REPORT DOCUMENTATION

1. AGENCY USE ONLY (Leave blank)

2. REPORT DATE
January 1991

4. TITLE AND SUBTITLE
DYNAMIC GENERATION OF BINDS AND DEFINES IN OCI

5. FUNDING NUMBERS
In-house

6. AUTHOR(S)
M. P. Moser

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Ocean Systems Center
San Diego, CA 92152-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Naval Ocean Systems Center
San Diego, CA 92152-5000

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

A ‘C’ structure and the ORACLE Call Interface (OCI) describe function, (ODSC), can be combined to provide a method of dynamically building OCI bind and define statements for various SQL statements. The resulting program requires no special knowledge about any tables or columns and is unaffected by database alterations. This paper describes the construction of the ‘C’ structure, the usage of the OCI functions ODSC, ODFINE, and OBNDRV, and the procedure of combining them into a finished program.

Published in *Proceedings* of the 1990 International Oracle User Group, September 1990, pp. 347-356.

14. SUBJECT TERMS

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAME AS REPORT

Dynamic Generation of BINDs and DEFINEs in OCI

Mike Moser
Naval Ocean Systems Center

Abstract

A 'C' structure and the ORACLE Call Interface (OCI) describe function, (ODSC), can be combined to provide a method of dynamically building OCI bind and define statements for various SQL statements. The resulting program requires no special knowledge about any tables or columns and is unaffected by database alterations. This paper describes the construction of the 'C' structure, the usage of the OCI functions ODSC, ODFINE. and OBNDRV, and the procedure of combining them into a finshed program.

Introduction

Consider an application that is running in batch mode or that does internal data queries in a non-interactive mode. Frequently data must be retrieved from numerous tables or different columns of data from the same table. Both cases require new bind statements to be executed before a new query can be processed. Considerable coding can be saved, if only one bind statement is repeated within a loop while being updated with new values. This technique provides greater flexibility while reducing maintenance. The method to be described dynamically prepares simple select and insert SQL statements for processing. The coding examples run under UNIX on SUN 3 or 4 computers using ORACLE versions 5 and 6.

Structure

The key component of this procedure is a 'C' structure designed to hold the information returned by the ODSC and other data that is returned when a query is run. The structure is:

```
struct attributes
     {char          char_data[50];
      char          field_name[30];
      int           ctype_flag;
      short int     field_wid;
      short int     fldname_len;
      short int     fld_len;
      short int     field_null;
     }
```

The "char_data" variable is the location to which queried data is returned from ORACLE or where input data is stored just prior to insertion into ORACLE. Its address is the one bound or defined to the cursor. The char_data field is defined as character string data, because ORACLE is very good at manipulating this type of data. When data is extracted out of number or date fields in ORACLE tables it is easily and automatically converted by ORACLE to character string format to match the defined variable. Likewise data passed from character string variables to ORACLE fields defined as number or date is also automatically

converted.
The "field_name" variable holds the column name in the specified table, which is obtained with the ODSC call. The "ctype_flag" variable holds the ORACLE code describing the column's data type. A one indicates character data, two indicates number data, and twelve indicates date data. It is obtained from the ODSC call. The "field_wid" variable is an integer obtained from the ODSC call. For character columns it is the number specified in the creation command. Number columns are always 22 and date columns are 7. The fldname_len variable is an integer obtained from the ODSC call that gives the number of characters in the column name. Both the "fld_len" and "field_null" variables only recieve values when a select statement is executed. The "fld_len" is an integer indicating the actual length of the data returned and the "field_null" is set to one (or true) if the column contains no data. The structure is envoked as an array of structures and as a pointer to the array.

```
attributes   att[MAXCOLS], *field_no;
```

The array "att" has as many members as the maximum number of columns (MAXCOLS) expected in any table. Field_no is set to point at the member of the array containing the column (or field) to be described.

