the employees in that department. After listing the individual employees, the department's salary summary is reported.

D i v i s i o n P a y r o l l R e p o r t

by Department

****

DEPTNO: 010 - - DEPTNAME: ADMINISTRATION

****

EMPNO	NAME	MONTHLY SALARY	COMM	ANNUAL COMPENSATION
7782	CLARK	$3162.50		$37,950.00
7934	MILLER	$1300.00		$15,600.00
7839	OATES	$5750.00		$69,000.00
	Department Summary			
AVG = $3,404.17	MIN = $1,300.00	MAX = $5,750.00		$122,550.00

****

DEPTNO: 020 - - DEPTNAME: RESEARCH

****

EMPNO	NAME	MONTHLY SALARY	COMM	ANNUAL COMPENSATION
7369	SMITH	$800.00		$9,600.00
7566	JONES	$3421.25		$41,055.00
7876	ADAMS	$1100.00		$13,200.00
7788	SCOTT	$3000.00		$36,000.00
7902	FORD	$3000.00		$36,000.00
7955	WILSON			
7956	JAKES	$1000.00		$12,000.00
	Department Summary			

" Example 2 - Nested Report "

Figure 4.2A - Part 1 of 3

AVG = $2,053.54 MIN = $800.00 MAX = $3,421.25 $147,855.00

****

DEPTNO: 030 - - DEPTNAME: SALES

****

EMPNO	NAME	MONTHLY SALARY	COMM	ANNUAL COMPENSATION
7499	ALLEN	$1600.00	$300.00	$19,500.00
7521	WARD	$1437.50	$500.00	$17,750.00
7698	BLAKE	$3277.50		$39,330.00
7654	MARTIN	$1437.50	$1400.00	$18,650.00
7844	TURNER	$1500.00	$0.00	$18,000.00
7900	JAMES	$950.00		$11,400.00
7989	CARTER	$1500.00	$0.00	$18,000.00

Department Summary ----------

AVG = $1,671.79 MIN = $950.00 MAX = $3,277.50 $142,630.00

****

DEPTNO: 040 - - DEPTNAME: OPERATIONS

****

"Example 2 - Nested Report"

Figure 4.2A - Part 2 of 3

D i v i s i o n T o t a l s

MINIMUM SALARY	MAXIMUM SALARY	AVERAGE COMPENSATION	TOTAL COMPENSATION
$800.00	$5,750.00	$25,814.69	$413,035.00

E n d O f R e p o r t

"Example 2 - Nested Report"

Figure 4.2A - Part 3 of 3

A block diagram of this example is provided in Figure 4.2C. The 'inner' report which lists the employee information is explicitly executed within the body macro ("deptbody") of the 'outer' department report. Associated with this 'inner' report are head, body, and foot macros labeled "emphead", "empbody", and "empfoot". Notice that within the "empfoot" macro, two other macros are explicitly executed. The "deptsum" select macro uses a SQL query to compute the summary salary information. The procedural macro "compsum" adds the department summary to the division totals. The division summary is generated by the "deptfoot" macro after all departments have been processed.

Figure 4.2B is the program listing for this example. The line numbers and titles at the bottom of the page were added after the report was executed. The data for this report was obtained from the 'DEPT' and 'EMP' tables within the 'PERSONNEL' database (Line 16).

This report contains three SELECT macros. "seldept" (Lines 42-46) drives the outer report berequesting each department's name and number from the 'DEPT' table. The absence of the "WHERE" clause will cause all the rows in the table to be returned. The "selemp" macro will drive the inner report returning all the employees in a particular department. The substitution variable '&deptno' will cause the current value of the program variable 'deptno' to be substituted into the "WHERE" clause.

The fifth value returned in the SELECT list(Line 51) is the computed annual compensation. The null function was required for the 'comm' column because all employees other than salesman have a null value for commission. If 'nvl' was omitted the annual compensation for non-salesman would be assigned the null value and printed as blanks.

The "deptsum" SELECT macro is explicitly executed in the foot macro of the inner report. After all the employees in a department have been listed this query uses the SQL built-in functions to compute the count, minimum, maximum, average, and sum of the salaries for the reported department. This technique for computing summary information requires a second pass of the data.

```
001: .REM ************************************************************  **
002: .REM ********  SAMPLE   REPORT  2   ---  N E S T E D   R E P O R T  **  **
003: .REM ************************************************************
004: .REM
005: .REM    This sample demonstrates the capability to nest a report within the
006: .REM    head, body, or foot of another report. In this example the "payroll
007: .REM    status for each department within a division will be reported. The
008: .REM    compensation for each employee is listed ,                 followed
009: .REM    by the department's summary information. At the end of the report
010: .REM    the entire division's summary information is reported.
011: .REM
012: .REM ************************************************************
013: .REM
014: .REM            D e c l a r e              V a r i a b l e s
015: .REM
016: .DATABASE personnel
017: .DECLARE  deptno 000
018: .DECLARE  dname a15
019: .DECLARE  empno 0999
020: .DECLARE  ename a15
021: .DECLARE  monsal $9999.99
022: .DECLARE  comm   $9999.99
023: .DECLARE  annsal $99,999.99
024: .DECLARE  deptsum $999,999.99
025: .DECLARE  deptmin $9,999.99
026: .DECLARE  deptmax $9,999.99
027: .DECLARE  deptavg $9,999.99
028: .DECLARE  deptcnt 9999
029: .DECLARE  divsum  $999,999.99
030: .DECLARE  divavg  $99,999.99
031: .DECLARE  divmin  $9,999.99
032: .DECLARE  divmax  $9,999.99
033: .DECLARE  empcnt  9999
034: .REM  ************************************************************
035: .REM
036: .REM          D e f i n e    S E L E C T    M a c r o s
037: .REM
038: .REM  ************************************************************
039: .REM
040: .REM          Select   Department   Information
041: .REM
042: .DEFINE seldept
043:         SELECT deptno,dname
044:         INTO   deptno,dname
045:         FROM   dept
046: ..
047: .REM
048: .REM          Select Employee Data Within a Department
049: .REM
050: .DEFINE selemp
```

*** Example 2 - Nexted Report Program Listing ***

Figure 4.2B - Part 1 of 5

```
051:          SELECT empno,ename,sal,comm,(sal*12 + nvl(comm,0))
052:          INTO   empno,ename,monsal,comm,annsal
053:          FROM   emp
054:          WHERE  deptno = &deptno
055: ..
056: .REM
057: .REM           Select Department Summary Information
058: .REM
059: .DEFINE deptsum
060:         SELECT count(sal),min(sal),max(sal),avg(sal),sum(sal*12+nvl(comm,0)
061:         INTO   deptcnt,deptmin,deptmax,deptavg,deptsum
062:         FROM   emp
063:         WHERE  deptno= &deptno
064: ..
065: .REM    ***************************************************************
066: .REM
067: .REM         D e f i n e    P r o c e d u r a l    M a c r o s
068: .REM
069: .REM    ***************************************************************
070: .REM
071: .REM         D i v i s i o n    R e p o r t    " H e a d "
072: .REM
073: .DEFINE depthead
074:           .REM   This head  macro just processes the first row of
075:           .REM   Department report. It is required only because a foot
076:           .REM   is specified on the REPORT statement.
077:           .deptbody
078: ..
079: .REM
080: .REM      D E P A R T M E N T   R e p o r t    " B o d y "
081: .REM
082: .DEFINE deptbody
083:           .REM
084:           .REM    E x e c u t e    E m p l o y e e       R e p o r t
085:           .REM
086:           .REM      P r i n t   H e a d i n g   F o r   D e p a r t m e n t
087:           .REM
088:          #s 2
089:          #cul  ************************************************** #
090:          #b
091:          #cen
092:          DEPTNO:
093:          .PRINT deptno
094:          - - DEPTNAME:
095:          .PRINT dname
096:          #
097:          #b
098:          #cul  ************************************************** #
099:          #s 2
100:          .REM      Report Each Employee within the current Department
```

" Example 2 - Nexted Report Program Listing "

Figure 4.2B - Part 2 of 5

```
101:          .REM
102:          .REPORT selemp empbody  emphead  empfoot
103: ..
104: .REM
105: .REM       D E P A R T M E N T    R e p o r t    " F o o t "
106: .REM
107: .DEFINE deptfoot
108:         .REM   Compute Division summary information
109:         .REM
110:         .REM      Compute "Average Monthly Salary"
111:         .REM
112:         .DIV divavg divsum empcnt
113:         .REM
114:         .REM      Output: Maximum and Minimum Salary
115:         .REM              Average and Total Yearly Compensation
116:         .REM
117:         #np
118:         #cul D i v i s i o n  \  T o t a l s #
119:         #s 1
120:         #t 4
121:         #r #nc #r #nc #r #nc #r #nc
122:         MINIMUM #nc MAXIMUM #nc AVERAGE #nc  TOTAL   #nc
123:         SALARY #nc SALARY #nc COMPENSATION #nc COMPENSATION #nc
124:         ------- #nc --------     #nc ------------ #nc ------------ #nc
125:         .PRINT divmin
126:         #nc
127:         .PRINT divmax
128:         #nc
129:         .PRINT divavg
130:         #nc
131:         .PRINT divsum
132:         #te
133:         #s 2
134:         #cul E n d  \  O f \ R e p o r t   #
135:         #te
136: ..
137: .REM    ***************************************************************************
138: .REM
139: .REM             E M P L O Y E E        "H e a d"    M a c r o
140: .REM
141: .DEFINE  emphead
142:           #t 2
143:           #s 1
144:           EMPNO #nc #cen NAME # #nc MONTHLY #nc COMM #nc ANNUAL #nc
145:           #nc #nc SALARY #nc #nc COMPENSATION #nc
146:           ----- #nc --------------- #nc -------- #nc -------- #nc
147:           ------------ #nc
148:           .REM   Set-up Right/Left Justification  Switches
149:           .REM        Insert One Blank Line
150:           #nc #nc #r #nc #r #nc #r #nc
```

" Example 2 - Nexted Report Program Listing "

Figure 4.2B - Part 3 of 5

```
151:          .REM
  2:          .REM    Execute 'Body' Macro to Process First Row
  3:          .empbody
154: ..
155: .REM
156: .REM         E M P L O Y E E        R e p o r t    " B o d y "
157: .REM
158: .DEFINE  empbody
159:          .PRINT empno
160:          #nc
161:          .PRINT ename
162:          #nc
163:          .PRINT monsal
164:          #nc
165:          .PRINT comm
166:          #nc
167:          .PRINT annsal
168:          #nc
169: ..
170: .REM
171: .REM         E M P L O Y E E        R e p o r t    " F o o t "
172: .REM
173: .DEFINE  empfoot
174:          .REM
175:          .REM   Compute Department Summary Information
176:          .REM
177:          .EXECUTE deptsum
  8:          #te
179:          #t 3
180:          #s 2
181:          #cul Department Summary # #nc ------------ #nc
182:          #s 1
183:          \ AVG =
184:          .PRINT deptavg
185:          \ MIN =
186:          .PRINT deptmin
187:          \ MAX =
188:          .PRINT deptmax
189:          #nc #r
190:          .PRINT deptsum
191:          #te
192:         .REM   Execute macro to compute division totals
193:          .compsum
194: ..
195: .REM  ***************************************************************
196: .REM
197: .REM        C o m p u t e     D i v i s i o n    T o t a l s
198: .REM
199: .DEFINE compsum
200:         .ADD empcnt empcnt deptcnt
```

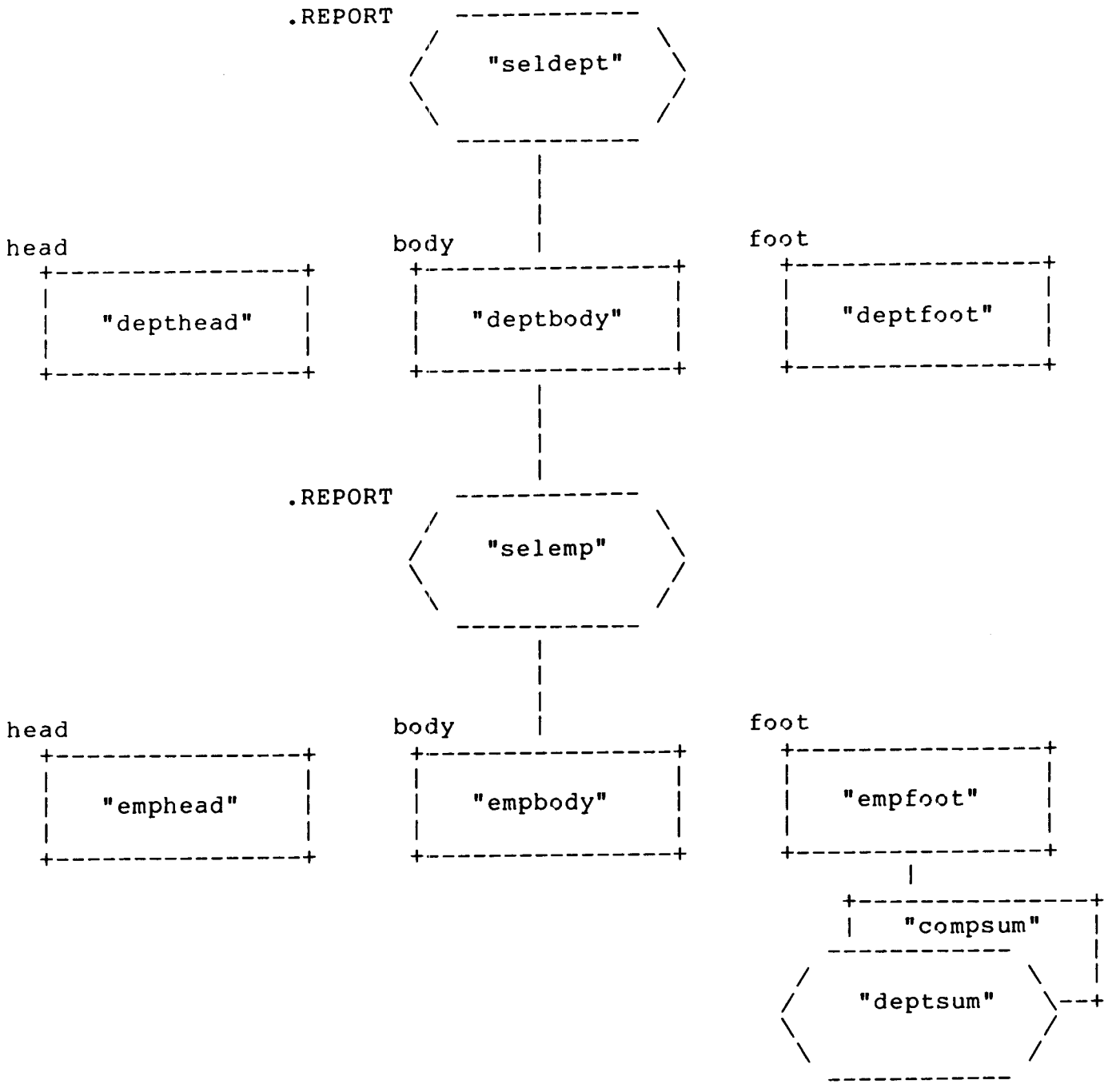
" Example 2 - Nexted Report Program Listing "

Figure 4.2B - Part 4 of 5

```
201:          .ADD divsum divsum deptsum
202:          .IF "&deptmax <= &divmax" THEN compl0
203:          .REM     S e t     N e w    M a x i m u m
204:          .EQUAL divmax deptmax
205:          .&compl0
206:          .IF "&deptmin >= &divmin" THEN comp20
207:          .REM     S e t     N e w    M i n i m u m
208:          .EQUAL divmin deptmin
209:          .&comp20
210:   ..
211: .REM
212: .REM  ***************************************************************
213: .REM
214: .REM
215: .REM            R e p o r t :    M a i n     S e c t i o n
216: .REM
217: .REM
218: .REM            D e f i n e     " R P F "   C o m m a n d s
219: .REM
220: #dt 1 10 75 #
221: #dt 2 5 9 13 27 31 38 42 49 55 0 #
222: #dt 3 1 51 55 0 #
223: #dt 4 1 9 12 20 23 37 47 0 #
224: #page 6 58
225: #spn 2 42 2 1
226: #sp 1
227: .REM
228: .REM            E x e c u t e     D i v i s i o n   R e p o r t
229: .REM
230: .REM           Print    Report    Title
231: #t 1
232: #s 4
233: #cul D i v i s i o n  \  P a y r o l l  \  R e p o r t #
234: #s 2
235: #cul by Department #
236: .REM      Initialize Division Minimum and Maximum Values
237: .REM         Highest value for Minimum; Lowest Value for Maximum
238: .SET divmin 9999.99
239: .SET divmax 0
240: .REM
241: .REPORT  seldept  deptbody depthead deptfoot
242: .STOP
```

" Example 2 - Nexted Report Program Listing "

Figure 4.2B - Part 5 of 5



.REPORT seldept deptbody depthead deptfoot

"Example 2 - Nested Report"

Figure 4.2C

An alternative approach would be to accumulate the values in program variables as each row is processed, then compute the summary data using the RPT mathematical functions. This technique was used to generate the "Division Totals" shown on the third page of the report (Figure 4.2A). However, instead of computing for each employee row, division totals were computed using each department's summary data. This choice was made to improve performance by reducing the frequency of execution of the "compsum" macro. In most cases the difference would not be significant, however to analyze the maximum and minimum values of 'salary' two "IF" statements are required. As utlined in section 4.4.4.3.1 "IF Statement Guidelines", "IF" statements processing is fairly time consuming and therefore the frequency of execution should be minimized.

The procedural macro "depthead" was included only because a "foot" macro was required for the outer report. Because the procedural macros on the .REPORT statement (Line 241) are positionally defined a 'head' macro name must precede the 'foot' name. The explicit execution of the body macro "deptbody" causes the first row to be processed.

The body of the "Department Report" lists the department's name and number, then executes the inner employee report (Line 102). In the 'foot' of the department report the average compensation is computed (Line 112). The headings for the division totals are printed in Lines 121 to 124. Line 121 does not output any text, but has the effect of skipping a single line. The '#r' commands executed in each column forces right justification.

The head macro of the "Employee Report" ("emphead") prints the headings for this inner report and sets the justification for the output columns (Line 150). Again, the body must be executed to cause the first row to be processed. The 'body' outputs each column of data retreived, and the 'foot' computes and lists the department summary information.

"compsum" uses the 'ADD' statement to accumulate the sum of the compensation and the total number of employees. The "IF" statements compare current department minimum and maximum salaries with those values for the division. Since blanks were included within the expressions they must be enclosed within double quotation marks ("). If a new max or min values is **not** established then the expressions are true and control is passed to the macro label specified after the "THEN" clause. Since no "ELSE" statement was specified, a false condition causes the next instruction to be executed. The "EQUAL" statements cause the division values to be set equal to the corresponding department values.

Line 211 is the end of the macro definitions and the beginning of the Procedure section of this report program. As defined on lines 220 through 223, four tables were required for this report. Tables 2, 3, and 4 are invoked within table 1 at various places in the report. The '#spn' command (Line 225) causes the pages of the report to be numbered. Type 2 format (Letter Page Numbering) was selected to be printed beginning in print position '42'. Two lines will be skipped after printing the page number. Numbering will begin with '1', however with type 2 the number is not printed on page one.

The "SET" statements (Lines 238,239) initialize the values of the division minimum and maximum salaries. The "REPORT" statement executes the outer report which drives the entire program.

4.4 **Example 3 - Sample Letter**

This example demonstrates the flexibility of the report writer to construct a computer generated letter. Fixed text is combined with database information to form the completed memo. The power of SQL allows the desired data to be obtained from the database, and the RPF commands and RPT statements permits the data to be interleaved within the letter format.

Figure 4.3A shows the sample which is composed of data from the 'EMP' and 'DEPT' table. The report was executed to generate multiple copies of this memo, each addressed to a different manager. The manager's name was obtained from the 'EMP' table, and the manager's department number, name and location from the 'DEPT' table.

The current 1981 and proposed 1982 salary figures are included within the body of the memo. This information is obtained by computing the sum of the salaries of each employee in the 'EMP' table who is in the addressee's department. This information will be different for each manager's copy of the memo. Although this memo is very simple, it demonstrates the report writer's ability to combine and include data from various tables in the database into a single letter.

Figure 4.3C provides a block diagram of the report program. A single '.REPORT' statement with only a body macro was used. The SELECT statement which drives the report generation joins the 'EMP' table to the 'DEPT' table to retrieve the adressee data. Within the body a second SELECT is explicitly executed to compute the current and proposed salary data for the addressee's department.

A program listing of the report is provided in Figure 4.3B. The line numbers and titles were added after execution. In the 'DECLARE' variables section, the variable 'curdate' has a format of 'date'. This means that the value of curdate is in the internal Julian day number format. If this variable was assigned its value from a column in a SELECT list, that column value must also be in Julian day format. Presently, only IAF supports the creation of a column value in this format. In this program 'curdate' is used to output the current date at the time of execution. Assigning the variable to the current date is accomplished by the .SET statement on line 22. The literal '$$DATE$$' instructs RPT to obtain the date from the system, convert it into Julian format, and then store it into the variable referenced on the SET statement.

The first of the two SELECT macros, 'seladdr', joins the 'EMP' to the 'DEPT' table to acquire the addressee information. The second predicate in the WHERE clause restricts the list of addressees to department managers. The 'selsum' macro uses the SQL arithmetic capabilities to compute both the current sum and the sum incremented by 10% for employees in the manager's department (Line 44).

Date : 03/25/81

To: BLAKE

Department : 30 - SALES

Location : PARIS

From : Bill James
Director of Personnel

Subj : 1982 Employee Compensation Plan

Its budget time again! To aid you in completing the salary portion of your budget I have computed your department's current salary expenses. For planning purposes we are presently estimating an across the board increase of 10% in the 1982 salary pool.

Your department's 1981 figure is: $140,430.00

Estimated 1982 figure is: $154,473.00

In completing your salary plan you should break this total down by individual employees. Please let me know if you have any questions on this or related issues.

Bill

Example 3 - Sample Letter

Figure 4.3A

```
001: .REM ************************************************************************
002: .REM **********  SAMPLE REPORT 3  -----  GENERATED LETTER    **********
003: .REM ************************************************************************
004: .REM
005: .REM   This sample demonstrates the capability to create a computer
006: .REM   generated letter or memorandum. Information from the database is
007: .REM   combined with the predefined text to form the completed letter.
008: .REM   A REPORT statement is used to drive the generation of multiple
009: .REM   letters, each with a different addressee and variable data.
010: .REM
011: .REM ************************************************************************
012: .REM
013: .REM              D E C L A R E     V A R I A B L E S
014: .REM
015: .DATABASE personnel
016: .DECLARE addressee a10
017: .DECLARE deptno 999
018: .DECLARE dname a15
019: .DECLARE location a15
020: .DECLARE curdate date
021: .REM   Set the variable 'curdate' equal to today's date
022: .SET curdate $$DATE$$
023: .DECLARE sumsal  $999,999.99
024: .DECLARE sumsal+ $999,999.99
025: .REM   ************************************************************
026: .REM
027: .REM          D e f i n e      S E L E C T    Macros
028: .REM
029: .REM   ************************************************************
030: .REM
031: .REM          Select   Managers   and their Departments
032: .REM
033: .DEFINE  seladdr
034:          SELECT emp.ename,dept.deptno,dept.dname,dept.loc
035:          INTO   addressee,deptno,dname,location
036:          FROM   emp,dept
037:          WHERE  emp.deptno=dept.deptno
038:          AND    job = 'MANAGER'
039: ..
040: .REM
041: .REM          Select  Department's  Current Payroll
042: .REM
043: .DEFINE  selsum
044:          SELECT sum(sal*12),sum((sal*1.1)*12)
045:          INTO   sumsal,sumsal+
046:          FROM   emp
047:          WHERE  deptno = &deptno
048: ..
049: .REM    ************************************************************
050: .REM
```

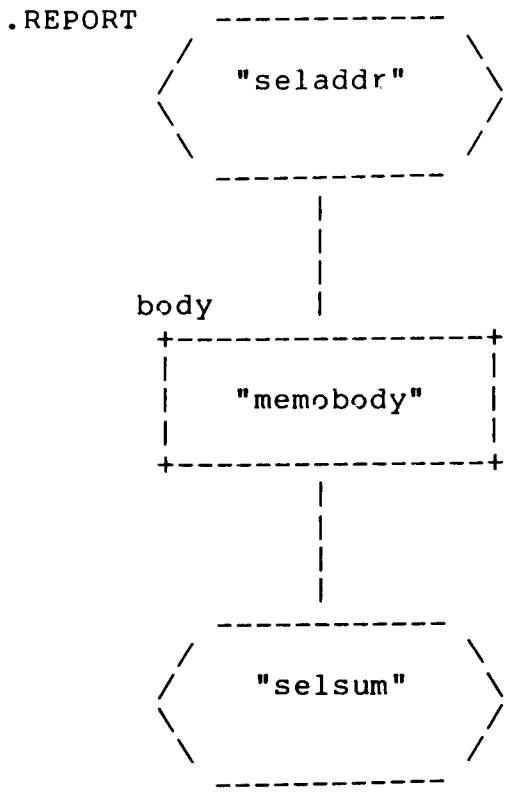
" Example 3 - Sample Letter Program Listing "

Figure 4.3B - Part 1 of 2

```
051: .REM            D e f i n e   P r o c e d u r a l   M a c r o s
0?   .REM
05   .REM    ****************************************************************
054: .DEFINE  memobody
055:          #np
056:          Date :
057:          .PRINT curdate
058:          #s 2
059:          To:
060:          .PRINT addressee
061:          #b Department :
062:          .PRINT deptno
063:          -
064:          .PRINT dname
065:          #b Location :
066:          .PRINT location
067:          #s 2  From : Bill James #p \ \ Director of Personnel
068:          #s 2  Subj : 1982 Employee Compensation Plan
069:          #s 2
070:          Its budget time again! To aid you in completing the salary portion
071:          of your budget I have computed your department's current salary
072:          expenses. For planning purposes we are presently estimating an
073:          across the board increase of 10% in the 1982 salary pool.
074:          #s 2
075:          .REM
076:          .REM        S E L E C T  Department's Salary Total
077:          .REM
0             .EXECUTE selsum
079:          #i 10 Your department's 1981 figure is:
080:          .PRINT sumsal
081:          #b Estimated 1982 figure is:
082:          .PRINT sumsal+
083:          #s 3
084:          #te
085:          In completing your salary plan you should break this total down by
086:          individual employees. Please let me know if you have any questions
087:          on this or related issues.
088:          #s 3
089:          Bill
090: ..
091: .REM ********************************************************************
092: .REM
093: .REM         P R O C E D U R E          S E C T I O N
094: .REM
095: .REM ********************************************************************
096: #dt 1 13 73 #
097: #t 1
098: #page 8 56
099: .REPORT seladdr memobody
100: .STOP
```

"Example 3 - Sample Letter Program Listing"

Figure 4.3B - Part 2 of 2



"Example 3 - Sample Letter "

Figure 4.3C

The .REPORT statement on line 99 drives the generation of these multiple memos. Only a body macro "memobody" is specified. Since no head macro was included, the body will be executed for the first row returned. The structure of the body is similar to the RPF examples presented in Section 3. Interspersed within the text are PRINT statements which cause the appropriate database information to be included within the memo output. The EXECUTE statement on line 78 executes the 'selsum' to compute the salary sums. The '#p' command and double back slash '\' on line 67 are used to force the author's title to be alligned under his name. Each '\' causes one additional blank to be included. The '#np' on line 55 causes each copy of the memo to be printed on a separate page.

R E P O R T - W R I T E R - U T I L I T Y

(R P T)

5.1 **Overview**

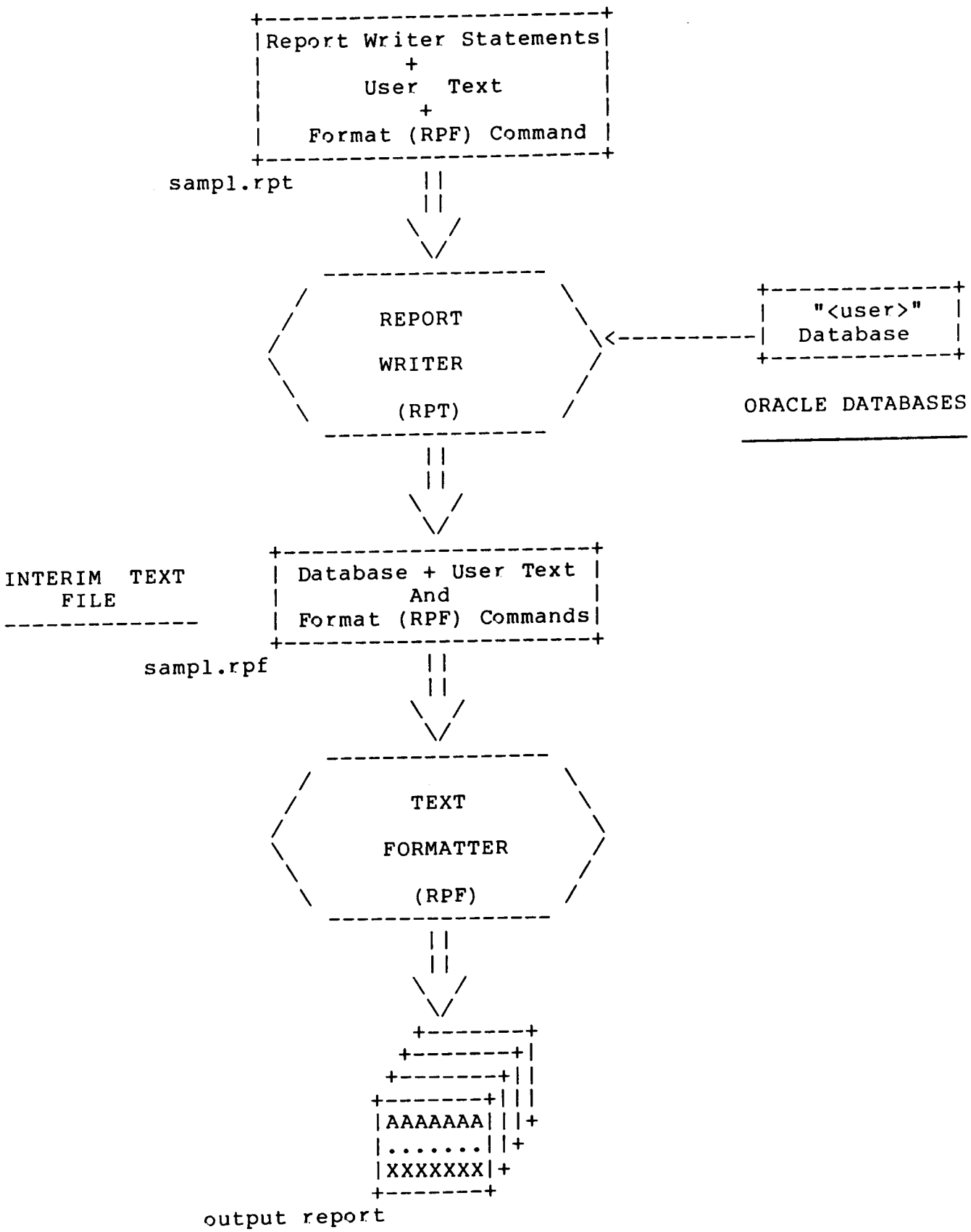
The ORACLE Report Writer Utility (RPT) interprets and executes a report generation program. A **Report Program** may consist of report writer statements, RPF commands, and user text. The use of RPF to format text is described in an earlier section, and a working knowledge will be assumed.

This section will concentrate on the statements necessary to construct a report program. The structure of a program will be described along with a detailed discussion of each programming statement.

5.2 **Report Generation Process**

Figure 5.1 is an overview of the report generation process. The report program is created using a standard text editor. This file is passed as input to the RPT utility.

The program file may contain report text and RPF commands in addition to the RPT program statements. The text and RPF commands are ignored by RPT and are copied as encountered directly to the "Interim File". On the other hand, Report Program statements are interpreted and executed. The main purpose of the report program is to direct the retrieval of database information, and properly place that data into the "Interim File". Each data item which RPT places into the output file is treated by RPF as a "word". This "Interim File" must subsequently be processed by the RPF utility.



" Report Generation Process "

Figure 5.1

In this manner database information is merged with text to form the finished report. Text can be included for report titles, column headings, descriptive information, or the body of a letter for computer generated correspondence. RPF commands can be used to control text and data placement into a tabular format, spacing, underlining, margin control, and page numbering. Any valid RPF command is permitted in a report program.

5.2.1 **Executing RPT**

The RPT utility may be executed from the user's terminal or scheduled within a batch procedure. The utility is executed with the following command:

```
RPT <input file> <output file> [userid/password] [-c]
```

<input file>	Is the name of the file which contains the RPT report program.
<output file>	Name to be assigned to the "Interim File" created by RPT and subsequently processed by RPF.
userid / password	This parameter is required only if the ORACLE database to be processed is secure.The 'userid' must have been defined using the SQL Define User Statement, and have been granted 'read' privileges to the desired data.
-cN	Where N specifies the size of the SQL work area to be requested for each SQL query defined in the report. If omitted, a default value of 3K bytes will be used.

5.3 **Report Structure**

The RPT programming language is similar in many ways to conventional languages like COBOL or PL/I. Although no explicit declaration of program sections is required (ie. COBOL Data Definition Section) a logical grouping of statements is helpful in providing clarity. The three sections of a RPT program are:

- Data Declaration Section
- Macro Definition Section
- Procedure Section

The Data Declaration Section contains the definition of the user's ORACLE database, and "local" variables to temporarily store retrieved database information. Variables may also be defined for counters, to store totals, and as temporary storage.

Two types of "macros" may be defined within a Macro Definition Section. The "SELECT" macro is used to define a SQL select statement. A "Procedural" macro is similar to a COBOL program subroutine, and is a collection of executable RPT statements.

The Procedure Section contains executable RPT statements which comprise the main body of a report program. Within this section "SELECT" and "Procedural" macros can be either explicitly or implicitly executed.

