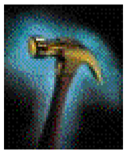


Using the Oracle® Forms Application Programming Interface (API)

An Oracle Technical White Paper

EIT-DE-WP-56

January 2000

TABLE OF CONTENTS

TYPOGRAPHICAL CONVENTIONS

This white paper uses the following typographical conventions to set apart important elements from the body of the text. C code and filenames appear in a `monotype` font, as follows:

```
d2fctxcr_Create();
d2fctx.h
```

Boldface words are used to add emphasis when introducing new concepts.

FORMS API OVERVIEW

WHAT IS THE FORMS API?

Oracle Forms includes an Application Programming Interface (API) that enables C programmers to create, load, edit, save, and compile Forms module files (`.fmb`, `.mmb`, `.olb`, and `.pll` files) from self-written C programs. Form module files are normally created and edited using the Form Builder, the design-time component of Oracle Forms. The Forms API gives you access to almost all the Form Builder functionality, and because it is a programmatic interface and not an interactive development environment, it is an ideal tool for writing scripts that perform repetitive tasks on large numbers of form module files.

BENEFITS OF THE FORMS API

Consider the following example problem that you might face using Form Builder. Let's say your application contains 100 form module files, each one having several screens (canvases). Your manager reviews your work and complains that the 8-pt font you used everywhere is too small, dictates that all usage of the word "business" should be replaced by "e-business", and asks that you double-check that you no longer refer to the old ORDERS table.

One option is to manually open, check, modify, and save each of the 100 form module files. This solution will not only take a long time, but is also tedious and error-prone. A better solution would be to write a simple C program that automates the task of opening each form module file, making the appropriate checks and modifications, and finally saving and closing the file. Assuming your C program was correct, you could ensure that all the necessary changes were correct, using only a small fraction of the time and effort.

In this paper, we demonstrate how to effectively use the Forms API. Using several sample C programs, we show you how to work with form module files, get and set properties, create and delete objects, compile files, search for arbitrary objects, and so on. Our intention is that this paper describes the API in enough detail that you can quickly apply the API to solving your particular business problems.

Assumptions

In this paper, we assume that you have a working knowledge of C programming, and understand the basic terms and concepts in Form Builder. For more detailed information concerning Form Builder, refer to the section titled *Related Documents*.

The steps required to write a C program using the Forms API will vary from platform to platform, as well as among the various development environments. The examples in this paper require only a basic command-line C compiler and linker. In general, each of your C files will `#include` several of the Forms API header files, and make a number of calls to the macros and functions defined in those header files. After compiling your C files, link them with the Forms API library included with the Oracle Forms release. On some platforms, a sample makefile is included with the release of the Forms API to make this easier.

Note: remember to keep backup copies of your Form Builder (.fmb) files before running a Forms API program that might alter those files, in case you later discover errors in your Forms API program.

EXAMPLE 1: BASIC PROGRAM STRUCTURE AND METHODS

We'll start off by presenting a simple C program written using the Forms API, and then explain the purpose of each line of code in detail. This example will show you how to:

- Create and destroy the Forms API context
- Open and close form module (.fmb) files
- Get a simple property value

Example 1 – FormName.c

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>

#include <d2fctx.h>  /* Forms API context */
#include <d2ffmd.h>  /* Form module header file */

int main (int argc, char *argv[])
{
    d2fctxa     ctx_attr;
    d2fctx     *ctx;
    d2ffmd     *form;
    text       *form_name;
```

```
        /* Check arguments */
        if ( argc != 2 )
        {
            fprintf(stderr, "USAGE: %s <filename>\n", argv[0]);
            exit(1);
        }

        /* Create Forms API context */
        ctx_attr.mask_d2fctxa = (ub4)0;
        if ( d2fctxcr_Create(&ctx, &ctx_attr) != D2FS_SUCCESS )
        {
            fprintf(stderr, "Error creating Forms API context\n");
            exit(1);
        }