C Program Design
The following code is designed as a set of subroutines that can be inserted into a C program which is doing OCI calls. They each involve the 'attribute' structure, so the suggestion is to place it before the program main() in order to give it global scope. For organizational purposes, storing the structure in a header file and including it with a #include seems reasonable. In the following program outline the starred (*) portions are the ones related to this discussion. The basic program logic is:

```
        Logon to ORACLE and open a cursor

        Get the SQL statement

    *   Parse the SQL statement to determine:
          the type of statement
          the columns involved
          the table referenced

    *   Use odsc to get data about each column and store it in an
        array of structures

        Pass the SQL statement to the cursor with OSQL3

    *   Bind or define the variables as appropriate

    *   Execute the SQL statement and fetch or insert the data

        Clean up and logoff
```

Logging on to ORACLE and opening a cursor is the first step. The only issue here is having the cursor variable available as either a global or passed parameter.

The second step is simply providing some method of passing the text of the SQL statement to be processed to the subroutines. In the developmental version the SQL statement is stored in a flat text file and the name of the file is passed as a command line parameter. Then inside the program the file is opened and read into a character string variable.

The third step is crucial, if the method is to handle insert or update SQL statements. The ODSC call only works on select statements, so the insert and update statements must be parsed. The parsing is done to determine which columns in which tables are going to be recieving data. This column and table information is then used to build a select statement for the ODSC call. It is this select statement that is passed to the next step, not the original SQL statement. The appendix contains a sample subroutine for doing this.

The next step is the basic subroutine that uses the ODSC call, it is 'fld_desc'. In the call to 'fld_desc' a string variable is passed that contains a select statement devired from the SQL statement that will eventually be processed. If the SQL statement to be processed is a select, then there is no problem and it should be used. However, if the SQL statement is an insert or update, then the procedure described above is needed. The following code segment gives the entire 'fld_desc' subroutine.

```
/****************
**  FLD_DESC() **
****************
        This function fills the 'att' structure with the data about
    the fields referenced in a sql-statement as described by the
    ORCACLE OCI function 'odsc'. */

fld_desc(sqlstmnt,qrystmnt)
char        *sqlstmnt;
char        *qrystmnt;
{
   extern char *strim();
   ORACHAR (cname,30);
   ORACHAR (coltype,6);
   ORACHAR (tname,30);
   short width, scale, ctype, cnamelen;
   int good_field, all_fields, no_fields, field_count, cur_siz=0;
   int i,total_fields=0,got_table;

/*   Initialize variables and structures  */
    init_att_data();

/*   Try query to see if any values would be returned. */
    if (stmt_type == 1) { /*  run user's select statement */
        if (OSQL3(curs,sqlstmnt))
```

```
            oracerr(curs,10);
    }
    if (stmt_type == 2) { /* run select statment made from insert stmt */
        if (OSQL3(curs,qrystmnt))
            oracerr(curs,10);
    }
    if (!(*curs)) OEXEC(curs);
/*    if (!curs) OFETCH(curs);  */      /*send dummy select*/
/*    if (curs == 4) {
        printf("\nThe table contains no data relating to your conditions!!\n
    }
    else */
    if (!(*curs)) {                     /*got good table*/
        field_count = 1;
        got_table = TRUE;
        field_no = att;
        cnamelen = 30;
        if (odsc(curs,1,&width,(short *)-1,(short *)-1,&ctype
            ,cname,&cnamelen,&scale))
            oracerr(curs,9);
        while (!(*curs)) {      /*load column descriptions & define data buf
            strim(field_no->field_name,cname,cnamelen); /*pack name*/
            good_field = TRUE;
            if (good_field) {
                no_fields = FALSE;
                if (scale > 240) scale = 0;
                field_no->fldname_len = cnamelen;
                field_no->field_wid = width;
                field_no->ctype_flag = ctype;
                total_fields++;
                field_no++;
            }
            width = 0;
            cnamelen = 30;
            odsc(curs,++field_count,&width,(short *)-1,(short *)-1
                  ,&ctype,cname,&cnamelen,&scale);
        }
        if ((curs[0] != 4) && (curs[0] != -303)) {
            oracerr(curs,5);
            got_table = FALSE;
        }
    }
    flds_in_tab = total_fields;
    return;
}
```

The fifth step is to apply the OSQL3 call. One of its parameters is the string containing the SQL statement to be processed.

