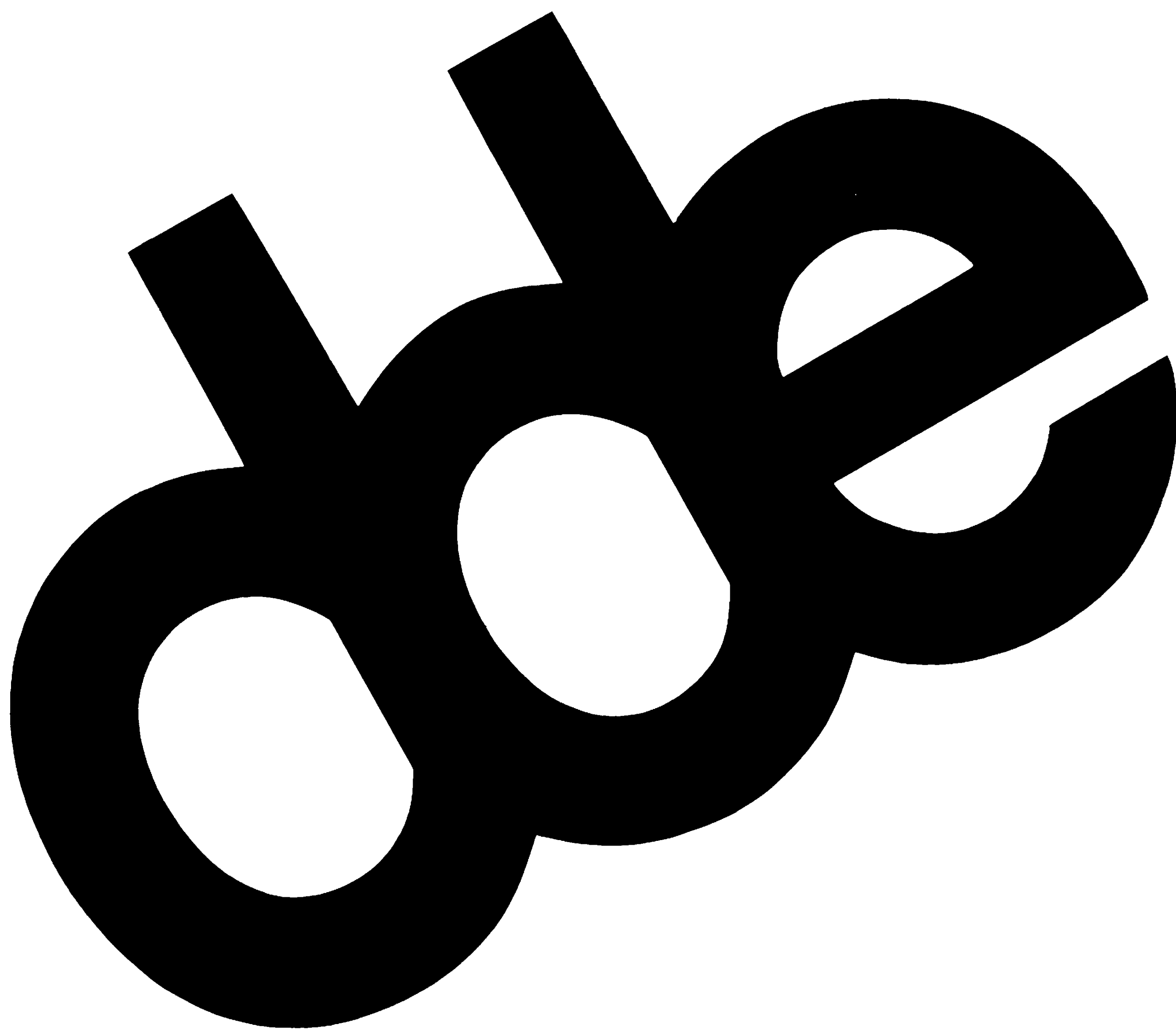
Oracle Forms 4.0 and 4.5 Hints



© Dansk Data Elektronik A/S
December 1995

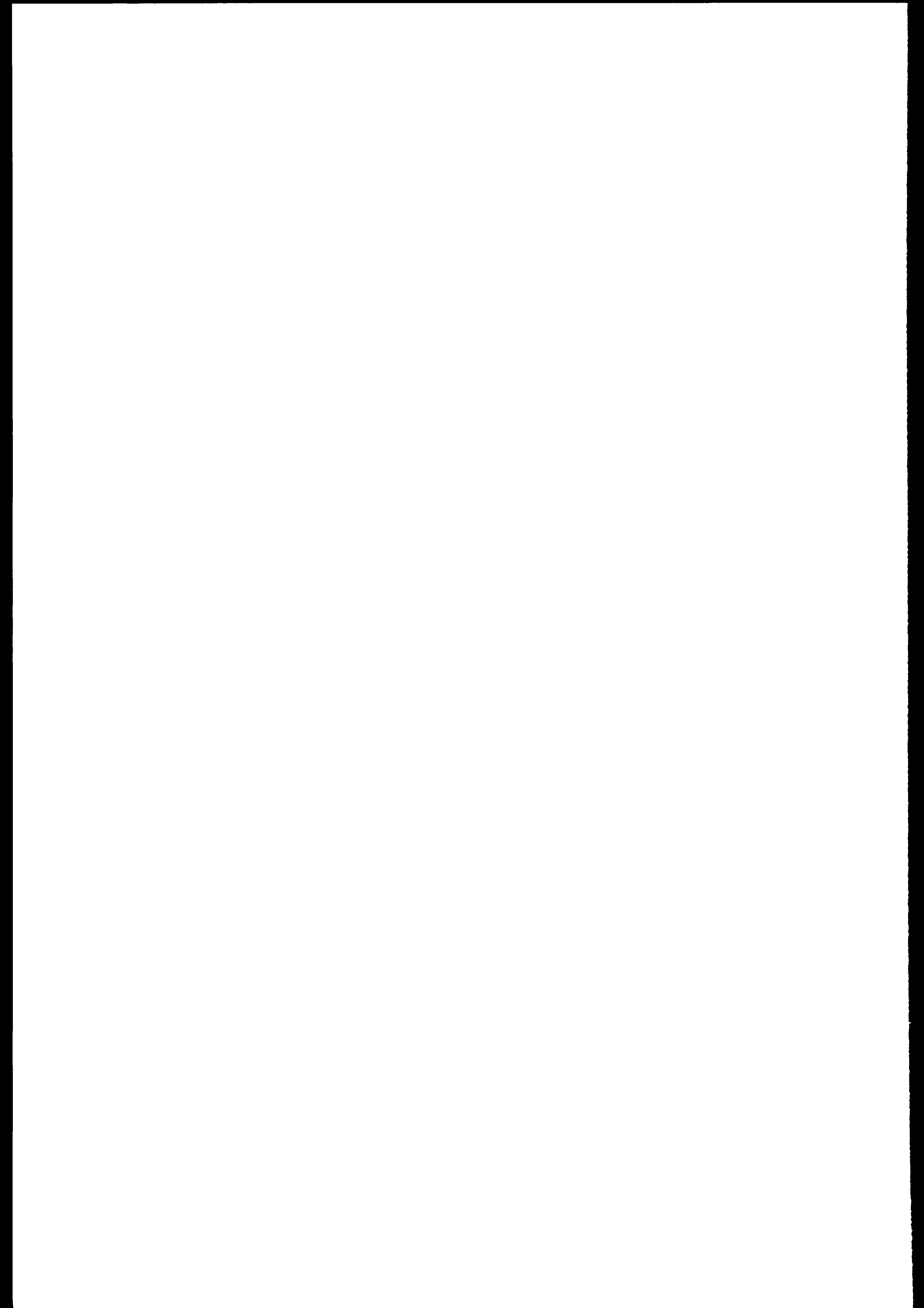


Table of Contenst

1 Objective....1

2 Programmatic Advice....3
- 2.1 Generic Procedures and Triggers....3
  - 2.1.1 Register Procedures....5
  - 2.1.2 Careful using Rowtype....5
  - 2.1.3 Changing the Makefile....6
  - 2.1.4 Calling f40runm from a Program....7
  - 2.1.5 Get Shell Variable - a User-exit....8
- 2.2 Timers....12
- 2.3 Buttons....13
- 2.4 Parameter Lists....13
  - 2.4.1 Relations between Blocks....14
- 2.5 Alerts....15
- 2.6 List Items....15
- 2.7 List-of-Values (LOV)....16
- 2.8 Management....16
  - 2.8.1 Oracle Forms 4.0 Tables....16
  - 2.8.2 Use SCCS or RCS....22
  - 2.8.3 Simplify the System....23
  - 2.8.4 Extract Common Code from Oracle Forms....23
  - 2.8.5 Adapt a Naming Convention....24
  - 2.8.6 Make the Oracle Forms Tables big enough....24
  - 2.8.7 Migrating from SQL*Forms version 3.0....25
  - 2.8.8 Running in Character Mode....26
  - 2.8.9 Looking for Forms....26
  - 2.8.10 Transport to / from MS-Windows....26
  - 2.8.11 Trigger execution Sequence....26
- 2.9 Navigation....27
  - 2.9.1 Prepare for Mouse Navigation....28
- 2.10 Tuning....28
  - 2.10.1 Direct Reference....28
  - 2.10.2 Save System Variable Overhead....28
  - 2.10.3 Use Explicit Cursors....28
  - 2.10.4 Be careful using Default Where / Order By Clauses....29
  - 2.10.5 Reuse Cursors....30
  - 2.10.6 Response time Logging....30

2.11 Transactions 35
2.11.1 Select Transaction Strategy 35
2.12 Tricks 36
2.12.1 Allocate Global Variables 36
2.12.2 Security 37
2.12.3 Getting Alerts from the RDBMS 37
2.13 Debugging 39
2.14 Flexible Presentation 40
2.14.1 Existing Posibilities 40
2.14.1.1 Alert Objects 40
2.14.1.2 Application Objects 41
2.14.1.3 Block Objects 41
2.14.1.4 Canvas Objects 41
2.14.1.5 Forms Objects 41
2.14.1.6 Item Objects 41
2.14.1.7 LOV Objects 41
2.14.1.8 Menu Objects 42
2.14.1.9 Radio Button Objects 42
2.14.1.10 Relation Objects 42
2.14.1.11 View Objects 42
2.14.1.12 Window Objects 42
2.14.2 Load Properties to a Table 42
2.14.3 Get Properties from the Table 45

3 Layout Advise 48

Development Environments in DDE - Oracle Forms version 4.x

The present paper is written as a contribution to the debate about which ingredients are applicable to a development environment in order to develop good and robust bug fixed programs.