Again, these are purely logical sections; no language statements exist to define the beginning of one or the end of another. There are virtually no restrictions on where RPT statements may be used within a program. Variables however, must be defined before they are used. RPF commands and report text may be included anywhere within a program, and will be copied into the "Interim File" as encountered within the program's execution.

5.4 RPT Language Statements

This section will describe each RPT statement. Figure 5.2 is a summary of these statements.

The general form of a report statement is:

.<command> <arg1> <arg2> ... <argn>

Report statements begin with a period (".") immediately followed by the RPT command. <command> must be one of the commands listed in Figure 5.2. Although commands and reserved words will be specified in upper case, either upper or lower case is permitted.

A RPT command may begin anywhere on the input line. Each command occupies the entire line; report text or other commands may not be included. All the arguments associated with a command must be specified on the same line.

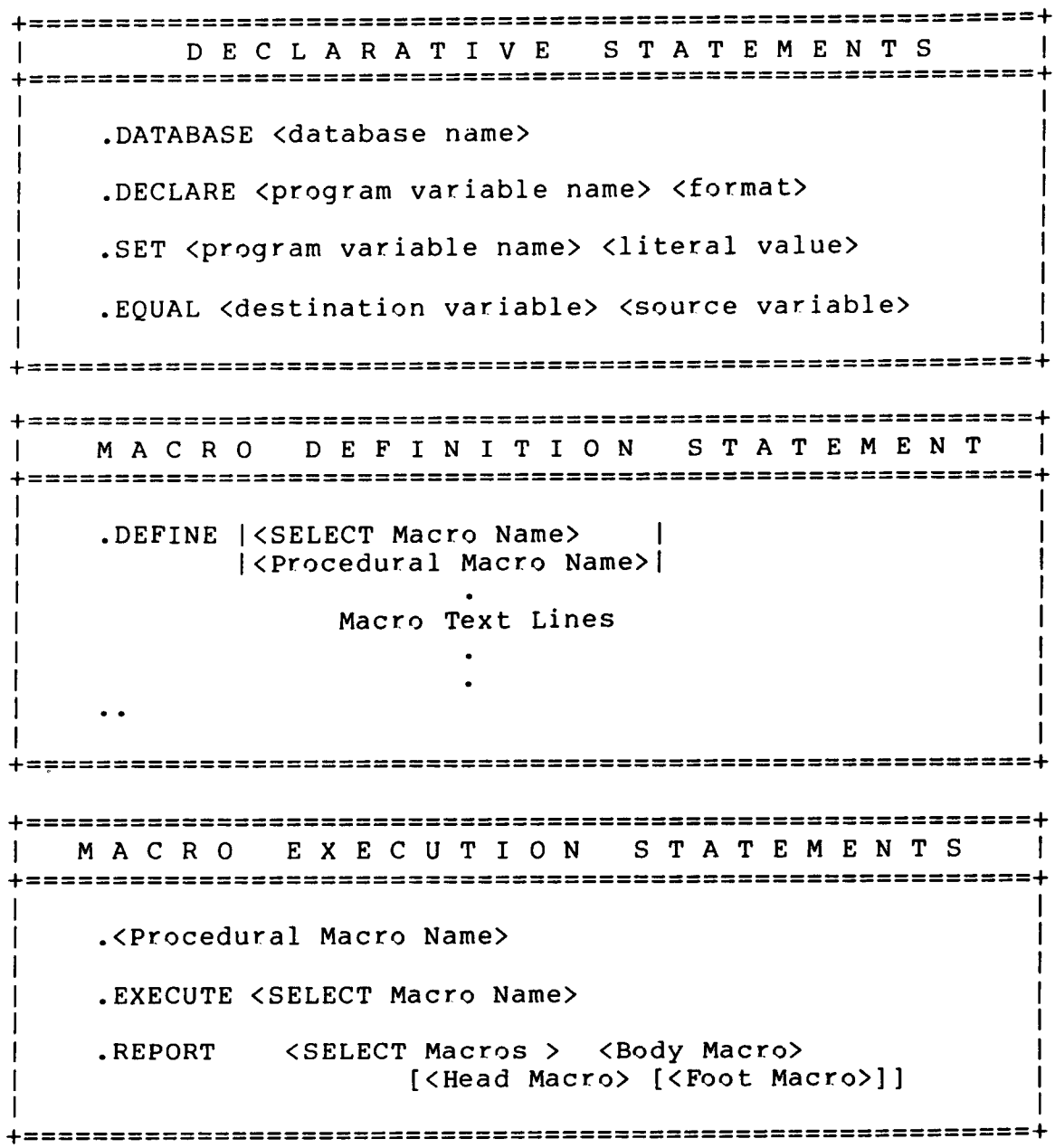
A command may have one or more arguments. Arguments are specified positionally in the defined order. Each argument must be separated by at least one blank space. Arguments are specified by replacing <arg1> <arg2> ... <argn> with either a numeric or alphabetic character string.

Some arguments may contain blank spaces. If one or more imbedded blanks are included the entire argument must be enclosed in double quotes. For example

```
.ASK "Please enter request date : " date
```

contains two arguments; the message "Please enter request date : " and the variable 'date'. The double quotes serve to delimit the beginning and end of an argument, and are not considered a part of the string. Arguments which may contain blanks are denoted with optional quotes (["]<arg>["]).

Arguments enclosed within square brackets "[ ]" are optional; those enclosed within vertical bars "| |" indicates a choice of one.



RPT Statements - Figure 5.2 (Part 1 of 2)

```
+============================================================+
|   P R O G R A M    C O N T R O L    S T A T E M E N T S  |
+============================================================+
|                                                            |
|     .&<Label Name>                                         |
|                                                            |
|     .GOTO <Label Name>                                     |
|                                                            |
|     .IF ["]<Expression>["] THEN <Label1> [ ELSE <Label2>]  |
|                                                            |
|     .STOP                                                  |
|                                                            |
+============================================================+

+============================================================+
|       A R I T H M E T I C      S T A T E M E N T S       |
+============================================================+
|                                                            |
|      |ADD |                                              |
|     .|SUB | <Dest Var> |<Source Var 1>||<Source Var 2>|    |
|      |MUL |            |  <Literal>   ||   <Literal>  |    |
|      |DIV |                                              |
|      |DSUB|                                              |
|                                                            |
+============================================================+

+============================================================+
|   M I S C E L L A N E O U S     S T A T E M E N T S      |
+============================================================+
|                                                            |
|     .PRINT <Program Variable Name>                         |
|                                                            |
|     .ASK '<Message>' <Program Variable Name>               |
|                                                            |
|     .REM <Comment Text>                                    |
|                                                            |
+============================================================+
```

RPT Statements - Figure 5.2 (Part 2 of 2)

5.4.1 Declarative Statements

5.4.1.1 DATABASE

This statement specifies the ORACLE database to be processed within this report. Only one database may be processed per report program.

.DATABASE <database name>

<database name>	is the name of any valid database. If the database is secure a userid/password must be supplied at RPT execution time as part of the RPT command line.

5.4.1.2 DECLARE

This statement is used to declare a program variable and the edit format for printing. All program variables are initialized to null and must be referenced in other statements.

.DECLARE <var name> <format>

<var name>	Is the name assigned to this program variable. The name may be from 1 to x characters, with the first character alphabetic.
<format>	is the edit format used when outputting this variable to the "Interim File". The format will also control rounding and overflow when used in arithmetic statements. A variable's data type is indicated by its format specification. Three data type are currently supported; alpha-numeric, numeric, and date.

Date

A 'Date' variable is defined by specifying a format of 'DATE'.

The output format of a 'date' type variable is MM/DD/YY. Internally, a 'date' variable is a numeric variable which contains the associated date in an absolute julian day number. This approach allows one date to be subtracted from another using a standard arithmetic subtraction function (see section 5.4.5.5 - DSUB). For example:

01/20/81 - 12/29/80 = 22

provides the numeric result of 22. Numeric values may also be added to or subtracted from a date. This is useful in printing invoices where the due date may be computed to be the current date plus 30 days.

A 'date' variable may have its value initialized in a number of ways. It may be assigned a literal value of the form MM/DD/YY, or may be set equal to the value of another 'date' variable. If its value is assigned as a result of a column returned in a SQL SELECT, the database value must be in internal 'Julian Day' format. Presently, the only method for storing internal date data items within a database is by entering the data using an IAF application.

To create a 'date' compatible column, the column must be defined as a 'number' in the SQL CREATE TABLE statement. The column's value must be initially entered and always updated using an IAF application, where the field type is defined as 'date'.

Alpha-numeric

An 'Alpha-numeric' variable may contain any printable character, and is defined by specifying a format of 'An' (where n = number of characters). When a report is executed all alpha-numeric variables are assigned the 'NULL' value.

Numeric

'Numeric' variables are specified using the following symbols:

9 - defines each digit of a numeric variable. Leading zeros are not displayed.

. - Defines the position of the decimal point within a numeric variable. The position is used for arithmetic alignment and is displayed on output.

, - Causes a comma to be inserted on output. Omitted on output if there are no digits to the left of this position.

$ - causes a dollar sign to precede the number on output.

MI - Causes the minus sign to be displayed to the right of a negative number. The default is to the left.

PR - Causes the variable to be displayed within "< >" brackets when negative.

0 - May be used instead of a 9 to designate a digit. Normally leading zeroes are suppressed, however a zero in the format will cause every digit position to be filled.

V - Defines the position of the decimal point within a numeric variable. The position is used for alignment in arithmetic statement, but the decimal point is not displayed on output.

B - Causes the variable to be output as blanks if its value is zero.

The following are examples of various formats:

Format	Value	Displayed
999.99	56.478	56.48
999V99	56.478	5648
9,999	8410	8,410
9,999	639	639
99999	607	607
09999	607	00607
9999	-5609	-5609
9999MI	-5609	5609-
9999PR	-5609	<5609>
B999	564	564
B999	0	blanks
99.99	124.98 (1)	##.##
		24.98
$99.99	45.23	$45.23
DATE	2441453 (2)	12/23/80
A5	Customer	Custo
A20	Customer	Customer

(1) - If the value retrieved into this variable from the database is greater than can be displayed by the format, #'s will be displayed. If the variable is overflowed due to an arithmetic operation, a truncated value will be displayed.

(2) - Julian day number for 12/23/80

5.4.1.3 **SET**

This statement sets the value of the variable equal to the specified literal value.

.SET <variable name> <literal value>

<variable name>	any previously defined program variable.
<literal value>	numeric or character literal. The literal type must match the variable type. The special literal "$$DATE$$" may be used to assign the system date to a date variable.

.SET name JONES - sets the current value of the variable 'name' equal to JONES.

.SET empno 5647 - sets the current value of the numeric variable 'empno' equal to 5647.

.SET today $$DATE$$ - sets the value of the date variable 'today' equal to the current date.

5.4.1.4 **EQUAL**

This statement will set the value of one variable equal to the value of another variable. Both variables must be of the same data type.

.EQUAL <dest var> <source var>

The <dest var> will be set equal to the value of the <source var>.

For character variables, the value of the <source var> will be truncated if longer than the <dest var>, and blank filled if shorter.

For numeric variables, if the format of the <dest var> contains fewer decimal places, the value of the <source var> will be rounded. No provisions are made for overflow.

If the value of the <source var> cannot be stored within the format of the <dest var> variable, the value will be truncated in the destination.

5.4.2 **Macro Definition Statements**

RPT recognizes two types of macro statements; SELECT and Procedural. Both types are defined in the same manner, and RPT will distinguish them by the way they are invoked and the type of statements they contain.

5.4.2.1 **DEFINE**

This statement is used to define a SELECT or Procedural Macro. Execution of this statement stores the macro away for future use. Nothing is output to the interim file. Notice that ".." (two periods) on a line by themselves are used to complete the macro definition.

```
.DEFINE |<SELECT Macro Name>     |
        |<Procedural Macro Name>|
              .
      Macro Text Lines
              .
              .
              .
..
```

<SELECT Macro Name> <Procedural Macro Name>	Name of the macro being defined.
..	Ends the Macro definition.

5.4.2.2 **SELECT Macro**

A SELECT Macro contains the text of a SQL query. Only one query may be specified within each SELECT macro. These queries are used to retrieve the data which will be included within the report. The macro may include any SQL clause or parameter which is valid within a SELECT statement. In this manner the full power of the SQL query language may be used to extract the database information.

In addition to the standard SQL clauses, an INTO clause must also be included. This clause specifies the program variables which will receive the column values returned in the SELECT clause. For example a report program has three variables defined; alpha, beta, and gamma. If the following SELECT macro named 'sample' were executed:

```
.DEFINE sample
        SELECT empno,ename,loc
        INTO   alpha,beta,gamma
        FROM   emp,dept
        WHERE  emp.deptno=dept.deptno
               and sal > 5000;
..
```

the values returned for 'empno', 'ename', 'loc' are stored in the program variables 'alpha','beta','gamma' respectively. The program variable must be of the same data type as the column or expression in the SELECT clause. It should be noted that an INTO clause, as well as any SQL clause, may be defined in a 'free format', and the structure used here is for readability purposes only.

The value of a program variable may be substituted for any literal defined in the WHERE or SELECT clause of a SQL query. When used in this manner, the variable name must be preceded with an '&'. The variable name may not include an underscore character '_'. For example, if the following SELECT macro:

```
.DEFINE seldept
        SELECT deptname,location
        INTO   dname,loc
        FROM   dept
        WHERE  deptno = &dno
..
```

were executed with 'dno' equal to 20, the value of 20 would be substituted into the WHERE clause, and the 'name' and 'location' of department 20 would be stored into the program variables 'dname' and 'loc'. Both the program variable and the database column must be the same date type.

From the above examples, it can be seen that the results of one query may be stored into a program variable, and used as a substitution variable within another query. This technique is used to construct queries which produce nested reports. Nested reports are discussed in Section 5.4.3.3.1. The following two SELECT statements would be used to generate a report of each course in the 'PHYSICS' department, and for each course a list of students and their grades:

```
.DEFINE courses
        SELECT cname,cnumber,cdate
        INTO   cname,cnumber,cdate
        FROM   course
        WHERE  cdept = 'PHYSICS'
..

.DEFINE students
        SELECT sname,grade
        INTO   sname,grade
        FROM   students
        WHERE  course  =   &cnumber
..
```

5.4.2.3 **Procedural Macro**

A 'Procedural Macro' is similar to a programming language subroutine. It may contain both RPT language statements, and user text and RPF commands. When invoked, the RPT statements within the macro are executed, and user text and RPF commands copied to the 'Interim File'.

A macro may not be **defined** within a macro. Macros may be invoked from within other macros. The following procedural macro outputs the heading of a report. The .PRINT statement outputs the value of the variable 'year', and is explained in a later section.

```
.DEFINE heading
        #t 2
        #cul Corporate Finance Report #
        #s 2
        #cul Fiscal Year
        .PRINT  year
        #
..
```

5.4.3 **Macro Execution Statements**

5.4.3.1 **Procedural Macro Execution**

A procedural macro may be explicitly executed anywhere within an RPT program including from within another procedural macro. The named macro is executed by specifying:

.<procedural macro name>

The macro must have been defined in the program prior to its execution. The statement:

.summary

will cause the procedural macro named 'summary' to be executed. Following the macro execution the next sequential statement is executed.

5.4.3.2 **SELECT Macro Execution**

A SELECT macro may be explicitly executed with the following RPT statement:

.EXECUTE <SELECT macro name>

The EXECUTE statement will cause the specified SELECT macro to be executed. Following the execution of the query, the first row will be returned. The values of the selected columns will be placed into the corresponding program variables as defined on the INTO clause.

Executing a query in this manner will always return only the first row. If the macro is re-executed, the entire query will be reprocessed, and again the first row is returned.

Explicit execution of SELECT macros is useful where only one row is returned. For example, if the average sal for a department was to be reported in a summary section, the following query could be executed to return this result:

```
.DEFINE deptavg
        SELECT avg(sal)
        FROM   dept
        INTO   avgsal
        WHERE  deptno = &dno;
..

        .
        .
        .
.SET   dno 30
.EXECUTE deptavg
.PRINT avgsal
```

Another use of the EXECUTE would be to look up a customer's name and address, based on the customer number, for a computer generated form letter. The following example would accomplish that task:

```
.DEFINE custaddr
        SELECT custname,custaddr
        INTO   name,addr
        FROM   custlist
        WHERE  custno = &customer
..
         .
         .
         .
.EXECUTE custaddr
.PRINT name
#n
.PRINT addr
#n
```

5.4.3.3. **REPORT Statement**

The **REPORT** statement causes the automatic execution of SELECT and procedural macros which drives the generation of most reports. Unlike the EXECUTE statement, the REPORT statement will cause every row returned from the SELECT to be processed. For each row the specified procedural macros are executed. Figure 5.3 depicts the structure of a REPORT statement.

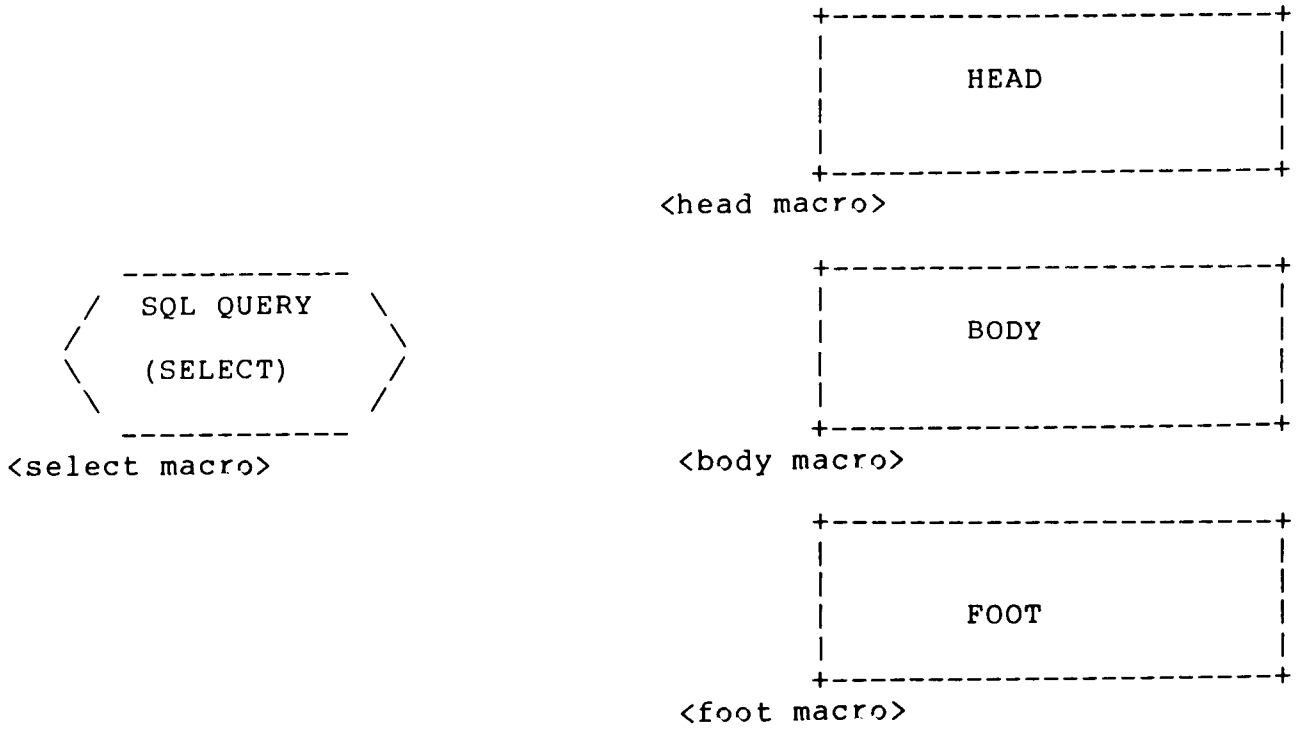
The three procedural macros correspond with the head, body, and foot of a report. The **head macro** is executed once within a REPORT statement, when the first row is returned. Included within this macro would be the column headings, descriptive text, and report titles. Since the body macro is not executed for the first row returned (if the heading argument is included) you should either execute the body macro as part of the heading or make other arrangements to print the first row. This allows flexibility in setting up reports since it may be desirable to handle the first row differently than succeeding rows.

The **body macro** is executed for the second through the last rows. If a foot macro was not specified, the body would also be executed for the first row. Its function is to output each row of data within the desired format. Other functions could be to accumulate totals, maintain counters, and control page breaks.

The **foot macro** is executed after the last row of the query has been processed. Within this macro summary calculations and footnotes could be included.

The head, body, and foot are standard procedural macros and may contain any valid RPT statement. Within these macros, other macros can be executed. For example, within the foot of a report the following SELECT could be executed to compute salary statistics for the department 10:

```
.DEFINE summary
        SELECT max(sal),min(sal),avg(sal),sum(sal)
        INTO   maxsal,minsal,avgsal,sumsal
        FROM   emp
        WHERE  deptno=10
..
```



.REPORT <select macro> <body macro> [<head macro> <foot macro>]

REPORT Statement Structure

Figure 5.3

5.4.3.3.1 **Nested Reports**

A REPORT statement can be executed within the head, body, or foot macros of another REPORT. Figure 5.4 shows the structure of such a report.

In this example, the departments within a company are reported. For each department, a list of the projects within that department is generated. Within the body of the 'department' report, is a REPORT statement to generate the projects report.

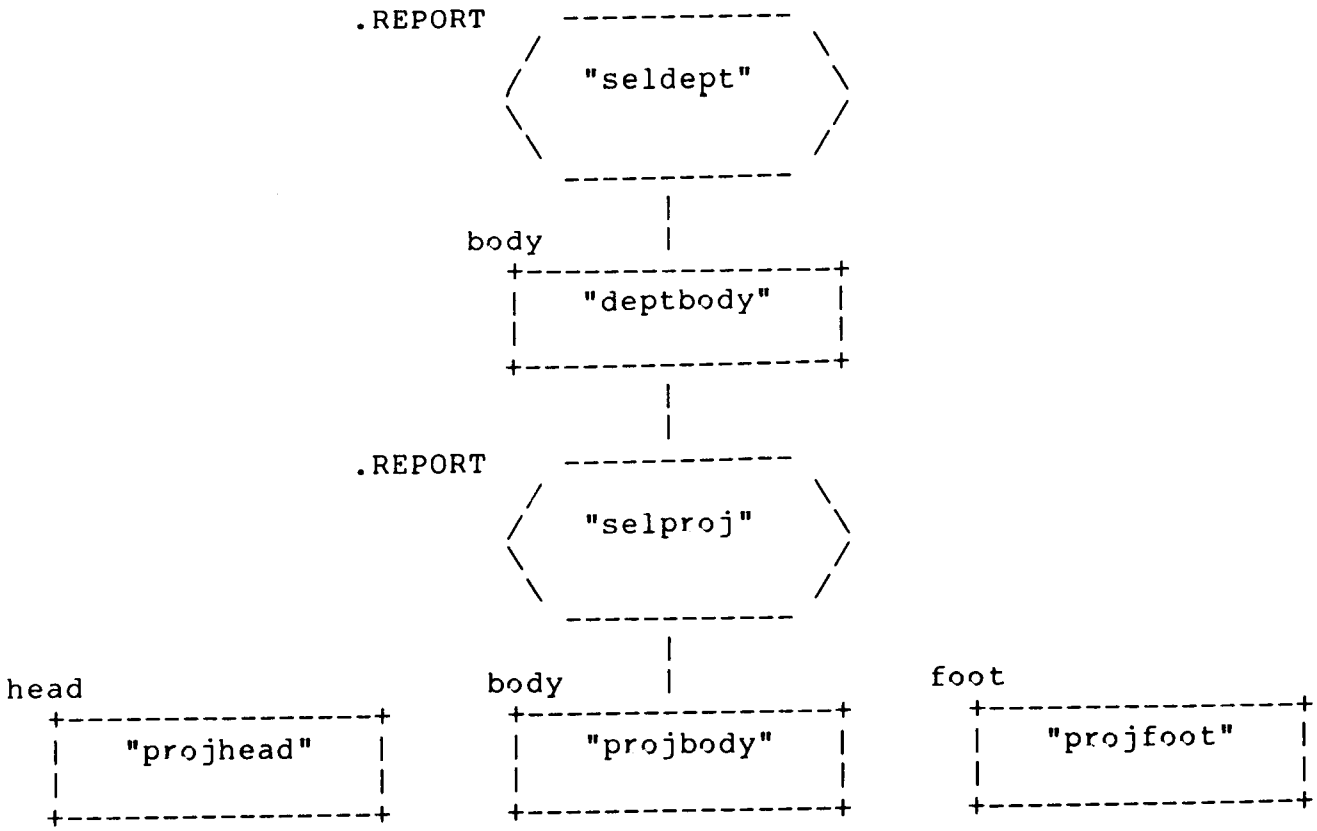
Multiple levels of **nested** REPORT statements may be constructed. Additionally, multiple REPORT statements may be included within a head, body, or foot of another report. Figure 5.5 shows the structure and format of an 'employee personnel' report, listing for each employee, the employee's job history, salary history, and project assignments.

5.4.3.3.2 **Disjunctive Reports**

Two or more SELECT macros may be specified in a REPORT statement. In this case multiple SELECT's will be executed, and a row from each returned. The head, body, and foot macros would be executed exactly as with a single SELECT, however data from multiple selects may now be printed. The rows from each SELECT are returned in step with each other.

The data retrieved from multiple selects may be entirely independent, referencing different tables. For example, an employee's job history may be retrieved by one SELECT, while a list of the employee's current projects by another. This data could be printed side by side in separate columns across the page as shown in figure 5.6.

The multiple SELECT macros may be specified with either an 'AND' or an 'OR' between each select macro. 'AND' indicates that the report should be executed only if both selects (on either side of the 'AND') return at least one row. An 'OR' indicates that the report should be executed if at least one of the selects (on either side of the 'OR') returns at least one row. Note that if only one SELECT is specified in a report and no rows are returned, the report will not be executed. Also note that if two or more selects are specified the select argument will need to be enclosed in quotes as in "sel1 AND sel2".



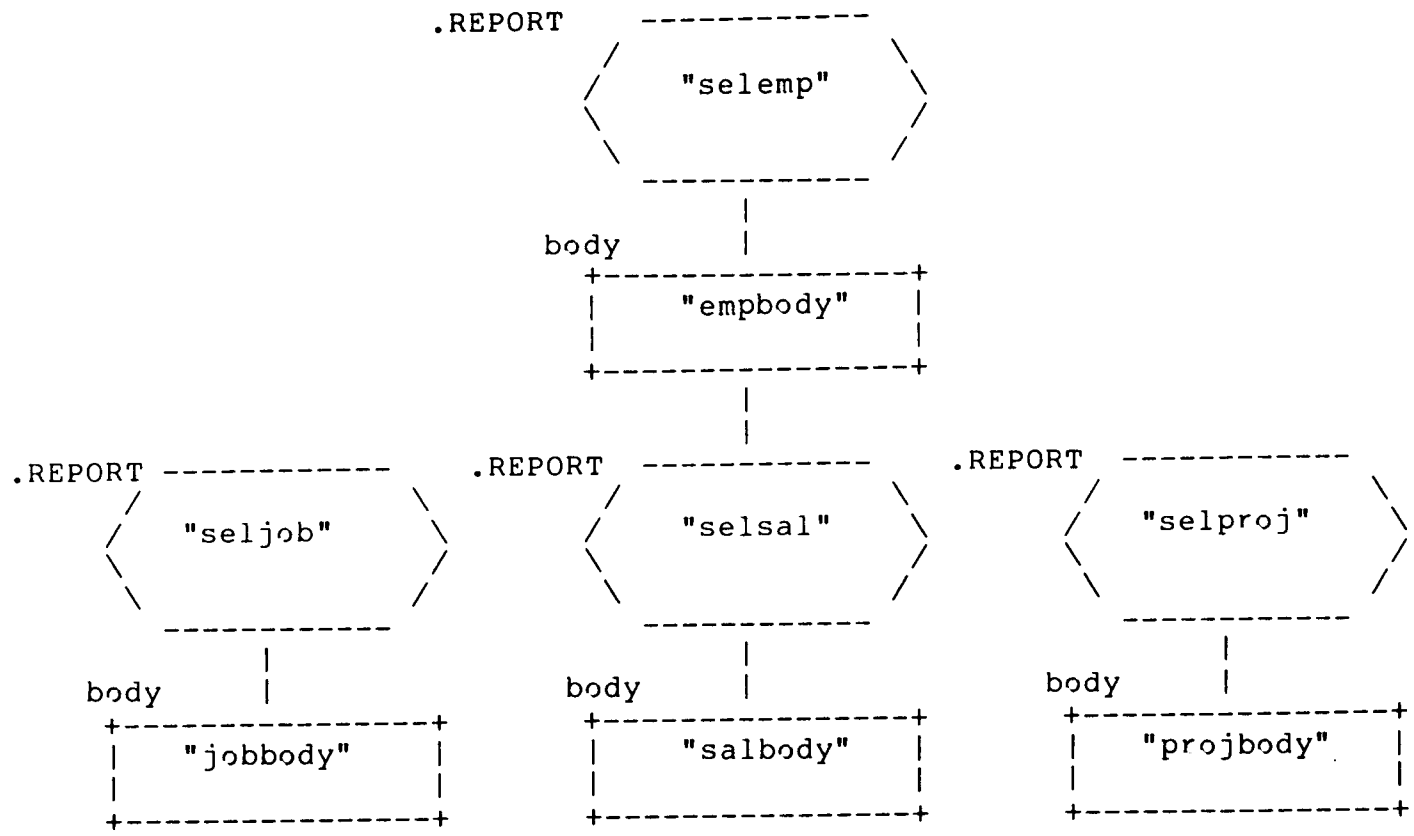
DEPARTMENT / PROJECT REPORT

Department Number	Department Name	Department Location
067	Circuit Dev	Boise

Project No.	Project Name	Comp Date
563	Board Design	04/30/81
894	Chip Manufact	06/01/81

NESTED REPORT

Figure 5.4



EMPLOYEE PERSONNEL REPORT

Employee Number	Employee Name
8945	Martin, R.J.

Job Title	Start Date
Prog Trainee	06/14/72
Programmer	12/06/75
Analyst	05/23/77

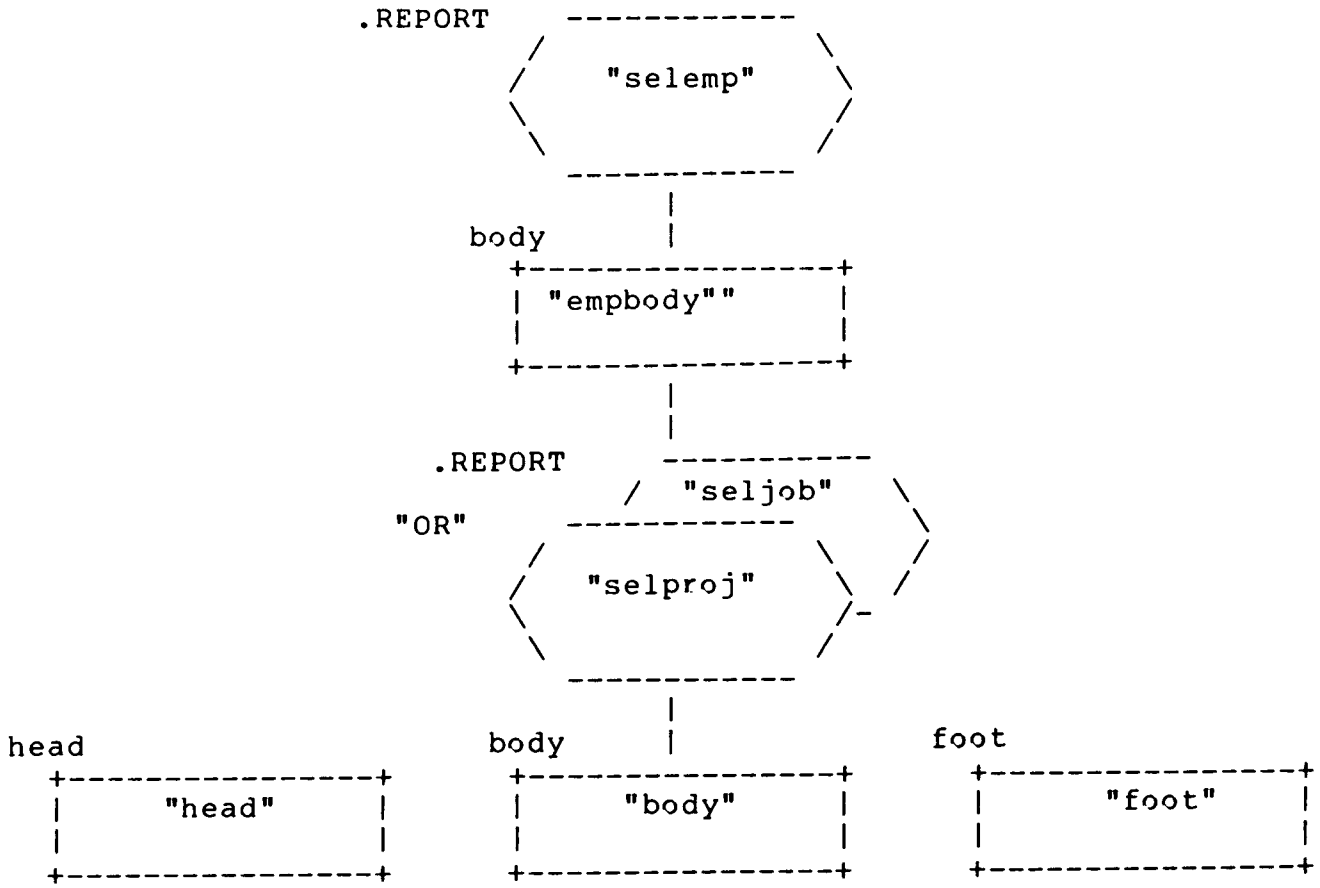
Salary	Start Date
$9,600.00	06/14/72
$13,200.00	02/01/74

Project Assignments

456 Sales Report System

MULTIPLE REPORTS WITHIN A REPORT

Figure 5.5



E M P L O Y E E R E P O R T

Employee Number	Employee Name	Job History		Current
		Title	Date	Projects
4028	Murphy, P.E.	Pgmr Lvl1	06/72	Design 33
		Pgmr Lvl2	12/74	Doc 45
		Analyst	07/75	
		Sr. Anal	08/78	

" DISJUNCTIVE REPORT "

Figure 5.6

5.4.3.3.4 **REPORT Statement Format**

The format of the REPORT statement is:

.REPORT <SELECT macros> <body macro> [<head macro> <foot macro>]

<SELECT macros>	name of SELECT macro executed for this report. If two are specified, they must be joined by an 'AND' or 'OR' and enclosed within double quotes.
<body macro>	name of the body macro which will be executed for the 2nd thru last returned rows.
<head macro>	name of the head macro which is executed for the first row returned. This parameter is optional.
<foot macro>	name of the foot macro which is executed after the last row is returned. This is an optional parameter and can only be specified if a head macro is also specified.