The sixth step is to use the data collected in the structure to do the actual bind or define calls. Some type of flag can be used to indicate the type of SQL statement being processed so the appropriate call can be made. The following code demostrates how a "stmt_type" flag (defined in step 3) is used:

```
/******************
**  DO_BND_DFN() **
******************
        This function binds or defines the variables in the SQL
    statement based on the 'ctype' of the statement as determined
    in the PARSE_SQL_STMT function. */

do_bnd_dfn(arry)
char arry[][MAXFLEN];
{
/*  GLOBAL variables: stmt_type, att, field_no, flds_in_tab, flds_in_sql*/
    int    i, j;

    field_no = att;
        for (j=0; j<flds_in_tab; j++) {
            switch (stmt_type) {
                case 1:
                    dfin_4_slct(j);
                    break;
                case 2:
                    bnd_4_nsrt(j);
                    break;
                case 3:
/*                  bnd_4_updt();*/
                    break;
            } /*  end switch  */
        } /*  end  for j  */
    return;
}

dfin_4_slct(i)
int   i;
{
    if (odefin(curs,i+1,field_no->char_data,50,1,-1,
        &field_no->field_null,(char *)-1,-1,-1,&field_no->fld_len,
        (short *)-1))
        oracerr(curs,7);
    field_no++;
    return;
}

bnd_4_nsrt(j)
int   j;
{
    char   orafld_name[20];

    strcpy(orafld_name,"                    ");
    strcpy(orafld_name,":");
    strcat(orafld_name,field_no->field_name);
    if (OBNDRV(curs,orafld_name,field_no->char_data))
        oracerr(curs,8);
    field_no++;
    return;
```

The seventh step is where the inputs and outputs are tailored to the specific application. Different cases will be needed for each type of SQL statement. In the example given below the retrieved columns from the select statement are written as specially formatted records to a flat file. The data to be inserted is read directly from an ASCII file into the fields of the structure. This requires coordination between the order of the data and the order of the fields in the insert statement. The update data is handled in a fashion similar to the insert data.

```
    if (stmt_type ==1) {
        if (OEXEC(curs))     /* EXECUTE THE SQL STATEMENT */
            oracerr(lda, 4);
        field_no = att;
        while (!*curs && !end_of_table){
            OFETCH(curs);
            if (*curs == 4)
                end_of_table = TRUE;
            else if (*curs)
                oracerr(lda,6);
            else {
                for (i=0,field_no = att; i<flds_in_tab; i++,field_no++) {
                    pad();
                    printf("%s ",field_no->char_data);
                } /*  end for */
                printf("\n");
                field_no = att;
            } /* end else */
        } /*  end while  */
    } /*  end if   */
    else if (stmt_type ==2) {
      if (!get_datafile_name(argv[2]))
        fprintf(stderr,"bad command line specification\n");
      while (fgets(line,BUFLEN,file_in)) {
        if (feof(file_in) || ferror(file_in)) {
          printf ("Processing complete--EOF");
          break;
        }
        while (line[0] == '\n')
          fgets(line,BUFLEN,file_in);
        line_ptr = line;
        for (i=0,field_no=att; i<flds_in_tab; i++,field_no++) {
          field_no->char_data[0] = '\0';
          start = nxtwrd(line_ptr);
          end = start;
          while ((c = *(end++)) != ','&& c != '\n') ;
          line_ptr = end;
          --end;
          strncat(field_no->char_data,start,end-start);
          field_no->char_data[end-start] = '\0';
        } /*  End for */
        if (!OEXEC(curs))    /* EXECUTE THE SQL STATEMENT */
          oracerr(lda, 11);
      }  /*  End while */
```

```
      fclose(file_in);
    } /*  end else if */
    oclose(curs);
    ologof(lda);
}
```

The final step consists of the normal housekeeping chores of closing open files, closing open cursors, and logging off of ORACLE.

This paper demostrates how a few short subroutines can replace hard coded bind and define statements. In cases where many tables are involved, the amount of code will be greatly reduced. Also more flexibility will be gained in situations where many different queries need to be run. Hopefully, the code examples will make it easy to impliment into new developments as well as into existing applications.