From version 1.1.12 to 1.1.16 of this document remarks on Oracle Forms version 4.5 are also supplied. Spelling errors are corrected in version 1.1.17 and 1.1.18.

1. Objective

The objective is to make as good use of the features in Oracle Forms version 4.0 as possible, and prevent developers from falling into the same traps as others did before them.

As forms applications are often used by skilled as well as non-skilled users, you may observe the following elements of userfriendliness:

Simplicity A user interface composed of simple elements allows the user to keep the general view of the elements, not to be confused by an abundance of choices.

General View Allow the user to keep the general view of the application, even if the user needs to manipulate some details for some periods of time.

Adapting As many presentation parameters as possible should allow themselves to be adapted at runtime, to fit the actual need. This is however not to used as an excuse for distributing non-reasonable default settings.

Oracle, Oracle7, PL/SQL, SQL*Forms, SQL*ReportWriter, Oracle Forms, Oracle ReportWriter, ORACLE Case, Oracle Forms and Oracle Report are registered trademarks of Oracle Corporation.

Intuitive The user interface should be able to reason what semantic function, the user is in fact processing. And it should be clear how different legal actions are to be activated.

Forgiving Any user initiated event changing or destroying information, should have an inverse function to lift them again. Without this capability the user will not dare to try yet undiscovered parts of the application.

Fast Any legal action should be activated with very few user initiated events. If an action takes more than about one second, the user interface should give the user an idea of the course of the execution.

Oracle Forms 4.5 adds a number of features providing improved productivity and performance to Forms 4.0, some of which are:

* New developer productivity features like a Object Navigator to find the object in question rapidly as well as an PL/SQL debugger.

* Better integration with MS Windows since OLE2 and VBX custom controls are supported.

* Better reusability of code through property clases and object groups.

* Better GUI control through better mouse awareness, combo boxes and tool bars.

* Intergrated timing and debugging through PECS (Performance Event Collection Service).

2. Programmatic Advice

The following list of advice is by no means complete, and skilled developers will be able to find cases, where the advice is not very applicable. Still, I suggest the advice should be discussed among Oracle Forms developers.

2.1 Generic Procedures and Triggers

Instead of writing complex triggers, develop some generic procedures, and call these from the triggers. This should increase the possibility of reusing the code, saving execution time, storage and development effort.

The following example is selected from Steve Muench - Oracle Corporation.

Suppose we have two different fields being foreign keys to the **emp** table, then we could write two **ON-VALIDATE-FIELD** triggers like this:

```
begin
  select ename
    into :block1.first_ename_field
    from emp
    where empno = :block1.first_empno_field;
exception
  when no_data_found then
    Message( 'This employee does not exist!' );
    raise form_trigger_failure;
end;
```

And

```
begin
  select ename
    into :block2.second_ename_field
    from emp
    where empno = :block2.second_empno_field;
exception
  when no_data_found then
    Message( 'This employee does not exist!' );
    raise form_trigger_failure;
end;
```

Instead of developing a generic routine to test the matter:

```
procedure Validate_Employee (
  fp_EmpNo in Number,
  fp_EName out Char) is
  cursor sel_emp is
    select ename from emp
    where empno = fp_EmpNo;
begin
  open sel_emp;
  fetch sel_emp into fp_EName;
  if sel_emp%notfound then
    Message( 'This employee does not exist!' );
    raise form_trigger_failure;
  end if;
  close sel_emp;
end;
```

The two validate triggers may now look like this:

```
Validate_Employee(:block1.first_empno_field,:block1.first_ename_field );
:      :     :     :     :     :     :     :     :
Validate_Employee(:block2.second_empno_field, :block2.second_ename_field );
```

You may want to use the fact that the first parameter in the trigger call should be the contents of the field in which the cursor is right now. In that case the trigger call could look like this:

```
Validate_Employee( Name_In( :system.cursor_field ), :block1.first_ename_field );
:      :     :     :     :     :     :     :     :
Validate_Employee( Name_In( :system.cursor_field ), :block2.second_ename_field );
```

You may wonder why the second parameter is not changed accordingly. This is due to the fact that we expect a value to be returned from the procedure.Also you might want to implement generic routines handling different argument types.

This may be accomplished by calling the procedure by **name** instead of by **value / reference**.

```
procedure Validate_Employee (
  fp_Field in Char,
  fp_Block in Char) is
  fp_EmpNo emp.empno%type;
  fp_EName emp.ename%type;
  var      Char(61);
  cursor sel_emp is
    select ename from emp
    where empno = fp_EmpNo;
begin
  fp_EmpNo := Name_In( fp_Field );
  open sel_emp;
  fetch sel_emp into fp_EName;
  if sel_emp%notfound then
    Message( 'This employee does not exist! ');
    raise form_trigger_failure;
  end if;
  close sel_emp;
  var := fp_Block||'.'||Get_Block_Property( fp_Block, FIRST_ITEM );
  var := fp_Block||'.'||Get_Item_Property( var, NEXTITEM );
  Copy( fp_EName, var );
end;
```

The call of the procedure could look like this:

```
Validate_Employee( :system.cursor_field, :system.cursor_block );
```

2.1.1 Register Procedures

Try to label the generic procedures in some for validation and some for user interface purpose that could reside on the client side in the form, and those checking database integrity and propagating changes (**db_proc**) that could reside in the database (Oracle7).

2.1.2 Careful using Rowtype

In the PL/SQL procedures and triggers, you have the possibility of declaring variables spanning a whole row, which is very convenient in generic procedures.

Note, however that you might get errors if the underlying table is changed without regenerating the form.

2.1.3 Changing the Makefile

Oracle recommend that the makefile for relinking Oracle Forms modules supporting user-exits and SQL*Net drivers, are taken as an example rather than used as it is. This often leads to tough debugging sessions because an adapted makefile is not automatically updated whenever a new version of Oracle Forms is installed.

It is therefore advisable to change the makefile a little, to support the usage of a generic makefile, in order to minimize the potential problems as well as maintenance efforts.

The following changes are thus recommended to **$ORACLE_HOME/forms40/lib/sqlforms40.mk**, after a copy has been taken:

* Add -I$(ORACLE_HOME)/forms40/lib as a C-compiler option in order to be able to run the makefile from a dictionary different from the forms40 /lib dictionary. Two positions have to be changed and the resulting line will look like:

  $(CC) -c $(CFLAGS) -I. -I$(ORACLE_HOME)/forms40/lib $*.c

* Change the definition of **EXIT** to:

  EXITS=iapxtb.o $(OTHERXIT)

  In order to be able to specify other exits on the command file, without having to add the files to the makefile.

* You may also add lines to the makefile supporting the linking of C-programs calling **f40runmx** directly.

```
callfrm: $(EXITS)
		@$(ECHO) $(CC) $(LDFLAGS) -o $@ $@.c \
		$(SSLIFTAB) \
		$(EXITS) \
		$(P2CSPECWODIANA) \
		$(ISTUIC) \
		$(UI10) \
		$(UIICXD) \
		$(SSLLIBS) \
		$(FORMS40LIBS) \
		$(SSLLIBS) \
		$(NNLIBS) \
		$(VGSLIBS) \
		$(DELIBS) \
		$(PLSLIBS) \
		$(CALIBS) \
		$(MMMLIBS) \
		$(TK2UIMLIBS) \
		$(ZOMLIBS) \
		$(SQLPLUSLIBS) \
		$(PROLIBS) \
		$(TTLIBS) \
		$(CLIBS) \
		$(MOTIFLIBS)
```

2.1.4 Calling f40runm from a Program

It is possible to call f40runm directly from a C program, either to hide the database account, or to reuse an already open database connection.

```
#include <sys/types.h>
#include <std.h>
#include <stdio.h>
/* #include <ifzcal.h> */

extern int ifzcal();

main(argc,argv)
  int    argc ;
  char   *argv[];
  {
  int res;
```

```
  res = ifzcal("f40run module=timertst
userid=system/manager",44);
  printf("Exiting, returncode is: %d\n", res);
  }
```

Figure: "callfrm.c"

2.1.5 Get Shell Variable - a User-exit

Here we shall demonstrate how to make a user-exit in C, to copy a **shell** variable into a forms variable. And how the user-exit might be used from a form.