5.4.4 **Program Control Statements**

RPT provides statements to control the program execution within a **procedural** macro. Only the .STOP statement may be used outside of a procedural macro.

5.4.4.1 **Label Definition Statement**

A label may be defined within a procedural macro using the following statement:

.&<label name>

<label name>	From 1 to 8 character name; first character must be alphabetic.

A label may be referenced only within the macro in which it was defined. Label definitions do not span macros. Since label definitions are local, the same label name may be used in multiple macros.

5.4.4.2 **GOTO Statement**

The **GOTO** statement causes an unconditional branch to the specified label. The format of this statement is:

```
.GOTO <label name>
```

<label name>	The name of a label defined within the current procedural macro.

Figure 5.7 lists a sample program which demonstrates the use of the GOTO statement.

5.4.4.3 **IF Statement**

This statement causes a branch to the specified macro label depending on the result of an expression. The IF statement is only valid within a procedural macro. The format is:

```
IF <expression> THEN <label1> [ ELSE <label2> ]
```

<expression>

An <expression> may compare the value of a program variable with other program variables or literal constants. Program variable names must be preceded with an ampersand '&'. If the '&' is omitted an "Invalid Column Name" message will be displayed.

The IF statement supports the complete set of relational and logical operators, and arithmetic expressions permitted within a WHERE clause of a SQL statement.

Character constants must be enclosed within single quotation marks. All literal values and program variables within an expression must be of the same data type.

```
.REM   ************************************************************
.REM   **********      EXAMPLE OF "GOTO" STATEMENT         *********
.REM   ************************************************************
.REM
.REM                        Define Macro
.REM
.DEFINE macro1
       .GOTO label1
       .&label2
          two
       .GOTO end
       .&label1
          one
       .GOTO label2
       .&end
..
.REM
.REM                       Execute Macro
.REM
.macro1
```

==========

Generated Output

```
one two
```

"Sample GOTO Program"

Figure 5.7

If the literals, program variables, and relational operators are separated by one or more blank characters, the entire expression must be enclosed within double quotation marks(").

A program variable used in an expression may not have the NULL value. If the value is NULL, the report program will be terminated with an error. To prevent this situation all variables should be initialized to either zeros or blanks.

If the variable is assigned a value from a database column which allows NULLs, the null value function (NVL) should be used to assign a non-NULL value. For example the variable salary used in following IF statement:

```
.IF " &salary > 10000 " THEN ...
```

then the SELECT macro should define the NVL function in the SELECT list:

```
SELECT NVL(sal,0)
INTO salary
FROM ...
```

If any variable participating in the expression has the null value the ORACLE error: "unexpected end of sequel statement" will be displayed.

The following are valid expressions:

```
&name='SMITH'

&salary*2<4500

"&dept=10 or &div='MOTOR'"

" &sal > 5000 and &job = 'plumber' "
```

<label1>

If the <expression> evaluates to 'True', processing control will be transferred to <label1>

<label2>

If the <expression> evaluates to 'False', processing control will be transferred to <label2>. If "ELSE <label2>" is omitted, and the expression is 'False', the next sequential statement is executed.

```
.REM   ************************************************************************
.REM   **********     EXAMPLE OF  "IF  THEN"  STATEMENT              ********
.REM   ************************************************************************
.DATABASE IAFDEMO
.REM
.REM                        Define   Loop    Variable
.REM
.DECLARE x   999
.REM
.REM                             Define     Macro
.REM
.DEFINE macro1
         .&label1
         Loop Counter =
         .PRINT x
         .ADD x x  1
         .IF "&x <= 5 "  THEN label1
         End of Loop
.
.REM
.REM                           Execute     Macro
.REM
.macro1
```

Generated Output

```
Loop Counter =  1
Loop Counter =  2
Loop Counter =  3
Loop Counter =  4
Loop Counter =  5
End of Loop
```

" Sample IF THEN Program "

Figure 5.8

5.4.4.3.1 IF Statement Guidelines

IF statement processing is fairly time consuming, and should therefore be used cautiously within a report program. It can be used effectively for controlling the overall flow of a report, where the statement will be executed infrequently. For example, depending on the current date or a terminal input variable, certain sections of a report may be included or excluded. The IF statement(s) controlling these conditionally executed sections would be executed once for the entire report.

In contrast, executing an IF statement for each row processed, could severely lengthen the report execution time if a large number of rows were processed. For example, although RPF supports page control, there is no mechanism to reprint page headings for each page break. One method of reprinting the headings would be to count each line which is generated, then at the top of page execute a special heading routine. However, for each line printed an IF statement would have to be executed to test for an end of page condition. This use of the IF statement is not recommended for reports of substantial length.

Some uses of the IF statement can be replaced with SQL functions. For example computing the maximum and minimum value of a variable over a range of rows could be accomplished by comparing each new value with the previous maximum and minimum values. This requires two IF statements for each row. Another approach would be to execute a SELECT in the report 'foot' which used the MIN and MAX SQL functions. The second approach would be more efficient.

5.4.4.4 STOP Statement

The STOP statement will terminate the execution of the report program. STOP may be included in the procedure section or a procedural macro. The format of the statement is:

```
.STOP
```

If a STOP is not included, the program will terminate following the execution of the last program statement.

5.4.5 **Arithmetic Statements**

RPT provides statements to perform addition, subtraction, multiplication, and division between two program variables. The results of the operation are stored in a third variable. The formats are:

```
 |ADD |
.|SUB |  <dest var>  <source var1>  <source var2>
 |MUL |
 |DIV |
 |DSUB|
```

<dest var>
<source var1>
<source var2>

For all arithmetic statements except DSUB, a numeric literal may be substituted for the input arguments <source var1> and <source var2>.

These numeric variables must have been previously defined. The maximum number which can be stored in a variable is determined by the format specified on the DECLARE statement. If the result of the arithmetic operation overflows the <dest var>, the high order digits will be lost, and no error will be indicated. These digits are lost both on output and in subsequent arithmetic operations.

For example, if the result is '456' and the format of <dest var> is '99', '56' is stored in <dest var> and the high order digit '4' is lost.

If the decimal portion of the arithmetic result contains more digits them defined in the <dest var> format, the low order digits will be rounded off. For example, if the the result was '46.576' and the format of the <dest var> was '99.99', the number '46.58' will be stored.

5.4.5.1 **ADD**

The value of <source var1> is added to <source var2> and the result is stored in <dest var>. For example, if X=5 and Y=6 then:

```
.ADD Z X Y
```

will set Z=11. The result may be stored into one of the source variables with:

```
.ADD X X Y
```

results in X=11.

5.4.5.2 **SUB**

The value of <source var2> is subtracted from <source var1> and the result is stored in <dest var>. For example, if X=5 and Y=3 then:

```
.SUB Z X Y
```

will set Z=2. The statement:

```
.SUB X X Y
```

set the value of X=2.

5.4.5.3 **MUL**

The value of <source var1> is multiplied by <source var2> and the result is stored into <dest var>. For example, if X=5 and Y=6 then:

```
.MUL Z X Y
```

will set Z=30. The statement:

```
.MUL X X Y
```

sets the value of X=30.

5.4.5.4 **DIV**

The value of <source var1> is divided by <source var2> and the result is stored into <dest var>. For example, if X=10 and Y=2 then

```
.DIV Z X Y
```

sets the value of Z=5. The statement:

```
.DIV X X Y
```

sets the value of X=5. If a number is divided by zero (ie. <source var2>=0), a "** DIVIDE BY ZERO (X/Y) **" error message will be issued and the report program will be terminated.

5.4.5.5 **DSUB**

This statement subtracts one 'date' variable from another, storing the result into a 'numeric' variable. This allows the number of days between two dates to be computed. For example, if DATE1=01/24/81 and DATE2=12/25/80 then

```
.DSUB RESULT DATE1 DATE2
```

set the value of RESULT=30. Note that neither DATE1 nor DATE2 may be literals. 5.4.6 **Miscellaneous Statements**

5.4.6.1 **PRINT Statement**

The PRINT statement inserts the contents of the specified program variable into the output "Interim File". This is the only mechanism for inserting database information into the output report. The content of the variable will be formatted according to the format defined on the DECLARE statement. The data will be treated as a separate word when processed by RPF.

The format of the statement is:

```
.PRINT <variable name>
```

PRINT statements may be interspersed with RPF commands to print the data in the various columns of a tabular report. Refer to the Sample Reports in Section 5 for examples of the use of PRINT.

5.6.2 **ASK Statement**

The ASK statement displays a message on the user's terminal, and allows the user to enter a value to be assigned to the specified program variable. ASK provides a means for the user to dynamically control the flow and output of a report.

The format of this statement is:

```
.ASK "<message>" <variable>
```

<message> is a 1 to x character message which will be displayed on the user's terminal. If the message contains blank characters, the message text must be enclosed within double quotes(").

<variable> is the name of the program variable whose value will be set equal to the data value entered by the user. The entered data value must be of the same data type as the program variable.

If a numeric variable is specified, and the entered data is alphanumeric, the variable is set equal to zero and no error is indicated.

If a 'date' variable is specified, the format of the input is MM/DD/YY. The date routine will verify that the entered date is valid. If invalid, the operator will be requested to re-enter the data.

5.4.6.3 **Remark Statement**

The REM Statement allows the report programmer to include comment lines within the program source file. The entire line of text following the REM statement is treated as a comment and ignored by RPT. The remark statement will not be output to the interim file. The format of this statement is:

.REM <comment text>

R S I

ORACLE

HOST LANGUAGE INTERFACE

Oracle Programmer's Guide - Version 2.3

O R A C L E

HOST LANGUAGE INTERFACE

TABLE OF CONTENTS

O R A C L E

HOST LANGUAGE INTERFACE

INTRODUCTION

SQL is designed to be used as a stand-alone terminal language for interactive users, and as a data sublanguage embedded in a host programming language. All SQL query, data manipulation, data definition, and data control facilities are available from both the interactive and data sublanguage interface.

ORACLE interfaces to FORTRAN, COBOL, PL/1, "C" and other high level host programming languages by means of program calls. ORACLE interfaces to assembly language via macro instructions.

A program establishes communication with ORACLE by issuing the LOGON call. Communication takes place via the Logon Data Area (LDA) defined within the user program. A user program issues one and only one LOGON to ORACLE.

A program opens a database and creates a "cursor" by issuing an OPEN call. A cursor is the name of a data area which is used to identify and control an active SQL statement. The Cursor data area is defined within the user program. ORACLE permits user programs to have multiple active SQL statements. This is accomplished by a single program issuing multiple OPEN's to establish multiple cursors.

The SQL call is used to associate a SQL statement with a cursor. In the case of a query, the SQL call defines a set of rows to be retrieved and logically positions the cursor just before the first row.

Subsequent calls related to the same SQL statement reference the same cursor.

The following sections assume that the reader is familiar with SQL, section 3.12 of the "ORACLE Introduction", and at least one programming language.

O R A C L E

HOST LANGUAGE INTERFACE

PROGRAM CALL SUMMARY

There are eleven different ORACLE calls or macros that may be used to access the database.

The LOGON call establishes communication between a program and ORACLE.

The OPEN call connects a program to a database and creates a cursor.

The SQL call is used to pass a SQL statement to ORACLE.

The DESCRIBE call is used to dynamically determine the number and data types of fields to be retrieved during a query operation.

The NAME call is used to retrieve table and column names.

The DEFINE call identifies to ORACLE the location of data field buffers in the user program.

The BIND call allows programs to dynamically substitute variables into SQL statements.

The EXECUTE call causes ORACLE to process the SQL statement.

The FETCH call is used to retrieve rows, one at a time, during a program query operation.

The CLOSE call closes the database and disconnects the cursor from ORACLE.

The LOGOFF call disconnects the user program from ORACLE.

O R A C L E

COMPILER LANGUAGE INTERFACE

CODING RULES

The following general rules apply to user programs calling ORACLE:

literals	may be used within the CALL parameter list if they are permitted by the compiler. Note that the use of a literal must generate a pointer to the literal as a result of the CALL; the PDP-11 compilers always generate pointers to literals, but the VAX compilers may generate a pointer to a character string descriptor which cannot be used by the ORACLE Host Language Interface. An example is the VAX FORTRAN compiler which generates pointers to descriptors for all references to character data types. In order to circumvent the FORTRAN calling defaults, use the "%REF" function to force a call by reference.
variable-length fields	variable length parameters are passed to ORACLE with an accompanying length field in the form: parameter,length.
length field formats	Length fields are binary numbers (FORTRAN INTEGER; COBOL PICTURE S9999 COMP; etc.) of the standard word size for the computer on which ORACLE is running (32 bits for the VAX, 16 bits for the PDP-11). The length field may be omitted if the variable parameter is terminated by a binary zero.

missing parameters

If a length field or other optional parameter is omitted from a call parameter list, the user may code comma comma (,,) to indicate the absence of the parameter. For example: "param1,len1,param2,len2 vs. param1,,param2,,". In some languages ("C" for example) the comma comma notation is not allowed to indicate a missing parameter in a call parameter list. If comma comma notation is not permitted, the user may code a minus one (-1) to indicated the missing parameter. For example: (param1,-1,param2,-1).

The LOGON Call

CALL OLOGON (lda[,areacount])

The LOGON call establishes communication between ORACLE and a user program.

Communication takes place via the Logon Data Area (LDA) defined within the user program. The LOGON call connects the LDA to ORACLE. A program logs on to ORACLE one and only one time. A program has one and only one LDA.

The LDA is a 64 byte data area defined within the user program. The first two bytes of the LDA contains a return code indicating the result of the LOGON. A zero return code indicates a successful LOGON. Error return codes are listed in the "Messages and Return Codes" section of this manual.

lda	specifies the name of the 64 byte Logon Data Area defined within the user program.
areacount	specifies an integer number indicating the number of ORACLE SQL statement work areas (SWA's) to be concurrently maintained in main storage. This optional parameter is used only if the user program is opening multiple cursors and and the user does not want ORACLE to swap SWA's. (See the section on "Program Interface Data Areas." at the end of this manual.) Areacount should be equal to or less than the maximum number of cursors the program will open. The default value for areacount is one.

LOGON Examples:

FORTRAN

Log on to ORACLE using a Logon Data Area named LDAREA.

```
CALL OLOGON(LDAREA)
```

COBOL

Log on to ORACLE using a Logon Data Area named LDAREA and a integer named AREACOUNT.

```
CALL "OLOGON" USING LDAREA,AREACOUNT.
```

MACRO-11 ASSEMBLY LANGUAGE

Log on to ORACLE specifying the address of the LDA in register 2.

```
CCALL OLOGON,R2
```

The OPEN Call

```
CALL OOPEN (cursor,lda,dbn,dbnlen[,areasize][,uid,uidlen])
```

The OPEN call establishes a cursor to operate on a specific database.

A cursor is a data area defined within the user program. The OPEN connects the cursor to ORACLE. The cursor name is used to identify an active SQL statement within the user program.

Each cursor may control only one SQL statement at a time. The same cursor may be reused to control another SQL statement after the first statement's operation has been completed.

A single user program can have multiple SQL statements active at the same time. This is accomplished by issuing multiple OPEN's to establish multiple cursors within the program. These OPEN's can be to the same or different databases.

The cursor data area contains status information on an active SQL operation. All subsequent ORACLE calls referencing a SQL statement reference it by cursor name. The first two bytes of the cursor contain a return code indicating the result of the OPEN. A zero return code indicates a successful OPEN. Error return codes are listed in the "Messages and Return Codes" section of this manual.

cursor	specifies the name of a 64 byte data area within the user program. The cursor data area is connected to ORACLE by the OPEN call. Each cursor defines an active SQL statement within the program.
lda	specifies the name of the Logon Data Area specified in the LOGON call.
dbn	specifies the name of the ORACLE database as defined in the Database File (DBF) utility.
dbnlen	specifies a binary integer indicating the length of the database name. If the database name was specified as a literal, this parameter may be omitted.

areasize	specifies a binary integer indicating the size of the ORACLE SQL Work Area (SWA) in increments of 1K bytes. This optional parameter is used only if the user wants a work area other than the default size of 3K bytes. The SWA must be large enough to contain the compiled SQL statement plus one row of data of the table or view being processed. ORACLE SQL work areas can vary in size from 1 to 16. See the section on "Program Interface Data Areas" at the end of this chapter. Note that if multiple cursors of different sizes are to be opened, the one with the largest SWA size must be opened first.
uid	specifies the user identification and password as defined by either the Database File (DBF) utility or the SQL "DEFINE USER" function.
uidlen	specifies a binary integer indicating the length of the user identification and password. If they were specified as a literal, this parameter may be omitted.

OPEN Examples:

FORTRAN

Open the PERSONNEL database and establish the cursor CURS1. The name of the Logon Data Area (LDA) is LDAREA. Specify the database name and user id as literals and take the default 3K SQL work area size.

```
CALL OOPEN(CURS1,LDAREA,'PERSONNEL',,,'SCOTT/TIGER')
```

Open the personnel database and establish the cursor CURS2. The name of the program's LDA is LDAREA. Specify the database name and the length of the database name as program variables DBN and DBL. Specify a 5K SQL work area. The user identification and password are not specified, i.e., the database is not secure.

```
CALL OOPEN(CURS2,LDAREA,DBN,DBL,5)
```

COBOL

Open the personnel database and establish a cursor CURS1. The name of the Logon Data Area is LDAREA, the database name is contained in a variable named DBN, its length is in DBNLEN, the area size is in a variable named AREASIZE, the user identification and password are contained in a variable named UID, and its length is in UIDL.

```
CALL "OOPEN" USING CURS1,LDAREA,DBN,DBNLEN,AREASIZE,UID,UIDL.
```

MACRO-11 ASSEMBLY LANGUAGE

Open the personnel database and pass the address of the cursor to ORACLE in register Zero. The address of the program's LDA is in register 2. Specify the database name in program variable DBN and pass the length of the database name as a literal. Specify a 1K SQL work area. The user identification and password are not specified, i.e., the database is not secure.

```
OOPEN R0,R2,BN,#9,#1
```

The SQL Call

CALL OSQL (cursor,sqlstatement,sqllen)

The SQL call passes a SQL statement to ORACLE, and associates that SQL statement with an open cursor. Subsequent calls reference the SQL statement by cursor name.

The SQL call may contain any valid SQL query, data manipulation, data definition or data control statement. ORACLE parses the statement and selects an optimal access path to perform the requested function, however, the operation is not executed at this time.

SQL syntax error codes will be returned in the cursor RETURN-CODE area along with a pointer to the text in error. See the parse errors section in Messages and Return Codes for a complete list of syntax errors.

cursor	specifies the name of a 64 byte data area within the user program. The cursor data area contains status information on an active SQL operation. Each cursor defines an open SQL statement within the user program. The SQL call attaches a SQL statement to the cursor. A cursor may be serially reused by subsequent SQL calls within a user program, or the program may define multiple concurrent cursors.
sqlstatement	specifies any valid SQL query, data manipulation, data definition, or data control statement. The statement may contain substitution variables anywhere a constant is permitted. Substitution variables are identified by preceding the variable name with an ampersand, i.e. &EMPNO. These substitution variables then may be referenced symbolically in a BIND call.
sqllen	specifies a binary integer containing the length of the SQL statement. If the SQL statement was specified as a literal this parameter may be omitted.

SQL Examples:

FORTRAN

Pass a SQL query statement to ORACLE using the cursor CURS1. Specify the SQL statement as a literal.

```
CALL OSQL(CURS1,'SELECT ENAME,SAL FROM EMP WHERE DEPTNO =
&DNO;')
```

Pass a SQL statement to ORACLE using the cursor CURS2. Specify the SQL statement as a program variable named QUERY1 with the length of the SQL statement specified as as literal.

```
CALL OSQL(CURS2,QUERY1,28)
```

COBOL

Pass the SQL statement contained in a variable named SQLSTM with length contained in SQLSTML to ORACLE. Use the cursor named CURS1.

```
CALL "OSQL" using CURS1,SQLSTM,SQLSTML).
```

MACRO-11 ASSEMBLY LANGUAGE

Pass a SQL statement to ORACLE using the cursor CURS1. Register 1 points to the length of the SQL statement. Register 1 plus 2 points to the the SQL statement itself.

```
CCALL OSQL,#CURS1,2(R1),(R1)
```

The DESCRIBE Call

CALL ODSRBN (cursor,position[,dbsize][,dbtype][,fsize])

The DESCRIBE call returns internal data type and size information for a field or expression listed in the SELECT clause of a query statement.

DESCRIBE operates positionally, one field at a time, referencing each field in the SELECT clause as if each were numbered consecutively, left to right, beginning with 1.

DESCRIBE can be used after a SQL, EXECUTE, or FETCH call to determine the maximum size and internal data type (dbsize & dbtype) of fields to be returned as the result of a query. If DESCRIBE is used after a FETCH Call, the actual size of the field just fetched (fsize) may be returned in addition to dbsize and dbtype.

If the user specifies a position number greater than the number of fields in the SELECT list, DESCRIBE returns an end-of-file indicator in the RETURN-CODE of the cursor data area. This allows programs to dynamically determine the number of fields to be returned as the result of a query. This is necessary if the program does not know in advance how many fields there are in the SELECT list, as in the case of SELECT *.

cursor — specifies the name of a 64 byte data area within the user program. The cursor data area contains status information on an active SQL operation. Each cursor defines an active SQL statement within the user program. DESCRIBE uses the cursor name to reference a specific SQL query statement that had been passed to ORACLE in a prior SQL call. The RETURN CODE area of the cursor indicates success (code of 0) or failure (non-zero) of the DESCRIBE call. All error return codes are listed in the "Messages and Return Codes" section of this manual.

position	specifies the position of a field or expression listed in the SELECT clause of a SQL query statement. Fields and expressions in a SELECT list are separated by commas. Each field or expression is then referenced positionally as if the fields were numbered left to right consecutively beginning with 1. The position number is specified as a binary integer. If the user specifies a position number greater than the number of fields in the SELECT list, DESCRIBE returns an end-of-file indicator (+4) in the RETURN-CODE of the cursor data area.
dbsize	returns a binary integer specifing the maximum size of the field. If the field is defined as CHAR in the CREATE or EXPAND TABLE, the length returned is the maximum length specified for the field in that particular CREATE or EXPAND TABLE. Fields defined as NUMBER in the CREATE TABLE, and fields that contain the results of expressions always return a size of 8.
dbtype	returns a binary integer that indicates the internal data type of the field as it is stored in the database. Fields defined as CHAR in the CREATE or EXPAND TABLE are stored as variable length ASCII strings and return a value of 1. Fields defined as NUMBER are stored in ORACLE extended precision floating-point and return a value of 2. Fields that contain expression results also return a value of 2.

fsize	returns a binary integer that indicates the actual size of the data field returned by the last FETCH operation. The value returned is the actual length of the field as stored in the database before it is moved to the user buffer where padding or truncation may take place. ORACLE suppresses leading zeros on numeric data and trailing blanks on character data before storing the fields in the database. This field is valid only if the DESCRIBE is issued after a FETCH call.

DESCRIBE Examples:

FORTRAN

Request a description of the first data field in the SELECT list in the SQL query statement referenced by the cursor CURS1. Specify the position as a literal. Return the maximum size of the field and the internal data type into the program variables SIZE and TYPE.

```
CALL ODSRBN(CURS1,1,SIZE,TYPE)
```

COBOL

Request a description of the data field in the SELECT list in the SQL statement referenced by cursor CURS1 whose number is given in variable POS.

```
CALL "ODSRBN" USING CURS1,POS,SIZE,TYPE,ACTSIZE.
```

MACRO-11 ASSEMBLY LANGUAGE

Describe the second data field in a SELECT list controlled by the cursor pointed to by register 0. Return the data fields maximum size and internal data type into the program variables SIZE and TYPE.

```
ODSRBN R0,#2,#SIZE,#TYPE
```

The NAME Call

CALL ONAME (cursor,position,tbuf,tbufl,cbuf,cbufl)

The NAME call is used to retrieve the names of the tables and columns used in a SELECT clause of a SQL program query. NAME operates positionally one field at a time, referencing each field or expression in a SELECT clause as if each were numbered sequentially from left to right beginning with 1.

NAME may be used after a SQL call to determine the table name, column name, or expression string of fields to be returned. If the requested field is an expression (e.g., SAL+COMM), then no table name can be returned, and the column name is the literal expression text.

If the user specifies a position number greater than the number of fields in the SELECT list, NAME returns an end-of-file indicator in the RETURN-CODE of the cursor data area. Programs can use the end-of-file status to dynamically determine the names of fields to be returned as the result of a query for which the number of fields is unknown, as in the case of SELECT *.

cursor	specifies the name of a 64 byte data area within the user program. The cursor data area contains status information on an active SQL operation. Each cursor defines an active SQL statement within the user program. The NAME call references the cursor name to associate a data field buffer with a specific SQL statement.
position	specifies the position of a field or expression in the SELECT clause of a SQL query statement. Fields are separated by commas and numbered left to right consecutively beginning with 1. The position number is specified as a binary integer. NAME uses the position number to relate buffers to fields in the SELECT list.

tbuf	specifies the location of the data field buffer within the user program where the name of the table of which the field is a column is to be stored. If tbuf is zero, then the table name will not be stored.
tbufl	specifies the location of a binary integer which indicates the length of tbuf. If the table name to be stored is longer than tbufl, then the table name will be truncated; if it is shorter, then tbufl will point to a binary integer which is the length of the table name stored in tbuf. If tbufl is zero, then the table name will not be stored.
cbuf	specifies the location of the data field buffer within the user program where the name of the column or expression is to be stored. If cbuf is zero, then the column name will not be stored.
cbufl	specifies the location of a binary integer which indicates the length of cbuf. If the column name to be stored is longer than cbufl, then the column name will be truncated; if it is shorter, then cbufl will point to a binary integer which is the length of the column name stored in cbuf. If cbufl is zero, then the column name will not be stored.

NAME Examples:

FORTRAN

Retrieve the table and column names for the second field in the SELECT list associated with the cursor CURS1.

```
CALL ONAME(CURS1,2,TABLE,TABLEL,COL,COLL)
```

COBOL

Retrieve the table and column names in the SELECT list defined by CURS4 which has its position number specified by a variable named SELPOS.

```
CALL "ONAME" USING CURS4,SELPOS,TABLE,TABLEL,COL,COLL.
```

MACRO-11 ASSEMBLY LANGUAGE

Retrieve the table and column names for the second field in the SELECT list associated with cursor CURS2.

```
CCALL ONAME,#CURS2,#2,#TABLE,#TABLEL,#COL,#COLL
```

The DEFINE Call

```
CALL ODFINN (cursor,pos,buffer,bufl[,ftype][,rcode][,fdig])
```

The DEFINE call is used to define one output buffer for each field in a SELECT list within a SQL program query.

DEFINE specifies the location and size of a data field buffer in the user program. Define also passes the external data type of the field as defined by the user program, and optionally specifies a field return code. DEFINE defines one data field buffer at a time. Buffers are defined after the SQL call and prior to the FETCH call.

SELECT buffers are always defined positionally. Fields within the SELECT list are referenced as if they were numbered consecutively, left to right, beginning with 1. During a FETCH, ORACLE will convert each field from internal to the specified external data type and then store the fields in the defined buffers.

ORACLE provides return code information at the row level, "cursor" return code, and optionally at the field level, "field" return code. During each FETCH, ORACLE establishes a return code for each field processed. This code indicates either successful completion (return code = 0) or an exceptional condition such as: null field fetched, field truncated, etc. The field return code is stored in the rcode variable for each defined field. At the completion of each FETCH, the last non-zero "field" return code encountered is placed in the "cursor" return code.

cursor — specifies the name of a 64 byte data area within the user program. The cursor data area contains status information on an active SQL operation. Each cursor defines an active SQL statement within the user program. The DEFINE call references the cursor name to associate a data field buffer with a specific SQL statement.

pos	specifies the position of a field or expression in the SELECT clause of a SQL query statement. Fields are separated by commas and numbered left to right consecutively beginning with 1. The position number is specified as a binary integer. DEFINE uses the position number to relate buffers to fields in the SELECT list.
buffer	specifies the location of the data field buffer within the user program.
bufl	specifies the length of the buffer being defined. The buffer length is specified as a binary integer.
ftype	is a binary integer that specifies the data type that the field is to be converted to before it is moved to the user buffer. If the ftype parameter is omitted no conversion takes place. A list of external data types and type codes is contained in the section on data types in this manual.
rcode	specifies a two byte binary field defined in the user's program into which ORACLE will place a field return code. Field return codes are filled in after a FETCH operation.
fdig	The number of fractional digits (to the right of the decimal point) to be returned for datatype 7 (COBOL implied decimal). fdig is required for datatype 7, and is ignored for all others.

DEFINE Examples:

FORTRAN

Define a data field buffer for the second field in the SELECT list associated with the cursor CURS1. The data field is to be fetched into a program variable named DEPT which is defined as INTEGER*2. ORACLE is to convert the field to integer external data type (3). At the completion of each fetch ORACLE will place a return code into the program variable RC2.

```
CALL ODFINN(CURS1,2,DEPT,2,3,RC2)
```

COBOL

Define a buffer for the field in the SELECT list defined by CURS4 which has its position number specified by a variable named SELPOS. The buffer is EMPNO with its length in variable EMPNOL. The data type is specified in the variable CBL and has FRAC fractional digits. The picture for EMPNO is S99999V99 USAGE IS DISPLAY.

```
MOVE 8 TO EMPNOL. MOVE 2 TO FRAC. CALL "ODFINN" USING
CURS4,SELPOS,EMPNO,EMPNOL,CBL,FRAC.
```

MACRO-11 ASSEMBLY LANGUAGE

Define a data field buffer for the second field in the SELECT list associated with cursor CURS2. The data field buffer in the user program is named DNAME and has a length of 20, specified as a literal. The field is to be returned to the program in ASCII format (1). After each fetch, ORACLE will place a return code for the data field into the program variable ERR1.

```
CCALL ODFINN,#CURS2,#2,NAME,#20.,#1,#ERR1
```

The BIND Call

```
CALL OBIND (cursor,sqlvar,sqlvl,progvar,progvl[,ftype])

CALL OBINDN (cursor,sqlvarnum,progvar,progvl[,ftype])
```

The BIND call is used to dynamically modify a SQL statement after it has been passed to ORACLE in a SQL call. The statement may then be executed, modified again, re-executed, etc.

The BIND call specifies that a program value is to be assigned to a SQL substitution variable within a SQL statement.

BIND and BINDN are exactly the same except in the way they reference SQL substitution variables. BINDN references SQL substitution variables numerically. BIND references SQL substitution variables symbolically by name. The name of the variable to be bound must be specified in upper case.

At the time of the BIND, ORACLE converts the program variable from external to internal format, and then moves the data value into the SQL statement. BIND is used after the SQL call and prior to the EXECUTE call. The completion status of the BIND is indicated in the RETURN CODE area of the cursor. All return codes are specified in the "Messages and Return Codes" section of this manual.

cursor	specifies the name of a 64 byte data area within the user program. The cursor data area contains status information on an active SQL operation. Each cursor defines an active SQL statement within the user program. BIND uses the cursor name to reference a specific SQL statement.
sqlvar	specifies the character string name of a substitution variable within the SQL statement, i.e. WHERE EMPNO = &EMPLOYEE. BIND moves the program variable value into the SQL substitution variable &EMPLOYEE.

sqlvl	specifies a binary integer indicating the length of the character string specified for the sqlvar parameter. For example, &EMPLOYEE has a length of 9. If the substitution variable was specified as a literal this parameter may be omitted.
sqlvarnum	specifies a binary integer that references a SQL substitution variable within the SQL referenced by the cursor. For example, if sqlvarnum contains the value 2, it references a SQL substitution variable defined as &2.
progvar	specifies the name of a variable defined within the user program. The value within the program variable is substituted into the SQL statement at the time of the BIND.
progvl	specifies a binary integer containing the length of the program variable. A length of zero indicates a null value is to be bound to the progvar parameter.
ftype	specifies a binary integer that indicates the data type of the program variable as it is defined within the user program. ORACLE converts the program variable from external to internal format before it is bound to the SQL statement. A list of external data types and type codes is contained in a separate section of this manual.

BIND Examples:

FORTRAN

Bind the value contained in the program variable DEPT to the SQL substitution variable &DNO. DEPT is defined in the user program as INTEGER*4.

```
CALL OBIND(CURS1,'&DNO',,DEPT,4,3)
```

Bind the value contained in the program variables EMPNO and DEPT to the SQL substitution variables &1 and &2

```
 CALL OSQL(CURS1,'SELECT EMPNO FROM EMP
1 WHERE EMPNO=&1 AND DEPTNO=&2')
 CALL OBINDN(CURS1,1,EMPNO,4,3)
 CALL OBINDN(CURS1,2,DEPT,4,3)
```

COBOL

Bind the value contained in program variable EMPNO to the SQL substitution variable which is specified in variable EMPNO-NAME which has a length specified in EMPNO-N-L.

```
CALL "OBIND" USING CURS1,EMPNO-NAME
  ,EMPNO-N-L,EMPNO,EMPNO-L,INT4.
```

MACRO-11 ASSEMBLY LANGUAGE

Bind the value contained in program variable DEPT to the SQL substitution variable pointed to by ADRDNO. ADRDNO is a program variable containing the 4 byte ASCII string &DNO. The data type of the value contained in DEPT is integer with a length of 4.

```
CCALL OBIND,R0,#ADRDNO,#4,EPT,#4,#3
```

EXECUTE Call

CALL OEXEC (cursor)

The EXECUTE call causes the SQL statement currently associated with the cursor to be processed.