Appendix

1) Header Files, Macros, and Global Variables:

```
#include <stdio.h>
#include <ctype.h>
#include "otabatt.h"
#include "oracle.h"
#include "genmacros.h"

#define MAXFIELDS  50
#define MAX_ROWS   20
#define MAXFLEN    80
#define NUMLEN     10
#define DATELEN     9
#define BUFLEN    200
#define DEBUG       1

struct  attributes att[MAXFIELDS], *field_no;

short int lda[32], curs[32];
static char     uidpw[10] = "tabi/net";
char *otable;
FILE *file_in;

char    table_name[20];
int     flds_in_sql, flds_in_tab, stmt_type;
```

2) Sample Main Program:

```
main(argc,argv)
int  argc;
char **argv;
{
    static char  sql_stmnt[1000];
    static char  qry_stmnt[1000] = "select ";
    char  fld_arry[MAXFIELDS][MAXFLEN];
    char  line[BUFLEN], *line_ptr, *end, *start;
    char  c, *nxtwrd();
    int   i,end_of_table=FALSE, wrdlen;

    if (!get_sql_stmt(argv[1],sql_stmnt))
        fprintf(stderr,"bad command line specification\n");
    cvtupper(sql_stmnt);

    parse_sql_stmt(sql_stmnt,qry_stmnt);

    if (OLON(lda,uidpw)) {        /* LOGON TO ORACLE */
        oracerr(lda, 1);
    }
    if (OOPEN(curs,lda)) {        /* OPEN A CURSOR    */
        oracerr(lda, 2);
    }
    fld_desc(sql_stmnt,qry_stmnt); /* CREATE THE TABLE_COLUMNS ATTRIBUTE FIL

    if (OSQL3(curs,sql_stmnt)) {          /* DEFINE THE SQL STATEMENT */
        oracerr(lda, 3);
    }
    dc_bnd_dfn(fld_arry);         /* DYNAMIC BIND & DEFINE ROUTINE */

/* Main processing code taken from here and put on page 6 */

} /* end of main */
```