First create the source of the user-exit **getenvr.pc**:

```
/* ************************************************************************
GETENVR will get the value of the <envr-name> and store it in the

        <field-name> with the maximum length of <max-length>

GETENVR is called with the following syntax for the parameter string p:

USAGE (in forms 4.0): USER-EXIT('GETENVR envr-name field-name max-
length')

13. May 91 - MJ - DDE
************************************************************************ */

#include <stdio.h>
#ifndef IFUXIT
# include <ifuxit.h>
#endif
/* #include <usrxit.h> is now obsolete on sqlforms 40 */

#define MAXARGS               128
#define ARBUFSIZ              512

EXEC SQL BEGIN DECLARE SECTION;
      VARCHAR formname [30];                /* form variable name */
      VARCHAR varvalue [128];           /* Variable to store value */
EXEC SQL END DECLARE SECTION;
EXEC SQL INCLUDE SQLCA.H;
```

```
char *wordb[MAXARGS];         /* Holds pointers to the blank separated  */
                              /* tokens in the string passed to GETENVR */
int getenvr(p, paramlen, erm, ermlen, qry)
char *p;                                         /* Parameter string */
  int  *paramlen;                         /* Ptr to param string length */
  char *erm;                          /* Error message if doesnt match */
  int  *ermlen;                         /* Ptr to error message length */
  int  *qry;                               /* Ptr to query status flag */

  {
  int listsiz;                                   /* Number of values */
  int i;                                             /* Temp counter */
  char arbuf[ARBUFSIZ];               /* To hold string that is passed */
  int maxlen;                               /* The max length to copy */
  char *cpoint;                                      /* Temp counter */

  EXEC SQL WHENEVER SQLERROR GOTO sqlerr;

  remblank( p );   /* Remove leading, trailing, and and double spaces */

  strncpy(arbuf, p, ARBUFSIZ-1 );
                  /* get envrname, fieldname and maxlen in wordb[] */

  listsiz = countargs( arbuf );

  if ( listsiz != 4 )                        /* Check for 4 arguments */
    return IAPFTL;

  strncpy(formname.arr, wordb[2], 30);          /* Store the form-name */
  formname.len = strlen(formname.arr);

  sscanf(wordb[3], "%d", &maxlen);                    /* get the maxlen */

  strncpy(varvalue.arr, getenv(wordb[1]), maxlen);/* Get and store env */
  if (strlen(varvalue.arr) == 0) return IAPFAIL;
  varvalue.len= strlen(varvalue.arr);

  EXEC TOOLS SET :formname values ( :varvalue );/* Store value in form */
  /*EXEC IAF PUT :formname values ( :varvalue );        is obsolete   */
  return IAPSUCC;
  }
```

```
sqlerr:                                  /* General error exception */
  sqlca.sqlerrm.sqlerrmc[sqlca.sqlerrm.sqlerrml] = '\0';
  sqliem( sqlca.sqlerrm.sqlerrmc, &sqlca.sqlerrm.sqlerrml );
  return IAPFAIL;

/* countargs -counts arguments in string and sets pointers in wordb[] */
/*            to point to individual, null-terminated tokens.  Relies */
/*            on string having no leading, trailing or double blanks. */

int countargs(textp)
char *textp;
{
  int numargs;
  char *sp;
  extern char *strchr();

  for ( numargs = 0, sp = textp; numargs < MAXARGS && sp; numargs++ ) {
    wordb[numargs] = sp;
    sp = strchr( textp, ' ');

    if ( sp ) {
      *sp = '\0';
      textp = ++sp;
    }
  }
  return(numargs);
}

/* Remove leading, trailing, and and double spaces */

int remblank (textp)
char *textp;
{
  register char *pin = textp;
  register char *pout = pin;
  register char c;
  while ( *pin == ' ' ) pin++;
  for (; c = *pin; ++pin) {
    if (c == ' ') {
      *pout++ = ' ';
       for (++pin; (c = *pin) && c == ' '; ++pin) ;
       if ( !c ) pout--;
    }
```

```
if (c) {
      if (c != ' ')
        *pout++ = c;
      else
        --pin;
    } else
      break;
  }
  *pout = '\0';
}
```

Figure: "The getenvr.pc User exit"

Append an entry in the **iapxtb** table, by editing the **iapxtb.c** source table.

The **iapxtb.c** file should contain the following entry:

```
#include <ifuxit.h>

extern int getenvr();

externdef exitr iapxtb[] = {              /* Holds exit routine pointers */
    "GETENVR", getenvr, XITCC,
    (char *) 0, 0, 0                       /* zero entry marks the end */
    } ;
```

Relink **f40runx** and **f40runmx** with the new user-exit:

OTHERXIT=getenvr.o make -f sqlforms40.mk f40runmx

The following trigger will use the user-exit to get the environment name stored in the field **ENVR**, and store the environment value in the field **VAL**. If it succeeds, the trigger step will validate the **VAL** field:

```
User_Exit( 'getenvr '||Name_In( 'ENVR' )||' VAL  '||
           Get_Item_Property ('VAL', MAX_LENGTH ));
if form_fatal then
  Message( 'GETENVR USAGE: env_var form_var max_len' );
elsif form_failure then
  Message( 'Envr >'||Name_In( 'ENVR' )||'< does not exist');
else
  Go_Item( 'VAL' );
  Next_Item;
end if;
```

Note that all literal text in English has been moved from the user-exit to the calling trigger, to make it easier to translate. You may even copy values into fields other than character fields, just remember to validate.

2.2 Timers

In order to do some internal periodical actions, the use of timers is handy. Create e.g. in a when-new-form-instance trigger a timer:

```
declare
  timer_id Timer;
begin
  timer_id := Create_Timer( 'TEST_TIMER', 10000, REPEAT );
  if Id_Null( timer_id ) then
    Message( 'Error creating the TEST_TIMER timer' );
    raise form_trigger_failure;
  end if;
end;
```

Create an when-timer-expired trigger to define the actions to fire when one of the timers expire, e.g. to update a field showing current time:

```
declare
  a_time   Char(20);
begin
  :global.timer_name:= Get_Application_Property(TIMER_NAME);
  if :global.timer_name = 'TEST_TIMER' then
    a_time := Substr( :system.current_datetime, 13, 99 );
    Message( 'Time: '||a_time, NO_ACKNOWLEDGE );
    :forms_app.local_time := a_time;
  else
    Message( 'Another timer did expire: '||:global.timer_name );
  end if;
end;
```

Verified to work for Oracle Forms 4.5 as well.

Note however, that even as timers do expire when the forms is in query mode, they will not expire during an alert or when the operator is in progress of selecting a menu item.

Likewise, if a timer is requested to expire extremely often (like once every millisecond), the execution of the corresponding trigger would neither be stacked, nor recursivly executed.

2.3 Buttons

You may create buttons in your canvas. You will however need to attach actions to the buttons using **when-button-pressed** triggers on a per button
basis.

Unfortunately you cannot develop more generic **when-button-pressed** triggers, say on the form level, since information on which button is most recently pressed is not generally available.

2.4 Parameter Lists

In order to be able to do more sophisticated parameter passing between forms, reportwriter modules and Graphics applications, the notion of **parameter lists** is introduced. Create e.g. in a **when-new-form-instance** trigger a parameter list:

```
declare
  pl_id    ParamList;
begin
  pl_id := Create_Parameter_List( 'PARAMETER_TEST' );
  if Id_Null( pl_id ) then
    Message( 'Error creating the PARAMETER_TEST parameter list' );
    raise form_trigger_failure;
  end if;
end;
```

Then before calling another module, add some elements into the parameter list:

```
declare
  pl_id    ParamList;
begin
  pl_id := Get_Parameter_List( 'PARAMETER_TEST' );
```

```
  if not Id_Null( pl_id ) then
    Add_Parameter( pl_id, 'PARAM_TIME', TEXT_PARAMETER,
      To_Char( :local_date,'DD-MM-YYYY' ) );
    Add_Parameter( pl_id, 'PARAM_USER', TEXT_PARAMETER, user );
    Call_Form('paramtst', NO_HIDE, DO_REPLACE, NO_QUERY_ONLY, pl_id);
    Delete_Parameter( pl_id, 'PARAM_TIME' );
    Delete_Parameter( pl_id, 'PARAM_USER' );
  end if;
end;
```

Then in the called form when-new-form-instance trigger, read the parameter values:

```
declare
  pl_id ParamList;
  pl_type  Number;
  pl_value Char(30);
begin
  Get_Parameter_Attr( 'DEFAULT', 'PARAM_TIME', pl_type, pl_value );
  :text_date := pl_value;
  Get_Parameter_Attr( 'DEFAULT', 'PARAM_USER', pl_type, pl_value );
  :text_user := pl_value;
end;
```

Note that the name of the parameter list is **DEFAULT**.

2.4.1 Relations between Blocks

Use relations rather than 'master - detail' options in 'default block'.

The relations are not 'safe' though, even if the foreign fields in the detail are protected against update, because an update of the master key does not either restrict or cascade - it simply changes, leaving unreferenced details.

Also locking a detail for update does not lock the appropriate master. Another user could be deleting the master, as it is not locked.

And relations only support delete cascade on one level, so we should use reference integrity and requery instead.

Note that if some of the primary key columns may be null, then the join condition becomes rather complicated. So keep all primary key columns not null whenever possible.

2.5 Alerts

In Oracle Forms 4.5, it is possible to define alerts and to populate alert messages in seperate windows. The following PL/SQL block will change the message of an existing alert object (TEST_ALERT), and make it appear to the user.

```
declare
  alert_id Alert;
  alert_result Number;
  alert_message Varchar2( 80 );
begin
  alert_id := Find_Alert( 'TEST_ALERT' );
  if Id_Null( alert_id ) then
    Message( 'Alert TEST_ALERT does not exist!' );
  else
    alert_message := 'This is a text (ÆØÅ æøå)';
    Set_Alert_Property(alert_id,ALTER_MESSAGE_TEXT, alert_message);
  end if;
end;
```

Note however, that until the user acknowledges the message, no processing is done by Forms. Even timers will not expire in this period. That is, timers will expire during alerts, but the associated action will be processed after the alert - and only once per timer! (This has been verified in Oracle Forms version 4.5.5).