If the SQL statement is a data manipulation, data definition, or data control statement, the entire SQL function is performed at this time; the RETURN CODE is set and a count of the rows processed by the statement is placed in ROW COUNT field of the cursor data area. If the SQL statement is a query, the user program must explicitly request each row of the result using the FETCH call.

cursor	specifies the name of a 64 byte data area within the user program. The cursor data area contains status information on an active SQL operation. Each cursor defines an active SQL statement within the user program. The EXECUTE call executes the SQL statement attached to the cursor.

EXECUTE Examples:

FORTRAN

Execute the SQL statement which was passed to ORACLE using CURS1.

```
CALL OEXEC(CURS1)
```

COBOL

Execute the SQL statement which was passed to ORACLE using CURS1.

```
CALL "OEXEC" USING CURS1.
```

MACRO-11 ASSEMBLY LANGUAGE

Execute the SQL statement which was passed to ORACLE using the cursor pointed to by register zero.

```
CCALL OEXEC,R0
```

The FETCH Call

CALL OFETCH (cursor)

The FETCH call returns rows of a query result to the user program, one row at a time. Each field of the query is placed into a buffer identified by a previously executed DEFINE call. Fields that are requested by the user program in character string format are left justified and padded with trailing blanks. Character strings that are too large for the field buffer are truncated and the ORACLE field return code is set to +3.

If null values are encountered in any field of the fetch, the ORACLE field return code for that field is set to +2 and the user buffer remains unchanged. To determine which specific fields are null or have been truncated, the user program must have specified field return codes in the DEFINE buffer calls, or a DESCRIBE call may be issued. If multiple non-zero field return codes are encountered in a single FETCH, the cursor return code will contain the last non-zero field return code.

After the last row of the query result has been returned to the user program, the next fetch will return an end-of-file return code of +4. After each FETCH the cursor row count is updated. When end-of-file has been reached, the row count will contain the total number of rows found by the query.

cursor	specifies the name of a 64 byte data area within the user program. The cursor data area contains status information on an active SQL operation. Each cursor defines an active SQL statement within the user program. The cursor maintains position on a set of rows that satisfy a query as those rows are retrieved, one at a time by the FETCH call.

FETCH Examples:

FORTRAN

Fetch a row for the SELECT statement passed to ORACLE using CURS1. CALL OFETCH(CURS2)

COBOL

Fetch a row for the SELECT statement passed to ORACLE using CURS1.

```
CALL "OFETCH" USING CURS1.
```

MACRO-11 ASSEMBLY LANGUAGE

Fetch a row for the SELECT statement passed to ORACLE using CURS2.

```
CCALL OFETCH,#CURS2
```

The CLOSE Call

CALL OCLOSE (cursor)

The CLOSE call disconnects a cursor from ORACLE and frees all resources obtained by the OPEN, SQL, and EXECUTE functions using this cursor. If the CLOSE fails, the RETURN CODE area of the cursor contains the status indicator. All of the return codes are listed in the "Messages and Return Codes" section of this manual.

cursor	specifies the name of a 64 byte data area within the user program. The cursor data area contains status information on an active SQL operation. Each cursor defines an active SQL statement within the user program. CLOSE disconnects the cursor from ORACLE.

CLOSE Examples:

FORTRAN

Close the cursor CURS1.

```
CALL OCLOSE(CURS1)
```

COBOL

Close the cursor CURS1.

```
CALL "OCLOSE" USING CURS1.
```

MACRO-11 ASSEMBLY LANGUAGE

Close the cursor pointed to by register zero.

```
OCLOSE R0
```

The LOGOFF Call

CALL OLOGOF (lda)

The LOGOFF call disconnects a program from ORACLE and frees all ORACLE resources owned by this program. If the LOGOFF fails, the reason is indicated in the first two bytes of the Logon Data Area (LDA). A complete list of return codes is given in the "Messages and Return Codes" section of this manual.

lda	specifies the name of the Logon Data Area specified in the LOGON call.

LOGOFF Examples:

FORTRAN

Log off from ORACLE.

```
CALL OLOGOF(LDAREA)
```

COBOL

Log off from ORACLE

```
CALL "OLOGOF" USING LDAREA.
```

MACRO-11 ASSEMBLY LANGUAGE

Log off from ORACLE.

```
CCALL OLOGOF,R2
```

SQL SUBSTITUTION VARIABLES

When SQL is used within a host programming language, substitution variables may be used within the SQL statement. SQL substitution variables allow user programs to dynamically modify SQL statements, and then execute those modified statements.

SQL substitution variables are identified by an ampersand. Substitution variables may be used anywhere in a SQL statement that a constant may be used. For example, in FORTRAN:

```
 CALL OSQL (CUR1,'SELECT ENAME,SAL
1 FROM EMP
2 WHERE DEPTNO = &DEPT;')
```

The BIND call is used to substitute values into a SQL substitution variable.

```
 CALL OBIND (CUR1,'&DEPT',5,DEPT,2,3)
```

DEPT is a variable defined in the user program as a 2 byte fixed point number.

SQL substitution variables may also be used to BIND values in INSERT and SET statements.

```
 CALL OSQL (CUR2,'INSERT INTO DEPT:
1<&A,&B,&C,NULL>;')
```

Null values may be inserted into the database by specifying NULL in the INSERT list or by binding a value with a zero length to a SQL substitution variable.

```
CALL OBIND (CUR2,&C,2,LOC,0,1)
```

O R A C L E

HOST LANGUAGE INTERFACE

DATA TYPES

ORACLE performs data conversions for most data types provided by the supported languages. On retrieval (SELECT) operations, ORACLE will convert from the internal format of the data as stored in the database, to an external format as defined by the user program. On storage (INSERT and UPDATE) operations, ORACLE will convert from external to internal data types.

The user specifies the external data type for SELECT operations with the DEFINE call. The user specifies external data types for INSERT and UPDATE operations with the BIND call.

Internally, ORACLE stores characters in ASCII and numbers in a variable length extended precision (maximum 22 bytes) floating point format.

If the user does not want ORACLE to do any conversion on numeric data, the data may be defined as a character string for both the internal and external data type.

The following is a list of the external data types supported by ORACLE:

DATA TYPE	CODE	FORTRAN	COBOL
Varying-length character string	01	LOGICAL*1	PIC X...X
ORACLE internal numeric	02	N/A	N/A
8 bit fixed point	03	LOGICAL*1	N/A
16 Bit fixed point	03	INTEGER*2	PIC S9(4) COMP
32 Bit fixed point	03	INTEGER*4	PIC S9(9) COMP
32 Bit floating point	04	REAL*4	N/A
64 Bit floating point	04	REAL*8	N/A
Null terminated string	05	LOGICAL*1	PIC X...X
Raw data	06	LOGICAL*1	PIC X...X
COBOL implied decimal	07	N/A	PIC S9V9

O R A C L E

HOST LANGUAGE INTERFACE

DATA TYPE DESCRIPTIONS

The use of each of the ORACLE external data types is described below.

DATA TYPE	DESCRIPTION
01	The varying length character string format is a string of ASCII characters whose length is determined by a length field. Trailing blank characters are discarded on input. ORACLE pads the string with trailing blanks on output. If the length specification is missing on input, the string length is determined by scanning the string until a null character (a zero byte) is encountered. The length is required on output. An all blank field, or one whose length is specified as zero on input is treated as a NULL by ORACLE if the internal datatype is CHAR; if the internal type is NUMBER, then an all blank field is converted to zero. A zero length output length specification is invalid. When the ORACLE internal data type is NUMBER, input character strings are converted to internal numeric format until an invalid numeric character is encountered, e.g., '1234.45bcd' is converted to 1234.45, and the 'bcd' is ignored without any error indication. Output to an ASCII buffer from an internal numeric datatype to a character string which contains the ascii character representation of the internal number. The field width determines the precision. The number will be converted to scientific notation if required by the field size., e.g., if the number is 123456789 and the field width is 6, the output string will be '1.2E08'.
02	See the section on internal numeric format.

03 The integer number format is used to process numbers which have no fractional parts. The integer number is a signed binary number of one, two, or four bytes. The significance order is determined by the host language being used. The length specification is required for input and output. If the number being output from ORACLE is not integral, the fractional part is discarded. Integer numbers may be used only with internal numeric columns.

04 The floating number format is used to process numbers which have fractional parts, or which exceed the capacity of integer number format. The number is represented using the computer's floating point format with a length of either 4 or 8 bytes (REAL*4 or REAL*8). The length specification is required for input and output. Since the internal numeric format is decimal based, some precision may be lost during the conversion from the computer's binary floating point format to and from ORACLE's decimal format.

05 The null terminated string format is exactly like the varying length character string format, except that the string is always terminated by one byte of zero (the NULL character). On input, the string length is ignored and the string scanned for the null. On output, the null is placed after the last character returned. If the string exceeds the field length specified, the string is truncated and the last character position of the buffer contains the null. Trailing blanks are discarded on input. A zero length string (null in the first position or an all blank field) is treated as NULL by ORACLE.

06 The raw data format is used for binary data. The contents of the buffer are not interpreted in any way by ORACLE for either input or output. The length is required for input and output. On output, only the number of characters stored in the database are returned; the remainder of the output buffer is not modified. The number of characters actually returned may be determined using the DESCRIBE call.

07	The COBOL implied decimal data type is used to return non-integral numbers from ORACLE to a COBOL data type which is suitable for calculation. The COBOL data area must be a signed numeric display field with an implied decimal point. The number of digits to the right of the decimal point is specified with the DEFINE call. The value returned may be used as is for COBOL calculations, or may be moved to a computational field prior to calculations. The number will never be converted to scientific notation. If the number to be returned loses significant digits during the conversion, ORACLE fills the buffer with "*" characters.

DATA CONVERSIONS

The following table specifies the data conversions supported by ORACLE.

\ SYSTEM FORMAT / USER FORMAT	TO ORACLE: CHAR	TO ORACLE: NUMBR	FROM ORACLE: CHAR	FROM ORACLE: NUMBR
ASCII	YES	YES	YES	YES
INTERNAL NUMERIC	YES	YES	YES	YES
8 BIT FIXED PT.	-	YES	-	YES
16 BIT FIXED PT.	-	YES	-	YES
32 BIT FIXED PT.	-	YES	-	YES
32 BIT FLOATING	-	YES	-	YES
64 BIT FLOATING	-	YES	-	YES
NULL TERM. STR.	YES	YES	YES	YES
RAW DATA	YES	YES	YES	YES
COBOL IMPLIED	NO	NO	NO	YES*

* NOTE: The COBOL 'number' datatype must have a picture of the following form:
PICTURE S9(N)V9(N) USAGE DISPLAY SIGN LEADING SEPARATE.

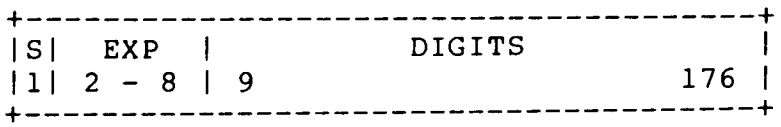
O R A C L E

HOST LANGUAGE INTERFACE

INTERNAL NUMERIC FORMAT

Database fields defined as NUMBER in the CREATE TABLE are stored in the database in ORACLE's variable length extended precision floating point format.

ORACLE floating point numbers vary in length and occupy from 1 byte to 22 bytes of real storage. The ORACLE internal numeric format is depicted below:



The number consists of a sign bit (0 is negative, 1 is positive), a 7 bit exponent and up to 21 bytes of significant digits.

The exponent represents the number of digit positions to shift the radix point (which is assumed to be to the left of the leftmost digit). The exponent is expressed in excess 64 format, so that an exponent of 64 indicates a 0 shift, 63 a shift to the left of one digit, 65 a shift to the right of one digit, and so on.

Each digit is stored in one byte and is a base 100 digit represented as a binary number from 0 to 99. Each shift of the radix point changes the magnitude of the number by a power of 100. Trailing zero digits are discarded. Negative numbers are indicated by a sign bit of zero and the digits are stored as the 100's complement of the number.

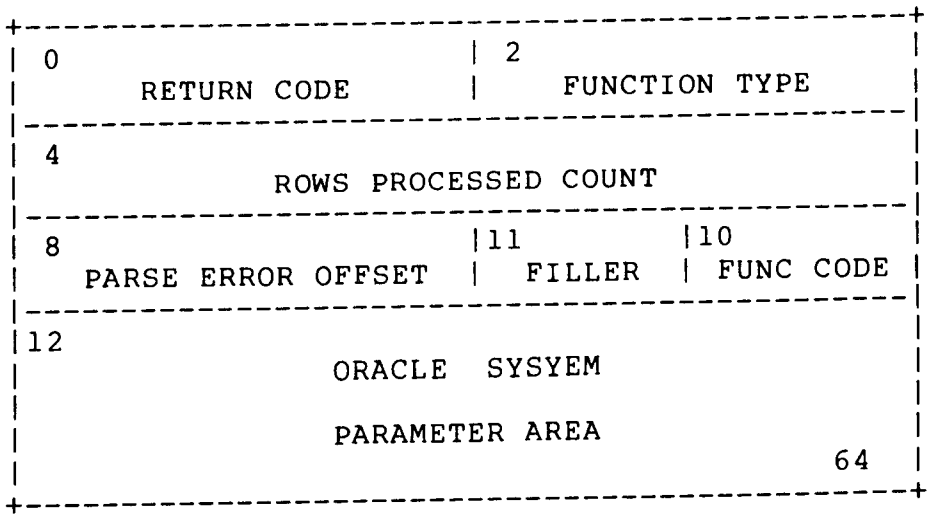
The ORACLE numeric format can represent numbers ranging from 10 ** -128 to (10 ** 128) - 1 with up to 42 digits of precision.

O R A C L E

HOST LANGUAGE INTERFACE

CURSOR DATA AREA

The cursor is a 64 byte data area defined within a user program. A cursor is identified to ORACLE in an OPEN call. The cursor data area contains status information on an active SQL operation. Each cursor defines an active SQL statement within a user program. All ORACLE calls that reference a SQL statement reference it by cursor name. The cursor format is depicted below:



RETURN CODE — contains a two byte binary number that indicates the completion code for the specified operation. Zero indicates a successful result. A positive return code indicates a successful result with an exceptional condition. A negative return code indicates an error was encountered in attempting to perform the specified operation. See ORACLE Messages and Codes for a complete list of return codes.

FUNCTION CODE	contains an operation code indicating the type of ORACLE function requested. The function codes are: 02 - SQL 04 - EXECUTE 06 - BIND 08 - DEFINE 10 - DESCRIBE 12 - FETCH 14 - OPEN 16 - CLOSE 18 - BINBN
ROWS PROCESSED COUNT	contains a four byte binary number indicating the count of the number of rows processed by a SQL operation. The count will contain the number of rows inserted, updated, or deleted by a data manipulation statement, or the number of rows fetched in a query statement. This field is valid only after an EXECUTE or FETCH operation.
PARSE ERROR OFFSET	contains a two byte binary number indicating the offset in characters into the SQL text where the parse error occured.

O R A C L E

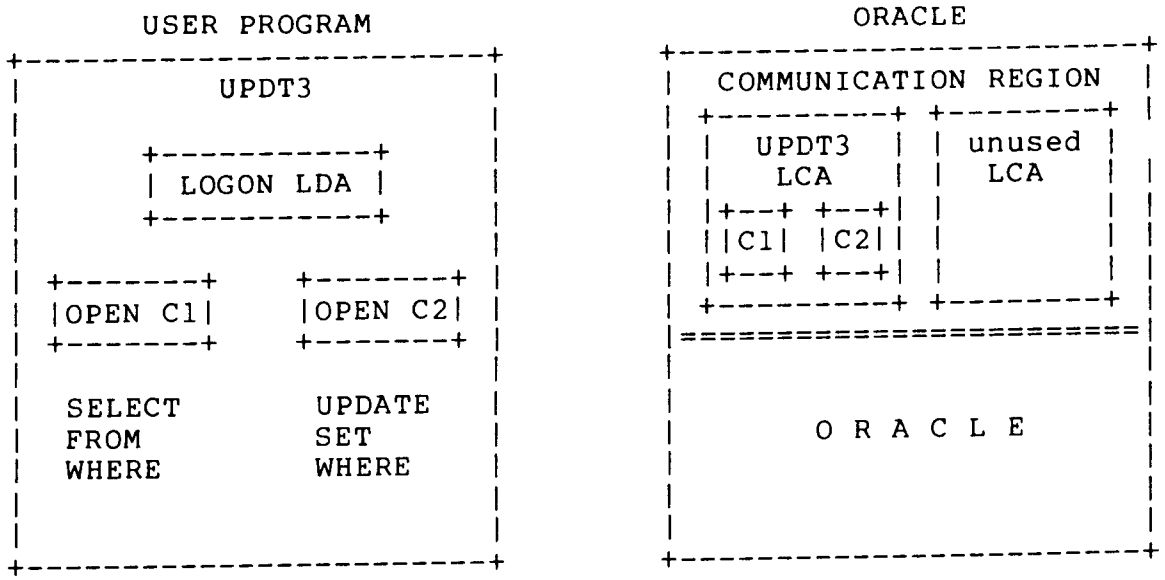
HOST PROGRAM INTERFACE

PROGRAM INTERFACE DATA AREAS

ORACLE allows a single program to have multiple cursors open at the same time.

To optimize program performance, it is useful to have an understanding of the communication between ORACLE and a user program.

The following is a diagram of a user program named UPDT3 with two open cursors.



When program UPDT3 issues the LOGON call, ORACLE allocates an Logon Control Area (LCA) for UPDT3 in the ORACLE Communication Region. ORACLE connects this LCA to the Logon Data Area (LDA) defined within UPDT3. ORACLE will allocate one and only one LCA for each terminal and program currently logged on to ORACLE.

When program UPDT3 issues an OPEN call, ORACLE allocates a SQL Work Area (SWA) for UPDT3 in the ORACLE Communication Region. ORACLE connects the SWA to the Cursor (C1) defined within the UPDT3.

When program UPDT3 issues a second OPEN call, ORACLE allocates a second SWA and connects it to UPDT3's second cursor (C2). ORACLE will allocate one SWA for every open cursor.

If a program opens multiple cursors causing ORACLE to allocate multiple SWA's to that program, some of that program's SWA's may be swapped to disk. However, ORACLE will maintain at least one SWA in main memory for each program or terminal logged on to ORACLE.

The default number of SWA's to be maintained in memory for a given program is one, unless the user specifies a different number in the optional "areacount" parameter in the LOGON call.

The default size of each SWA is 3K bytes. The user can override the default by specifying the "areasize" parameter of the OPEN call. The SWA must be large enough to contain the compiled SQL statement plus one row of data of the table or view being processed.

```
*****************************************
*                                       *
*            F O R T R A N              *
*                                       *
*   E X A M P L E   P R O G R A M S     *
*                                       *
*****************************************
```

Table of Contents

```
      PROGRAM SAMPL1
C
C     SAMPL1 IS WRITTEN FOR THE VAX FORTRAN COMPILERS. NOTE
C     THE %REF IS USED WHENEVER A LITERAL STRING IS PASSED
C     TO ORACLE.
C
C     SAMPL1 IS A SIMPLE EXAMPLE PROGRAM WHICH ADDS NEW EMPLOYEES
C     ROWS TO THE PERSONNEL DATA BASE.  VERY LITTLE CHECKING IS
C     DONE TO INSURE THE INTEGRITY OF THE DATA BASE.  THE PROGRAM
C     QUERIES THE USER (VIA DEVICE 5) FOR DATA AS FOLLOWS:
C
C        Enter employee number:
C        Enter employee name  :
C        Enter employee job   :
C        Enter employee salary:
C        Enter employee dept  :
C
C     THE NEW EMPLOYEE ROW IS INSERTED AND THE DEPARTMENT
C     TABLE IS UPDATED TO INCREASE THE EMPLOYEE COUNT.  IF
C     THE EMPLOYEE NUMBER IS ENTERED AS '0', THEN THE
C     PROGRAM TERMINATES.
C
      IMPLICIT INTEGER*2 (A-Z)
      LOGICAL*1 ENAME(10),JOB(9)
      INTEGER*2 LDA(32),CUR1(32),CUR2(32)
C
C     LOGON TO ORACLE
C
      CALL OLOGON(LDA)
      IF (LDA(1).NE.0) GO TO 10000
C
C     OPEN TWO CURSORS FOR THE PERSONNEL DATA BASE
C

      CALL OOPEN(CUR1,LDA,
     X   %REF('PERSONNEL'),,,%REF('QA/TEST'))
      IF (CUR1(1).NE.0) GO TO 10000

      CALL OOPEN(CUR2,LDA,
     X   %REF('PERSONNEL'),,,%REF('QA/TEST'))
      IF (CUR2(1).NE.0) GO TO 10000
C
C     PASS THE SQL STATEMENTS TO ORACLE
C
      CALL OSQL(CUR1,
     1   %REF('INSERT INTO EMP(EMPNO,ENAME,JOB,SAL,DEPTNO)
     2   <&EMPNO,&ENAME,&JOB,&SAL,&DEPTNO>;'))
      IF (CUR1(1).NE.0) GO TO 10000
```

```
C
C     NOTE THAT THE 'NULLF' FUNCTION FORCES THE EMPCNT TO ZERO
C     IF IT IS NULL
C
      CALL OSQL(CUR2,
     1   %REF('UPDATE DEPT SET EMPCNT=NULLF(EMPCNT,0)+1
     2   WHERE DEPTNO=&DEPTNO;'))
      IF (CUR2(1).NE.0) GO TO 10000

C
C READ THE USER'S INPUT FROM DEVICE 5 (NORMALLY, THE TERMINAL)
C
10    WRITE(5,100)
100   FORMAT('$Enter employee number: ')
      READ (5,110,END=5000,ERR=10)EMPNO
110   FORMAT(I5)
      IF (EMPNO.EQ.0) GO TO 5000
      WRITE (5,120)
120   FORMAT('$Enter employee name  : ')
      READ (5,130)ENAME
130   FORMAT(10A1)
      WRITE (5,140)
140   FORMAT('$Enter employee job   : ')
      READ (5,150)JOB
150   FORMAT(9A1)
153   WRITE (5,155)
155   FORMAT('$Enter employee salary: ')
      READ (5,110,ERR=153)SAL
158   WRITE (5,160)
160   FORMAT('$Enter employee dept  : ')
      READ (5,110,ERR=158)DEPT
C
C     BIND ALL SQL SUBSTITUTION VARIABLE VALUES.
C     IF ANY ERRORS OCCUR, PRINT AN ERROR MESSAGE,
C     BUT CONTINUE.
C
      CALL OBIND(CUR1,%REF('&EMPNO'),,EMPNO,2,3)
      IF (CUR1(1).NE.0) GO TO 1000
      CALL OBIND(CUR1,%REF('&ENAME'),,ENAME,10,1)
      IF (CUR1(1).NE.0) GO TO 1000
      CALL OBIND(CUR1,%REF('&JOB'),,JOB,9,1)
      IF (CUR1(1).NE.0) GO TO 1000
      CALL OBIND(CUR1,%REF('&SAL'),,SAL,2,3)
      IF (CUR1(1).NE.0) GO TO 1000
      CALL OBIND(CUR1,%REF('&DEPTNO'),,DEPT,2,3)
      IF (CUR1(1).NE.0) GO TO 1000
      CALL OBIND(CUR2,%REF('&DEPTNO'),,DEPT,2,3)
      IF (CUR2(1).NE.0) GO TO 1000
```

```
C
C     EXECUTE THE SQL STATEMENTS: CUR1 INSERTS A ROW INTO THE
C     'EMP' TABLE.
C
      CALL OEXEC(CUR1)
      IF (CUR1(1).NE.0) GO TO 1000
C
C     ...CUR2 UPDATES THE 'EMPCNT' COLUMN OF THE 'DEPT' TABLE.
C
      CALL OEXEC(CUR2)
      IF (CUR2(1).NE.0) GO TO 1000
      GO TO 10
1000  CALL ERRRPT(LDA,CUR1,CUR2)
      GO TO 10
C
C     CLOSE THE TWO CURSORS
C
5000  CALL OCLOSE(CUR1)
      CALL OCLOSE(CUR2)
C
C     LOGOFF FROM ORACLE
C
      CALL OLOGOF(LDA)
      STOP 'END OF SAMPL1'
10000 CALL ERRRPT(LDA,CUR1,CUR2)
      GO TO 5000
      END

      SUBROUTINE ERRRPT(LDA,C1,C2)
C
C     ERRRPT PRINTS THE CURSOR NUMBER, THE ERROR CODE, AND THE
C     ORACLE FUNCTION CODE.  IF THE LDA CONTAINS AN ERROR CODE,
C     A LOGON ERROR IS ASSUMED
C
C     LDA IS THE LOGON DATA AREA ARRAY
C     C1  IS THE FIRST CURSOR ARRAY
C     C2  IS THE SECOND CURSOR ARRAY
```

```
C
      INTEGER*2 LDA(32),C1(32),C2(32)

      IF (LDA(1).EQ.0) GO TO 100
      WRITE (5,10)LDA(1)
10    FORMAT('0Logon error: ',I5)
      GO TO 500
100   IF (C1(1).EQ.0) GO TO 200
      WRITE (5,110) 1,C1(1),C1(6)
110   FORMAT('0ORACLE error on cursor ',I1,
     1 ': CODE IS ',I5,', OP IS ',I5)
      GO TO 500
200   IF (C2(1).EQ.0) GO TO 300
      WRITE (5,110),2,C2(1),C2(6)
      GO TO 500
300   WRITE (5,310)
310   FORMAT('0Unknown ORACLE error')
500   RETURN
      END
      PROGRAM SAMPL2
C
C     SAMPL2 IS A SIMPLE EXAMPLE PROGRAM WHICH ADDS NEW EMPLOYEE
C     ROWS TO THE PERSONNEL DATA BASE.  SOME CHECKING
C     IS DONE TO INSURE THE INTEGRITY OF THE DATA BASE.
C     THE PROGRAM QUERIES THE USER (VIA DEVICE 5) FOR DATA AS FOL
C
C        Enter employee number:
C        Enter employee name  :
C        Enter employee job   :
C        Enter employee salary:
C        Enter employee dept  :
C
C     THE NEW EMPLOYEE ROW IS INSERTED AND THE DEPARTMENT
C     TABLE IS UPDATED TO INCREASE THE EMPLOYEE COUNT.  IF
C     THE EMPLOYEE NUMBER IS ENTERED AS '0', THEN THE
C     PROGRAM TERMINATES.
C
      IMPLICIT INTEGER*2 (A-Z)
      LOGICAL*1 ENAME(11),JOB(10),DEPT(20)
      INTEGER*2 CURS(32,4)
C
C     LOGON TO ORACLE
C
      CALL OLOGON(CURS(1,1))
      IF (CURS(1,1).NE.0) GO TO 10000
C
C     OPEN THREE CURSORS FOR THE PERSONNEL DATA BASE
C
      CALL OOPEN(CURS(1,2),CURS(1,1),'PERSONNEL',,,'QA/TEST')
      IF (CURS(1,2).NE.0) GO TO 10000

      CALL OOPEN(CURS(1,3),CURS(1,1),'PERSONNEL',,,'QA/TEST')
      IF (CURS(1,3).NE.0) GO TO 10000
```

```
      CALL OOPEN(CURS(1,4),CURS(1,1),'PERSONNEL',,,'QA/TEST')
      IF (CURS(1,4).NE.0) GO TO 10000
C
C     PASS THE SQL STATEMENTS TO ORACLE
C
      CALL OSQL(CURS(1,2),'INSERT INTO EMP(EMPNO,ENAME,JOB,SAL,DE
     1   <&EMPNO,&ENAME,&JOB,&SAL,&DEPTNO>;')
      IF (CURS(1,2).NE.0) GO TO 10000
C
C     NOTE THAT THE 'NULLF' FUNCTION FORCES THE EMPCNT TO ZERO
C     IF IT IS NULL
```

```
C
      CALL OSQL(CURS(1,3),'UPDATE DEPT SET EMPCNT=NULLF(EMPCNT,0)
     1   WHERE DEPTNO=&DEPTNO;')
      IF (CURS(1,3).NE.0) GO TO 10000

      CALL OSQL(CURS(1,4),'SELECT DNAME FROM DEPT WHERE
     1   DEPTNO=&DEPTNO;')
      IF (CURS(1,4).NE.0) GO TO 10000
C
C     DEFINE A BUFFER TO RECEIVE THE DEPARTMENT NAME FOR ORACLE
C
      CALL ODFINN(CURS(1,4),1,DEPT,20,5)
      IF (CURS(1,4).NE.0) GO TO 10000

C
C READ THE USER'S INPUT FROM DEVICE 5 (NORMALLY, THE TERMINAL)
C
10    WRITE(5,100)
100   FORMAT('$Enter employee number: ')
      READ (5,110,END=5000,ERR=10)EMPNO
110   FORMAT(I5)
      IF (EMPNO.EQ.0) GO TO 5000
      WRITE (5,120)
120   FORMAT('$Enter employee name  : ')
      DO 125 I=1,11
125   ENAME(I)=0
      READ (5,130)ENAME
130   FORMAT(11A1)
      WRITE (5,140)
140   FORMAT('$Enter employee job   : ')
      DO 145 I=1,10
145   JOB(I)=0
      READ (5,150)JOB
150   FORMAT(10A1)
153   WRITE (5,155)
155   FORMAT('$Enter employee salary: ')
      READ (5,110,ERR=153)SAL
158   WRITE (5,160)
160   FORMAT('$Enter employee dept  : ')
      READ (5,110,ERR=158)DEPTNO
C
C     BIND ALL SQL SUBSTITUTION VARIABLE VALUES
C     IF ANY ERRORS OCCUR, PRINT AN ERROR MESSAGE,
C     BUT CONTINUE.
```

```
C
      CALL OBIND(CURS(1,2),'&EMPNO',,EMPNO,2,3)
      IF (CURS(1,2).NE.0) GO TO 1000
      CALL OBIND(CURS(1,2),'&ENAME',,ENAME,10,5)
      IF (CURS(1,2).NE.0) GO TO 1000
      CALL OBIND(CURS(1,2),'&JOB',,JOB,9,5)
      IF (CURS(1,2).NE.0) GO TO 1000
      CALL OBIND(CURS(1,2),'&SAL',,SAL,2,3)
      IF (CURS(1,2).NE.0) GO TO 1000
      CALL OBIND(CURS(1,2),'&DEPTNO',,DEPTNO,2,3)
      IF (CURS(1,2).NE.0) GO TO 1000
      CALL OBIND(CURS(1,3),'&DEPTNO',,DEPTNO,2,3)
      IF (CURS(1,3).NE.0) GO TO 1000
      CALL OBIND(CURS(1,4),'&DEPTNO',,DEPTNO,2,3)
      IF (CURS(1,4).NE.0) GO TO 1000
C
C     EXECUTE THE SQL STATEMENTS.  CURSOR NUMBER 3 SELECTS
C     'DNAME' FROM THE 'DEPT' TABLE.  IF THERE IS NO SUCH DEPARTM
C     (AS DETECTED BY A RETURN CODE OF 4 TO THE FETCH CALL), THEN
C     AN ERROR MESSAGE IS DISPLAYED.
      CALL OEXEC(CURS(1,4))
      IF (CURS(1,4).NE.0) GO TO 1000
      DO 450 I=1,20
450   DEPT(I) = 0
      CALL OFETCH(CURS(1,4))
      IF (CURS(1,4).EQ.0) GO TO 500
      IF (CURS(1,4).NE.4) GO TO 1000
      WRITE (5,400)
400   FORMAT('0No such department number')
      GO TO 10
C
C     CURSOR NUMBER 1 INSERTS A NEW ROW INTO THE 'EMP' TABLE.
C
500   CALL OEXEC(CURS(1,2))
      IF (CURS(1,2).NE.0) GO TO 1000
C
C     CURSOR NUMBER 2 UPDATE THE 'EMPCNT' COLUMNS IN THE 'DEPT' T
C
      CALL OEXEC(CURS(1,3))
      IF (CURS(1,3).NE.0) GO TO 1000
      WRITE (5,600)ENAME,DEPT
600   FORMAT(' ',11A1,' added to the ',20A1,' department')
      GO TO 10
1000  CALL ERRRPT(CURS(1,1),4)
      GO TO 10
```

```
C
C CLOSE THE THREE CURSORS
C
5000  CALL OCLOSE(CURS(1,2))
      CALL OCLOSE(CURS(1,3))
      CALL OCLOSE(CURS(1,4))
C
C     LOGOFF FROM ORACLE
C
      CALL OLOGOF(CURS(1,1))
      STOP 'END OF SAMPL2'
10000 CALL ERRRPT(CURS(1,1),4)
      GO TO 5000
      END

      SUBROUTINE ERRRPT(CURS,N)
C
C     ERRRPT PRINTS THE CURSOR NUMBER, THE ERROR CODE, AND THE
C     ORACLE FUNCTION CODE.  IF THE LDA CONTAINS AN ERROR CODE,
C     A LOGON ERROR IS ASSUMED
C
C     CURS IS THE CURSOR ARRAY
C     N IS THE NUMBER OF CURSORS (INCLUDING THE LDA) IN THE ARRAY
C
      INTEGER*2 CURS(32,N)

      IF (CURS(1,1).EQ.0) GO TO 100
      WRITE (5,10)CURS(1,1)
10    FORMAT('0Logon error: ',I5)
      GO TO 500
100   DO 110 I=1,N
      IF (CURS(1,I).NE.0) GO TO 150
110   CONTINUE
150   IF (I.LE.N) GO TO 300
      WRITE (5,200)
200   FORMAT('0Unknown ORACLE error')
      GO TO 500
300   WRITE (5,400)I,CURS(1,I),CURS(6,I)
400   FORMAT('0ORACLE error on cursor ',I1,
     1   ': code is ',I5,', op is ',I5)
500   RETURN
      END
```

```
      PROGRAM SAMPL3
C
C     SAMPL3 IS A SIMPLE EXAMPLE PROGRAM WHICH ADDS NEW EMPLOYEE
C     ROWS TO THE PERSONNEL DATA BASE.  CHECKING
C     IS DONE TO INSURE THE INTEGRITY OF THE DATA BASE.
C     THE EMPLOYEE NUMBERS ARE AUTOMATICALLY SELECTED USING
C     THE CURRENT MAXIMUM EMPLOYEE NUMBER AS THE START.
C     IF ANY EMPLOYEE NUMBER IS A DUPLICATE, IT IS SKIPPED.
C     THE PROGRAM QUERIES THE USER (VIA DEVICE 5) FOR DATA AS FOL
C
C        Enter employee name  :
C        Enter employee job   :
C        Enter employee salary:
C        Enter employee dept  :
C
C     THE NEW EMPLOYEE ROW IS INSERTED AND THE DEPARTMENT
C     TABLE IS UPDATED TO INCREASE THE EMPLOYEE COUNT.  IF
C     THE EMPLOYEE NAME IS NOT ENTERED, THEN THE PROGRAM
C     TERMINATES.
C
C     IF THE ROW IS SUCCESSFULLY INSERTED, THE FOLLOWING
C     IS PRINTED:
C
C     ENAME added to DNAME department as employee # NNNNN
C
      IMPLICIT INTEGER*4 (A-Z)
C
C     THE MAXIMUM LENGTHS OF THE 'ENAME', 'JOB', AND 'DNAME'
C     COLUMNS WILL BE DETERMINED BY AN ORACLE CALL.  THE
C     PROGRAM ASSUMES THAT THE SUM OF THE LENGTHS WILL BE
C     LESS THAN 100 BYTES.  THE COLUMNS WILL ALL BE STORED
C     IN ONE ARRAY -- STRNGS.
C
      LOGICAL*1 STRNGS(100),ENMFMT(6),JOBFMT(6),DEPFMT(70)
      INTEGER*2 CURS(32,6)
C
C     LOGON TO ORACLE
C
      CALL OLOGON(CURS(1,1))
      IF (CURS(1,1).NE.0) GO TO 10000
C
C     OPEN FIVE CURSORS FOR THE PERSONNEL DATA BASE
```

```
C
      CALL OOPEN(CURS(1,2),CURS(1,1),'PERSONNEL',,,'QA/TEST')
      IF (CURS(1,2).NE.0) GO TO 10000

      CALL OOPEN(CURS(1,3),CURS(1,1),'PERSONNEL',,,'QA/TEST')
      IF (CURS(1,3).NE.0) GO TO 10000

      CALL OOPEN(CURS(1,4),CURS(1,1),'PERSONNEL',,,'QA/TEST')
      IF (CURS(1,4).NE.0) GO TO 10000

      CALL OOPEN(CURS(1,5),CURS(1,1),'PERSONNEL',,,'QA/TEST')
      IF (CURS(1,5).NE.0) GO TO 10000

      CALL OOPEN(CURS(1,6),CURS(1,1),'PERSONNEL',,,'QA/TEST')
      IF (CURS(1,6).NE.0) GO TO 10000
C
C     RETRIEVE THE CURRENT MAXIMUM EMPLOYEE NUMBER
C
C     PASS THE SQL STATEMENT TO ORACLE
C
      CALL OSQL(CURS(1,2),'SELECT MAX(EMPNO) + 10 FROM EMP;')
      IF (CURS(1,2).NE.0) GO TO 10000
C
C     DEFINE A BUFFER TO RECEIVE THE MAX(EMPNO)+10 FROM ORACLE
C
      CALL ODFINN(CURS(1,2),1,EMPNO,4,3)
      IF (CURS(1,2).NE.0) GO TO 10000
C
C     EXECUTE THE SQL STATEMENT
C
      CALL OEXEC(CURS(1,2))
      IF (CURS(1,2).NE.0) GO TO 10000
C
C     FETCH THE DATA FROM ORACLE INTO THE DEFINED BUFFER
C
      CALL OFETCH(CURS(1,2))
      IF (CURS(1,2).EQ.0) GO TO 50
      IF (CURS(1,2).NE.4) GO TO 10000
C
C     CURSOR RETURN CODE 4 MEANS THAT NO ROW SATISFIED THE QUERY,
C     SO GENERATE THE FIRST EMPNO
C
      EMPNO=10
50    CONTINUE
C
C     DETERMINE THE MAX LENGTH OF THE EMPLOYEE NAME AND JOB TITLE
C
C     PASS THE SQL STATEMENT TO ORACLE.  IT WILL NOT BE EXECUTED.
```

```
C
      CALL OSQL(CURS(1,2),'SELECT ENAME,JOB FROM EMP;')
      IF (CURS(1,2).NE.0) GO TO 10000
C
C     CALL ORACLE TO DESCRIBE THE TWO FIELDS SPECIFIED IN THE ABO
C     SQL STATEMENT.  WE ARE ONLY CONCERNED ABOUT THE LENGTH.
C
      CALL ODSRBN(CURS(1,2),1,ENAMEL)
      IF (CURS(1,2).NE.0) GO TO 10000

      CALL ODSRBN(CURS(1,2),2,JOBL)
      IF (CURS(1,2).NE.0) GO TO 10000
C
C     PUT THE LENGTHS INTO THE FORMATS SO THAT THE ENAME AND JOB
C     COLUMNS WILL BE PRINTED CORRECTLY.
C
      ENCODE(6,60,ENMFMT)ENAMEL
60    FORMAT('(',I2,'A1)')
      ENCODE(6,60,JOBFMT)JOBL
C
C     PASS THE SQL STATEMENTS TO ORACLE
C
      CALL OSQL(CURS(1,2),'INSERT INTO EMP(EMPNO,ENAME,JOB,SAL,DE
     1   <&EMPNO,&ENAME,&JOB,&SAL,&DEPTNO>;')
      IF (CURS(1,2).NE.0) GO TO 10000

      CALL OSQL(CURS(1,3),'UPDATE DEPT SET EMPCNT=NULLF(EMPCNT,0)
     1   WHERE DEPTNO=&DEPTNO;')
      IF (CURS(1,3).NE.0) GO TO 10000
      CALL OSQL(CURS(1,4),'SELECT DNAME FROM DEPT WHERE
     1   DEPTNO=&DEPTNO;')
      IF (CURS(1,4).NE.0) GO TO 10000

      CALL OSQL(CURS(1,5),'BEGIN TRANSACTION 1 ON TABLE EMP,DEPT
     1   UPDATE;')
      IF (CURS(1,5).NE.0) GO TO 10000

      CALL OSQL(CURS(1,6),'END TRANSACTION 1;')
      IF (CURS(1,6).NE.0) GO TO 10000
C
C     CALL ORACLE TO DESCRIBE THE 'DNAME' COLUMNS - ONLY THE LENG
C     IS OF CONCERN
C
      CALL ODSRBN(CURS(1,4),1,DEPTL)
      IF (CURS(1,4).NE.0) GO TO 10000
C
C     PUT THE MAXIMUM 'DNAME' LENGTH INTO A FORMAT SO THAT IT WIL
C     BE PRINTED CORRECTLY.
C
      ENCODE(70,70,DEPFMT)ENAMEL,DEPTL
70    FORMAT('('' '',',I2,'A1,'' added to the '',',I2,'A1,''
     1 department as employee # '',I5)')
```

```
C
C     DEFINE THE BUFFER TO RECEIVE 'DNAME' FOR ORACLE
C
      CALL ODFINN(CURS(1,4),1,STRNGS(ENAMEL+JOBL+4),DEPTL,5)
      IF (CURS(1,4).NE.0) GO TO 10000

C
C READ THE USER'S INPUT FROM DEVICE 5
C
100   WRITE (5,120)
120   FORMAT('$Enter employee name  : ')
      READ (5,ENMFMT,END=1000)(STRNGS(J),J=1,ENAMEL)
      IF (STRNGS(1).EQ.' ') GO TO 1000
      WRITE (5,140)
140   FORMAT('$Enter employee job   : ')
      READ (5,JOBFMT)(STRNGS(J),J=ENAMEL+2,ENAMEL+2+JOBL-1)
153   WRITE (5,155)
155   FORMAT('$Enter employee salary: ')
      READ (5,158,ERR=153)SAL
158   FORMAT(I5)
300   WRITE (5,160)
160   FORMAT('$Enter employee dept  : ')
      READ (5,158,ERR=300)DEPTNO
C
C     BIND THE DEPTNO VARIABLE
C
      CALL OBIND(CURS(1,4),'&DEPTNO',,DEPTNO,4,3)
      IF (CURS(1,4).NE.0) GO TO 700
C
C     EXECUTE THE SQL STATEMENT
C
      CALL OEXEC(CURS(1,4))
      IF (CURS(1,4).NE.0) GO TO 700
C
C     FETCH THE ROWS:  DEPTNO IS A UNIQUE COLUMN, SO A MAXIMUM OF
C     ONE ROW WILL BE FETCHED.  IF CURSOR RETURN CODE 4 IS
C     RETURNED, THEN THERE IS NO SUCH DEPARTMENT.
C
C     NOTE THAT THE DNAME AREA OF STRNGS WILL BE SET TO ALL NULLS
C     PRIOR TO THE CALL TO ORACLE
C
      DO 390 I=ENAMEL+JOBL+4,ENAMEL+JOBL+4+DEPTL+2
390   STRNGS(I)=0
      CALL OFETCH(CURS(1,4))
      IF (CURS(1,4).EQ.0) GO TO 410
      IF (CURS(1,4).NE.4) GO TO 700
      WRITE (5,400)
400   FORMAT('0No such department number')
      GO TO 300
```

```
C
C     BIND ALL SQL SUBSTITUTION VARIABLE VALUES
C     IF ANY ERRORS OCCUR, PRINT AN ERROR MESSAGE,
C     BUT CONTINUE.
C
410   CALL OBIND(CURS(1,2),'&ENAME',,STRNGS(1),ENAMEL,1)
      IF (CURS(1,2).NE.0) GO TO 700
      CALL OBIND(CURS(1,2),'&JOB',,STRNGS(ENAMEL+2),JOBL,1)
      IF (CURS(1,2).NE.0) GO TO 700
      CALL OBIND(CURS(1,2),'&SAL',,SAL,4,3)
      IF (CURS(1,2).NE.0) GO TO 700
      CALL OBIND(CURS(1,2),'&DEPTNO',,DEPTNO,4,3)
      IF (CURS(1,2).NE.0) GO TO 700
      CALL OBIND(CURS(1,3),'&DEPTNO',,DEPTNO,4,3)
      IF (CURS(1,3).NE.0) GO TO 700
C
C     EXECUTE THE SQL STATEMENTS.  CURSOR 5 ASKS ORACLE TO
C     BEGIN AN UPDATE TRANSACTION ON TABLES 'EMP' AND 'DEPT'.
C     ALL OTHER TRANSACTIONS ON THOSE TWO TABLES WILL BE
C     BLOCKED UNTIL AN 'END TRANSACTION': CURSOR 6.
C
      CALL OEXEC(CURS(1,5))
      IF (CURS(1,5).NE.0) GO TO 700
C
C     BIND THE EMPNO
C
450   CALL OBIND(CURS(1,2),'&EMPNO',,EMPNO,4,3)
      IF (CURS(1,2).NE.0) GO TO 700
C
C     EXECUTE THE INSERT (CURSOR 2)
C
500   CALL OEXEC(CURS(1,2))
      IF (CURS(1,2).EQ.0) GO TO 600
C
C     IF THE CALL RETURNS CODE -9 (DUPLICATE VALUE IN INDEX), THE
C     GENERATE THE NEXT POSSIBLE EMPLOYEE NUMBER
C
      IF (CURS(1,2).NE.-9) GO TO 700
      EMPNO=EMPNO+10
      GO TO 450
C
C     EXECUTE THE UPDATE (CURSOR 3)
C
600   CALL OEXEC(CURS(1,3))
      IF (CURS(1,3).NE.0) GO TO 700
530   WRITE (5,DEPFMT)(STRNGS(J),J=1,ENAMEL),(STRNGS(K),
     1    K=ENAMEL+2+JOBL+2,ENAMEL+2+JOBL+2+DEPTL-1),EMPNO
      GO TO 800
700   CALL ERRRPT(CURS(1,1),6)
800   CALL OEXEC(CURS(1,6))
      EMPNO=EMPNO+10
      GO TO 100
```

```
C
C     CLOSE THE FIVE CURSORS
C
1000  CALL OCLOSE(CURS(1,2))
      CALL OCLOSE(CURS(1,3))
      CALL OCLOSE(CURS(1,4))
      CALL OCLOSE(CURS(1,5))
      CALL OCLOSE(CURS(1,6))
C
C     LOGOFF FROM ORACLE
C
      CALL OLOGOF(CURS(1,1))
      STOP 'END OF SAMPL3'
10000 CALL ERRRPT(CURS(1,1),6)
      GO TO 1000
      END

      SUBROUTINE ERRRPT(CURS,N)
C
C     ERRRPT PRINTS THE CURSOR NUMBER, THE ERROR CODE, AND THE
C     ORACLE FUNCTION CODE.  IF THE LDA CONTAINS AN ERROR CODE,
C     A LOGON ERROR IS ASSUMED
C
C     CURS IS THE CURSOR ARRAY
C     N IS THE NUMBER OF CURSORS (INCLUDING THE LDA) IN THE ARRAY
C
      INTEGER*2 CURS(32,N)

      IF (CURS(1,1).EQ.0) GO TO 100
      WRITE (5,10)CURS(1,1)
10    FORMAT('0Logon error: ',I5)
      GO TO 500
100   DO 158 I=1,N
      IF (CURS(1,I).NE.0) GO TO 150
158   CONTINUE
150   IF (I.LE.N) GO TO 300
      WRITE (5,200)
200   FORMAT('0Unknown ORACLE error')
      GO TO 500
300   WRITE (5,400)I,CURS(1,I),CURS(6,I)
400   FORMAT('0ORACLE error on cursor ',I1,
     1   ': code is ',I5,', op is ',I5)
500   RETURN
      END
```

```
*****************************************
*                                       *
*                  C                    *
*                                       *
*   E X A M P L E    P R O G R A M S    *
*                                       *
*****************************************
```