        /* Load the form module into memory */
        if ( d2ffmdld_Load(ctx, &form, argv[1], FALSE) != D2FS_SUCCESS )
        {
            fprintf(stderr, "Failed to load form module: %s\n", argv[1]);
            exit(1);
        }

        /* Get the name of the form module */
        if ( d2ffmdg_name(ctx, form, &form_name) != D2FS_SUCCESS )
        {
            fprintf(stderr, "Error getting the name of the form module\n");
        }
        else
        {
            /* Print the name of the form, then free it */
            printf ("The name of the form is %s\n", form_name);
            free(form_name);
        }

        /* Destroy the in-memory form */
        if ( d2ffmdde_Destroy(ctx, form) != D2FS_SUCCESS )
        {
            fprintf(stderr, "Error destroying form module\n");
        }

        /* Close the API and destroy context */
        d2fctxde_Destroy(ctx);

        return 0;
    }
```

EXPLANATION OF EXAMPLE 1

Now, we'll walk through the example program line-by-line, explaining the purpose and function of each line of code. Although simple, this example illustrates the general structure and principles present in every Forms API program.

Including Header Files

Every Forms API program must include one or more of the Forms API header files. The header file `d2fctx.h` is required in all programs. In the example above, it's the first file included. Other header files, like the other included file in our example, `d2ffmd.h`, define the functions and macros associated with a particular object type. We'll talk more about object types in the next section, but for now we need to include `d2ffmd.h` because this program will be working on form module objects, and `d2ffmd.h` is the header file corresponding to form module objects.

Creating the Forms API Context

After validating the number of arguments, our program creates the Forms API context. Creating the Forms API context is required in every Forms API program, because the context pointer is the first argument passed into all Forms API functions. The second argument to `d2fctxcr_Create()` allows you to specify optional attributes that we'll discuss later. For now, we initialize the mask field of `ctx_attr` to zero, indicating that none of these options will be specified.

Loading a Form Module

In the next part of the program, we load in the form module file specified by the user on the command line by calling `d2ffmdld_Load()`. Notice that the context is passed in as the first argument as usual. The third argument is the file name, and the fourth argument tells whether to look for the file in the database instead. Passing in `FALSE` as we've done means look for the file on the file system. If the function returns successfully, the second argument `form` will point to a form module object of type `d2ffmd`. A form module (`d2ffmd`) is just one type of object; most of the other object types will be familiar to Oracle Forms programmers: blocks (`d2fblk`), items (`d2fitm`), canvases (`d2fcnv`), and so on. Again, we'll discuss objects more later.

Getting a Property Value

Just as in the Form Builder development environment, every Forms object has a set of properties associated with it. In this next part of the example, we ask the form module to look in its list of properties and tell us what the value of its "name" property is. The function (or rather, macro) that's used to get this information is `d2ffmdg_name()` – you can see the definition of this macro in the `d2ffmd.h` header file. Don't worry about the cryptic-looking name of this function now; we'll reveal the naming conventions for all the functions and macros later in this document.

Like many other Forms API functions, this function returns `D2FS_SUCCESS` or `D2FS_FAIL`. When returning successfully, the desired value is placed in one of the "out" parameters; in our case `form_name` will be a string holding the name of the form module object. We'll print this name to the display, and then (importantly) free the memory that was allocated when this string was retrieved.

Destroying objects

The call to `d2ffmdde_Destroy()` destroys the in-memory form module object, freeing up all the memory resources reserved by that object. This is analogous to "closing" the form module from the Form Builder development environment (the .fmb file is not changed by this operation, because you didn't save the object before destroying it).

Final clean-up

The final line of code, `d2fctxde_Destroy()`, should be the final call to the Forms API in every program. This operation frees up all memory reserved by the Forms API and terminates all internal processes. Once `d2fctxde_Destroy()` is called, the context (`ctx`) ceases to be valid, and no further calls to the Forms API are possible.

BASIC CONCEPTS: OBJECTS, PROPERTIES, CONVENIENCE MACROS

In this section, we'll drill deeper into the basic concepts introduced in Example 1. We'll describe in detail the following Forms API concepts:

- Forms objects
- Forms properties
- Getting and setting property values using the convenience macros.

WHAT ARE FORMS OBJECTS?