2.6 List Items

List items are indeed very useful. You may programaticcally add elements to and delete elements from a list. Only remember that the function **Get_List_Element_Count** returns a character string, where we think **to_number** could be applied.

By the way, in version 4.5.5 the **Get_List_Element_Count** function returns '1', if the list is empty.

2.7 List-of-Values (LOV)

List of value objects may be defined in order to assist the operator selecting an appropriate value (or tupple). And the values forming the list may be received from an SQL-statement in a record group.

Such a statement will normally extract valid values from a given column, such as: select distinct job from emp order by job. But more complex SQL-statements may be composed, like the following:

```
select distinct job from emp
union
select 'Unknown' from dual
union
select distinct job||'-'||to_char( deptno ) from emp
```

The selected value are returned from the LOV object into the designated field / variable, where a post-change trigger may do some actions.

2.8 Management

2.8.1 Oracle Forms 4.0 Tables

Like in SQL*Forms 3.0, the 4.0 version supports the possibility of having the total form stored in the database. This is still a superb way to make cross references and management on a lot of forms. It is however increasingly difficult to read the forms tables, because more products are involved and because the datatypes LONG and LONG RAW are used.

The table structure of the forms tables is described in Appendix F of the forms Reference Manual. (Table prefix is frm40__).

Also the Tool Kit tables are used for module definitions, PL/SQL code, and long text and images. (Table prefix is tool__).

Also the Resource Object Store (ROS) tables are used for literals of any sort. (Table prefix is ros). The program rosstr delivered on request, may dump the rosstrings in a readable format.

Also the Virtual Graphical System (VGS) ststem is used in order to support color and font information. (Table prefix is vg).

A number of deviations between the forms table definition and the one described in appendix F have been located, they are listed below: Manual Version: A11989-1 July 1993, Oracle Forms version: 4.0.11.0.1.

* Column frm40__app.appmname is varchar(255) instead of varchar(30).
* Column frm40__app.appfname is varchar(255) instead of varchar(30).
* Column frm40__app.appnlslang is new varchar(40) to store NLS information.
* Column frm40__blk.blktowner is varchar(61) instead of varchar(30).
* Column frm40__grp.grpname is varchar(30) instead of varchar(80).
* Column frm40__itm.itmcopy is varchar(61) instead of varchar(80).
* Column frm40__itm.itmquality is new number.
* Column frm40__lov.lovname is varchar(30) instead of varchar(80).
* Column frm40__lov.lovgroup is varchar(30) instead of varchar(80).
* Column frm40__mnuapp.appnlslang is new varchar(49) to store NLS information.

* Column frm40__mnuitm.itmname is varchar(50) instead of varchar(40).

* Column frm40__mnuitm.itmtxt is new varchar(80).

* Column frm40__namelist.nlvar is new varchar(80).

* Column frm40__window.winicontit is renamed to wincontvw.

Oracle Forms 4.5 still support an Oracle based repository although the structure has changed a lot from version 4.0.

The Forms tables are from version 4.5 not documented any more, and may be subject to changes. The tables form however a unique source for maintenance oriented operations, and are thus listed below:

Be careful, most of the tables do not have proper primary key, reference or not null definitions. The **OWNER** columns is the name of the actual Oracle account holding the form (In upper case in the **FRM45** tables and in lower case in the **PECS** tables). **MODID** is the module id uniquely defined.

```
Table DE__ATTACHED__LIBS:
   OWNER Varchar2(30),
   MODID Number(10,0),
   ITEMID Number(10,0),
   LIBNAME Varchar2(255),
   LOCATION Varchar2(30));
Primary key on (OWNER, MODID, ITEMID);

Table FRM45__BINDVAR:
   OWNER Varchar2(32),
   MODID Number(9,0),
   ITEMID Number,
   NEXTBPOS Number,
   PLSQLBV_EP Number,
   TOTAL_BINDVAR Number);

NONUNIQUE index on ( MODID, ITEMID);
```

```
Table FRM45__BUFFER:
   OWNER Varchar2(32),
   MODID Number(9,0),
   STARTADDR Number,
   STARTREF Number,
   DATATYPE Number,
   LONGID Number);
NONUNIQUE index on (MODID);

Table FRM45__GRP:
   OWNER Varchar2(30),
   MODID Number(9,0),
   ITEMID Number(9,0),
   GRPNAME Varchar2(30),
   GRPFLAG Number);
Primary key on (MODID, ITEMID);
```

The **frm45__object** table describes all forms related objects. **NAME** is the actual name of the object, **OBEJCTTYPE** its internal type, and **SCOPE1**, **SCOPE2** and **SCOPE3** defines the scope hierarchy for the object. Scope1 would be canvas-name, block-name, formstrigger-name or null. Scope2 would be field-name, blocktrigger-name or null. And scope3 would be fieldtrigger-name or null.

```
Table FRM45__OBJECT:
   OWNER Varchar2(32),
   MODID Number(9,0),
   ITEMID Number,
   NAME Varchar2(32),
   OBJECTTYPE Number,
   SEQUENCE Number,
   RAWLEN Number,
   TEXTLEN Number,
   CHUNKNO Number,
   SCOPEID Number,
   SCOPE1 Varchar2(32),
   SCOPE2 Varchar2(32),
   SCOPE3 Varchar2(32),
   RAWDATA Raw(250),
   TEXTDATA1 Varchar2(2000),
   TEXTDATA2 Varchar2(2000),
   TEXTDATA3 Varchar2(2000),
   TEXTDATA4 Varchar2(2000),
   PROGRAMUNITID Number);
NONUNIQUE index on (MODID, ITEMID);
```

```
Table PECS__CLASS:
   CLSID Number(9,0),            /* class id - unique for each class */
   CLSTYP Number(9,0),       /* class type - i.e. forms, reports, etc. */
   CLSNAM Varchar2(30),                                /* class name */
   CLSVER Varchar2(30),                       /* version of this class */
   CLSCOM Varchar2(32),                             /* class comment */
   CLSOWN Varchar2(32),                         /* owner for this class */
   CLSDAT Date);
NONUNIQUE index on (CLSID);

Table PECS__DATA:
   EXPID Number(9,0),          /* owning experiment id for this event */
   RUNID Number(9,0),            /* run number for this event instance*/
   CLSID Number(9,0),               /* owning class id for this event */
   EVTTYP Number(9,0),         /* type of event (trigger, proc, etc.) */
   DATTYP Number,        /* type of data (matched, mismatched, point) */
   DATID Number,              /* data id - unique for this experiment */
   DATCOM Varchar2(32),                                   /* comment */
   ELATIM Number,                                    /* elapsed time */
   CPUTIM Number,                                        /* cpu time */
   ARG1 Number(9,0),              /* used for lookup of various form */
   ARG2 Number(9,0));                 /** names (block, item, etc.) **/
NONUNIQUE index on (CLSID);
NONUNIQUE index on (EXPID);

Table PECS__EVENTS:  a static code table
   CLSTYP Number(9,0),/* default value: -3 */ /* class for this event */
   EVTTYP Number(9,0),                          /* type for this event */
   EVTNAM Varchar2(32),                         /* name for this event */
   EVTCOM Varchar2(32));                     /* comment for this event */

Types are:
      1:  Application
      2:  Form
      3:  Block
      4:  Field
      5:  Key
      6:  Trigger
      7:  PLSQL
      8:  Commit
      9:  ExeQuery
     10:  LOV
     11:  Page
     12:  20Trigger
     13:  Procedure
```

```
     15:  Alert
     17:  Editor
     18:  Window
     19:  Canvas
     21:  ProcLine
     22:  TrigLine

Table PECS__EVENT_TYPE: a static code table.
   EVTID Number,
   EVTTYP Varchar2(32));

      1:  Duration
      2:  Point
      3:  Invalid
      4:  Invalid

Table PECS__EXPERIMENT:
   EXPID Number(9,0),   /* experiment id - unique for each experiment */
   EXPOWN Varchar2(32),                    /* oner of this experiment */
   EXPNAM Varchar2(32),                     /* Name of the experiment */
   EXPCOM Varchar2(255));             /* Comment for this experiment */
NONUNIQUE index on (EXPID);

Table PECS__PLSQL:
   MODID Number(9,0),
   PRCID Number,
   LINNUM Number,
   LINTXT Varchar2(255),
   LINTYP Number);
NONUNIQUE index on (MODID, PRCID);

Table PECS__RUN:
   EXPID Number(9,0),
   RUNID Number(9,0),
   RUNDAT Date,
   RUNCOM Varchar2(32));
NONUNIQUE index on (EXPID);

Table PECS__SUMMARY:
   EXPID Number(9,0),                                /* experiment id */
   CLSID Number(9,0),                                     /* class id */
   CLSTYP Number(9,0),                                  /* class type */
   MODID Number(9,0),
   EVTTYP Number(9,0),                               /* type of event */
   EVTNAM Varchar2(32),                          /* name of the event */
   SUMTYP Number,      /* summary event (matched,mismatched,point */
   SUMCOM Varchar2(32),                                    /* comment */
   SUMCNT Number(9,0),                   /* occurrences of this event */
   ELAAVG Number(9,4),                        /* average elapsed time */
```

```
   ELAMIN Number(9,4),                               /* minimum ... */
   ELAMAX Number(9,4),                               /* maximum ... */
   ELASTD Number(9,4),                      /* standard deviation ... */
   CPUAVG Number(9,4),                           /* average cpu time */
   CPUMIN Number(9,4),
   CPUMAX Number(9,4),
   CPUSTD Number(9,4),
   ARG1 Number(9,0),              /* used for lookup of various form */
   ARG2 Number(9,0),                /** names (block, item, etc.) **/
   BLKID Number(9,0),
   ITMID Number(9,0),
   BLKNAME Varchar2(32),
   ITMNAME Varchar2(32));
NONIQUE index on (CLSID);
NONUNIQUE index on (EXPID);
```

2.8.2 Use SCCS or RCS

In order to identify the different forms and libraries, it is advisable to assign a global variable - say **SCCS_VERS**, the string @(#) %\M% %\I% %\&H% (Without the '\\'). And then let the sccs system control and expand the symbols of the .fmt and .mmt files. Unfortunately though default text is not stored as literals in the .fmt files, but in hex format, so the sccs string must be stored elsewhere.