Table of Contents

```
/* VOID sample1

   sample1 is a simple example program which adds new employee
           records to the personnel database.  Very little
           checking is done to insure the integrity of the
           database.  The program queries the user for data
           as follows:

           Enter employee number:
           Enter employee name:
           Enter employee job:
           Enter employee salary:
           Enter employee dept:

           The new employee record is inserted and the department
           table is updated to increase the employee count.  If
           the employee number is entered as '0', then the progra
           terminates.
*/
#include <std.h>
char dbn[]{"personnel"};                    /* data base name
char uid[]{"qa/test"};                      /* user id/password
char insert[]{"INSERT INTO EMP(EMPNO,ENAME,JOB,SAL,DEPTNO):\
<&EMPNO,&ENAME,&JOB,&SAL,&DEPTNO>;"};
char update[]{"UPDATE DEPT SET EMPCNT=\
NULLF(EMPCNT,0)+1 WHERE DEPTNO=&DEPTNO;"};
main()
{
   int empno,sal,deptno;                    /* employee number, salary
                                               department number
   int lda[32],curs1[32],curs2[32];         /* lda and two cursors
   char ename[11],job[11];                   /* employee name and job
/*
   log on to ORACLE, open the database (two cursors), and pass
   the SQL statements to ORACLE.  The program exits if any error
   occur.
*/
   if (ologon(lda,-1) ||
       oopen(curs1,lda,dbn,-1,-1,uid,-1) ||
       oopen(curs2,lda,dbn,-1,-1,uid,-1) ||
       osql (curs1,insert,-1) ||
       osql (curs2,update,-1))
           {
           errrpt(lda,curs1,curs2);
           goto errexit;
           }
```

```
/*
   read the user's input from STDIN.  If the employee number is
   entered as zero (or just <cr> or if -eof- (control Z) is
   encountered, exit.
*/
   for(;0 < askn("Enter employee number: ",&empno) && empno != 0;
       {
       asks("Enter employee name  : ",ename);
       asks("Enter employee job   : ",job);
       askn("Enter employee salary: ",&sal);
       askn("Enter employee dept:   ",&deptno);
/*
   bind all SQL substitution variable values and execute the SQL
   insert and update.  If any errors occur, print an error messag
   but continue.
*/
       if (obind(curs1,"&EMPNO",-1,&empno,2,3)   ||
           obind(curs1,"&ENAME",-1,ename,-1,1)   ||
           obind(curs1,"&JOB",-1,job,-1,1)       ||
           obind(curs1,"&SAL",-1,&sal,2,3)       ||
           obind(curs1,"&DEPTNO",-1,&deptno,2,3)||
           obind(curs2,"&DEPTNO",-1,&deptno,2,3)||
           oexec(curs1)                          ||
           oexec(curs2))
               errrpt(lda,curs1,curs2);
       }
errexit:
/*
   close the cursors and log off from ORACLE
*/
       oclose(curs1);
       oclose(curs2);
       ologof(lda);
       return(0);
}
/*
COUNT askn(text,variable)

   print the 'text' on STDOUT and read an integer variable from
   SDTIN.

   text points to the null terminated string to be printed
   variable points to an integer variable

   askn returns a 1 if the variable was read successfully or a
       -1 if -eof- was encountered
*/
int askn(text,variable)
   char text[];
   int *variable;
   {
   return(ask("%i",text,variable));
   }
```

```
/*
COUNT asks(text,variable)

   print the 'text' on STDOUT and read up to 10 characters into
   the buffer pointed to by variable from STDIN.

   text points to the null terminated string to be printed
   variable points to a buffer of at least 11 characters
   (to insure room for the trailing NULL)

   asks returns a 1 if the variable was read successfully or a
       -1 if -eof- was encountered
*/
asks(text,variable)
   char text[],variable[];
   {
   return(ask("%10.10p",text,variable));
   }
```

```
/*
COUNT ask(fmt,text,variable)

   print the 'text' on STDOUT and read from STDIN according to th
   format text pointed to by 'fmt'.  The format string is passed
   directly to the c library routine 'getfmt'.

   fmt points to a format string for getfmt
   text points to tne null terminated string to be printed
   variable points to a buffer of sufficient length to hold the
       input specified by the format.  No length checking is
       performed.

   ask returns a 1 if the variable was read successfully or a
       -1 if -eof- was encountered
*/
ask(fmt,text,variable)
   char fmt[],text[],variable[];
   {
   putfmt("\n%p",text);
   putch(-1);
   return(getfmt(fmt,variable));
   }
```

```
/*
VOID errrpt(lda,curs1,curs2)

   errrpt prints the cursor number, the error code, and the
   ORACLE function code.  If the lda contains an error code,
   a log on error is assumed.

   lda points to an ORACLE log on data area.
   curs1 points to an ORACLE cursor area.
   curs2 points to another ORACLE cursor area.
*/
errrpt(lda,curs1,curs2)
   int lda[32],curs1[32],curs2[32];
   {
   int curserr,cursfcn,cursnum;
   if (lda[0])
       putfmt("Logon error: %i\n",lda[0]);
       else
       {
       if (curs1[0])
           {
           cursnum=1;
           curserr = curs1[0];
           cursfcn = curs1[5];
           }
           else
           {
           cursnum=2;
           curserr = curs2[0];
           cursfcn = curs2[5];
           }
       putfmt("ORACLE error on cursor %i: \
code is %i, op is %i\n",cursnum,curserr,cursfcn);
       }
   return(0);
   }
```

```
/* VOID sample2

   sample2 is a simple example program which adds new employee
           records to the personnel database.  Some checking
           is done to insure the integrity of the data base.
           The program queries the user for data as follows:

           Enter employee number:
           Enter employee name:
           Enter employee job:
           Enter employee salary:
           Enter employee dept:

           The new employee record is inserted and the department
           table is updated to increase the employee count.  If
           the employee number is not entered, then the program
           terminates.

*/
#include <std.h>
char dbn[]{"personnel"};                    /* data base name
char uid[]{"qa/test"};                      /* user id/password
char insert[]{"INSERT INTO EMP(EMPNO,ENAME,JOB,SAL,DEPTNO):\
<&EMPNO,&ENAME,&JOB,&SAL,&DEPTNO>;"};
char update[]{"UPDATE DEPT SET EMPCNT=\
NULLF(EMPCNT,0)+1 WHERE DEPTNO=&DEPTNO;"};
char select[]{"SELECT DNAME FROM DEPT WHERE \
DEPTNO=&DEPTNO;"};
main()
{
   int empno,sal,deptno;                    /* employee number, salary
                                               department number
   int curs[4][32];                         /* lda and three cursors
   char ename[11],job[11],dept[21];         /* employee name,job,dept
/*
   log on to ORACLE, open the data base (three cursors), and pars
   the SQL statements.  The program exits if any errors occur.
*/
   if (ologon(curs[0],-1) ||
       oopen(curs[1],curs[0],dbn,-1,-1,uid,-1) ||
       oopen(curs[2],curs[0],dbn,-1,-1,uid,-1) ||
       oopen(curs[3],curs[0],dbn,-1,-1,uid,-1) ||
       osql (curs[1],insert,-1) ||
       osql (curs[2],update,-1) ||
       osql (curs[3],select,-1) ||
       odfinn (curs[3],1,dept,21,5,-1))
           {
           errrpt(curs[0],4);
           goto errexit;
           }
```

```
/*
   read the user's input from STDIN.  If the employee number is
   entered as zero (or just <cr> or if -eof- (control Z) is
   encountered,exit.
   Verify that the entered department number is valid and echo th
   departments name
*/
   for(;0 < askn("Enter employee number: ",&empno) && empno != 0;
        {
        asks("Enter employee name  : ",ename);
        asks("Enter employee job   : ",job);
        askn("Enter employee salary: ",&sal);
        askn("Enter employee dept:   ",&deptno);
/*
   bind all SQL substitution variable values and execute the SQL
   statements.  If any errors occur, print an error message,
   then continue.
*/
        if (obind(curs[1],"&EMPNO",-1,&empno,2,3)   ||
            obind(curs[1],"&ENAME",-1,ename,-1,1)   ||
            obind(curs[1],"&JOB",-1,job,-1,1)       ||
            obind(curs[1],"&SAL",-1,&sal,2,3)       ||
            obind(curs[1],"&DEPTNO",-1,&deptno,2,3)||
            obind(curs[2],"&DEPTNO",-1,&deptno,2,3)||
            obind(curs[3],"&DEPTNO",-1,&deptno,2,3)||
            oexec(curs[3]) ||
            ofetch(curs[3])||
            oexec(curs[1])                          ||
            oexec(curs[2]))
                {
                if (curs[3][0]==4)
                    putfmt("\nNo such department number\n");
                    else
                    errrpt(curs[0],4);
                }
            else
            putfmt ("\n%p added to the %p department\n",ename,dept
        }
errexit:
/*
   close the cursors and log off from ORACLE
*/
        oclose(curs[1]);
        oclose(curs[2]);
        ologof(curs[0]);
        return(0);
}
```

```
/*
COUNT askn(text,variable)

   print the 'text' on STDOUT and read an integer variable from
   SDTIN.

   text points to the null terminated string to be printed
   variable points to an integer variable

   askn returns a 1 if the variable was read successfully or a
       -1 if -eof- was encountered
*/
int askn(text,variable)
   char text[];
   int *variable;
   {
   return(ask("%i",text,variable));
   }
```

```
/*
COUNT asks(text,variable)

   print the 'text' on STDOUT and read up to 10 characters into
   the buffer pointed to by variable from STDIN.

   text points to the null terminated string to be printed
   variable points to a buffer of at least 11 characters
   (to insure room for the NULL)

   asks returns a 1 if the variable was read successfully or a
       -1 if -eof- was encountered
*/
asks(text,variable)
   char text[],variable[];
   {
   return(ask("%10.10p",text,variable));
   }
/*
COUNT ask(fmt,text,variable)

   print the 'text' on STDOUT and read from STDIN according to th
   format text pointed to by 'fmt'.  The format string is passed
   directly to the c library routine 'getfmt'.

   fmt points to a format string for getfmt
   text points to tne null terminated string to be printed
   variable points to a buffer of sufficient length to hold the
       input specified by the format.  No length checking is
       performed.

   ask returns a 1 if the variable was read successfully or a
       -1 if -eof- was encountered
*/
ask(fmt,text,variable)
   char fmt[],text[],variable[];
   {
   putfmt("\n%p",text);
   putch(-1);
   return(getfmt(fmt,variable));
   }
/*
VOID errrpt(cur,n)

   errrpt prints the cursor number, the error code, and the
   ORACLE function code.  If the lda contains an error code,
   a log on error is assumed.

   cur points to an ORACLE cursor array. curs[0] is assummed to b
       the lda.
   n the the number of cursors in the array (including the lda)
*/
errrpt(cur,n)
```

```
    int n;
    int cur[][32];
    {
    int i;
    if (cur[0][0])
        putfmt("Logon error: %i\n",cur[0][0]);
        else
        {
        for (i=1;i>=n||cur[i][0]!=0;i+=1){}
        if (i==n)
        putfmt("Unknown ORACLE error\n");
        else
        putfmt("ORACLE error on cursor %i: \
code is %i, op is %i\n",i,cur[i][0],cur[i][5]);
        }
    return(0);
    }
```

```
/* VOID sample3

   sample3 is a simple example program which adds new employee
           records to the personnel data base.  Checking
           is done to insure the integrity of the data base.
           The employee numbers are automatically selected using
           the current maximum employee number as the start.
           If any employee number is a duplicate, it is skipped.
           The program queries the user for data as follows:

           Enter employee name:
           Enter employee job:
           Enter employee salary:
           Enter employee dept:

           The new employee record is inserted and the department
           table is updated to increase the employee count.  If
           the employee name is not entered, then the program
           terminates.

           If the record is successfully inserted, the following
           is printed:

           ename added to department dname as employee # nnnnnn

*/
#include <std.h>
char dbn[]{"personnel"};                    /* data base name
char uid[]{"qa/test"};                      /* user id/password
char insert[]{"INSERT INTO EMP(EMPNO,ENAME,JOB,SAL,DEPTNO):\
<&EMPNO,&ENAME,&JOB,&SAL,&DEPTNO>;"};
char update[]{"UPDATE DEPT SET EMPCNT=\
NULLF(EMPCNT,0)+1 WHERE DEPTNO=&DEPTNO;"};
char select[]{"SELECT DNAME FROM DEPT WHERE \
DEPTNO=&DEPTNO;"};
char maxemp[]{"SELECT MAX(EMPNO) + 10 FROM EMP;"};
char selemp[]{"SELECT ENAME,JOB FROM EMP;"};   /* used to determi
                                                  ename, job size
char begtrn[]{"BEGIN TRANSACTION 1 ON TABLE EMP,DEPT UPDATE;"};
char endtrn[]{"END TRANSACTION 1;"};
main()
{
   int empno,sal,deptno;                    /* employee number, salary
                                               department number
   int curs[6][32];                         /* lda and three cursors
   char strings[100];                       /* employee name,job,dept
   int enamel,jobl,deptl;                   /* the max length of cols
```

```
/*
   log on to ORACLE, open the data base (three cursors), and pars
   the SQL statements.  The program exits if any errors occur.
   Determine the lengths of the variable length strings via ODSRB
*/
   if (ologon(curs[0],-1) ||
       oopen(curs[1],curs[0],dbn,-1,-1,uid,-1) ||
       oopen(curs[2],curs[0],dbn,-1,-1,uid,-1) ||
       oopen(curs[3],curs[0],dbn,-1,-1,uid,-1) ||
       oopen(curs[4],curs[0],dbn,-1,-1,uid,-1) ||
       oopen(curs[5],curs[0],dbn,-1,-1,uid,-1))
           {
           errrpt(curs[0],6);
           goto errexit;
           }
       /*
           retrieve the current maximum employee number
       */
       if (osql(curs[1],maxemp,-1) ||
           odfinn(curs[1],1,&empno,2,3,-1) ||
           oexec(curs[1]) ||
           ofetch(curs[1]))
           {
           if(curs[1][0]==4) empno=10;
           else
               {
               errrpt(curs[0],6);
               goto errexit;
               }
           }
/*
   determine the max length of the employee name and job title
*/
       if (osql(curs[1],selemp,-1)||
           odsrbn(curs[1],1,&enamel,-1,-1) ||
           odsrbn(curs[1],2,&jobl,-1,-1)   ||
           odfinn(curs[1],1,strings,enamel,1,-1) ||
           odfinn(curs[1],2,&strings[enamel+1],jobl,1,-1))
           {
           errrpt(curs[0],6);
           goto errexit;
           }
```

```
/*
   parse the insert, select, and update statements
*/
        if (osql (curs[2],update,-1) ||
        osql (curs[3],select,-1) ||
        osql (curs[1],insert,-1) ||
        osql (curs[4],begtrn,-1) ||
        osql (curs[5],endtrn,-1) ||
        odsrbn(curs[3],1,&deptl,-1,-1) ||
        odfinn (curs[3],1,&strings[enamel+jobl+2],deptl,5,-1))
            {
            errrpt(curs[0],6);
            goto errexit;
            }
/*
   read the user's input from STDIN.  If the employee name is
   not entered, exit.
   Verify that the entered department number is valid and echo th
   department's name
*/
   for(;0 < asks("Enter employee name  : ",\
        strings,enamel);empno+=10)
        {
        asks("Enter employee job   : ",&strings[enamel+1],jobl);
        askn("Enter employee salary: ",&sal);
        for (;0>=askn("Enter employee dept  :   ",&deptno)||
        obind(curs[3],"&DEPTNO",-1,&deptno,2,3)||
            oexec(curs[3]) ||
            ofetch(curs[3]);
            putfmt("\nNo such department\n")){}
/*
   bind all SQL substitution variable values and execute the SQL
   statements.  If any errors occur, print an error message,
   then continue.
*/
        if (obind(curs[1],"&ENAME",-1,strings,-1,1)  ||
            obind(curs[1],"&JOB",-1,&strings[enamel+1],-1,1)
            obind(curs[1],"&SAL",-1,&sal,2,3)       ||
            obind(curs[1],"&DEPTNO",-1,&deptno,2,3)||
            obind(curs[2],"&DEPTNO",-1,&deptno,2,3)||
            oexec(curs[4]))
                errrpt(curs[0],6);
                else
                {
                for (; 0==obind(curs[1],"&EMPNO",-1,&empno,2,3) &&
                      -9==oexec(curs[1]);empno+=10)
                      /*  test code */
                      {
                      errrpt(curs[0],6);
                      }
                      /* end of test code */
```

```
                if (curs[1][0])
                    errrpt(curs[0],6);
                    else
                    {
                    if (oexec(curs[2]))
                        errrpt(curs[0],6);
                    else
                        putfmt("\n%p added to the %p department \
as employee number %i\n",\
                        strings,&strings[ename1+job1+2],empno);
                    }
                if (oexec(curs[5]))
                    errrpt(curs[0],6);
                }
        }
errexit:
/*
   close the cursors and log off from ORACLE
*/
        oclose(curs[1]);
        oclose(curs[2]);
        oclose(curs[3]);
        oclose(curs[4]);
        oclose(curs[5]);
        ologof(curs[0]);
        return(0);
}
/*
COUNT askn(text,variable)

   print the 'text' on STDOUT and read an integer variable from
   SDTIN.

   text points to the null terminated string to be printed
   variable points to an integer variable

   askn returns a 1 if the variable was read successfully or a
       -1 if -eof- was encountered
*/
int askn(text,variable)
   char text[];
   int *variable;
   {
   return(ask("%i",text,variable));
   }
```

```
/*
COUNT asks(text,variable,len)

   print the 'text' on STDOUT and read up to 'len' characters int
   the buffer pointed to by variable from STDIN.

   text points to the null terminated string to be printed
   variable points to a buffer of at least 'len'+1 characters

   asks returns the number of character read into the string, or
       -1 if -eof- was encountered
*/
asks(text,variable,len)
   char text[],variable[];
   {
   char fmt[14],lens[6];
   int x;
   x=itob(lens,len,10);
   lens[x]='\0';
   cpystr(fmt,"%",lens,".",lens,"p",NULL);
   return(EOF==ask(fmt,text,variable)?EOF:lenstr(variable));
   }
/*
COUNT ask(fmt,text,variable)

   print the 'text' on STDOUT and read from STDIN according to th
   format text pointed to by 'fmt'.  The format string is passed
   directly to the c library routine 'getfmt'.

   fmt points to a format string for getfmt
   text points to tne null terminated string to be printed
   variable points to a buffer of sufficient length to hold the
       input specified by the format.  No length checking is
       performed.

   ask returns a 1 if the variable was read successfully or a
       -1 if -eof- was encountered
*/
ask(fmt,text,variable)
   char fmt[],text[],variable[];
   {
   putfmt("\n%p",text);
   putch(-1);
   return(getfmt(fmt,variable));
   }
```

```
/*
VOID errrpt(cur,n)

   errrpt prints the cursor number, the error code, and the
   ORACLE function code.  If the lda contains an error code,
   a log on error is assumed.

   cur points to an ORACLE cursor array. curs[0] is assummed to b
       the lda.
   n the the number of cursors in the array (including the lda)
*/
errrpt(cur,n)
   int n;
   int cur[][32];
   {
   int i;
   if (cur[0][0])
       putfmt("Logon error: %i\n",cur[0][0]);
       else
       {
       for (i=1;i<n&&cur[i][0]==0;i+=1){}
       if (i==n)
       putfmt ("Unknown ORACLE error\n");
       else
       putfmt("ORACLE error on cursor %i: \
code is %i, op is %i\n",i,cur[i][0],cur[i][5]);
       }
   return(0);
   }
```

```
*****************************************
*                                       *
*              C O B O L                *
*                                       *
*    E X A M P L E   P R O G R A M      *
*                                       *
*****************************************
```