3)Related Subroutines:

```
/************
**  PAD()   **
*************
        This subroutine is an example of how some of the other fields in
    the "att" structure can be use.  It produces a screen output similar
    to sqlplus.*/

pad()
{
    char  tempstr[30], padstr[30], *padptr;
    char  *blank_fill();
    int   nul_val, ctype, i, max_wid, act_wid;

    nul_val = field_no->field_null;
    ctype = field_no->ctype_flag;
    max_wid = field_no->field_wid;
    act_wid = field_no->fld_len;
```

```
    tempstr[0] = '\0';
    if (nul_val > -1) strcat(tempstr,field_no->char_data);
    switch(ctype){
      case 1:    for (i=0, padptr = padstr; i<max_wid-act_wid; i++,padptr--
                   *padptr = ' ';
                 }
                 padstr[i] = '\0';
                 if (nul_val > -1) strcat(padstr,tempstr);
                 strcpy(field_no->char_data,padstr);
                 break;
      case  2:   for (i=0, padptr = padstr; i<NUMLEN-act_wid; i++,padptr--)
                   *padptr = ' ';
                 }
                 padstr[i] = '\0';
                 if (nul_val > -1) strcat(padstr,tempstr);
                 strcpy(field_no->char_data,padstr);
                 break;
      case 12:   for (i=0, padptr = padstr; i<DATELEN-act_wid; i++,padptr--
                   *padptr = ' ';
                 }
                 padstr[i] = '\0';
                 if (nul_val > -1) strcat(padstr,tempstr);
                 strcpy(field_no->char_data,padstr);tempstr[DATELEN] = '\0'
                 break;
    }

    return;
}

/********************
** INIT_ATT_DATA() **
********************
        Function to zero out the structure "att" pointed to by field_no. *

int init_att_data()
   {
   int i;

   for (field_no = att, i = 0; i++ < MAXFIELDS; field_no++) {
      *field_no->char_data = NUL;
      *field_no->field_name = NUL;
      field_no->ctype_flag = 0;
      field_no->field_wid = 0;
      field_no->fldname_len = 0;
      field_no->fld_len = 0;
      field_no->field_null = 0;
      }
   return(0);
   }
}
```

```
/********************
**  GET_SQL_STMT() **
********************
        This routine takes the first command line argument as the name of
    a file containing a sql statement.  It tries to open it for reading
    and if successful, puts the contents into the  variable sql_stmnt)*/

int get_sql_stmt(f_name,sql)
char *f_name, *sql;
{
    int  buf_size = 1024;
    /* Open Input File */
    if(f_name==NULL) {
        fprintf(stderr,"No input file has been specified\n" );
        return(0);
    }
    else {
        file_in= fopen(f_name,"rb");
        if(file_in==NULL) {
            fprintf(stderr,"\nERROR Can't Open Input File= %s \n",f_name);
            exit(0);
        }
        fprintf(stderr,"Opened Input File =%s \n",f_name);
        fgets(sql,buf_size,file_in);
    }
    fclose(file_in);
    return(1);
}

/************************
**  GET_DATAFILE_NAME() **
*************************
        This routine takes the second command line argument as the name of
    a file containing the input data.  It tries to open it for reading.*/

int get_datafile_name(f_name)
char *f_name;
{
    int  buf_size = 1024;
    /* Open Input File */
    if(f_name==NULL) {
        fprintf(stderr,"No input file has been specified\n" );
        return(0);
    }
    else {
        file_in= fopen(f_name,"rb");
        if(file_in==NULL) {
            fprintf(stderr,"\nERROR Can't Open Input File= %s \n",f_name);
            exit(0);
        }
        fprintf(stderr,"Opened Input File =%s \n",f_name);
    }
    return(1);
}
```

```
/**********************
**  PARSE_SQL_STMT() **
**********************
        This routine takes the sql statement and parses the first word
    into the variable "stmnt_type".  Then, depending on the type of
    oracle statement, puts all the field names into "arry" and the
    table name into "table_name". */

parse_sql_stmt(s,q)
char  s[], q[];
{
    int   fld_cnt, i, j, wrdlen;
    char  *end, lstwrd[20], *cur, *start, *wrdend(),*nxtwrd(),*newwrd();
    char  stmnt_type[10];

    cur = start = s;

/*   Put the first word in the variable "stmnt_type"   */
    end = wrdend(cur);
    strncpy(stmnt_type,cur,end - start);
    cur = end;

    if (strcmp(stmnt_type,"SELECT") == 0) {
      stmt_type = 1;
      pars_slct(cur,q);
    }
    else if (strcmp(stmnt_type,"INSERT") == 0) {
      stmt_type = 2;
      cur = newwrd(cur,lstwrd,&wrdlen);
      setnul(lstwrd);
      cur = newwrd(cur,lstwrd,&wrdlen);
      strncpy(table_name,lstwrd,wrdlen);
      setnul(lstwrd);
      cur = newwrd(cur,lstwrd,&wrdlen);