We recommend you to create a program unit like this:

```
function Get_Sccs_Version return Char is
begin
  return '@(#) %\M% %\I% %\&H%';
end;
```

And a when-new-form-instance trigger like this to save the value for inspection at runtime:

```
begin
  :sccs_vers := Get_Sccs_Version;
end;
```

When a new version of the form should be generated the following actions must be performed:

```
                                                # Make the '.fmt'
file ready for an update.
f40desm                                         # Make the changes
in the designer.
f40genm script=yes                      # .fmb -> .fmt
                                                # Check in and out
in the SCCS system
f40genm parse=yes                       # .fmt -> .fmb
f40genm generate=yes        # .fmb -> .fmx
```

The function will then reflect the actual version of the form:

```
function Get_Sccs_Version return Char is
begin
  return '@(#) sccs.fmt 1.1.1.1 4/26/94';
end;
```

You may also get the versions using what:

```
$ what sccs.*
sccs.fmb:
         sccs.fmt 1.1.1.1 4/26/94';
sccs.fmt:
         sccs.fmt 1.1.1.1 4/26/94';
sccs.fmx:
         sccs.fmt 1.1.1.1 4/26/94
```

2.8.3 Simplify the System

Use a single big code table (zip, country, ...), instead of many small ones. This keeps the code manipulation forms down to one, and the select statements may be reused much more easily.

2.8.4 Extract Common Code from Oracle Forms

If a number of forms have to have the same external capabilities, such as spawning reports, tracing SQL statements, checking mails, etc; then an associated menu application may just do it, since it runs in the same

process (with the same Oracle session) as the forms process. See the chapter later in this document.

2.8.5 Adapt a Naming Convention

In order to distinguish local PL/SQL variables from database object names, always prefix the local PL/SQL variables with something. (like local for local PL/SQL variables, fp_ for formal parameters, ...)

2.8.6 Make the Oracle Forms Tables big enough

It is sometimes advisable to keep the forms applications loaded in the forms tables, as you may use the referencing mechanism, or want to use the tables for maintenance purpose.

But as the forms tables are not created with any special storage parameters, you may increase database fragmentation and decrease forms development performance. A SQL*Plus script alt_frm.sql is available to put more reasonable parameters on the tables. If your forms tables is in use already, you may export them compressed, import them, and then run the script. Or you may wish to recreate the tables:

```
sqlplus $SYSTEM_PASS
@?/guicommon/tk2/admin/sql/tooldrop
@?/guicommon/tk2/admin/sql/rosdrop
@?/guicommon/vgs/admin/sql/vgdrop
@?/forms40/admin/sql/frm4drop

@?/guicommon/tk2/admin/sql/toolbild
@?/guicommon/tk2/admin/sql/rosbild
@?/guicommon/vgs/admin/sql/vgbild
@?/forms40/admin/sql/frm4bild

@?/guicommon/tk2/admin/sql/toolgrnt <oracle user>
@?/guicommon/tk2/admin/sql/rosgrnt <oracle user>
@?/guicommon/vgs/admin/sql/vggrnt <oracle user>
@?/forms40/admin/sql/frm4grnt <oracle user>
exit
```

2.8.7 Migrating from SQL*Forms version 3.0

Version 4.0 differs from version 3.0 a great deal.

* It allows usage of Windows oriented techniques to facilitate a rich user interface, such as fonts, colors, images, buttons, radio buttons, etc.

* It incorporates more features to make it easier to integrate with other tools through parameter lists, record groups, etc.

* It introduces more objects such as Master-Detail Block Relations and List of Values to make the form more easy to maintain.

* The internal structure of the database tables as well as the equivalent definition files have also changed substantially.

* Also the syntax in the user exits on how to exchange information with the running form has changed - check the manuals og see the getenvr.pc example in this document. <ifuxit.h> should be included instead of <usrxit.h>, and EXEC TOOLS must be used instead of EXEC IAF.

  If the syntax of setting and retrieving information from the actual form is not changed according to the manual, the following kind of error will appear linking f40runx:

```
ld:
 /wd/sql15/DR/lib/libsql.a(sqlgfo.o): jump relocation out-of-range,
   bad object file produced, can't jump from 0x845558 to 0x101838e4 (iapprs)
 /wd/sql15/DR/lib/libsql.a(sqlpfo.o): jump relocation out-of-range,
   bad object file produced, can't jump from 0x8458e4 to 0x101838e8 (iappfo)
```

It is rather simple to transform a version 3.0 SQL*Form to version 4.0. You simply has to call **f40genm** with the option upgrade=yes. Please remember however to take the hint **Prepare for Mouse Navigation** serious. Afterwards run **f40genm** with the **insert=yes** option in order to store the form in the database.

2.8.8 Running in Character Mode

Of course, running the form in character mode makes the form present itself somehow different from the windows-orientated look-and-feel. You may have to set the environment variable **term** to **t3**, though.

It is however, possible to execute the very same form in character mode as well as in the windows environment.

Buttons may be acticated by navigating to them, and then pressing select.

The feature of recording keystrokes and executing forms using the very same strokes only works in character mode.

2.8.9 Looking for Forms

The environment variable **ORACLE_PATH** may now be used to specify a list of directories where f40run[m] will look for a given form. The environment variable **FORMS40_PATH** is not used.

Oracle Forms 4.5 also supports the use of the variable FORMS45_PATH , but before ORACLE_PATH.

2.8.10 Transport to / from MS-Windows

Although the binary files (.**fmb**, .**mmb** and .**pll**) should be hardware platform independent, you may get error FRM-40030 (File **filename** is not a Oracle Forms 4.0 file) message. As the underlying modules might be of different versions. In this case use either the script files or extract the module from the database itself.

Remember to generate any library before any form or menu.

2.8.11 Trigger execution Sequence

A number of trigger events in forms 4.5 allows you to trap various events, but you would have to know the sequence in which some of the events are fired in order to utilize the mechanism.

Suppose triggers have been written for PRE-LOGON, POST-LOGON, PRE-LOGOUT, POST-LOGOUT, PRE-FORM, POST-FORM and for WHEN-NEW-FORM-INSTANCE.

If we call the first form (it must logon to the database), and we simply enter and leave again, the following sequence of trigger-execution would be observed: PRE-LOGON, POST-LOGON, PRE-FORM, WHEN-NEW-FORM-INSTANCE, POST-FORM, PRE-LOGOUT and POST-LOGOUT.

Assume however, that the POST-LOGON trigger does check that the application is only used between 7 am. and 6 pm.

```
declare
  a_check Varchar2(1);
  cursor check_access is
  select 'x' from dual
  where to_number(to_date(sysdate, 'hh24'))between 7 and 18;
begin
  open check_access;
  fetch check_access into a_check;
  if check_access%notfound then
    raise form_trigger_failure;
 end if;
  close check_access;
end;
```

Observe that if the exception is raised, then forms will terminate immediately without raising other events. (Not even ON-LOGOUT!)

2.9 Navigation

Make a choice on whether or not the operator should control navigation. Skilled users usually fancy free navigation.

This issue should not be misused to let the operator bother about **blocks, pages**, ...

2.9.1 Prepare for Mouse Navigation

Be sure not to execute any procedure to validate, compute, or perform any other semantic action when a navigation event is raised (such as going to next field, pre- or post-field, ...) because these events will be fired differently if the operator is going to run the same application in a non-character mode environment.

2.10 Tuning

2.10.1 Direct Reference

Instead of referring a variable like **:variable**, always (if possible) prefix with the block name **:block.variable**, to let forms find the variable faster.

2.10.2 Save System Variable Overhead

If the same PL/SQL block needs to read a system variable over and over, it is faster to save a local copy of the system variable once, and then do the referencing.

2.10.3 Use Explicit Cursors

If you know that a certain select does not return more than one row, use an explicit cursor, because an implicit cursor will always execute two fetches in order to get the **NO_MORE_ROWS** flag back.

Example: Assume we want the name of the president in a PL/SQL block. Using an implicit cursor, this could be done like this:

```
declare
   EName emp.ename%type;
begin
   select ename into EName
      from emp
      where job = 'PRESIDENT';
   if EName = 'JENSEN' then ...
   exception
   ...
end;
```

Unfortunately this example does not react properly on conditions like more than one president, or none at all. And even if only one exists in the table two fetches have to be performed according to the ISO standard, to see if in fact more rows could be availabel.

Instead, use an explicit cursor:

```
declare
   cursor president is
      select ename from emp where job = 'PRESIDENT';
   EName emp.ename%type;
begin
   open president;
   fetch president into EName;
   if president%found then
      if EName = 'JENSEN' then ...
   if end;
   close president;
end;
```

2.10.4 Be careful using Default Where / Order By Clauses

Using the default where/order by clause for blocks, be sure that the database supports the specified statement to prevent massive sorting. Also make sure that this facility is not used as an access prevention mechanism - because it is possible to get around the mechanism!