Table of Contents

```
IDENTIFICATION DIVISION.
PROGRAM-ID. ORACBL.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.  PDP-11.
OBJECT-COMPUTER.  PDP-11.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  LDA.
    02  LDA-RC  PIC S9999 COMP.
    02  FILLER  PIC S9999 COMP OCCURS 31 TIMES.
01  CURSOR.
    02  C-RC    PIC S9999 COMP.
    02  C-TYPE  PIC S9999 COMP.
    02  C-ROWS  PIC S9(9) COMP.
    02  C-OFFS  PIC S9999 COMP.
    02  C-FNC   PIC S9999 COMP.
    02  FILLER  PIC S9999 COMP OCCURS 26 TIMES.
77  AREA-COUNT  PIC S9999 VALUE 1 COMP.
77  AREA-SIZE   PIC S9999 VALUE 3 COMP.
77  DATA-BASE   PIC X(16) VALUE "personnel".
77  DATA-BASE-L PIC S9999 VALUE 9 COMP.
77  USER-ID     PIC X(8)  VALUE "qa/test".
77  USER-ID-L   PIC S9999 VALUE 7 COMP.
77  SQL-SEL     PIC X(40) VALUE "SELECT EMPNO,ENAME FROM EMP".
77  SQL-SEL-L   PIC S9999 VALUE 40 COMP.
77  SQL-INS     PIC X(40) VALUE "INSERT INTO EMP:<&EMPNOX,&ENAMEX
77  SQL-INS-L   PIC S9999 VALUE 40 COMP.
77  EMPNO-RC    PIC S9999 COMP.
77  EMPNO-N     PIC S9999 VALUE 1 COMP.
77  ENAME       PIC X(19).
77  ENAME-L     PIC S9999 VALUE 10 COMP.
77  ENAME-RC    PIC S9999 COMP.
77  ENAME-N     PIC S9999 VALUE 2 COMP.
77  FTYPE       PIC S9999 COMP.
77  ERRTYPE     PIC +9999.
77  EMPNOX      PIC S9(9) COMP VALUE 0.
77  EMPNOX-L    PIC S9999 VALUE 4 COMP.
77  EMPNOX-N    PIC X(7) VALUE "&EMPNOX".
77  EMPNOX-N-L  PIC S9999 COMP VALUE 7.
77  ENAMEX-N    PIC X(7) VALUE "&ENAMEX".
77  ENAMEX-N-L  PIC S9999 COMP VALUE 7.
77  INT4        PIC S9999 COMP VALUE 3.
77  ASC         PIC S9999 COMP VALUE 1.
77  EMPNOX-A    PIC S9(9) SIGN LEADING SEPARATE DISPLAY
                VALUE -1.
77  EMPNOX-A-X  REDEFINES EMPNOX-A PIC X(10).
77  EMPNOX-A-L  PIC S9999 COMP VALUE 10.
77  MSG-TO-OP   PIC X(45) VALUE
                   "+^^^^^^^^^^ Enter new empno (+|- and 9 digits)
```

```
PROCEDURE DIVISION.
BEGIN.
*
* LOGON TO ORACLE
*
    CALL "OLOGON" USING LDA-RC,AREA-COUNT.
    IF LDA-RC NOT = 0 PERFORM ORA-ERR GO TO EXIT-STOP.
*
* OPEN THE PERSONNEL DATA BASE
*
    CALL "OOPEN" USING C-RC,LDA-RC,DATA-BASE,DATA-BASE-L,
    AREA-SIZE,USER-ID,USER-ID-L.
    IF C-RC NOT = 0 PERFORM ORA-ERR GO TO EXIT-LOGOF.
*
* INSERT A RECORD
*
    CALL "OSQL" USING C-RC,SQL-INS,SQL-INS-L.
    IF C-RC NOT = 0 PERFORM ORA-ERR GO TO EXIT-CLOSE.
INSERT-ONE.
    DISPLAY MSG-TO-OP.
    ACCEPT EMPNOX-A.
    IF EMPNOX-A = 0 GO TO SELECT-IT.
    MOVE EMPNOX-A TO EMPNOX.
    CALL "OBIND" USING C-RC,EMPNOX-N,EMPNOX-N-L,EMPNOX,
     EMPNOX-L,INT4.
    IF C-RC NOT = 0 PERFORM ORA-ERR GO TO INSERT-ONE.
    CALL "OBIND" USING C-RC,ENAMEX-N,ENAMEX-N-L,EMPNOX-A-X,
       EMPNOX-A-L,ASC.
    IF C-RC NOT = 0 PERFORM ORA-ERR GO TO INSERT-ONE.
    CALL "OEXEC" USING C-RC.
    IF C-RC NOT = 0 PERFORM ORA-ERR.
    GO TO INSERT-ONE.
*
* PARSE THE SELECT
*
SELECT-IT.
    CALL "OSQL" USING C-RC,SQL-SEL,SQL-SEL-L.
    IF C-RC NOT = 0 PERFORM ORA-ERR GO TO EXIT-CLOSE.
*
* DEFINE THE RECEIVING DATA AREAS
*
    CALL "ODFINN" USING C-RC,EMPNO-N,EMPNOX,EMPNOX-L,INT4,
    EMPNO-RC.
    IF C-RC NOT = 0 PERFORM ORA-ERR GO TO EXIT-CLOSE.
    CALL "ODFINN" USING C-RC,ENAME-N,ENAME,ENAME-L,ASC,
    ENAME-RC.
    IF C-RC NOT = 0 PERFORM ORA-ERR GO TO EXIT-CLOSE.
```

```
*
* EXECUTE THE QUERY BLOCK
*
    CALL "OEXEC" USING C-RC.
    IF C-RC NOT = 0 PERFORM ORA-ERR GO TO EXIT-CLOSE.
FETCH-ONE.
*
* BLANK ALPHA AREAS AND FETCH THE ROWS
*
    MOVE SPACES TO ENAME.
    CALL "OFETCH" USING C-RC.
    IF ENAME-RC NOT = 0
       MOVE ENAME-RC TO ERRTYPE
       DISPLAY "NON-ZERO RETURN ON FETCH OF ENAME; CODE IS :",
           ERRTYPE.
    IF C-RC NOT = 0 PERFORM ORA-ERR  GO TO EXIT-S.
    MOVE EMPNOX TO EMPNOX-A.
    DISPLAY "EMPNO = ",EMPNOX-A," ;ENAME = ",ENAME.
    GO TO FETCH-ONE.
EXIT-S.
EXIT-CLOSE.
*
* CLOSE THE DATA BASE
*
    CALL "OCLOSE" USING C-RC.
    IF C-RC NOT = 0 PERFORM ORA-ERR.
EXIT-LOGOF.
*
* LOG OFF FROM ORACLE
*
    CALL "OLOGOF" USING LDA-RC.
    IF LDA-RC NOT = 0 PERFORM ORA-ERR.
EXIT-STOP.
    STOP RUN.
ORA-ERR.
*
* PRINT ORACLE ERROR NOTICE
*
    DISPLAY "ORACLE ERROR"
    MOVE C-FNC TO ERRTYPE.
    DISPLAY "ORACLE FUNCTION = ",ERRTYPE.
    MOVE LDA-RC TO ERRTYPE.
    DISPLAY "LDA ERROR = ",ERRTYPE.
    MOVE C-RC TO ERRTYPE.
    DISPLAY "CUR ERROR = ",ERRTYPE.
```

O R A C L E

ASSEMBLY LANGUAGE INTERFACE

To use the ORACLE user interface from a MACRO-11 language program, the programmer must invoke and use the CCALL macro as described below. The CCALL macro works as follows:

1. The CCALL macro is the SDLLIB.MLB. The user must include this library as part of his assembly.

2. The CCALL macro is invoked with the .MCALL directive.

3. The format of the CCALL macro is:

```
CCALL   FUNC,P1,...,Pn
```

where:

FUNC — is the function to be called (e.g. OOPEN, OSQL,...)

P1,...,P2 — is the parameter list as defined in the programming interface. All of the parameters must be able to be objects of a MOV instruction. A parameter can be a list of values, in which case the values are added together before they are put in the parameter list. An example of this is <R1,#OFFSET>. This would cause R1 and #OFFSET to be added together to produce a single parameter for the parameter list. This mechanism allows users to easily pass pointers to a data area in a structure pointed to by a register. Also, if the first item in this list is 'B' then the next item in the list is a byte and must be able to be the object of a MOVB instruction.

4. An example of the CCALL macro:

```
CURSOR: .BLKB   64.
LDA:    .BLKB   64.
DBNAME: .ASCIZ  /ACCOUNTING/
        CCALL   OOPEN,#CURSOR,#LDA,#DBNAME,#-1,#-1
```

5. When calling an ORACLE function, registers R0 and R1 are volatile and may be destroyed.

6. Upon return from an ORACLE function, R0 contains the return code (the first word of the cursor).

7. To include the MACRO-11 interface to ORACLE the user should follow the instructions that C programmers do to include the C interface.

ORACLE

HOST LANGUAGE INTERFACE

LINKING INSTRUCTIONS FOR RSX/IAS

When a user wishes to create his or her own database task, the distributed user interface modules must be included in the task image. The user code interface is provided in four object libraries:

* OFOLIB.OLB

* OCELIB.OLB

* ORALIB.OLB

* CLIB.OLB

The first two libraries, OFOLIB and OCELIB, provide the Host Language Interface between a host language, such as FORTRAN, and ORACLE. The libraries are used in the following way in the task build:

OCELIB/LB:ORACEE for C and assembly language programs
OFOLIB/LB:ORAFOR for FORTRAN and COBOL programs

The third library, ORALIB, contains object code which has several responsibilities. The first is to provide the primary intertask communication and handshake between the user task and ORACLE via the SDAT$ and RCVD$ system service routines. The second function provides the user interface to the ORACLE context region. This region is used by the user interface to pass commands and data to ORACLE and used in turn by ORACLE to pass data back to the user. This module will identify the user task name to ORACLE and then wait to receive a mapping context message from ORACLE. Upon the completion of this initialization, the user task may enter into transactions with ORACLE.

The fourth library contains common routines used by the 'C' language programs in OFOLIB, OCELIB, and ORALIB.

LINKING AN RSX-11M TASK

Two special steps must be taken to include the ORACLE Interface program in a user task.

1. Include OFOLIB or OCELIB, ORALIB, and CLIB in the list of library files.

2. Add an extra address window at task build time.

Examples:

The TKB command input which builds the SAMPL1 FORTRAN program:

```
SAMPL1/-FP,SAMPL1/-SP
SAMPL1,[1,1]F4PEIS,[1,1]F4POTS/LB
[1,1]OFOLIB/LB:ORAFOR
[1,1]ORALIB/LB
[1,1]CLIB/LB
/
WNDWS=1
STACK=3000
UNITS=13
//
```

The TKB command input which builds the sample3 "C" program:

```
sample3/cp,sample3/-sp
sample3
[1,1]ocelib/lb:oracee
[1,1]oralib/lb
[1,1]clib/lb:chdr
[1,1]clib/lb
/
stack=3000
units = 13
wndws = 1
extsct=$99998:3000
//
```

LINKING AN IAS TASK

Two special steps are required to include the ORACLE Interface in a user task.

1. Include OFOLIB, OCELIB or ORALIB, and CLIB in the list of object files;

2. Add an extra region descriptor block;

Examples:
The command input to TKB to build the SAMPL1 FORTRAN program:

```
SAMPL1/CP/-FP,SAMPL1/-SP
SAMPL1
F4POST/LB
OFOLIB/LB:ORAFOR
ORALIB/LB
CLIB/LB
/
STACK=3000
UNITS = 13
ATRG = 1
//
```

The command input to TKB to build the sample3 "C" program:

```
sample3/cp,sample3/-sp
sample3
ocelib/lb:oracee
oralib/lb
clib/lb:chdr
clib/lb
/
stack=3000
units = 13
atrg = 1
extsct=$99998:3000
//
```

Note: if a user task is multi-user, the task must have a name of the form: $$$abc

SYSTEM RESOURCES FOR PDP-11 PROGRAMS

ORACLE makes use of certain system resources which are discussed here to prevent conflict with user software.

SEND/RECEIVE DATA directives may not be used.

Neither SQL statements nor data to be used by ORACLE may reside in locations mapped to addresses 140000(8) through 177777(8).

None of the modules included from the ORACLE libraries (OFOLIB, OCELIB, ORALIB, and CLIB) may be mapped by an address window which also maps virtual locations 140000(8) through 177777(8).

Programs which are larger than 140000(8) bytes should use an overlay structure in which the ORACLE library modules are in the root and the remainder of the program is in a memory resident overlay. An example task builder imput file is contained in DBFTKB.CMD and DBF.ODL which follow.

DBFTKB.CMD (referenced by TKB @DBFTKB on RSX11M systems):

```
DBF/CP,DBF/-SP=DBF/MP
STACK=3000
UNITS=13
WNDWS=1
TASK=...DBF
ASG=TI:13
EXTSCT=$99998:4000
/
```

DBF.ODL (referenced above in the first TKB command line):

```
.ROOT   DBF010-*!(DBF040-DBF050)
DBF010: .FCTR   DBF020-DBF025-DBF030
DBF020: .FCTR   CLIB/LB:CHDR:.END:SBREAK
DBF025: .FCTR   DBFLIB/LB:DBFKNL:DBFDTA
DBF030: .FCTR   CLIB/LB-OCELIB/LB:ORACEE-ORALIB/LB-CLIB/LB
DBF040: .FCTR   DBFLIB/LB:MAIN-DBFLIB/LB
DBF050: .FCTR   ORALIB/LB-CLIB/LB
        .END
```

LINKING INSTRUCTIONS FOR VAX/VMS

Two object libraries provide the native mode user code interface on the VAX/VMS distribution:

* ORALIB.OLB

* CLIB.OLB

The first library, ORALIB, provides the Host Language Interface between a host language, such as FORTRAN, and ORACLE. The library also contains an object code which has several responsibilities. The first is to provide for the creation of and subsequent communication with a detached ORACLE process. The second function provides the user interface to the ORACLE context region. This region is used by the user interface to pass commands and data to ORACLE and used in turn, by ORACLE to pass data back to the user.

It should be noted a user image must reserve the address range 0C000-10000 (hex). This is may be accomplished by including the distributed pad option file in your link command. (Note the inclusion of ORAPAD.OPT in the example, below).

A sample command procedure for linking a "C" program is listed below:

```
$!
$!        UFILNK.COM
$!
$!       VAX/VMS UFI LINK COMMAND FILE
$!
$DEL UFI.EXE;*,UFI.MAP;*
$LINK /EXE=UFI/MAP=UFI -
      CLIB/INCLU=CHDR/LIBR,-
      UFILIB/INCLU=UFIPAD/LIBR,-
      ORAPAD/OPTIONS,-
      ORALIB/LIBR,-
      CLIB/LIBR
```

A sample command procedure for linking a FORTRAN program is listed below:

```
$!
$!       SAMPLE FORTRAN/ORACLE LINK PROCEDURE
$!
$LINK /EXE=SAMPL1/MAP=SAMPL1 -
      SAMPL1,-
      ORALIB/LIBR,-
      ORAPAD/OPTIONS,-              ! PADS 0C000-10000
      CLIB/LIBR
```

R S I

ORACLE

DATA BASE ADMINISTRATOR'S GUIDE

Oracle Programmer's Guide - Version 2.3

O R A C L E

DATABASE ADMINISTRATOR'S GUIDE

TABLE OF CONTENTS

I. INTRODUCTION

Within an ORACLE environment there are a number of functions which should be under control of a centralized database administration. These functions, summarized below, are described in detail in this manual and in the ORACLE Installation Guide.

- ORACLE Installation
  The ORACLE Installation facilities are described in the ORACLE Installation Guide.

- ORACLE Initialization
  The ORACLE Initialization procedures are described in the ORACLE Installation Guide.

- Database Creation & Directory Functions
  Each ORACLE system contains a database directory which describes existing databases and the operating system files utilized for them. The DBF utility provides functions for creating databases, and for adding to, removing from, and listing information in the database directory.

- Database Backup & Recovery Procedures
  Standard procedures should be established for periodically saving the files comprising databases, through use of operating system utilities. The ORACLE Journal provides a means to recover databases following hardware failures. The JNL utility provides functions for controlling the Journal and for applying the Journal in the recovery process.

- Database Loader
  The ORACLE Loader Utility provides facilities for loading raw data from one or more files into a database table.

- Table Unload & Reload
  The Unload Reload Utility provides functions for unloading tables from a database and subsequently reloading those tables into the same database or another database.

D B F

DATABASE FILE UTILITY

I. INTRODUCTION

DBF is the ORACLE utility supporting the functions of establishing a database and mapping that database onto operating system files. ORACLE maintains a directory of databases under its control. The DBF utility is used to establish, modify, and delete entries in this directory. When performing initialization functions on an existing database, any data in the existing database will be deleted. The directory contains information on the databases, their extents, and the physical files used. DBF provides functions to:

- Establish a database as an existing or new file;
- Add an existing or new file as an extent to an existing database;
- Remove an existing database from the database directory;
- Enter the files of an existing database into the database directory;
- Reallocate an existing database extent to another file;
- Identify the system database;
- List database directory information.

When ORACLE is installed, the ORACLE system database is the first database to be created. THE DBF utility is used to create the system database. Subsequently, whenever the operating system is initialized, the DBF utility must be used to identify the system database to ORACLE, before normal ORACLE processing can proceed.

II. OPERATION

When an ORACLE database is opened, ORACLE obtains the information about that database from the database directory. From that point on the information is maintained until the last active user of that database issues a CLOSE. Thus, if DBF is used to modify information about a database when that database is opened, ORACLE will not use the new information until that database is subsequently closed by all active users of that database, and then reopened.

When a database is created or initialized by DBF, that database is established as a secure or nonsecure database, through the user-name parameter. Once a database has been created as a secure database, access to it is restricted to the user who created it and to any users specified by him (see DEFINE USER statement in SQL). For a secure database, all DBF functions (except SYSTEM DATABASE, LIST, and ENTER) require specification of the user-name/password, and may only be performed by the creator of that database.

DBF is invoked with the following command:

```
DBF  dbf-command   [parameter-list]
```

where

dbf-command	is a code identifying the function to be performed;
parameter-list	is an optional list of one or more parameters used for the function.

Parameters must adhere to the following conventions:

1. Database-names consist of any printable characters and can be up to 16 characters long.

2. Database-names must be unique.

3. File-names must follow the conventions of the host operating system for naming files.

4. The database-names "ORACLE" and "ORAWRK" are reserved for the system database and the ORACLE Work database.

5. The user-name consists of any printable characters and can be up to 20 characters long.

6. The password consists of any printable characters and can be up to 20 characters long.

7. Database-names, user-names, and passwords are automatically converted to upper-case characters.

8. For the CREATE and EXTEND functions, the size parameter refers to the size of the operating system file being created. It may be expressed as a number of 512-byte blocks, or as a number followed by the letter "K", representing the number of 1024-block units, as follows:

   500 means 500 blocks (256,000 bytes)
   3K means 3x1024 = 3072 blocks (1,572,864 bytes)

   The minimum size of a database is 1024 blocks. The system database must be a single extent (single file) and its size must be at least 4096 blocks.

III. DBF FUNCTIONS

Create Database

This function creates a file under the host file system, initializes the created file as a database, and enters the new database into the database directory.

```
      C
DBF  CREATE  db-name  file-name  size  [user-name/paswrd]
```

where:

db-name	is the name of the newly created database;
file-name	is the name of the file to be created and initialized as a database;
size	is the size of the file to be created, expressed as a number of 512-byte blocks, or as a number followed by the letter "K" representing a multiple of 1024-block units;
user-name/paswrd	is the optional parameter used to identify the creator of the database, which must be provided if the database is to be secure (protected with the GRANT/REVOKE privileges). If only a user-name is specified, DBF will prompt the user for a password, which when entered will not be displayed on the screen.

Extend Database

This function creates a file under the host file system, initializes the created file as an extent to the specified database, and enters the new extent into the database directory.

```
      E
DBF   EXTEND  db-name  file-name  size [user-name/paswrd]
```

where:

db-name	is the name of an existing database which is to be extended;
file-name	is the name of the file to be created and initialized as an extent;
size	is the size of the file to be created, expressed as a number of 512-byte blocks, or as a number followed by the letter "K" representing a multiple of 1024-block units;
user-name/paswrd	is the optional parameter identifying the user, required for any secure database. If only a user-name is specified, DBF will prompt the user for a password, which when entered will not be displayed on the screen.

Initialize Database

This function initializes an existing file as a database and enters the new database into the database directory.

```
        I
DBF     INIT    db-name    file-name    [user-name/paswrd]
```

where:

db-name	is the name of the newly initialized database;
file-name	is the name of an existing file to be initialized as a database.
user-name/paswrd	is the optional parameter used to identify the creator of the database, which must be provided if the database is to be secure. If only a user-name is specified, DBF will prompt the user for a password, which when entered will not be displayed on the screen.

Initialize Database Extent

This function initializes an existing file as an extent to the specified database and enters the new extent into the database directory.

```
        IE
DBF     INITEXTENT    db-name    file-name    [user-name/paswrd]
```

where:

db-name	is the name of an existing database which is to be extended;
file-name	is the name of an existing file to be initialized as an extent.
user-name/paswrd	is the optional parameter identifying the user, required for any secure database. If only a user-name is specified, DBF will prompt the user for a password, which when entered will not be displayed on the screen.

Remove Database

This function removes a database from the database directory. Note that the files belonging to the specified database are **not deleted** or in any way removed from the host file system.

Removing a database from a directory would be required when performing the following functions:

- Moving a database from one ORACLE system to another;
- Renaming a database;
- Taking a database off-line.

```
        R
DBF     REMOVE    db-name    [user-name/paswrd]
```

where:

db-name	is the name of the database to be removed.
user-name/paswrd	is the optional parameter identifying the user, required for any secure database. If only a user-name is specified, DBF will prompt the user for a password, which when entered will not be displayed on the screen.

Enter Database

This function enters a set of existing database extent files (which have previously been initialized) as a single database into the database directory. The Enter Database function is also used to re-enter a previously removed (see REMOVE function) database, into the database directory.

```
        ED
DBF     ENTER     db-name     fn1 ... fnn
```

where:

db-name	is the name of the database to be entered into the database directory;
fn1 .. fnn	is the list of file-names of the files which make-up the database. These files must be entered in extent number order. If the file-name list is not entered, DBF will prompt for each file-name as follows: enter extent file n: Responding to a prompt with a CR or ^Z indicates that all files to be entered have been specified.

Move Extent

This function logically moves an extent of a database from one file to another and updates the database directory to reflect the change. Note that no files are copied or deleted. The file being moved to, must be an exact duplicate of the specified extent file. When extents are added to an existing database, the extents are assigned sequential numbers by ORACLE. The MOVE function requires identification of the extent to be moved, by extent number.

```
      ME
DBF   MOVE  db-name  file-name  ext-num  [user-name/paswrd]
```

where:

db-name	is the name of an existing database to be operated on;
file-name	is the name of an existing file which is an exact duplicate of the specified extent file;
ext-num	is number which specifies the extent to be moved.
user-name/paswrd	is the optional parameter identifying the user, required for any secure database. If only a user-name is specified, DBF will prompt the user for a password, which when entered will not be displayed on the screen.

Reinitialize Database

This function re-initializes an existing database. All data in the database is lost and the database is like a newly initialized database.

CAUTION! This function deletes the data in an existing database and should be used with extreme care.

```
        RI
DBF     REINIT    db-name     [user-name/paswrd]
```

where:

db-name	is the name of the database to be reinitialized.
user-name/paswrd	is the optional parameter used to identify the creator of the database, which must be provided if the database is to be secure (protected with the GRANT/REVOKE privileges). If only a user-name is specified, DBF will prompt the user for a password, which when entered will not be displayed on the screen.

System Database

This function identifies the system database to ORACLE. The specified file must already exist. When the system database is created or initialized, this function is automatically performed. This function is required when bringing up ORACLE (such as in a newly initialized operating system).

```
        SD
DBF     SYSTEMDB    file-name
```

where:

file-name	is the name of the file to be known to ORACLE as the system database.

List Database Information

This function displays information from the database directory. The display contains information about a database and its extents. If a database name is not specified, information will be displayed for all databases in the directory.

For each database the following information is provided:

(1) Database Name;
(2) User Name (Displayed only for secure databases);
(2) Security/Journal Flags
    ('S' for Secure database, 'J' for Journaling);
(3) For Each Extent:
        (a) Extent Number;
        (b) Number of Blocks;
        (c) File Name.
(4) If LU is specified, the number of unused blocks
    in each database is also displayed.

```
        L
DBF     LIST    [db-name]
        LU
```

where:

db-name	is the optional parameter specifying the name of the database to be listed.

SAMPLE DISPLAY

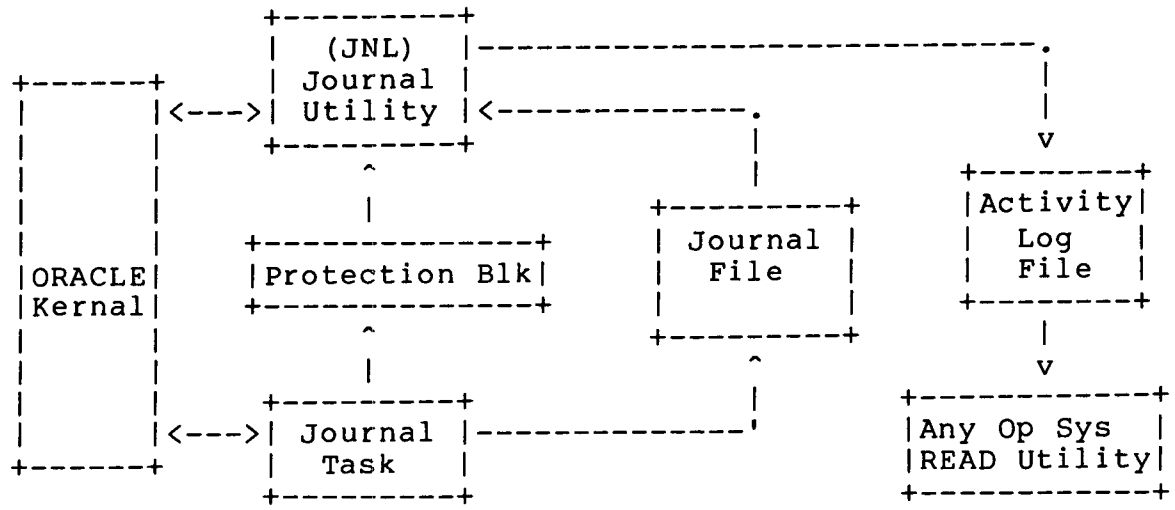
```
DATABASE NAME      FLAGS  EXT    BLOCKS  FILE
USER NAME                  #
----------------   ----   ---    ------  --------------------
demo               SJ       1      1024  DR0:[1,1]DEMO.DBS;1
                                  ------
                                    1024

kcm                S        1      4096  DR0:[1,1]KCM.DBS;1
qa                                ------
                                    4096

oracle                      1      4096  DR0:[1,1]ORACLE.DBS;1
ora5555                           ------
                                    4096
```

I. INTRODUCTION

An ORACLE system includes a Journal Task which controls all writing to the databases. A database may optionally be journaled, in which case the Journal Task will write afterimages of the modified blocks, to a Journal file for use in forward recovery. The Journal Task uses a single Journal file for the one or more databases being journaled. A Journal Utility provides functions for:

- o Initiating and stopping the Journal during an ORACLE session;
- o Initiating and stopping the Journal for a database;
- o Applying the Journal to one or more databases;
- o Querying the status of the Journal activity.

The ORACLE Task also maintains a special single-block Data Protection file on disk. The relationship is as follows:



Application of the Journal is used to restore the status of a database in the following situations:

- Disk Failure

  A disk hardware problem can result in the loss of one or more database extents (files). This can affect one or more databases, requiring those databases to be restored. Note that the LIST function in the DBF utility can be used to determine the extents which comprise each database.

- System Failure

  A system failure can be the result of an Operating System crash, a power failure, or an ORACLE problem. When ORACLE terminates as a result of an ORACLE problem, an ORACLE cleanup task assures that the database is left in a valid state. During an Operating System crash or power failure, the integrity of the database will be compromised only in the event that the incident occurs during the actual writing of blocks to that database. At system start up time (such as following a crash or power failure), the ORACLE initialization routine should include the DISPLAY STATUS function of the JNL utility to indicate which databases, if any, were affected. Note that since the ORACLE initialization procedure initializes the Data Protection Block, it is **imperative** that the DISPLAY STATUS function be performed prior to initialization, in order to ascertain the status of the databases.

The application of the Journal can only be used to restore database integrity. The determination as to whether or not specific SQL Data Manipulation (DML) statements (such as a global updates) were successfully executed, requires SQL queries to determine if the update activity was completed.

Note that for the special case of Data Definition (DDL) statements (CREATE TABLE, EXPAND TABLE, DROP TABLE, DEFINE VIEW, and DROP VIEW), integrity is automatically guaranteed. Thus if a DDL statement was in process, but not yet completed at the time of the failure, ORACLE will automatically back out any completed portion of the DDL statement the first time that database is reopened.

The files involved in the Journal functions are named as follows:

	Protection Blk Block File	Activity Log File	Journal File
VMS	SYS$ORACLE:PARAM.JNL	SYS$ORACLE:LOG.JNL	user-name
RSX-11M,IAS	LB:[1,1]PARM.JNL	LB:[1,1]LOG.JNL	user-name
UNIX	/oracle/param.jnl	/oracle/log.jnl	user-name

II. JOURNAL PROCEDURES

Proper use of the Recovery Functions in an ORACLE system requires the establishment of certain operational procedures. The most important of these involve the periodic saving of the database (or extents of the database). This is accomplished with any of a number of facilities available in a given operating system. Some of the available facilities and their characteristics are summarized in Table 1. These back-up procedures should be performed together with the other operational procedures instituted for a given computer configuration.

Another important procedure involves the saving and naming conventions for the files of the Journal. The Journal consists of a series of disc files. When the Journal is initialized (at ORACLE system start up), the name (and size) is provided for the Journal File. When this file is full, ORACLE will start another file and automatically name the next file by incrementing a number in the name (indicated by a special character). Thus during an ORACLE session, there will probably be multiple files (depending on the file size and the update activity), all having a common name with an increment. If the periodic SAVE spans multiple ORACLE sessions, a naming convention should be used to be able to easily identify the many files making up the Journal since the last periodic SAVE. It might also be appropiate to archive disc Journal files to tape.

The Journal Utility maintains an Activity Log which is useful in monitoring the historical series of Journal activity. This log contains date and time entries for the following:

- Journal Start and Journal File name
- Journal Start for particular database
- Journal Stopped
- Journal Stopped for particular database

This Activity Log file can be read with any convenient system utility for reading files. The following is an sample of information on the log.

```
Tue Mar 24 13:00:44 1981  Journal started on database pers
Tue Mar 24 13:00:48 1981  Journal started on file PER?.JNL
Tue Mar 24 18:06:50 1981  Journal stopped
Tue Mar 24 18:06:56 1981  Journal stopped on database pers
```

OPERATING SYSTEM	UTIL PGM	DISK LEVEL	FILE LEVEL	DISK TO DISK	DISK TO TAPE	SPEED
VMS	BCK	X	X		X	HIGH
	COPY	X	X	X	X	LOW
	DSC	X		X	X	MEDIUM
	FLX		X		X	LOW
RSX-11M,M+	BRU	X	X	X	X	HIGH
	PIP		X	X		LOW
RSX-11M,M+ and IAS	DSC	X		X	X	MEDIUM
	FLX		X		X	LOW
IAS	COPY		X	X	X	LOW
	BCK		X	X	X	HIGH
UNIX Ver6 and UNIX PWB	DUMP/ RESTORE		X		X	MEDIUM
	CP	X	X	X	X	LOW
	DD	X	X	X	X	MEDIUM
	TP	X	X		X	LOW
UNIX PWB only	VC	X		X	X	HIGH
	CPIO		X	X	X	LOW

TABLE 1

OPERATING SYSTEM BACK-UP UTILITIES

III. JNL FUNCTIONS

JNL is invoked with the following type of command:

JNL jnl-command [parameter-list]

where

jnl-command	is a term specifying the function to be performed. The jnl-command may be abbreviated to any unique subset of the statement verb (ie, STAR for START or STAT for STATUS).
parameter-list	is an optional list of one or more parameters used for the function.

Parameters must adhere to the following conventions:

1. File names must follow the conventions of the host operating system for naming files.

2. For the START function, the size parameter refers to the size of the operating system file being created. It may be expressed as a size of 512-byte blocks, or as a number followed by the letter "K", representing the number of 1024-block units, as follows:

   500 means 500 blocks (256,000 bytes)
   3K means 3x1024 blocks = 3072 blocks (1,572,864 bytes)

Start Journal

This function starts the ORACLE Journal. The Start Journal function would normally be performed as part of the ORACLE start up procedure.

```
JNL    START    filename    size
```

where:

filename	is the name of the operating system file to be used as the Journal. The name must contain a "?" in some character position. This special character will be replaced by the 3-digit number "001". successive files of the Journal during this ORACLE session will receive names with incremented numbers.
size	is the threshhold size used in determining when to terminate a physical file in the Journal and start another one. The actual size of each file will be slightly larger than this value. The size is expressed as a number of 512-byte blocks, or as a number followed by the letter "K" representing a multiple of 1024-block units.

The following example will start a Journal on a file called FEB01001. Each file of the Journal will be about 3,072 blocks in size.

```
JNL    START    FEB01?    3K
```

Stop Journal

This function stops the ORACLE Journal. It would normally be performed as part of ending an ORACLE session.

```
JNL    STOP
```

Start Database Journal

This function starts Journal activity for a database. Once Journal activity for a database has been initiated in an ORACLE session, Journal activity for that database will continue in future ORACLE sessions unless specifically turned off with the DBSTOP function.

```
JNL    DBSTART    dbname
```

where:

dbname	is the name of the ORACLE database.