      if (strcmp(lstwrd,"VALUES") == 0) {
        strcat(q," * from ");
        strcat(q,table_name);
      }
      else if (strcmp(lstwrd,"SELECT") == 0) {
        pars_slct(cur,q);
        strcat(q,table_name);
      }
      else {
        if (lstwrd[0] == '(')  {
            pars_lst(cur,lstwrd,q);
            strcat(q," from ");
            strcat(q,table_name);
        }
        else
          printf("Improper sql statement\n");
      }
    }

    return;
   }
```

```
/******************
**  PARS_LST()    **
******************
        This subroutine parses the field names out of a list surrounded
    by "( )".  The field names are stored in 'ARRY' and the count of
    fields is stored in flds_in_sql.  */

pars_lst(cur,lstwrd,qry)
char  *cur,*lstwrd,*qry ;
{
    int  fld_cnt, i, j, wrdlen;

    for (i=1,j=strlen(qry); i<strlen(lstwrd); i++,j++)
        qry[j] = lstwrd[i];
    qry[j] = '\0';
    fld_cnt = 1;
    while (lstwrd[strlen(lstwrd)-1] != ')') {
        setnul(lstwrd);
        cur = newwrd(cur,lstwrd,&wrdlen);
        strcat(qry,", ");
        if (lstwrd[strlen(lstwrd)-1] != ')')
            strcat(qry,lstwrd);
        fld_cnt++;
    }
    for (i=0,j=strlen(qry); i<strlen(lstwrd)-1; i++,j++)
        qry[j] = lstwrd[i];
    qry[j] = '\0';
    flds_in_sql = fld_cnt;
    return;
}

/******************
**  PARS_SLCT()   **
******************
        This subroutine parses the field names and the table name from
    a SQL select statement.  */

pars_slct(s,q)
char  s[], q[];
{
    int   fld_cnt=0,wrdlen;
    char  *end, curwrd[20], *cur, *start, *wrdend(),*nxtwrd(),*newwrd();

    cur = start = s;

    while (strncmp(curwrd ,"FROM",4)!=0) {
        setnul(curwrd);
        cur = newwrd(cur,curwrd,&wrdlen);
        if (curwrd[0] == '*') {
            strcat(q," * from ");
            cur = newwrd(cur,curwrd,&wrdlen);
            goto tbl;
        } /*End if (curwrd) */
        strcat(q,curwrd);
        strcat(q," ");
```

```
        fld_cnt++;
    }  /*End while (strncmp) */
    flds_in_sql = --fld_cnt;

/*   Get the table name and store it in tab  */
tbl : cur = newwrd(cur,curwrd,&wrdlen);
      strncpy(table_name,curwrd,wrdlen);
}


/***************
**  NEWWRD()  **
***************
        This subroutine returns a pointer to the character after the
    current word in the statement buffer.  It also returns the current
    word and the length of the word via a passed int pointer.  */

char *newwrd(buffptr,newwrd,wrdlen)
char *buffptr, *newwrd;
int  *wrdlen;
{
    char  *end, *start, *wrdend(),*nxtwrd();

    start = nxtwrd(buffptr);             /*This gets the next word  */
    end = wrdend(start);
    *wrdlen = end - start;
    strncpy(newwrd,start,*wrdlen);
    return(end);
}

/***************
**  NXTWRD()  **
***************
        This subroutine returns a pointer to the next non blank space
    character in the buffer 'str'.   The macros LF, CR, and TAB are ,
    found in the header file "*/

char *nxtwrd(str)
char  *str;
{
    char  c;
    while ((c= *(str++))==' '|| c==LF || c==CR || c==TAB || c==',') ;
    str--;
    if (c==NUL)
        str = NULL;
    return(str);
}


/****************
**  ISACOMMA() **
****************
        This subroutine searches character by character through white
    space in a string buffer for a comma. */
```

```
isacomma(str)
char  *str;
{
    char  c;
    while ((c= *(str++))==' '|| c==TAB || c==',')
      if (c == ',')
        return(1);
    return(0);
}

/***************
**  WRDEND()  **
****************
        This subroutine searches character by character through a
    string buffer looking for a non-alphanumeric character to
    indicate the end of a word string.  It returns a pointer to the
    first non-alphanumeric character it encounters. */

char *wrdend(w)
char *w;
{
    while ( *w !=' ' && *w !=LF && *w !=CR && *w !=TAB && *w!=',') {
        if ( *w==NUL)
            return(NULL);
        w++;
    }
    return(w);
}

/****************
**  ORACERR()  **
*****************
        This ORACLE error analysis routine. Prints ORACLE's explanation
    of the error and abandons execution of the program if the calling
    short int is non-zero.*/

#define ORACLE  1

int oracerr(cur,n)
   int n;
   short cur[];
   {
   extern short lda[];
   char  msg[80];
   if (!*cur) return(0);                  /*bail out of accidental call*/
   printf("\nORACLE Error %d at program location %d",cur[0],n);
   printf("\nFunction type %d, function code %d, error offset %d"
      ,cur[1],cur[5],cur[4]);
   oermsg(cur[0],msg);
   printf("\n%s\n\007\n",msg);
   orol(lda);
   ologof(lda);
   exit(0);
   }
/* end of file */
```