2.10.5 Reuse Cursors

How is it possible to reuse cursors in Oracle Forms? Often PL/SQL procedures and triggers will be invoked many times in a Oracle Forms session, how can we keep the cursors in these procedures open, so we do not have to re-parse them each and every time the procedure is called?

Actually Oracle Forms and PL/SQL is already handling this issue to a certain degree. Explicit and implicit cursors from PL/SQL procedures are not closed immediately when the programmer specifies the close call. Oracle Forms will keep a cache of open cursors, and if a cursor open request comes in, and the cursor is already there, no opening or parsing is done. The cursor is simply re-executed.

If no empty slots are found in the cursor cache, one of the not used cursors is closed, and the slot is reused for a new SQL-statement.

The number of entries in the cursor cache seems to be equal to the open_cursors init.ora parameter. This is unfortunate, since it makes it difficult to tune the size on a per-program-basis.

If you analyze your form with the -s option, the number of cursors reported exclude the cursors used in the PL/SQL procedures, as well as any recursive cursors used by the kernel to parse the actual SQL-statements.

Please also note that the term implicit in the -t option has nothing to do with implicit statements in PL/SQL, but only references the implicit generated SQL-statements to query, insert, delete and update blocks with underlying tables.

The advice is therefore to issue a close cursor call for every open call, not being worried about loss of performance as a result of this.

2.10.6 Response time Logging

Often there is a need to establish some material on what the response times are for different functions in the application. Either the customer

wants to log certain information in order to check the real values against those in the contract, or the developer wants to do some serious testing.

Unfortunately Oracle Forms does not include facilities to do timings on local events, so we should have to extend Oracle Forms with 2 user exits start_time, and get_time found in the iaptrace.pc file:

```
#include <stdio.h>
#include <time.h>
#ifndef IFUXIT
# include <ifuxit.h>
#endif
/* #include <usrxit.h> is obsolete under sqlforms 4.0 */
#include <sys/times.h>

extern  int    start_time();
extern  int    get_time();

EXEC SQL BEGIN DECLARE SECTION;
      varchar msg_val[100];
      char formfld[61];                  /* Store block.field here */
      char form_name[30];
      char from_form_fld[61];
      char trig_name[30];
      char id[30];
      long micro;
EXEC SQL END DECLARE SECTION;
EXEC SQL INCLUDE SQLCA.H;

static char trace_vers[]= "@(#) iaptrace.pc 1.1.5.1 5/18/94";

/* **********************************************************************
 * start_time is called with the following syntax for parameter string p:
 *
 * start_time
 * get_time form field trigger fieldname mode
 *
 * If mode include 'f' a message is written back to <fieldname>.
 * If mode include 'b' a log record is inserted in the FORMS_LOG table.
 *
 ********************************************************************** */
```

```
#define MAXARGS      7
#define ARBUFSIZ 512
char *wordb[6];

static long start_clock;

int start_time(p, paramlen, erm, ermlen, query)
  char *p;                                       /* Parameter string */
  int *paramlen;                          /* Ptr to param string length */
  char *erm;                           /* Error message if doesnt match */
  int *ermlen;                          /* Ptr to error message length */
  int *query;                              /* Ptr to query status flag */
{

  char msg[80];
  int  len;
  struct tms the_tms;

  start_clock = times(&the_tms);
  return IAPSUCC;
}

int get_time(p, paramlen, erm, ermlen, query)
  char *p;                                       /* Parameter string */
  int *paramlen;                          /* Ptr to param string length */
  char *erm;                           /* Error message if doesnt match */
  int *ermlen;                          /* Ptr to error message length */
  int *query;                              /* Ptr to query status flag */
 {
  extern int countwords();                /* Puts "words" into an array */
  extern char *cpystr();                 /* Copies one string to another */
  extern char *upper();                    /* Makes a string uppercase */
  extern char *remblank();                /* Get rid of extra whitespace */
  extern int strcmp();                          /* Compares two strings */
  int listsiz;                                      /* Number of values */
  int i;                                                /* Temp counter */
  char arbuf[ARBUFSIZ];               /* To hold string that is passed */
  char msg[80];
  char mode[10];
  int len;
  double df;
  short       to_forms, to_base;
```

```
struct tms  the_tms;

EXEC SQL WHENEVER SQLERROR GOTO sqlerr;

remblank( p );
strncpy(arbuf, p, ARBUFSIZ-1 );
listsiz = countargs( arbuf );

if ( listsiz < 1 ) {
  EXEC TOOLS MESSAGE 'get_time: Not enough arguments!';
  return IAPFTL;
}

if ( listsiz >= MAXARGS) {
  EXEC TOOLS MESSAGE 'get_time: Too many arguments!';
  return IAPFTL;
}

strcpy(form_name,wordb[1]);               /* get formname string   */
strcpy(from_form_fld,wordb[2]);           /* get fieldname string  */
strcpy(trig_name,wordb[3]);               /* get fieldname string  */
strcpy(formfld,wordb[4]);                 /* get fieldname string  */
strcpy(mode,wordb[5]);                       /* get mode string    */
to_forms = (strchr(mode, 'f') != NULL);
to_base  = (strchr(mode, 'b') != NULL);

strcpy(id,cuserid(NULL));
micro = times(&the_tms) - start_clock;
micro *= 10000;

if (to_forms) {
  df = micro * 1.0 / 1000;
 sprintf(msg_val.arr,"%s, Field: %s.%s, Trig: %s, Sec: %.6f",
       id, form_name, from_form_fld, trig_name, df);
 msg_val.len = strlen(msg_val.arr);

  EXEC TOOLS SET :formfld VALUES(:msg_val);
}

if (to_base) {
  EXEC SQL
    insert into forms_log (
      stamp, user_name, form_name, field_name, trigger_name, micro_sec)
```

```
    values
      (sysdate, :id, :form_name, :from_form_fld, :trig_name, :micro);
  }

  return IAPSUCC;                    /* Yes, return success code */

  sqlerr:
  sqlca.sqlerrm.sqlerrmc[sqlca.sqlerrm.sqlerrml] = '\0';
  sqliem( sqlca.sqlerrm.sqlerrmc, &sqlca.sqlerrm.sqlerrml);
  return IAPFAIL;
}
```

Figure: "The User Exit Rutines in **iaptrace.pc**"

Note that the log records will be committed when the form commits.

The **Forms_Log** table can be created with the following statement:

```
create table forms_log (
  stamp Date not null,
  user_name Char(30),
  form_name Char(30) not null,
  field_name Char(61),
  trigger_name Char(30) not null,
  micro_sec Number(10));
```

Two forms procedures are shown (**Start_Timer** and **Read_Timer**) - using the user exits, which should be made known to Oracle Forms:

```
procedure Start_Timer is
begin
  User_Exit('START_TIME');
end;

procedure Read_Timer ( trig_name Char, mode_val Char ) is
begin
  :global.micro := 'Nothing';
  User_Exit( 'GET_TIME '|| :system.current_form ||' '||
      :system.cursor_field ||' '|| trig_name ||' :global.micro '||
      mode_val );
```

```
if Instr( mode_val, 'f' ) > 0 then
    Message( :global.micro );
  end if;
end;
```

Figure: "Start_Timer and Read_Timer"

Now the procedures may be called from the trigger you want to time:

```
Start_Timer;
begin
        .   .   .
        .   .   .
        .   .   .
end;
Read_Timer( 'KEY-COMMIT', 'fb' );
```

In Oracle Forms 4.5, the PECS system (Performance Event Collection Service) offers the possibility to generate a statistics log, that could be loaded back into the PECS__% tables for analysis. If the PECS tables are installed in the system account, and you still get an table does not exist error message trying to load the statistical information, there is a good chance that grants to the actual user have not been performed. In this case you may have to execute the forms45/pecs/pecsgrnt.sql SQL*Plus script.

2.11 Transactions

Make a choice on whether or not the operator should control transactions. Or in other words: Does the operator or the application control the logical transaction by having control over **commit** and **rollback**.

2.11.1 Select Transaction Strategy

Be careful using the **ON-LOCK** trigger. Oracle recommend the use of the trigger in single user cases, where locking is not a hot issue. The **ON-LOCK** trigger may however also be used to change the locking and reservation mechanism of Oracle Forms.

Sometimes when many users have to change values in common rows in the database through Oracle Forms, you will find that the fastest users may be locked by users who for some reason did not release the reservation of the essential records yet.

Why not try to change the philosophy so the fastest users get through. If a user tries to commit changes, where the values once retrieved have changed in the meantime, then the user should receive an **Data changed by another user** - **Requery and change** message.

1. Find the block where the changed philosophy should be implemented.

2. Make sure that Oracle Forms does not reserve the rows when they are changed by the operator, by specifying the **ON-LOCK** trigger to **null** or something else.

3. Save the original value you are about to change.

4. In the **PRE-UPDATE** event, select the record to see if the record still matches the stored value, or fails.

2.12 Tricks

2.12.1 Allocate Global Variables

Note that it is possible to generate global variables in Oracle Forms at runtime. If you want to save some key values for later use, you would not know in advance how many variables to allocate. Here is a little example on how you might save the first field in every record as well as the rowid from the current block.

```
declare
  var   Char(30);
  field Char(30);
  rowidvar      Char(30);
```

```
begin
  field := Get_Block_Property( :system.cursor_block, FIRST_ITEM );
  rowidvar := :system.cursor_block||'.rowid';
  first_record;
  for i in 1 .. 100 loop
    if Name_In( field ) is null then exit; end if;
    var := ':global.empno_'||:system.trigger_record;
    Copy( Name_In( field ), var );
    var := ':global.rowid_'||:system.trigger_record;
    Copy( Name_In( rowidvar ), var );
    Next_Record;
  end loop;
end;
```

Note that I have limited the number of saved records, but there is no fixed upper limit on how many global variables a form may allocate - Only the amount of memory will put the limit.