Stop Database Journal

This function stops Journal activity for a database. This function would only be used after it was determined that there was no need to ever have to recover a particular database.

```
JNL    DBSTOP     dbname
```

where:

dbname	is the name of the ORACLE database.

Apply Journal Block Images

This is the function which applies the afterimages from the Journal to a saved copy of the databases in an ORACLE environment. When it is required to recover one or more databases in an ORACLE environment, those databases will be established by copying the saved versions to a disk and then performing the JNL APPLY function. Note that this function will apply the Journal images to all databases for which the Journal was activated, and that during the APPLY function those databases will be locked. If it is not necessary to involve all of those databases, the databases not being recovered **must** temporarily removed (DBF REMOVE function) from the database directory. The temporarily removed database could still be active by temporarily entering it (DBF ENTER function) with a different name.

The JNL APPLY function will automatically process the multiple files of the Journal for an ORACLE session in sequence. However if a Journal spans multiple ORACLE sessions, the JNL APPLY function must be executed for each session and in the proper sequence (note importance for effective file naming conventions). If, while processing the multiple Journal files of an ORACLE session, a file is required which is not available, a message will be displayed indicating the name of the file and processing will terminate. The APPLY function may be continued when the missing file is made available by invoking the JNL CONTINUE function.

```
JNL    APPLY    filename
```

where:

filename	is the name of the Journal file for an ORACLE session. The name must contain a "?" to indicate the position of the 3-digit number, incremented across the multiple files constituting the Journal for that session. On PDP-11 cpu's, the file may exist on disk or tape files. On VAX-11 cpu's, it must be a disk file.

Continue Applying Journal Images

The CONTINUE function is used when an APPLY function stops because a file is not available. This function permits continuation of the APPLY once the missing file is made available.

```
JNL    CONTINUE
```

Display Journal Status

JNL STATUS

This function causes a display of the status of the Journal Activity within the ORACLE environment. The Display function would always be used following an abnormal termination in order to determine the status of any database update activity. The Display function is also used to determine the size (in blocks) of the current Journal file and the total size of all Journal files of the current ORACLE session. The output format of the Display function is as follows:

```
+-------------------------------------------------------+
|                                                       |
|  DATABASE          DDL BAD JNL                        |
|  ---------------- --- --- ---                         |
|                                                       |
|  ORACLE            NO  NO  NO                         |
|                                                       |
|  PERSONNEL         NO  YES YES                        |
|                                                       |
|  DR1:[1,70]004.JNL is the current Journal file.       |
|  358 blocks have been written to the current file.    |
|  1969 blocks have been written to the Journal.        |
|                                                       |
|                                                       |
+-------------------------------------------------------+
```

where:

DATABASE	is the name of the database. All open databases will be included in the report
DDL	indicates that a DDL statement was in process during the failure, and that any portion of that DDL statement which was completed, will automatically be backed out the first time that database is reopened. Since DDL statements are single-threaded, if a DDL statement was interrupted by the failure, it is impossible for other activity to have also been interrupted.
BAD	indicates that the failure interrupted a database write and that the Journal must be applied to restore integrity to that database.

JNL	indicates that the Journal is active for that database. If integrity is lost in a database for which the Journal is not active, that database cannot continue to be used. A back-up copy must be restored before it can successfully be returned online.

A "YES" in the DDL column is only to indicate that the DDL statement in process at the time of the failure will automatically be backed out. Since the ORACLE system database is not journaled, a failure on it requires the use of the DBF utility to re-create it and enter the user databases.

IV JNL. Messages

There are 4 types of messages produced by the JNL utility:

1. Successful Completion
2. Task Failure Reason
3. General Error
4. Function-Specific Error

The messages are generally self-explanatory. The <item> in the message examples are replaced by an appropiate data value when the message is displayed. Those messages notated with the term "(ORACLE-error)", will be followed by the appropriate ORACLE error message (see Messages & Codes Manual).

Successful Completion Messages:

Journal processing complete
Journal started on database <database-name>
Journal stopped on database <database-name>
Journal started
Status report complete
Journal stopped

Task Failure Reason Messages:

Journal task canceled because
of bad ORACLE buffer cache
of missing ORACLE buffers
of missing KDA
of a database write failure
of a journal write failure
of a journal read failure

General Error Messages:

Not enough arguments
Ambiguous journal command
Invalid journal command
Cannot log on to ORACLE (ORACLE-error)
Cannot open the system database (ORACLE-error)
Journal utility in use
Cannot create journal log file

Function-Specific Error Messages:

APPLY Function:

Read failure on journal file <file-name>
Sequence number failure on journal file <file-name>
Incomplete ddl operation found on database <database-name>
Invalid journal block on journal file <file-name>
Journal file <file-name> not available
Type 'jnl continue' when file is available
Cannot create apply context file
Database <database-name> does not exist
Cannot open database <database-name>
Sequence number check
Question mark missing from file name
Invalid file name

DBSTART Function:

Invalid database name

DBSTOP Function:

Invalid database name

CONTINUE Function:

Cannot open apply context file
Cannot read apply context file

START Function:

Invalid size parameter
Question mark missing from file name
Invalid file name
Journal is already active

STATUS Function:

Cannot open parameter file <file-name>
Cannot read parameter file <file-name>

STOP Function:

Journaled databases are active
Journal is not active

DATABASE LOADER UTILITY

O D L

1.0 INTRODUCTION

ODL is the ORACLE utility for loading raw data from operating system files into an ORACLE database. ODL may be used to initially load a database table or to add records to an existing table. The ODL program functionally will:

1. read the user's input data;
2. perform the mapping from the raw data to a database table;
3. load the data records into a database table;

These functions are accomplished by using a description language as input to the ODL program. The general processing of the ODL program is as follows:

1. read the description language;
2. create the necessary SQL commands;
3. load the data using SQL INSERT commands.

The database table being loaded must have been created with the SQL "CREATE TABLE" statement prior to use of the ODL utility for that table.

2.0 RAW DATA REQUIREMENTS

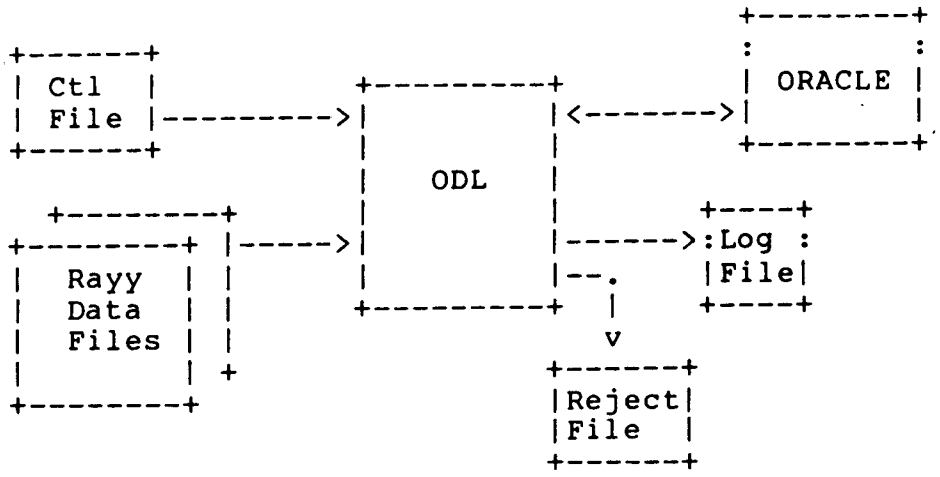
ODL requires the raw data to consist of one or more files of fixed-length records of a single format. The fields must also be fixed-length and those fields being loaded (not all fields need be loaded) must be of one of the following data types:

- Character String
- Binary Integer Number
- Floating Point Number

Future releases of ODL will support multiple record types, variable length records, and all host-supported data types.

3.0 ODL OPERATION

ODL will normally operate as a batch process. Therefore the control statements for ODL are stored on a control file and used during execution. A log file is utilized for error messages and statistics resulting from the execution of ODL. The raw data files are specified in the control statements. A reject file is maintained for up to 50 rejected raw data records (input records are rejected when they can not be loaded into a table).



ODL is invoked with the following statement:

```
ODL  ctl-file  log-file   database  [user-id/pswrd]
```

where:

ctl-file is the name of the file containing the ODL Control Language Statements.

log-file is the name of the file for logging error messages and statistics of the load processor.

database is the name of the database on which the load is being performed.

user-id is the name of the user as required for a secure database.

pswrd is the password for that user.

In the following example ODL is being invoked to load data into the Personnel database (secure). The ODL Control statements are on a file called load1.ctl and logging will be to a file called log1.

```
ODL  LOAD1.CTL  LOG1  PERSONNEL SCOTT/TIGER
```

The reject file will be given the name of the log file (less any extensions) plus the extension <.bad>. For example, it the log file was named <msg.log> (or just <msq>), the reject file would receive the name <msg.bad>. In the above example, the reject file will be named <LOG1.BAD>

Although ODL will allow up to 50 rejected raw data records before it terminates, the first record read must be acceptable or ODL will terminate.

4.0 ODL CONTROL LANGUAGE

The language is composed of definition and action statements. Definition statements define the raw data formats and the source files for the raw data. Action statements define the mapping of the input values and constants into the database table rows.

4.1 DEFINE RECORD STATEMENT

The DEFINE RECORD statement defines a particular input record. A list of one or more field definitions separated by commas, defines the raw data fields to be loaded.

```
DEFINE RECORD  rec-name  AS  [fld-def,...] fld-def;
```

where:

rec-name is a name used to refer to the raw data record.

fld-def is the data field definition described as follows:

```
fld-def = fld-name ( fld-type [, fld-loc] )
```

where:

fld-name is a name used to refer to a raw data field.

fld-type defines the data type as follows:

```
| FLOAT   |
| INTEGER | [ (size) ]
| CHAR    |
```

corresponding to a floating point, binary integer, and character. The size in bytes (enclosed in parenthesis) is optional and is required for sizes other than defaults. Acceptable sizes and defaults are as follows:

data type	sizes	defaults PDP-11	defaults VAX-11
FLOAT	8, 4	8	8
INTEGER	4, 2, 1	2	4
CHAR	1 thru 254	1	1

fld-loc defines the position of the field in the raw data record as follows:

```
                | + number |
LOC  ( | - number | )
                |   number |
```

An integer number without any sign is an absolute location from the beginning of the record. A sign in front of the number means the position is relative to the end of the last field. The minus sign is positioned towards the beginning of the record and the plus sign is positioned towards the end. The default position is at the end of the previous field or the beginning of the record for the first field.

If a database column is defined as NUMBER type and the raw data field is an ASCII string representing a numerical quantity, then the data type of the DEFINE RECORD filed must be CHAR. ORACLE handles the conversion to numerical format.

The following is a sample DEFINE RECORD statement:

```
DEFINE RECORD REC1 AS
   FLD1 (CHAR(6)), FLD2 (CHAR(25)),
   FLD3 (FLOAT, LOC (+3))
;
```

4.2 DEFINE SOURCE STATEMENT

The DEFINE SOURCE statement defines the input mediums to ODL and the records contained within this source.

```
DEFINE SOURCE   src-name   src-parms
    CONTAINING rec-name ;
```

where:

src-name is a symbolic name for this source.

src-parms provide the information necessary to read the source as follows:

```
FROM file-name [,file-name]...
LENGTH number
```

where:

file-name is the operating system file name of a raw data file. Multiple file names (separated by commas) can be used to refer to concatenated files.

length is the raw data record length.

rec-name is the record name used in the DEFINE RECORD statement.

Future releases will provide for multiple record types within a source and for multiple sources during a load.

The following is an example of a DEFINE SOURCE statement.

```
DEFINE SOURCE SRC1
   FROM FILE1, FILE2, FILE3
   LENGTH 80
   CONTAINING REC1
;
```

4.3 FOR EACH STATEMENT

The FOR EACH statement is a loop mechanism that reads one raw data record for each iteration. The FOR EACH statement defines the SQL INSERT statement to be executed for each record read. Future releases will provide for the use of multiple INSERT statements involving multiple database tables, and nested FOR EACH statements for hierarchical data.

```
FOR EACH RECORD    in-stmt    NEXT RECORD
```

where:

(in-stmt) is the SQL INSERT statement as follows:

```
INSERT INTO tab-name
    (col-name,...)    :  < src-data,... >
```

where:

tab-name identifies the database table.

col-name is the list of one or more columns, enclosed in parenthesis for which values are being loaded.

src-data is the list of raw data field names and data values, enclosed in angle brackets and separated by commas. The list can contain any of the following:

- o raw data field-name
- o character string
- o number
  (string of digits
  or scientific notation)
- o NULL

Note that the "NEXT RECORD" phrase is required (to terminate the loop function).

The following is an example of a FOR EACH statement.

```
FOR EACH RECORD
    INSERT INTO EMP ( EMPNO, EMPNAME, SALARY, COMM ) :
     < FLD1, FLD2, FLD3, NULL >
NEXT RECORD
```

5.0 ODL MESSAGES

All ODL messages are written to the log file. The following statistical messages are written to the log file:

```
# records successfully read
# records successfully loaded
```

When an error occurs on processing a record, the following message is produced:

```
RECORD # REJECTED
```

where # is the relative number of the record being read.

There are three types of errors which can occur during ODL processing:

- o Control Statement Errors
- o ODL Fatal Errors
- o ORACLE Errors

When an error occurs during processing of a raw data record, that record is written to the reject file. ODL will terminate after 50 records have been written to the reject file. If there is a problem during processing of the first raw data record, ODL will terminate without loading any records.

1. Control Statement Errors

These messages include information to help locate the error in the statement. There are 3 formats used as follows:

1) message ON LINE # COLUMN #
   where message is one of the following:
   - Number too large
   - String to large
   - Bad number
   - Missing exponent
   - Identifier too large
   - Illegal character
   - Quoted literal not ended

2) SYNTAX ERROR LINE # ON INPUT symbol
   where symbol identifies where the error was detected

3) symbol : message ON OR ABOUT LINE #
   where symbol and message are as follows:

symbol	message
rec field	Field name is ambiguous
rec field	Field is improperly aligned
rec field	Inconsistent rec field definition
rec field	Field location too negative
rec field	Bad location field expression
rec field	Inconsistent location field
rec field	Unsupported field length
source	Source from clause is missing
source	Source length clause missing
source	Multiple sources not supported
source	Bad source length clause
rec name	Record too large for source
rec name	Multiple records not supported
table	Table col/field count mismatch
table	Multiple tables not supported
symbol	Not a record field
symbol	Undefined record field
symbol	Previously defined
symbol	Record name expected
symbol	Undefined record name

2. ODL Fatal Errors

These errors occur because of an internal problem or because of a resource, such as memory, being exhausted. The format is as follows:

```
ODL FATAL ERROR : message
```

where message is as follows:

```
Out of parse stack space
Out of heap space *
Out of table space *
```

Those messages which can be corrected by linking ODL with more memory are indicated with an asterisk.

There are a set of fatal messages which can occur if other errors exist before the fatal error. These errors will usually disappear when the original error is corrected. These errors are as follows:

```
excsub : invalid kind
Illegal insert field definition
defsyn : unknown type
unreachable code - optional length
unreachable code - default length
```

3. ORACLE Errors

ORACLE error messages have the following format:

```
call ERROR : message
```

where:

call	is the name of the ORACLE interface subroutine which failed;
message	is the ORACLE error message as listed in the ORACLE Messages and Codes Manual.

6.0 SAMPLE ODL CONTROL LANGUAGE INPUT

The following ODL control language sequence defines a record containing an employee name, grade and salary. the salary field is located 10 bytes away from the end of the grade field.

The input source has two files with a record length of 80.

The database table EMPMAIN is loaded with fields from the EMPLOYEE raw data records and with some constants.

```
DEFINE RECORD EMPLOYEE AS
     NAME   (CHAR (20) ),
     GRADE  (INTEGER(2) ),
     SALARY (FLOAT(4), LOC( +10) );


define source tape
     from dr0:[1,110]file1, file 2
     length 80
     containing EMPLOYEE ;

FOR EACH RECORD
     INSERT INTO EMPMAIN
     ( NAME, PERFORMANCE, GRADE, GROSS,
        NULCOL, SALARY, START) :
     <NAME, 87.6543, EMPLOYEE GRADE, .86e + 6,
        NULL, SALARY, 356 >
NEXT RECORD
```

O R A C L E

UNLOAD/RELOAD DATABASE UTILITY

This utility provides the function of unloading tables of an ORACLE database onto a sequential file and later reloading them into a database. This is useful for:

- o physically reordering rows of a table;
- o unloading a table, changing column size, nonull, or image specifications, and reloading that table;
- o moving a database between versions of ORACLE when the internal format of the database changes.

The following specific functions are supported:

- o Unload all tables and/or views of a database;
- o Unload selected tables and/or views of a database ;
- o Unload tables and/or views without GRANT privileges;
- o Unload only the table and view definitions;
- o Reload the previously unloaded tables and/or views;
- o Reload the tables and/or views without GRANT privileges;
- o Reload previously unloaded tables by inserting into existing tables;
- o Display the names of tables and views on an unloaded database.

Tables and views of a secure database may only be unloaded and reloaded by the owners of the data (creators of the tables or definers of the views). Therefore Unload/Reload operations on a secure database require the use of the user-name/password parameter.

Unloading a database is called EXPorting and reloading a database is called IMPorting.

EXPORT (Unload Tables)

EXP will unload one or more tables and/or views of a database. EXP will display the names of tables and views unloaded, and will display row counts for each table unloaded.

```
EXP db-name [user-name/paswrd] [dumpfile] [- T V G D Cn ]
```

where:

db-name	is the name the database being unloaded.
dumpfile	is the optional name which if provided, will be used to name the output file. Otherwise, the file name will be <database-name>.
user-name/ paswrd	is the user-name and password required when unloading tables from a secure database.
T	is the option which causes EXP to prompt the user for the table-names identifying the tables to be unloaded. Entering a <return> in answer to the prompt signifies that no more tables are to be unloaded.
V	is the option which causes EXP to prompt the user for the view-names identifying the views to be unloaded. Entering a <return> in answer to the prompt signifies that no more views are to be unloaded.

G	is the option which precludes the unloading of GRANT privileges associated with the tables and views being unloaded.
D	is the option to unload **only** the table/view definitions and GRANT privileges.
Cn	is the option which overrides the default value for the size of the SWA (SQL Work Area) in ORACLE. The size is expressed as a number (n=1 to 16) of KBytes. The default is 3.

EXAMPLE 1 : EXP PERSONNEL SCOTT/TIGER PERS.SAV -C5

EXAMPLE 2 : EXP CREDIT -TV

The first example will cause the entire PERSONNEL database (secure) to be unloaded to a file called PERSONNEL.SAV. In this example the SWA is being overridden and set to 5 KB.

The second example will cause selected tables and views of the CREDIT (non-secure) database to be unloaded to a file called CREDIT.EXP. The users will will be prompted for the tables and views to be unloaded.

IMPORT (Reload Tables)

IMP will reload the one or more tables and/or views of a previously unloaded database. If for some reason, a table cannot be created in the reloaded database, processing continues with the next table. If a record can not be accepted, that record will be displayed and processing will continue with the next record. If a view definition is rejected, it will be displayed.

IMP will display the names of tables and views being reloaded, and for each table reloaded, will display the rows inserted. If the number of rows actually loaded is different than the rows read, this is indicated with a count of rows read.

```
IMP  db-name  [user-name/paswrd]  [dumpfile]  [- T V G D I Cn ]
```

where:

db-name	is the name of the database being reloaded. This may or may not be the same as the name of the database which was unloaded.
dumpfile	is the optional name which if provided, is used to reference the input file. Otherwise, the name of the input file will be assummed to be <database-name>.
user-name/ paswrd	is the user-name and password required when reloading tables to a secure database.

T	is the option which **prevents** reloading of tables.
V	is the option which **prevents** reloading of views.
G	is the option which **prevents** the inclusion of GRANT privileges on the tables and/or views being reloaded.
D	is the option which overrides all load options and simply displays the names of tables and views in the dump file
I	is the option which specifies that the reload is taking place against an existing table and the rows are to be inserted into that table. If this parameter is not specified, a reload against an existing table would result in no rows being inserted in that table.
Cn	is the option which overrides the default value for the size of the SWA (SQL Work Area) in ORACLE. The size is expressed as a number (n=1 to 16) of KBytes. The default is 3.

```
EXAMPLE 1 :  IMP  PERSONNEL  SCOTT/TIGER  PERS.SAV  -C5

EXAMPLE 2 :  IMP  CREDIT  -VI
```

The first example will cause the PERSONNEL database (secure) to be reloaded from a file called PERS.SAV with a SQL Work Area of 5KB.

The second example will cause only the tables of the unloaded CREDIT database (non-secure) to be reloaded from a file called CREDIT, and if any tables being reloaded are already in the database, the rows will be inserted.

R S I

ORACLE

INSTALLATION GUIDE

Oracle Programmer's Guide - Version 2.3

O R A C L E

INSTALLATION GUIDE

TABLE OF CONTENTS

1.0 INTRODUCTION

This document is a guide to the installation of the PDP-11 or VAX version of the ORACLE Relational Data Base Management System running under the RSX-11M, IAS or VMS operating systems. It is organized in the following manner:

Section 1 is the introduction;

Section 2 presents an architectural overview of ORACLE;

Section 3 presents the computer hardware required to install and use ORACLE;

Section 4 presents the procedure for installing ORACLE on an RSX-11M system;

Section 5 presents the procedure for installing ORACLE on an IAS system.

Section 6 presents the procedure for installing ORACLE on a VAX/VMS system.

Appendix A contains tuning information for ORACLE.

Appendices B-F are germane to the VAX/VMS installation.

Appendices G-J are germane to the PDP-11 RSX11M installation.

Appendices K-M are germane to the PDP-11 IAS installation.

Familiarity with the RSX-11M,IAS or VMS operating system is assumed in this document. For the purpose of review however, several terms are given brief definitions.

1. Task or Image -- a linked file which may be installed and loaded from disk by the operating system and subsequently executed.

2. Region or Partition -- a named area in memory which may be accessed by any task with appropriate privileges.

3. Library -- a collection of macros or routines in one file.

    a) A Macro library is maintained in ASCII format for use by the assembler.

b) An Object library contains assembled and/or compiled routines or subroutines. This is accessed by the task builder at link time. If all code in the library is reentrant, the library may be memory resident.

2.0 **ARCHITECTURAL OVERVIEW**

This section of the manual relates ORACLE's functional attributes to ORACLE's internal structure. This information is necessary to configure or tune ORACLE for an installation's particular application environment.

The ORACLE DBMS is composed of five functional areas. (See Figure 2.1) These are briefly described below:

1) ORACLE Code -- The ORACLE program itself contains three functional areas:

    a) The SQL Data Language - parses and compiles SQL statements.
    b) The Integrated Data Dictionary - decodes data records.
    c) The Kernel - manages space and provides database access routines.

2) Memory Resident Work Areas

    a) Database Buffer Pool Cache.
    b) ORACLE Interface Communication Region.
    c) Kernel Data Area.

3) User Interface Modules

    a) User Friendly Interface (UFI) - terminal interface.
    b) The Host Language Interface - user program interface.

4) The Oracle Cleanup Task

    a) Detects user task aborts and frees the database resources in use by abnormally terminating user programs and terminal processes.

5) The Journal Task

    a) Writes modified buffers to the journal files and to the database files.

    b) Synchronizes journal and database activities so that database integrity is insured.

ORACLE Components

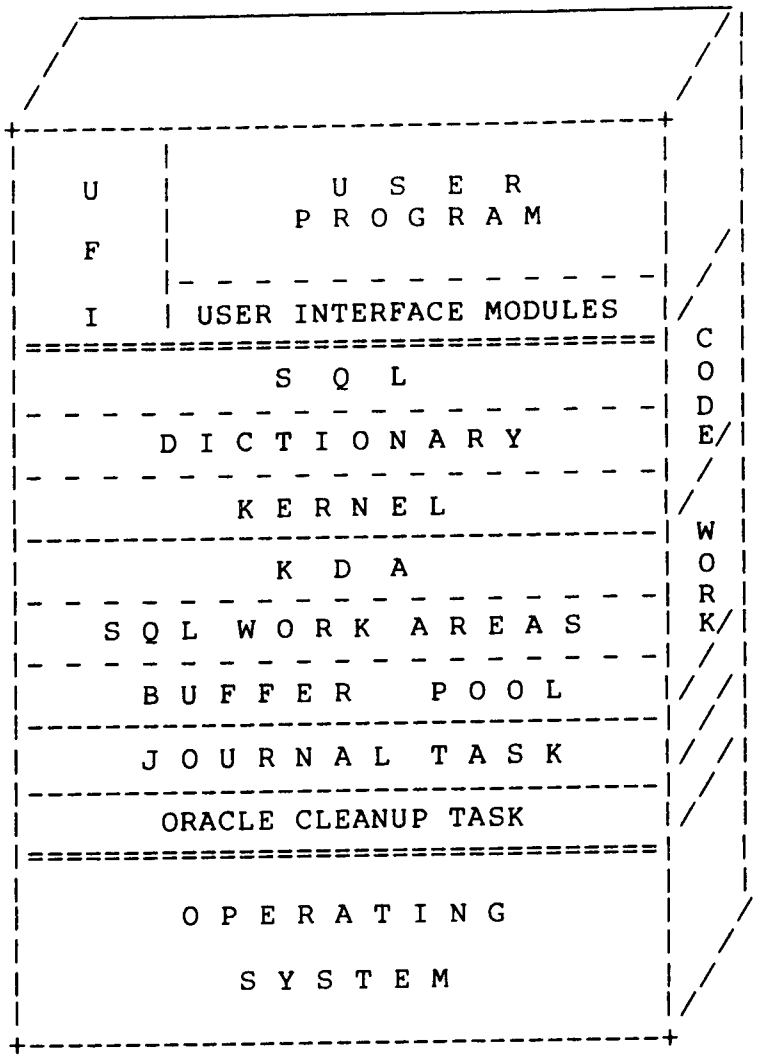


FIGURE 2.1

2.1 **ORACLE Task**

The ORACLE task is composed of a header and a stack area containing code which passes control to a shared code. The shared code consists of the resident root and eight memory resident overlays which logically contain the following: (1) the kernel; (2) the parser; (3) the SQL execute code; and (4) the space management segment (see the figure below).

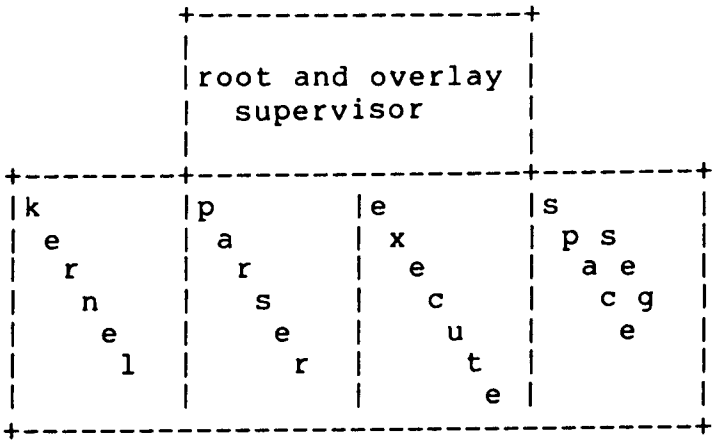


FIGURE 2.2

The Kernel provides the Database Management primitives, such as, insert or delete a record, scan an index, etc. It also contains the Data Dictionary manipulation routines.

The Parse and Execute segments comprise SQL. The SQL parser accesses the data dictionary to validate and translate to internal form all SQL statements. The Execute overlay then executes the SQL statement by making Kernel calls.

The Space Management segment is responsible for all disk space and buffer cache management in the database.

2.2 RegionsS

Three regions are required by ORACLE as work areas: a Buffer Pool Cache, a SQL work area Region, and the Kernel Data Area.

Buffer Pool
The Buffer Pool is used to store disk blocks that have been read into main storage. ORACLE disk blocks are 512 bytes long for the PDP-11. ORACLE maintains recently accessed data in the buffer pool. ORACLE uses a "least recently used" algorithm to control reuse of buffers from the pool. The buffer pool is shared by all concurrent copies of ORACLE.

SQL Work Area Region
The ORACLE SQL Work Area Region is used for passing data between the user task and ORACLE. The mapping context returned to the user by ORACLE during initialization is a window into this region.

When the user program issues an OPEN call, ORACLE allocates a SQL Work Area (SWA) from the SQL Work Area Region. The address of the calling program's cursor is placed into the SWA. The size of the SWA is 3K bytes unless otherwise specified in the OPEN call. SWA's may be from 1K to 16K bytes.

When the user program issues a SQL call ORACLE parses and then compiles the SQL statement. The compiled SQL statement is stored in the SWA.

The location of buffers and data type conversion information passed in the DEFINE call is stored in the SWA. BIND calls modify compiled SQL statements that are stored in the SWA.

When the user program issues an EXECUTE call for a SQL query statement and subsequently issues a FETCH call, the results of the query are temporarily stored in the SWA.

Kernel Data Area
The Kernel Data Area (KDA) is used to control allocation of serially reuseable resources within ORACLE. The KDA contains buffer and work area headers, ENQ lists, database control vectors and the dictionary cache.

The Buffer Headers control allocation and identify the contents of buffers in the buffer pool.

The Work Area Headers control the allocation of Logon Work Areas (LWA's) and SQL Work Areas (SWA's) from the SQL Work Area Region.

ENQ entries are used for locking out access to rows of tables as those rows are being updated.

Database and Extent Vectors are used to control an open database.

The Dictionary Cache maintains the definition of the active tables and the columns within those tables that are being accessed.

Each of these areas may be altered or tuned to reflect a specific environment. (See Appendix A)

2.3 USER INTERFACE MODULES

The primary interactive user interface to the ORACLE database system is the User Friendly Interface (UFI). UFI provides the user with the ability to enter all SQL commands at a terminal. Additionally, UFI can spool output to a file and optionally queue the file for printing, write SQL commands to a file, edit current SQL commands, and execute SQL commands from a saved file. UFI is installed with the task name SQL.

The Database File utility (DBF) is a task which provides the user with the ability to manipulate database files in an on-line environment.

When a user wishes to create his or her own database task, the distributed user interface modules must be included in the task image. The user code interface is provided in full in the HOST LANGUAGE INTERFACE Manual.

The Interactive Application Facilities (IAG and IAF) and the Report Writer (RPT and RPF) allow database application to be developed without the need for conventional programming.

2.4 **ORACLE Cleanup Task**

ORACLE maintains the status of each user/ORACLE task pair in the Global Task Table (GTT), which resides in the KDA. If a task terminates execution while ORACLE cursors are still open, certain cleanup of the ORACLE environment must be done. Hence, abnormal task termination must be sensed by the ORACLE data base system.

The ORACLE cleanup task (OCLNUP) is installed at system startup. It is run every 10 seconds to determine if any user task has aborted. OCLNUP serializes access to the GTT and scans it for entries belonging to the exiting task. For each entry found, the following functions are performed:

* Kill the appropriate ORACLE task
* Dequeue all owned resources
* Close open databases
* Free the Kernel context areas
* Free the SQL context area (in OCLCTX)
* Free any allocated I/O buffer
* Free the GTT entry

2.5 **Journal Task**

ORACLE uses a single task to control the updating of the databases and the creation of the database journal files. The journal task (JTS) synchronizes the writing of updated memory buffers to the database files on disk to insure that database integrity is maintained. The journal files are written prior to updating the databases so that the journal may always be used to restore the database to the most current state.

3.0 HARDWARE REQUIREMENTS

3.1 PDP-11

* CPU	PDP-11 with EIS
* Memory	128K bytes above the operating system requirements for ORACLE common partitions 64K bytes in a system controlled partition for UFI; 78K bytes in a system controlled partition for IAP or RPT
* Terminal	Any supported terminal
* Tape	9 track, 800 or 1600 BPI -- for tape distribution of ORACLE
* Disk Space	4000 blocks for ORACLE Distribution and task images
* Database Space	4096 blocks for the ORACLE system database, at least 4096 blocks for the ORAWRK database (optional), and at least 1024 blocks per user database. All databases require contiguous space for each extent (see the DBF users guide for a description of extents).

3.2 VAX

* CPU	VAX-11/780 or VAX-11/750
* Memory	300KB required above the operating system requirements
* Terminal	Any supported terminal
* Tape	9 track, 1600 BPI -- for tape distribution of ORACLE
* Disk Space	4000 blocks for ORACLE Distribution and task images
* Database Space	4096 blocks for the ORACLE system database, at least 4096 blocks for the ORAWRK database (optional), and at least 1024 blocks per user database. All databases require contiguous space for each extent (see the DBF users guide for a description of extents).

4.0 ORACLE/RSX-11M INSTALLATION PROCEDURE

In order to provide the appropriate RSX-11M environment for ORACLE, a number of options must be selected at system generation, and several new partitions must be added to the system. These issues are discussed below.

System Software:

RSX-11M Version 3.1 or later

The following sysgen options must be included:

* Mapped System
* SEND/RECEIVE Directives
* PLAS Directives
* STOP Directive

Partitions required in the system are as follows:

* ORACLE	302000 (8)	
* OCLCTX	12000 (8)	or larger
* OCLBUF	24000 (8)	or larger
* OCLKDA	20000 (8)	

Four partitions must be added to your system. This is most easily accomplished by the following procedure:

1. Make a new copy of all privileged tasks;
2. Make a new copy of RSX-11M.SYS from RSX-11M.TSK (assuming it was saved) and RSX-11M.STB;
3. Edit your VMR install command file to reflect partition changes;
4. INS [1,54]VMR;-1 INS [1,54]BOO;-1
5. Run VMR using the install command file and the new copy of RSX-11M.SYS;
6. Software boot the new system and save.

The following is a summary of the ORACLE installation procedure:

1. Log in under a privileged account.
2. Place distribution on appropriate device, allocate (and mount, if disk) the device.
3. Copy the distribution medium.
4. Execute the command file ORBLD.11M.
5. Log off the system.