2.12.2 Security

Oracle Forms 4.5 allow the programmer to extract the used Oracle username, password and connect string with get_application_property, in order to be able to spawn new sessions on the same account. Note however that the programmer must be sure to deal carefully with this information, and that users may request a verified guarantee stating that these functions (if used) are to be trusted.

2.12.3 Getting Alerts from the RDBMS

In Oracle7 it is possible to set up an alert (not to be mistaken from a forms alert!) telling all interested clients that a certain event did happen. Now here we shall see how the operators may be notified.

Assume a trigger have been placed to catch changes to the salary column of the emp table, like this:

```
create or replace trigger emp_sal_change
after insert or delete or update of sal on emp
for each row
declare
  string Varchar2( 80 );
```

```
begin
  if updating then
    string := 'Updating emp '||To_Char( :new.empno )||
              ', old sal: '||To_Char( :old.sal )||' changed to: '||To_Char( :new.sal  ;
  else
    if inserting then
      string := 'Inserting emp '||To_Char( :new.empno )||', new sal: '||To_Char( :new.sal
    else
      string := 'Deleting emp '||To_Char( :old.empno )||', old sal: '||To_Char(  :old.sal
    end if;
  end if;
  dbms_alert.Signal( 'emp_sal_alert', string );
end;
```

It is certainly not good programming style to include language dependent text in the messages, but as an example ... Note that the trigger will signal an alert called **emp_sal_alert** with a proper message. Note also that alerts instead of pipes are used since we would only like to get the signal if the transaction changing the salery column actually commits.

A client may now show interest in this specific RDBMS alert with the register call. We also need a timer to check is an alert has arrived. Unfortunately we would have to do this in a polling fasion, since there is no concepts of threads in Oracle Forms. It will however not be very expensive to do the polling, since only little database activity is involved.

The when-new-form-instance trigger may look like this:

```
declare
  timer_id Timer;
begin
  dbms_alert.Register( 'emp_sal_alert' );
  timer_id := Create_Timer( 'CHECK_TIMER', 10000, REPEAT );
  if Id_Null( timer_id ) then
    Message( 'Error creating the CHECK_TIMER timer' );
  end if;
end;
```

Now when the **CHECK_TIMER** timer expire, we need to see if any **emp_sal_alert** alerts are pending, so we set the timeout parameter to 0. Note that the Waitany procedure is used to allow for other alerts as well.

The **when-timer-expired** trigger may look like this:

```
declare
  alert_message Varchar2( 2000 );
  alert_name Varchar2( 2000 );
  status Number;
  alert_id Alert;
  alert_result Number;
begin
  :global.timer_name := Get_Application_Property( TIMER_NAME );
  if :global.timer.name = 'CHECK_TIMER' then
    dbms_alert.Waitany( alert_name, alert_message, status, 0 );
    if status = 0 then
      alert_id := Find_Alert( 'EMP_SAL_ALERT' );
      if Id_Null( alert_id ) then
        Message( 'Alert EMP_SAL_ALERT does not exist!' );
      else
        Set_Alert_Property( alert_id, ALERT_MESSAGE_TEXT, alert_message );
        alert_result := Show_Alert( alert_id );
      end if;
    end if;
  else
    Message( 'Another timer did expire: '||:global.timer_name );
  end if;
end;
```

2.13 Debugging

Using the debug switch in **frm4run (-d)** will only tell when the different triggers are fired. For more complex forms it is even more interesting to see when the different procedures are called. In order to be able to follow the procedure calls a little PL/SQL script **ondebug** has been invented. It will insert a message line after the first **begin** in every procedure of a particular form in the Oracle Forms tables.

If the forms_ddl routine is present, you may also create two developer buttons **Trace_On** and **Trace_off**, and let the **when-button-pressed** call forms_dll('alter session set sql_trace = true'); and forms_dll('alter session set sql_trace = false');.

Remember though, that DDL statements will issue an implicit commit, so always precede the **forms_dll** calls with **Commit_Form.**

2.14 Flexible Presentation

Oracle Forms 4.x is rather static in its view for possible presentational changes. In other words - If the designer does not implement with flexible presentation issues in mind, the customer is likely not to be able to adjust any presentational issues at all in a Oracle Forms based system.

If however the programmer wants to allow some degree of presentational freedom (without violating the integrity of the application), how should he or she act?

It is well known that more dramatical presentational changes, like adding fields on the fly, cannot easily be done at runtime in forms. Note however that many properties of existing object may be changed at runtime. This will allow a system to read in local presentational settings from file or database, and to issue appropriate **set_xxx_property** calls in the beginning of each form, in order to allow local customers to change the presentation of the forms, without having to ask the developers to maintain many different presentational variants of the same forms.

Unfortunately the **get_xxx_property** and **set_xxx_property** procedures, are not symmetric since the property we can get, cannot in all cases be used directly in the set procedure. As an example **the get_item_property** of DISPLAYED will give the character strings 'TRUE' or 'FALSE', but in the **set_item_property** procedure, the symbols PROPERTY_TRUE or PROPERTY_FALSE must be used.

2.14.1 Existing Posibilities

First we will look at the existing possibilities in Oracle forms 4.5.

2.14.1.1 Alert Objects

The alert message may be changed at runtime, unfortunately there is no equivalent way to extract the existing value of an alert message.

2.14.1.2 Application Objects

The cursor style may be changed at runtime, and equivalent ways to extract the existing value exist. The display height and width are unfortunately not settable properties.

2.14.1.3 Block Objects

The default where clause and order by clause may be changed at runtime. Likewise may the navigator style and optimizer hints may be changed. The number of records to display and records to fetch are unfortunately not settable properties.

2.14.1.4 Canvas Objects

The height and width as well as visual attributes may be changed at runtime.

2.14.1.5 Forms Objects

Neither the character cell height nor the width may be changed at runtime.

2.14.1.6 Item Objects

Properties like auto hint, case restriction, current record attribute, if the item is displayed or not, echo, if the item is enabled at all, fixed length information, the format mask, height and width, name of a corresponding icon, label text, position and visual attributes may be changed at runtime.

But properties like alignment, editor name and position, hint text, type of the item and the query length cannot be changed at runtime.

2.14.1.7 LOV Objects

Properties like auto refresh, the size and position may be changed at runtime.

2.14.1.8 Menu Objects

Properties like checked, displayed, enable and the label may be changed at runtime.

2.14.1.9 Radio Button Objects

Properties like displayed, enable, label, size, position and the visual attributes may be changed at runtime.

2.14.1.10 Relation Objects

The property auto query may be changed at runtime.

2.14.1.11 View Objects

Properties like display position, position on canvas, size and visibility may be changed at runtime.

2.14.1.12 Window Objects

Properties like position, size, remove on exit flag and visibility may be changed at runtime. The title can however not be changed at runtime.

2.14.2 Load Properties to a Table

First let us collect the properties of items, since they are relatively easy to start with.

The following procedure will visit all blocks and all items of a form, in order to ask another procedure to examine the properties of an item.

Note that **item-id** is used in order to cope with duplicate item names across blocks.

```
procedure Save_Properties is
  form_name Varchar2(30);
  block_name Varchar2(30);
  item_name Varchar2(30);
  item_id Item;
```

```
begin
  form_name := Get_Application_Property( CURRENT_FORM_NAME );
  block_name := Get_Form_Property( form_name, FIRST_BLOCK );
  loop
    Save_Obj_Prop( form_name, block_name, 'BLOCK' );
    item_name := Get_Block_Property( block_name, FIRST_ITEM );
    loop
      item_id := Find_Item( block_name||'.'||item_name );
      if not Id_Null( item_id ) then
        Save_Obj_Prop( form_name, block_name||'.'||item_name, 'ITEM' );
        item_name := Get_Item_Property( item_id, NEXTITEM );
        exit when item_name is null;
      end if;
    end loop;
    block_name := Get_Block_Property( block_name, NEXTBLOCK );
    exit when block_name is null;
  end loop;
  commit;
end;
```

The following routine will get as many settable properties from the item in question as possible - and pass them on to the next routine.