4.1 Log on and Copy the Distribution Files

Log into your system under a privileged account in the form [g,m] where the group number must be in the range of 1 to 7.

1. For magtape distribution use the command

   ALL dev:

where "dev" is the device specification of the drive containing the distribution medium.

2. The following command will copy the ORACLE TAPE distribution files into the target UIC:

   FLX SY:/RS=dev:[*,*]*.*/DO

4.2 **Building ORACLE**

The distribution kit contains the file ORBLD.11M. This command file builds and installs ORACLE. Upon successful completion the validation test may be run and the results verified. In order to use the Interactive Application Facility, video terminal description files must be compiled. The descriptions for VT100, VT52, and OWL terminals may be built as part of the ORACLE installation.

Type the following command to build and install ORACLE:

```
@ORBLD.11M
```

The command file will place the task images on the SY: device and current UIC for the account to which you are logged on.

4.3 **Log Off**

Upon successful termination of the validation test the ORACLE Installation is complete. Log off the system by typing the command

```
BYE
```

The user may wish to add the invocation of the ORACLE install file ORINS.11M to the system startup command file, LB0:[1,2]STARTUP.CMD. ORINS.11M installs four copies of ORATSK named ORAAAA, ORAAAB, ORAAAC, and ORAAAD to support up to four users. If additional users are desired, ORINS.11M may be modified to install as many copies of ORATSK as are required. The task names must continue the sequence given above.

4.7 ORACLE Files

The following files are reguired to be on the same device and UIC as the ORINS.11M command file for ORACLE initialization to succeed:

OCLCTX.TSK	Context Region task image
OCLBUF.TSK	Buffer Region task image
OCLKDA.TSK	Kernel Data Region
OCLKDA.STB	KDA Symbol Table
ORACLE.TSK	ORACLE Library
ORACLE.STB	Library symbol table
ORATSK.TSK	The ORACLE Task
OCHECK.TSK	The ORACLE Task initiator
JTS.TSK	The journal creation task
UFI.TSK	User Friendly Interface (SQL)
DBF.TSK	Create Database Utility
IAP.TSK	Interactive Application Procesor
IAG.TSK	Interactive Application Generator
IMP.TSK	Import Utility
EXP.TSK	Export Utility
OIM.TSK	Old Import Utility
RPT.TSK	Report Writer Utility
RPF.TSK	Report Formatter Utility
JNL.TSK	Journal Utility
CRT.TSK	CRT Definition Utility
IAGUPGRD.TSK	IAF Upgrade Utility
ORINS.11M	ORACLE Installation Command File
SDLLIB.MLB	System Development Library

The following files must remain in LB:[1,1] for ORACLE to operate correctly (they are moved to LB:[1,1] during ORACLE installation):

DBFOST.	DBF initialization statements
ORAERR.MSG	ORACLE error message text

The following files must remain on a disk for use in task building programs which use the ORACLE Host Language Interface:

OFOLIB.OLB	FORTRAN, COBOL, and PASCAL interface routines
OCELIB.OLB	C interface routines
ORALIB.OLB	ORACLE interface routines
CLIB.OLB	C library routines - this library is different from any you may have received from a C compiler vendor

4.8 ORACLE Installation for RSX11M 3.1

There is a deficiency in RSX11M version 3.1 which does not allow the same task file to be installed with more than one task name. The task file ORATSK is intalled once for each potential concurrent ORACLE user with the task names 'ORAAAA', 'ORAAAB', etc. To install ORACLE on RSX11M version 3.1, the ORATSK.TSK file must be duplicated for each ORACLE task, then installed with the correct names. The MCR commands used to accomplish this feat are listed below. The INS commands should be inserted into the ORINS.11M command file.

```
>PIP ORAAAA.TSK=ORATSK.TSK ! MAKES DUPLICATES OF ORATSK
>PIP ORAAAB.TSK=ORATSK.TSK
>PIP ORAAAC.TSK=ORATSK.TSK
>PIP ORAAAD.TSK=ORATSK.TSK
>PIP OCLNUP.TSK=ORATSK.TSK ! MAKES A COPY OF ORATSK FOR
>                          ! ...THE CLEAN UP TASK
>!
>INS ORAAAA/TASK=ORAAAA    ! INSTALL THE ORACLE TASKS
>INS ORAAAB/TASK=ORAAAB    ! ...THESE INSTALLS SHOULD
>INS ORAAAC/TASK=ORAAAC    ! ...IN BE ORINS.11M
>INS ORAAAD/TASK=ORAAAD
>INS OCLNUP/TASK=OCLNUP
```

5.0 ORACLE/IAS INSTALLATION PROCEDURE

System Software:

IAS Version 3.0 or later

Create a privileged account for ORACLE; include privilege for logical block I/O to a disk. Choose the disk on which the database will be built. If this is the system disk timesharing must be shut down and the disk must be mounted DCF from MCR.

Ex: MOU DB0:IASSYS/CHA=[DCF]

If the disk is other than the system disk it should be mounted (preferably at system start up) from a privileged account with the following command:

Ex: MOU/NOOP/GLO/CON dkx:name

The following is a summary of the IAS installation procedure:

1. Log in on the ORACLE account
2. Place distribution media on appropriate device and mount
3. Copy distribution media
4. replace the IAS task builder with the one from the distribution
5. Edit the following .SRC files to reflect site requirements: OCLKDA,OCLBUF,OCLCTX,ORBLD
6. Execute the command file ORBLD.IAS
7. Log off

5.1 LOG ON

Log into the privileged ORACLE Account with the command

LOGIN ORACLE A

5.2 Place Distribution Media on System

1. For magtape distribution

MOU/NOOP/FOR mtx: X

where "mtx" is the device and unit specification of the drive containing the distribution medium.

2. For disk cartridge distribution use the command

```
MOU/NOOP dkx: ORACLE
```

where "dkx" is the device specification of the disk drive containing the ORACLE distribution.

5.3 Copy the Distribution Files

1. The following command will copy the ORACLE distribution files into the target UIC:

```
MC FLX SY:/RS=mtx:[*,*]*.*/DO
```

2. For disk cartridge distribution use the command

```
COPY dkx:[1,50]*.*  *.*
```

5.4 Replace IAS Task Builder

The task builder distributed with IAS does not handle memory resident overlays properly. If you have not obtained the corrected task builder from DEC remove the one currently installed in the system and install the corrected version.

```
REM ...TKB
INS TKB
```

5.5 Edit Source Files

Edit the ORACLE build command file, ORBLD.IAS to set the creation of the system database to the proper disk. (It must be mounted Control Functions Enabled).

Edit the files OCLCTX.SRC and OCLBUF.SRC to reflect the number of users which must be concurrently supported by this installation.

The system may be further tuned by changing OPARAM (ref. APPENDIX A)

5.6 **Build ORACLE**

The distribution kit contains the file ORBLD.IAS. This command file builds and installs ORACLE.

Type the following command to build and install ORACLE:

```
@ORBLD.IAS
```

5.7 **Log Off**

Upon successful termination of the validation test, the ORACLE Installation is complete. Log off the system by typing the command

```
LOGO
```

5.8 **ORACLE FILES**

[1,x]	OCLCTX.TSK	Context Region task image
[1,x]	OCLBUF.TSK	Buffer Region task image
[1,x]	OCLKDA.TSK	Kernel Data Region
[1,x]	OCLKDA.STB	KDA Symbol Table
[1,x]	ORACLE.TSK	ORACLE Library
[1,x]	ORACLE.STB	Library symbol table
[1,x]	ORATSK.TSK	The ORACLE Task
[1,x]	UFI.TSK	User Friendly Interface
[1,x]	DBF.TSK	Database File Utility
[1,x]	CRT.TSK	CRT Definition Utility
[1,x]	IAG.TSK	Create Form Task
[1,x]	IAP.TSK	Forms Transaction Processor
[1,x]	IMP.TSK	Import Utility
[1,x]	EXP.TSK	Export Utility
[1,x]	OIM.TSK	Old Import Utility
[1,x]	RPT.TSK	Report Writer Utility
[1,x]	RPF.TSK	Report Formatter Utility
[1,x]	ORBLD.IAS	ORACLE Build Command File
[1,x]	ORINS.IAS	ORACLE Install Command File
[1,x]	ORREM.IAS	ORACLE Remove Command File
[1,x]	SQLBLD.IAS	Personnel Database Build
[1,x]	SQLDEMO.IAS	Demo using Personnel DB

6.0 ORACLE/VMS INSTALLATION PROCEDURE

This chapter outlines the procedures for installing the ORACLE system on a DEC 11/780 VAX computer running under version 2.0 of the VMS operating system. These procedures correspond with ORACLE version 2.3.

6.1 VAX Installation Tape

The VAX installation tape contains a container file of the files necessary to run the ORACLE Relational Database Management system. The files are restored into a single directory using the RMS RESTORE utility (RST).

The tape contains files necessary to install, operate, and maintain the ORACLE system. They fall into the following categories:

- Command files to load and install ORACLE;
- Image files of the ORACLE system and related utilities;
- Library files containing the native mode user interface;
- Source files for those modules containing database tuning parameters;

Appendix 'B' is a listing of the files contained on tape.

6.2 Pre-Installation Activities

Prior to installing the ORACLE system and restoring the installation tape the following activities must be completed:

1. A new account must be defined on VMS, for example 'ORACLE', which has the privileges and quotas defined in appendix 'C'. Note that all ORACLE accounts must have UIC's with the same group code as ORACLE (ex: [100,x], where x can have any value). The required privileges and quotas for a user account are listed in Appendix 'C'. The user log in command file should invoke SYS$ORACLE:ORACLE.COM

6.3 Installation Process

1. Log onto the ORACLE system account and use the following two commands to invoke the ORACLE bootstrapping loader on the installation tape:

```
MOUNT MTA0: ORACLE

@MTA0:ORBOOT
```

2. The loader will announce itself and ask for the name of the device to load from. Answer with the same device name used above (i.e., MTA0:).

3. You will be prompted for any additional information necessary as the procedure continues.

Note: The following is a summary of the operation of the loader.

After purging and renaming all files existing in the directory, the loader will proceed to load the new versions in from the tape. When done, the loader passes control to the newly loaded procedure INSTALL.COM. The INSTALL procedure first insures that there is a LOGIN.COM files which calls ORACLE.COM. Second, INSTALL configures GRPSYM.COM and ORACLE.COM so that they define SYS$ORACLE as the current directory. Next, it calls ORACLE.COM to define ORACLE symbols. If you are using a different VAX/VMS version than the one on which the ORACLE system was built, INSTALL will call LINKORA.COM to relink. The LINKORA procedure reads the required object libraries off the tape and relinks all ORACLE components.

6.4 ORACLE Installation and Validation

The following procedures will install and activate the ORACLE system and validate the installation.

1. Log on using the 'ORACLE' account.

2. Type @ORINIT to activate the ORACLE system.

3. Create a system database using the command below:

```
$ DBF C ORACLE ORACLE.DBS 4096
```

   The name of the system database (oracle.dbs) and the size (4096) are installation dependent. Refer to the ORACLE Utilities guide for a description of these parameters.

4. Create and define the 'personnel' user database using the following command:

```
$ DBF C PERSONNEL PERSONNEL.DBS 2K SCOTT/TIGER
```

5. Build the PERSONNEL database tables.

```
$ SQL @SQLBLD.SQL
```

6. Execute ORACLE validation demonstration.

```
$ SQL @SQLDOC.SQL
```

   At the completion of this step ORACLE is functionally installed and validated. The system is now available for general use by other users.

6.5 Daily ORACLE Initialization

To initialize ORACLE on a daily basis perform the following steps:

1. Log on to ORACLE
2. Enter @ORINIT to load the ORACLE system.
3. Define the system database:

```
$ DBF SD ORACLE.DBS
```

To bring down ORACLE at the end of the day perform the following steps:

1. Log on to ORACLE
2. Type @ORSTOP
3. Log off.

6.6 Site Specific Changes to ORACLE

RUNORA is the executable image which creates the detached ORACLE processes. It determines both the base priority at which an ORACLE process runs and how many concurrently active ORACLE processes can be created. As delivered, RUNORA activates an ORACLE process at priority 4 and supports six concurrent users. To change either of these parameters, edit the RUNORA options file, RUNORA.OPT. Relink RUNORA by typing the command:

```
@BLDRUNORA
```

Then reinitialize ORACLE.

APPENDIX A: SYSTEM TUNING

A.1 Kernal Data Area

There are several parameters in the KDA which may be altered by the user to reflect his environment. The KDA parameters are contained in a macro in SDLLIB.MLB which is supplied with ORACLE. The macro OPARAM must be extracted from the library, edited to change the desired parameters, the replaced in the library. The KDA must then be reassembled and the object program replaced in KDALIB.OLB using the librarian utility. ORACLE tasks rebuilt by the command file ORBLD distributed with your system. The maximum allowable size of the KDA is 8K bytes. The parameters of note are as follows:

BUF$N	Number of 512 byte buffers in OCLBUF; This is adjusted at run time to reflect the actual size of OCLBUF. ORACLE will not execute properly with fewer than 20 buffers (24000 bytes octal dedicated to the buffer region.
ENQ$N	Number of entries on Resource Wait lists.
DXV$N	Number of Database Context Vectors; Controls the number of databases which may be concurrently open.
EXV$N	The total number of extens which may comprise the concurrently open databases.
GTT$N	Number of User/ORACLE task pairs; Controls the number of users and user tasks which are concurrently logged onto ORACLE.
CST$N	Number of active cursors: Controls the number of cursors which may be concurrently open.
RLC$N	Number of Relations on Cache; This controls the size of the table definition cache. This parameter should approach or exceed the number of tables concurrently accessed to reduce dictionary access.

DOC$N — Number of Domains on Cache; This parameter controls the number of data item definitions which are cached in memory. Occurrence of dictionary access on disk will diminish as this parameter approaches the number of Domains concurrently used.

SQL$N — Maximum number of 1K SQL Work Areas; This is set at run time to reflect the size in 1K areas of OCLCTX if it is smaller than SQL$N.

SWP$N — Number of context swap areas in the ORACLE system database.

FLS$N — A threshold number to control the writing of modified buffers which should be equal to the number of buffers divided by 8. If there are fewer than 32 buffers (20 is the minumum number), FLS$N should be set to 2. Use of to small a value for FLS$N will result in degraded system performance, especially when database journaling is active.

A.2 **SQL Work Area**

The SQL Work Area (OCLCTX) must be large enough to accommodate the largest number of concurrent users you wish to support with your system. It is the area from which individual SWA's are allocated. A user may allocate a maximum of 16K bytes of work space. The default allocation is 3K bytes; thus, the distributed copy of OCLCTX will support a maximum of four users if each uses the default size. An interactive user may control the allocation of buffer space in 1K increments through UFI with the "#WORKSIZE" n command, where 1 <= n <= 16. A user program may control allocation at LOGON and when OPENing databases.

To change the size of OCLCTX edit the file OCLCTX.SRC to reflect the new size, reassemble the module and task build. (If you are using RSX11M, then the size of the context area may be altered by changing the size of the OCLCTX partition to the desired size. When OCLCTX is installed into the OCLCTX partition, its size will increase to fill the partition. There is no need to edit OCLCTX.SRC, or to rebuild OCLCTX.) After relinking OCLCTX, reinitialize the ORACLE system to incorporate the new region.

A.3 ORACLE Buffer Area

The ORACLE Buffer Region (OCLBUF) contains buffers for I/O operations between the user database, the system database, and the ORACLE Kernal. Since these buffers are used as a data cache, disk I/O may be reduced significantly by increasing the size of OCLBUF. The effect of size increase is application dependent, specifically on the number of concurrent users, concurrent active databases and the randomness of database use. The minimum size of the OCLBUF is 20 buffers (10240. bytes). The maximum (and recommended) size is 64 buffers (32768. bytes).

To change the size of OCLBUF, edit the file OCLBUF.SRC to reflect the new size, reassemble the module and task build. (If you are using RSX11M, then the size of the buffer area may be altered by changing the size of the OCLBUF partition to the desired the size. When OCLBUF is installed into the OCLBUF partition, its size will increase to fill the partition. There is no need to edit OCLBUF.SRC, or to rebuild OCLBUF.) After relinking OCLBUF reinitialize the ORACLE system to incorporate the new region.

A.4 Database Structure

An ORACLE installation consists of at least two databases: the system database and one or more user databases.

System Database

The system database contains control information used in locating user databases. The system database also provides work areas for sorting and other operations on user databases. The system database name is always 'ORACLE' and must be at least 4096 blocks in size. DBF commands are used to create, initialize, or re-initialize the system database. After it is created, ORACLE must be informed of its file name every time ORACLE is initialized by using the 'DBF SD' command.

ORACLE Work Database

An ORACLE work database may be created which will be used by ORACLE for performing ORDER BY, UNIQUE, and GROUP BY operations. It may be of any size greater than 4096 blocks and it my be composed of more than one extent. The name of the work database is always ORAWRK. It may be entered and removed using DBF at will.

User Databases

Each user database is composed of one or more operating system files. A user database is composed of a dictionary, indexes and user data clusters.

The following information may prove useful in approximating the size of a database:

A 500 block area is reserved in the database for user views.

Each record which is stored in the database has a 24 byte header and a 2 byte header for each non-null item in the row. Additionally, the database space allocation has a 64 byte granu arity which must be considered when sizing the database.

ORACLE allows any data item within a table to have an index associated with it. These compressed key indices allow rapid access to data when querying or updating the database. A key or image is (7 + key) bytes in length. The format is shown below:

L length byte
FB flag byte
FC Forward compression
RBA Relative Byte Address (4 bytes)
KEY Compressed key (1.1 bytes, average)

A balanced tree-structure is used to build the database keys. Thus, the use of forward and backward data compression on the lowest key level provides optimum space utilization and minimum access overhead. Index blocks split when they become full. Thus, on the average an index block will be approximately three-fourths full.

The following formulae approximate the size of a database in bytes:

TS = RS + IS + 262144

IS = 1.33 * 8.1 * NK

RS = SUM (NR(i) * CEIL { ARS(i) / 64 } * 64), i = 1 to NT

ARS(i) = 24 + (SUM (AS(i,j) + 2), j = 1 to NC(i))

where

TS is the total size of the database in bytes

IS is the size of the image (index) area in bytes

RS is the size of the row area in bytes

NK is the total number of non-null occurrences of imaged columns

NR(i) is the total number of rows in table i

ARS(i) is the average size of each row in table i

AS(i,j) is the average size of data in column j of table i

NC(i) is the number of columns in each row of table i

NT is the number of tables in the database

CEIL is a function which takes the minimum integer larger then the number, e.g., CEIL [2.1] = 3.

SUM (AS + 2) for each item is the sum of the average sizes of the column values

NR is the number of rows in a table

NK is the number of keyed (imaged) items in a table

NI is the number of nonull items in a row

APPENDIX B: VAX DISTRIBUTION DIRECTORY

Directory _DRA0:[ORACLE]

BLDRUNORA.COM	RUNORA link command file
BSE.EXE	RSI Full Screen Editor
CLIB.OLB	"C" Object library
CRT.EXE	CRT Definition Utility
DBF.EXE	DBF Utility
DBFOST.	System Database structure
DEMO.INP	IAF demo source file
DEMO.RPT	RPT demo source file
EXP.EXE	Export Utility
GRPSYM.COM	ORACLE group symbol definition command file
IAG.EXE	Interactive Application Generator Utility
IAFDEMO.SQL	IAF DEMO database initialization command file
IAP.EXE	Interactive Application Processor Utility
IMP.EXE	Import Utility
JNU.EXE	Journal Utililty
JTS.EXE	Journal File Creation/Buffer Flush Task
KDA.SRC	Kernal Data Area (OCLKDA) source
LINKORA.COM	ORACLE Component relink command file
LOCK.EXE	LOCK image -- Locks part of ORACLE in memory
LOCK1.EXE	LOCK1 image -- Locks part of ORACLE in memory
OCLBUF.EXE	OCLBUF global section
OCLBUF.SRC	OCLBUF Source
OCLCTX.EXE	OCLCTX global section
OCLCTX.SRC	OCLCTX source
OCLKDA.EXE	OCLKDA global section
OCLKDA.TSK	OCLKDA backup image
OCLNUP.EXE	ORACLE Cleanup image
OIM.EXE	Old Import Utility (Imports an ORACLE 2.2 file)
ORAAAx.EXE	ORACLE images (where x = A-X)
ORACLE.COM	ORACLE User Symbol definition (reference in LOGIN.COM)
ORACLE.EXE	ORACLE Global section
ORACLE.TSK	ORACLE Global section backup
ORAERR.MSG	ORACLE Error message file
ORAFOR.MAR	FORTRAN HLI Interface source
ORALIB.OLB	General User/ORACLE Interface Library
ORAPAD.OPT	User image link option file
ORDIR.COM	ORACLE account subdirectory creation
ORINIT.COM	ORACLE initialization command file
ORINS.COM	ORACLE "install" command file

OROBJRST.COM	ORACLE objects restore command file
ORSTOP.COM	ORACLE Shutdown command file
ORSTOP.EXE	ORACLE Shutdown executable image
ORVAX.OLB	Library containing RUNORA objects
REGDESC.EXE	ORACLE Region Descriptor Global Common
RJTS.COM	Command file to create detached process running JTS
RLOCK.COM	Command file to create detached process running LOCK
RLOCK1.COM	Command file to create detached process running LOCK1
RORA.COM	Command file to create detached process running RUNORA
RPF.EXE	Report Formatter Utility
RPT.EXE	Report Writer Utility
RPTDEMO.SQL	RPT demo database initialization command file
RSI.EXE	ORACLE AME
RUNORA.EXE	Executable image to create detached ORACLE processes
RUNORA.OPT	Link option file for RUNORA
SAMPL3.FOR	Sample FORTRAN program
SQLBLD.SQL	Validation Database build file
SQLDOC.CMD	Validation Command file with comments
SQLDOC.SQL	Validation Command file
UFI.EXE	USER FRIENDLY INTERFACE Utility

APPENDIX C: VAX PRIVILEGES AND QUOTAS

Main ORACLE
Process privileges :

ALTPRI	may set priority to any level
CMKRNL	may change mode to kernel
GRPNAM	may insert in group logical name table
DETACH	may create detached processes
EXQUOTA	may exceed quotas
LOG_IO	may do logical I/O
GROUP	may affect other processes in same group
PSWAPM	may change process swap mode
TMPMBX	may create temporary mailbox
SYSGBL	may create system wide global sections
MOUNT	may execute mount ACP functions

ORACLE User
Process priveleges :

GRPNAM	may insert in group logical name table
GROUP	may affect other processes in same group
TMPMBX	may create temporary mailbox

Process Quotas:	ORACLE	USER
CPU limit :	INFINITE	INFINITE
Buffered I/O byte count quota :	20480	4096
Timer queue entry quota :	10	10
Paging file quota :	10000	10000
Direct I/ O limit :	12	6
Buffered I/O limit:	12	6
Open file quota :	30	20
Subprocess quota :	8	2
AST limit :	10	10

APPENDIX D: VAX COMMAND FILES

Listing of ORACLE.COM

```
$ASS _DRA0: SYS$ORACLE
DBF:==$SYS$ORACLE:DBF
DTM:==$SYS$ORACLE:DTM
EXP:==$SYS$ORACLE:EXP
IAG:==$SYS$ORACLE:IAG
IAP:==$SYS$ORACLE:IAP
IMP:==$SYS$ORACLE:IMP
OIM:==$SYS$ORACLE:OIM
RSI:==RUN SYS$ORACLE:RSI
SQL:==$SYS$ORACLE:UFI
```

Listing of GRPSYM.COM

```
$ASS/GR DRA0: SYS$ORACLE
$ASS/GR SYS$ORACLE:OCLNUP OCLNUP
$ASS/GR SYS$ORACLE:ORAAAA ORAAAA
$ASS/GR SYS$ORACLE:ORAAAB ORAAAB
$ASS/GR SYS$ORACLE:ORAAAC ORAAAC
$ASS/GR SYS$ORACLE:ORAAAD ORAAAD
$ASS/GR SYS$ORACLE:ORAAAE ORAAAE
$ASS/GR SYS$ORACLE:ORAAAF ORAAAF
$ASS/GR SYS$ORACLE:ORAAAG ORAAAG
$ASS/GR SYS$ORACLE:ORAAAH ORAAAH
$ASS/GR SYS$ORACLE:ORAAAI ORAAAI
$ASS/GR SYS$ORACLE:ORAAAJ ORAAAJ
$ASS/GR SYS$ORACLE:ORAAAK ORAAAK
$ASS/GR SYS$ORACLE:ORAAAL ORAAAL
$ASS/GR SYS$ORACLE:ORAAAM ORAAAM
$ASS/GR SYS$ORACLE:ORAAAN ORAAAN
$ASS/GR SYS$ORACLE:ORAAAO ORAAAO
$ASS/GR SYS$ORACLE:ORAAAP ORAAAP
$ASS/GR SYS$ORACLE:ORAAAQ ORAAAQ
$ASS/GR SYS$ORACLE:ORAAAR ORAAAR
$ASS/GR SYS$ORACLE:ORAAAS ORAAAS
$ASS/GR SYS$ORACLE:ORAAAT ORAAAT
$ASS/GR SYS$ORACLE:ORAAAU ORAAAU
$ASS/GR SYS$ORACLE:ORAAAV ORAAAV
$ASS/GR SYS$ORACLE:ORAAAW ORAAAW
$ASS/GR SYS$ORACLE:ORAAAX ORAAAX
$ASS/GR SYS$ORACLE:ORAAAY ORAAAY
$ASS/GR SYS$ORACLE:ORAAAZ ORAAAZ
```

APPENDIX E -- Sample VAX Installation (VMS 2.1)

APPENDIX F -- Sample VAX Installation (VMS 2.0)

APPENDIX G -- RSX11M Distribution Directory

APPENDIX H -- RSX11M Command Files

APPENDIX I -- Sample RSX11M Installation

APPENDIX J -- ORACLE Installation in Small RSX11M Configurations

APPENDIX K -- IAS Distribution Directory

APPENDIX L -- IAS Command Files

APPENDIX M -- Sample IAS Installation

R S I

ORACLE

MESSAGES AND CODES

Oracle Programmer's Guide - Version 2.3

O R A C L E

MESSAGES & CODES MANUAL

TABLE OF CONTENTS

ORACLE STATUS CODES (POSITIVE NUMBERS)

1	000001	fetched item unchanged in join
2	000002	fetched item null
3	000003	fetched item truncated
4	000004	end of fetch operation

KERNEL ERRORS (-1 to -29)

-1	177777	invalid operation code
-2	177776	no context areas available
-3	177775	non primary item specified during tuple create
-4	177774	invalid owning relation id on tuple create
-5	177773	tuple not found
-6	177772	item not found
-7	177771	i/o error
-8	177770	invalid item definition
-9	177767	duplicate item value in index
-10	177766	duplicate item in tuple
-11	177765	user specified context area invalid
-12	177764	invalid relation definition
-13	177763	index only operation and item not indexable
-14	177762	tuple create and value not unique.
-15	177761	exceeded max owning tuples in record space
-16	177760	current record space has max leaf tuples
-17	177757	invalid item length on input
-18	177756	database full
-19	177755	item too long for users buffer
-20	177754	end of index
-21	177753	item replace, lengths not equal
-22	177752	sxv copy and type not index

-23	177751	sxv compare, both equal
-24	177750	sxv compare, sxv gt buf
-25	177747	sxv compare, sxv lt buf
-26	177746	invalid sxv on temporary index add
-27	177745	read by key and item not indexed
-28	177744	item too large to index
-29	177743	tuple not accessed for update

DICTIONARY ERRORS (-30 to -45)

-30	177742	journal is not active
-31	177741	invalid database name
-32	177740	database already exists
-33	177737	database doesn't exist
-34	177736	cannot open database
-35	177735	getbuf or relbuf error
-36	177734	invalid owning relation
-37	177733	invalid size specification
-38	177732	relation already exists
-39	177731	invalid item attributes
-40	177730	second primary domain specified
-41	177727	domain already exists
-42	177726	cannot lock the database
-43	177725	invalid dxv
-44	177724	not enough space
-45	177723	invalid database
-46	177722	database file does not exist
-47	177721	database file is too small
-48	177720	system database in use
-49	177717	maximum number of columns in table exceeded
-50	177716	bad database

ENQUEUE/DEQUEUE ERRORS (-51 to -59)

-51	177715	enqueue list area over flow
-52	177714	resource not available and wait not requested
-53	177713	invalid enq request block
-54	177712	deadlock detected

SEQUENTIAL SEGMENT ERRORS (-60 to -99)

-60	177704	segment not found
-61	177703	segment already exists

DDL ERRORS - -100 to -199

-100	177634	table created
-101	177633	table expanded
-102	177632	first column in table must be indexed or linked
-103	177631	link already exists
-104	177630	relations are already linked
-105	177627	invalid link name
-106	177626	column already exists
-107	177625	view name is already a table name
-108	177624	view already exists
-109	177623	complex item in primary select list
-110	177622	same column name specified twice
-111	177621	table referenced by a view
-112	177620	table not empty
-113	177617	view referenced by a view
-114	177616	user already defined

DML ERRORS (-200 to -299)

-201	177467	trying to update primary item
-202	177466	trying to delete mandatory item
-203	177465	missing mandatory item

QUERY ERRORS (-300 to -399)

-300	177324	end of fetch
-301	177323	trying to select nongroup attribute
-302	177322	user buffer too small
-303	177321	item specified not in select or set list
-304	177320	where clause insufficient for join

PARSE ERRORS (-400 to -499)

-401	177157	invalid sequel statement
-402	177156	missing right square bracket
-403	177155	missing 'table' keyword
-404	177154	missing 'add' keyword
-405	177153	missing 'column' keyword
-406	177152	invalid create statement
-407	177151	invalid table name
-408	177150	invalid column name
-409	177147	invalid column type
-410	177146	missing left parenthesis
-411	177145	invalid column size
-412	177144	invalid column modifier
-413	177143	missing right parenthesis
-414	177142	invalid link name
-415	177141	missing 'from' keyword
-416	177140	missing 'to' keyword
-417	177137	missing semi-colon
-418	177136	invalid column definition
-419	177135	missing where clause
-420	177134	invalid drop statement
-421	177133	table name not in from list
-422	177132	ambiguous column name
-423	177131	missing 'into' keyword
-424	177130	missing colon

-425	177127	missing right angle brackets
-426	177126	missing left angle brackets
-427	177125	invalid constant
-428	177124	too many constants in tuple
-429	177123	missing 'in' keyword
-430	177122	missing 'and' keyword
-431	177121	invalid comparision
-432	177120	set function not allowed here
-433	177117	missing 'by' keyword
-434	177116	too many items in select list
-436	177114	missing 'set' keyword
-437	177113	missing equal sign
-438	177112	unexpected end of sequel statement
-439	177111	invalid substitution string
-440	177110	missing period
-441	177107	invalid define statement
-442	177106	missing 'as' keyword
-443	177105	invalid raw text length
-444	177104	missing 'select' keyword
-445	177103	view table not allowed here
-446	177102	asterisk not allowed here
-447	177101	duplicate table name
-448	177100	wrong number of items in select list
-449	177077	invalid view name
-451	177075	invalid number
-452	177074	unknown substitution variable data

		type
-453	177073	incompatible data types
-454	177072	non-numeric data types
-455	177071	null not allowed here
-456	177070	invalid logical operator
-457	177067	'order by' not allowed in view or nested query
-458	177066	reserved table
-459	177065	quoted literal not ended
-460	177064	missing 'but' or 'rights' keyword
-461	177063	'all but' not allowed with update rights on columns
-462	177062	invalid grant option
-463	177061	missing 'on' keyword
-464	177060	invalid keyword
-465	177057	missing '/'
-466	177056	missing 'with' keyword
-467	177055	missing 'connect' keyword
-468	177054	missing 'transaction' keyword
-469	177053	missing comma
-470	177052	invalid transaction number
-471	177051	column name is too large
-472	177050	table name is too large
-473	177047	view name is too large
-474	177046	user name is too large
-475	177045	password is too large
-476	177044	null row indicator not allowed on first table

JOURNAL ERRORS (-500 to -599)

-500	177014	journal utility already in use
-501	177013	invalid journal file

MISCELLANEOUS ERRORS (-600 to -699)

-600	176650	invalid request type
-601	176647	cannot open user database
-602	176646	invalid sdg
-603	176645	invalid database
-604	176644	invalid data type
-605	176643	bind variable does not exist
-606	176642	prior logon not issued
-607	176641	prior parse not issued
-608	176640	not all variables bound
-609	176637	prior execute not issued
-610	176636	operation terminated by user
-611	176635	cursor control area modified by user
-612	176634	unimplemented operation
-613	176633	invalid length of input value
-614	176632	insufficient parameters
-615	176631	invalid context area size requested
-616	176630	invalid cursor
-617	176627	cannot open system database
-618	176626	cannot allocate system swap area
-619	176625	logon not issued
-620	176624	oracle routines in same window as context area
-621	176623	invalid user identifier
-622	176622	security violation

-623	176621	error creating security record
-624	176620	database does not support security
-625	176617	non supported conversion

SYSTEM ERRORS (-700 to -799)

-701	176503	insufficient work area
-702	176502	quad header missing
-703	176501	database name not specified
-704	176500	insufficient system sort space
-705	176477	abort lun assignment error
-706	176476	no rsx11m dynamic memory
-707	176475	global task table full

RSX11M AND IAS ERRORS (-800 to -899)

-801	176337	gtsk$s failure
-802	176336	cannot create oracle task
-803	176335	cannot attach context region
-804	176334	ast failure
-805	176333	gmcx$s failure
-806	176332	elaw$s failure
-807	176331	craw$s failure
-808	176330	default uic different than logon uic
-809	176327	VMS initialization failure (mb or ef cluster)

UNIX ERRORS (-900 to -999)

-901	176173	pipe creation error
-902	176172	fork error
-903	176171	pipe write error
-904	176170	pipe read error