Note that as long as **Get_Item_Property**(**<item>, ITEM_TYPE**) does not work (error 40738 is issued from Oracle Forms 4.5.5), only general properties may be selected. This error is said to be corrected in version 4.5.6.

```
procedure Save_Obj_Prop (form_name Varchar2,
  obj_name Varchar2, obj_type Varchar2) is

  item_id Item;
  item_type Varchar2(30);
  prop_value Varchar2(200);
begin
  if (obj_type = 'ITEM') then
    item_id := Find_Item( obj_name );
    if not Id_Null( item_id ) then
      item_type := Get_Item_Property( item_id, ITEM_TYPE );
      if (item_type in ('BUTTON', 'CHECKBOX')) then
        prop_value := Get_Item_Property( item_id, LABEL );
        Save_Prop_On_Db( form_name, obj_name, obj_type, 'LABEL', prop_value );
       end if;
      prop_value := Get_Item_Property( item_id, DISPLAYED );
      Save_Prop_On_Db( form_name, obj_name, obj_type, 'DISPLAYED', prop_value );
--    Get other General Properties of Items like AUTO_HINT, ENABLED, HEIGHT,
```

```
--    MOUSE_NAVIGATE, NAVIGABLE, VISUAL_ATTRIBUTE and WIDTH.
    end if;
  elsif (obj_type = 'BLOCK') then
    prop_value := Get_Block_Property( item_name, CURRENT_RECORD_ATTRIBUTE );
    Save_Prop_On_Db(form_name, obj_name,
obj_type,'CURRENT_RECORD_ATTRIBUTE',prop_value);
--  Get other General Properties of Blocks like DEFAULT_WHERE,
NAVIGATION_STYLE,
--  OPTIMIZER_HINT, ...
  end if;
end;
```

And now the routine to actually update the forms_properties table:

```
procedure Save_Prop_On_Db (p_form_name Varchar2,
  p_obj_name Varchar2, p_obj_type Varchar2,
  p_prop_type Varchar2, p_prop_value Varchar2) is

  l_prop_value Varchar2(200);
  cursor check_if_prop_there is
  select prop_value from forms_properties
  where form_name = p_form_name
    and obj_name = p_obj_name
    and obj_type = p_obj_type
    and prop_name = p_prop_type;
begin
  open check_if_prop_there;
  fetch check_if_prop_there into l_prop_value;
  if check_if_prop_there%FOUND then
    if (l_prop_value != p_prop_value) then
      update forms_properties
        set prop_value = p_prop_value
        where form_name = p_form_name
          and obj_name = p_obj_name
          and obj_type = p_obj_type
          and prop_name = p_prop_type;
    end if;
  else
    insert into forms_properties
      (form_name, obj_name, obj_type, prop_name, prop_value) values
      (p_form_name, p_obj_name, p_obj_type, p_prop_type, p_prop_value);
  end if;
  close check_if_prop_there;
end;
```

The forms_properties table does look like this:

```
create table forms_properties (
  form_name Varchar2(30) not null,
  obj_name Varchar2(61) not null, /* to account for block.item */
  obj_type Varchar2(30) not null,
  prop_name Varchar2(30) not null,
  prop_value Varchar2(200));
```

2.14.3 Get Properties from the Table

Now the user or administrator may modify or add the values in the **forms_properties** table.

And the forms application may ask for some of its objects to be reshaped according to the possibly changed properties. Here all Item objects in the EMP block are reshaped:

```
Get_Properties(
  Get_Application_Property( CURRENT_FORM_NAME ),
  'EMP.%', 'ITEM', '%' );
```

The Get_Properties procedure may look like this:

```
procedure Get_Properties (p_form_name Varchar2,
  p_obj_name Varchar2, p_obj_type Varchar2,
  p_prop_type Varchar2) is

  l_obj_name Varchar2(61);
  l_obj_type Varchar2(30);
  l_prop_type Varchar2(30);
  l_prop_value Varchar2(200);

  cursor scan_prop_from_db is
  select obj_name, obj_type, prop_name, prop_value
  from forms_properties
  where form_name = p_form_name
    and obj_name like p_obj_name
    and obj_type like p_obj_type
    and prop_name like p_prop_type;
```

```
begin
  open scan_prop_from_db;
  loop
    fetch scan_prop_from_db
      into l_obj_name, l_obj_type, l_prop_type, l_prop_value;
    exit when scan_prop_from_db%notfound;
    Set_Prop_In_Form( l_obj_name, l_obj_type, l_prop_type, l_prop_value );
  end loop;
  close scan_prop_from_db;
end;
```

For all fetched rows in the property table, the **Set_Prop_In_Form** procedure is called, and it may look like this:

```
procedure Set_Prop_In_Form (p_obj_name Varchar2,
  p_obj_type Varchar2, p_prop_type Varchar2, p_prop_value Varchar2) is
  item_id Item;
begin
  if (p_obj_type = 'ITEM') then
    item_id := Find_Item( p_obj_name );
    if not Id_Null( item_id ) then
      if (p_prop_type = 'DISPLAYED') then
        if (p_prop_value = 'TRUE') then
          Set_Item_Property( item_id, DISPLAYED, PROPERTY_TRUE );
        else
          Set_Item_Property( item_id, DISPLAYED, PROPERTY_FALSE );
        end if;
      elsif (p_prop_type = 'AUTO_HINT') then
        if (p_prop_value = 'TRUE') then
          Set_Item_Property( item_id, AUTO_HINT, PROPERTY_TRUE );
        else
          Set_Item_Property( item_id, AUTO_HINT, PROPERTY_FALSE );
        end if;
      end if;
    else
      Message( 'Could not find item >'||p_obj_name||'<' );
    end if;
  elsif (p_obj_type = 'BLOCK') then
    if (p_prop_type = 'DEFAULT_WHERE') then
      Set_Block_Property( p_obj_name, DEFAULT_WHERE, p_prop_value );
    elsif (p_prop_type = 'CURRENT_RECORD_ATTRIBUTE') then
      Set_Block_Property( p_obj_name, CURRENT_RECORD_ATTRIBUTE, p_prop_value );
    end if;
  end if;
end;
```

Note that it is rather awkward to set the boolean values, since we do need to call the **Set_xxx_Property** procedure with the right constants.

Also note that the call Set_Item_Property(item_id,DISPLAYED,PROPERTY_FALSE);, may raise the error 41014 in forms version 4.5.5, (cannot set attribute of null canvas item <nnn>).

3 Layout Advise

1. In general, do not load the screen with too many fields. Put the secondary fields on another page or on a pop-up page.

2. Fields corresponding to each other should be located together.

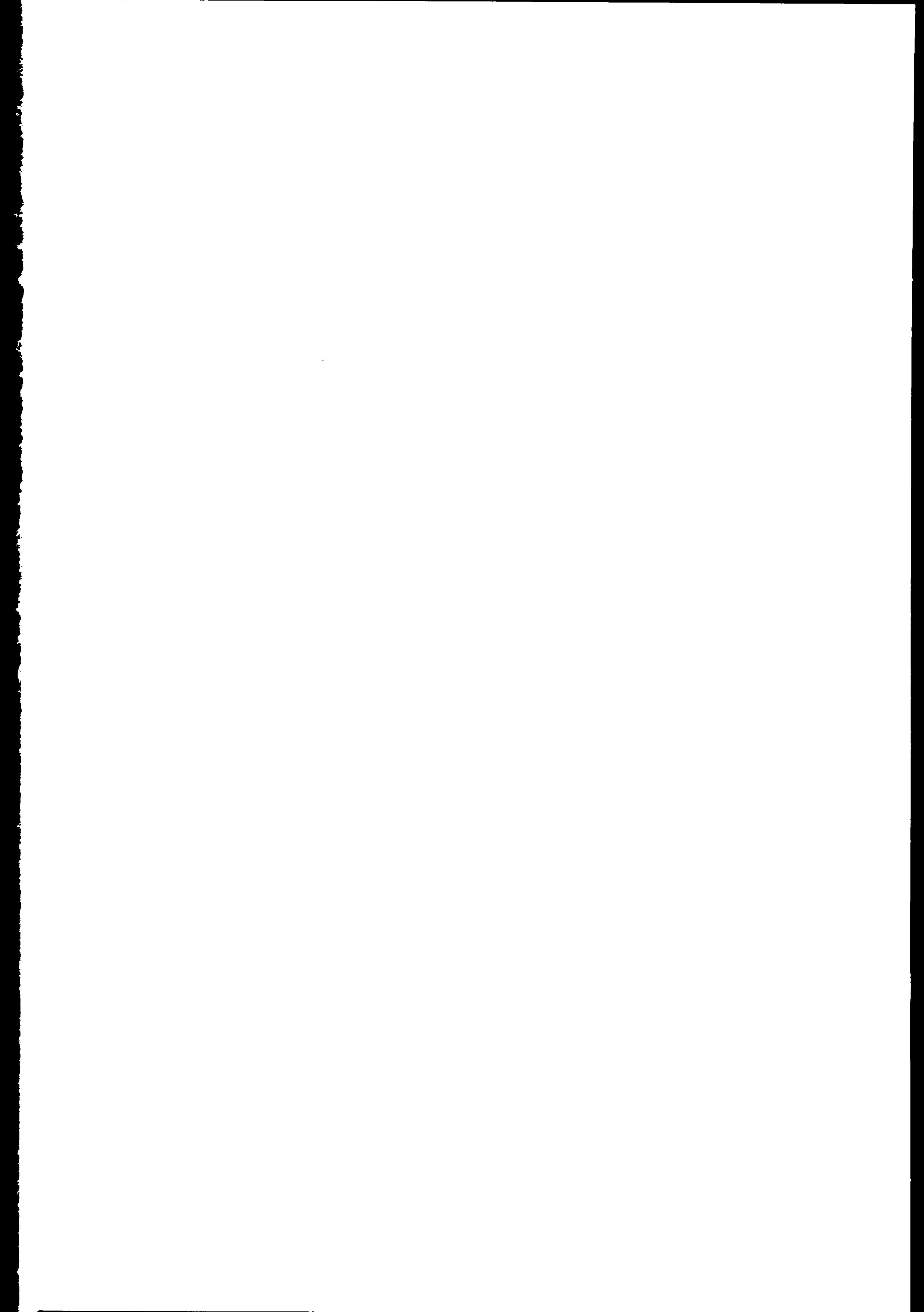
3. If a field contains a code of some sort, then the actual code should be looked up automatically.

4. Fields that accept abbreviated input, may expand their contents when the field is validated.

   Ex.: 'Y' gets expanded to 'Yes'.

5. If some fields (usually key fields) have influence on the contents of fields on another screen, then these key fields should be visible on the related screens as well.

6. Remember to add the list_of_value function to all applicable fields, especially the foreign key fields. You may want to develop the list_of_value screen yourself instead of using the default, which is hard to customize.



Dansk Data Elektronik A/S
Herlev Hovedgade 199
DK-2730 Herlev
Tel. (+45) 42 84 50 11
Fax (+45) 42 84 52 20