Form Builder is an object-oriented software application, in that all the visual (user interface) and data components of an application are represented as individual objects. All of the nodes you see in the Form Builder's Object Navigator are represented in the Forms API as objects. For example, form modules, data blocks, menu items, alerts, radio buttons, canvases, property classes, object libraries, and object groups are all forms objects. In addition, some lower-level objects such as the columns of a record group, or the font of a text item, are also forms objects.

Each object type is identified by a single unique **constant** in the `d2fdef.h` header file. You use this constant when querying the type of an object, or when creating a new object of a specific type. The constants begin with the prefix `D2FFO_`. For example, record group objects are all of type `D2FFO_REC_GROUP`, while alert objects are all of type `D2FFO_ALERT`. In Oracle Forms 6*i*, there are 37 types of objects. There's also a generic object type called `D2FFO_ANY`, which can be very useful when traversing lists of heterogeneous object collections, for example.

Each object type has an associated **header file** that defines the functions and macros associated with objects of that type. The name of the header file includes a unique three-letter abbreviation that describes the object type. For example, the header file for radio button objects is called `d2frdb.h`. All the functions and macros specifically associated with radio buttons are defined in that header file. Recall that in Example 1, we needed to include `d2ffmd.h` because we called functions and macros related to form module (`fmd`) objects.

Each object type is also associated with a C **typedef**. For example, pointers to item objects are defined as variables of type `d2fitm`, while record group objects are declared using the `d2frcg` type. In the example above, the form module object was declared as a variable of type `d2ffmd`. Variables of type `d2fob` represent Generic Objects. Importantly, any object (for example, `d2fitm`, `d2fcnv`, etc.) can be typecast to the generic object `d2fob`. The generic object is useful when you are either not sure of the type of a specific object, or are deliberately writing generic code that can apply to objects of more than one type. Please note, however, that using type-specific declarations, functions, and macros (where possible) offers stricter validation and type-checking at both compile-time and run-time.

WHAT ARE FORMS PROPERTIES?

All Forms objects have a number of properties that determine their behavior and/or appearance. Some object types have only two or three properties, while others have nearly one hundred. In the Form Builder development environment, the properties and their values are displayed in the Property Palette. You can view and modify the property values in the Property Palette. In the Forms API, you can similarly view (get) and modify (set) the property values of objects by using the appropriate functions and macros.

Much like object types, each property type is identified by a single unique constant in the `d2fdef.h` header file. This constant can be used when querying for a specific property of an object, when adding a property to a property class, etc. The constants all begin with the prefix `D2FP_`. For example, the "alert message" property is uniquely identified by the constant D2FP_ALT_MESG. In Oracle Forms *6i*, there are 540 types of properties. Note that the abbreviated property name (for example, ALT_MESG for the property "alert message") is important for the naming scheme explained later.

Each property is stored using one of five storage classes:

- Text (t)
- Number (n)
- Boolean (b)
- Blob (p)
- Object (o)

This means that when you ask an object for the value of one of its text-valued properties (for example, D2FP_NAME or D2FP_TITLE), it will give back a text string. Similarly, when you ask for the value of a Boolean-valued property (for example, D2FP_HIDE or D2FP_MINIMIZE_ALLOWED), you receive a simple TRUE or FALSE value in return. In the next section, we'll explain how the pre-defined macros in each object's header file help you use the correct type when working with property values. Later, we'll learn about functions that allow you to get the storage type for any property type.

GETTING AND SETTING PROPERTY VALUES USING THE CONVENIENCE MACROS

As we have seen, the Forms API defines many types of objects and even more types of properties that apply to one or more of the objects. They're the same objects and properties that you see in the Form Builder development environment. We've also seen that each object type has its own header file that defines functions and macros related to that type of object.

Some of the functions in those header files are used to get and set property values. Because there are five storage types (and unlike C++, the C language does not support polymorphism), there are five functions each for getting and setting property values, one for each storage type. For example, the file d2falt.h (for alert objects) defines these functions for getting property values:

```
d2faltgb_GetBoolProp(d2fctx *ctx, d2falt *obj, ub2 prop_typ, boolean *val)
d2faltgn_GetNumProp (d2fctx *ctx, d2falt *obj, ub2 prop_typ, number *val)
d2faltgt_GetTextProp(d2fctx *ctx, d2falt *obj, ub2 prop_typ, text **val)
d2faltgo_GetObjProp (d2fctx *ctx, d2falt *obj, ub2 prop_typ, d2fob **val)
d2faltgp_GetBlobProp(d2fctx *ctx, d2falt *obj, ub2 prop_typ, dvoid **val)
```

By using one of the property type constants (for example, D2FP_ALT_MSG, D2FP_BTN_1_LBL, etc.) for the prop_typ argument, you can use these functions to get property values of an alert object.

However, you will need to know which of the five to use, and that depends on the storage class of the property type.

A preferred alternative is to use the convenience macros defined in each object header file. The advantage of these macros is that the mapping between a property type and its storage class is built into the macro, and therefore enforced by the compiler. Look at two of the convenience macros in `d2falt.h`:

```
#define d2faltg_alt_msg(ctx,obj,val) \
                    d2faltgt_GetTextProp(ctx,obj,D2FP_ALT_MSG,val)

#define d2faltg_btn_1_lbl(ctx,obj,val) \
                    d2faltgt_GetTextProp(ctx,obj,D2FP_BTN_1_LBL,val)
```

Note that both macros cover the function `d2faltgt_GetTextProp()` that we saw above; they differ only in that they pass in a different property type constant. While functionally equivalent to calling `d2faltgt_GetTextProp()` with the various `D2FP_` constants, the convenience macros improve the code's readability and conciseness. This example demonstrates convenience macros with text-valued properties; similar macros exist for number-valued properties, Boolean-valued properties, etc. Refer to the section below on the Forms API function naming scheme for these.

All the Get and Set functions perform basic validation to ensure that the property is being get or set legally, and that the object is able to have properties of the given property type. In the case of an illegal operation, the function returns an error status such as `D2FS_TYPEMISMATCH` or `D2FS_BADPROP`.

GENERIC CONVENIENCE MACROS

There is also a header file called `d2fob.h` that is very similar to the object-specific header files, except that it is not associated with any particular object type; it is a kind of 'generic' or 'base' object header file. This file also has convenience macros for getting and setting property values. An example macro follows:

```
#define d2fobg_btn_1_lbl(ctx,obj,val) \
                    d2fobgt_GetTextProp(ctx,obj,D2FP_BTN_1_LBL,val)
```

The difference between calling `d2fobg_alt_msg()` and `d2faltg_alt_msg()` is that the latter type-specific function validates that the object you passed in (`obj`) is in fact an alert object (that is, of type `D2FFO_ALERT`). The former (generic) function doesn't check the object type before trying to set the property. The catch here is that if you pass in an object that does *not* have the property `D2FP_ALT_MSG` (a canvas, for example), then the function returns an error saying that that property is not associated with the type of the object you passed in. Despite the reduced amount of type-checking, the `d2fob*` functions are useful for writing generic code. For example, you could write a program that loops over every object in a form module and prints out the name of each object in a generic way.

FUNCTION AND MACRO NAMING SCHEME

By now you're probably wondering whether there's any pattern to the seemingly convoluted convenience function and macro names in the object header files. The answer is yes, there *is* a consistent naming scheme used throughout all header files, which we'll describe now.

Functions

Forms API function names consist of three parts using this pattern:

```
d2f + category + d_description
```

All Forms API functions begin with `'d2f'` without exception. A two- or three-character function "category" follows. In many cases, this is simply the object type: either `'ob'` (for the generic object) or one of the specific object types: `'alt'` for alerts, `'blk'` for blocks, etc. For those classes of utility functions that do not deal directly with objects, such as those in `d2fctx.h` (Forms context) and `d2fpr.h` (Property metadata), the letters following the `'d2f'` are more like a categorization of the function.

After that, two or three letters describe the purpose of the function, followed by an underscore and then a longer description of the function's purpose. Having both a short and long description is obviously redundant, but this makes the Forms API portable to nearly all operating system platforms.

For example, the function for duplicating alerts is `d2f + alt + du_Duplicate()`, and the function for finding out whether a property of a block is inherited is `d2f + blk + ii_IspropInherited()`. (Don't worry if you don't know what these functions *do* yet; we mention them here just to illustrate the function naming pattern.) As a general rule, if you know the category of the function (for example, the object type abbreviation) and you know the operation you want to perform, you can combine them and there should be a function with that name that you can use.

These rules also apply to the Get and Set functions we mentioned above, with only slight changes. These functions are named using the following (similar) rules:

```
d2f + category + (gt_GetTypeProp or st_SetTypeProp)
```

Here, `category` is again the abbreviated object type, and `t` and `Type` are one of the following five storage types:

- `t Text`
- `n Number`
- `b Boolean`
- `p Blob`
- `o Object`

For example, for the "alert" category, the functions would be:

```
d2faltgb_GetBoolProp()
d2faltgn_GetNumProp()
d2faltgt_GetTextProp()
d2faltgo_GetObjProp()
d2faltgp_GetBlobProp()

d2faltsb_SetBoolProp()
d2faltsn_SetNumProp()
d2faltst_SetTextProp()
d2faltso_SetObjProp()
d2faltsp_SetBlobProp()
```

Remember, the `'alt'` part shows the object type, so it could be `'blk'` or `'cnv'` or any other object type. In addition, the category could be `'ob'` which is used when treating the object generically, as in:

```
d2fobgb_GetBoolProp()
d2fobgn_GetNumProp()
...etc...
```

The following table shows the complete list of object type abbreviations, along with the object type constants from the `d2fdef.h` header file:

Object Type	**Constant from `d2fdef.h`**	**Abbreviation**
Generic	`D2FFO_ANY`	`ob`
Alert	`D2FFO_ALERT`	`alt`
Attached library	`D2FFO_ATT_LIB`	`alb`
Block	`D2FFO_BLOCK`	`blk`
Canvas	`D2FFO_CANVAS`	`cnv`
Column value	`D2FFO_COLUMN_VALUE`	`rcv`
Coordinate info	`D2FFO_COORD`	`crd`
Data source argument	`D2FFO_DAT_SRC_ARG`	`dsa`
Data source column	`D2FFO_DAT_SRC_COL`	`dsc`
Editor	`D2FFO_EDITOR`	`edt`
Font	`D2FFO_FONT`	`fnt`
Form module	`D2FFO_FORM_MODULE`	`fmd`
Form parameter	`D2FFO_FORM_PARAM`	`fpm`
Graphic (boilerplate)	`D2FFO_GRAPHIC`	`gra`

Object Type	Constant from `d2fdef.h`	Abbreviation
Item	`D2FFO_ITEM`	`itm`
Library program unit	`D2FFO_LIB_PROG_UNIT`	`lpu`
Library module	`D2FFO_LIBRARY_MODULE`	`lib`
List of values (LOV)	`D2FFO_LOV`	`lov`
LOV column mapping	`D2FFO_LV_COLMAP`	`lcm`
Menu	`D2FFO_MENU`	`mnu`
Menu item	`D2FFO_MENU_ITEM`	`mni`
Menu module	`D2FFO_MENU_MODULE`	`mmd`
Menu parameter	`D2FFO_MENU_PARAM`	`mpm`
Object group	`D2FFO_OBJ_GROUP`	`obg`
Object group child	`D2FFO_OBG_CHILD`	`ogc`
Object library module	`D2FFO_OBJ_LIB`	`olb`
Object library tab	`D2FFO_OBJ_LIB_TAB`	`olt`
Program unit	`D2FFO_PROG_UNIT`	`pgu`
Property class	`D2FFO_PROP_CLASS`	`ppc`
Radio button	`D2FFO_RADIO_BUTTON`	`rdb`
Record group	`D2FFO_REC_GROUP`	`rcg`
Record group column spec	`D2FFO_RG_COLSPEC`	`rcs`
Relation	`D2FFO_RELATION`	`rel`
Report object	`D2FFO_REPORT`	`rpt`
Tab page	`D2FFO_TAB_PAGE`	`tbp`
Trigger	`D2FFO_TRIGGER`	`trg`
Visual attribute	`D2FFO_VIS_ATTR`	`vat`
Window	`D2FFO_WINDOW`	`win`

Macros

Similar naming rules also apply to the get and set macros we've been using all along. All the macros for getting and setting properties are named using the following pattern:

`d2f + `*`category`*` + (g_ `*or*` s_) + `*`property_type`*

Again, the *`category`* is the two to three character abbreviation of the object type (for example, `'alt'` for alerts, `'blk'` for blocks, `'cnv'` for canvases, and so on). The `'g_'` indicates a property-get, while the `'s_'` indicates a property-set. Finally, the *`property_type`* is the full name of the property, and importantly, is identical to the patterns used in the `D2FP_` macros (recall we mentioned above that the `D2FP_` name would be important later). So for example, there's a property type in `d2fdef.h` called `D2FP_FORE_COLOR` that applies to items among other object types. Knowing that, you would construct the name of the macro that sets this property on item objects as follows:

```
d2f + itm + s_ + fore_color   =   d2fitms_fore_color()
```

In fact, you will see this macro in the `d2fitm.h` header file.

EXAMPLE 2: USING OBJECT AND PROPERTIES

That's a lot of information, but it gives us enough to walk through another somewhat more complicated example. In this example, we create a form object, add a canvas object, and use the type-specific and generic macros and functions to get and set properties. We then create data block and item objects, and set some properties on those objects as well. The type-specific and generic Duplicate functions are called at the end of the program.

EXAMPLE 2 – OBJPROP.C

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>

#include <d2fctx.h>
#include <d2ferr.h>
#include <d2ffmd.h>
#include <d2fcnv.h>
#include <d2fblk.h>
#include <d2fitm.h>
#include <d2fob.h>

/*
** Example Forms API program
**
** This program creates a few objects, gets and sets properties
** using both the convenience macros and the underlying functions,
** both type-specific versions and the generic versions.  At the
** end, we duplicate a canvas using both the type-specific and the
** generic function.  We also save the module so you can load it
** up in the Form Builder and see what was created.
*/
main(int argc, char *argv[])
{
   d2fctxa       attr;
   d2fctx       *ctx = (d2fctx *)0;
   d2ffmd       *form = (d2ffmd *)0;
   d2fcnv       *cnv1;
   d2fcnv       *cnv2;
   d2fcnv       *cnv3;
   d2fblk       *blk1;
   d2fitm       *itm1;
   d2fotyp       otyp;
   number        width;
   text         *color;
   text         *name;

   /*
   ** Create the Forms API context
   */
   attr.mask_d2fctxa = 0;
   if ( d2fctxcr_Create(&ctx, &attr) != D2FS_SUCCESS )
   {
         fprintf(stderr, "Could not create the context\n");
         goto error;
   }
```

```
/*
** Create a new form module
*/
if ( d2ffmdcr_Create(ctx, &form, (text *)"MOD1") != D2FS_SUCCESS )
{
       fprintf(stderr, "Could not create module\n");
       goto error;
}

/*
** Create a canvas owned by form object, using the type-specific
** function
*/
if ( d2fcnvcr_Create(ctx, (d2fob *)form, &cnv1, (text *)"CNV1")
       != D2FS_SUCCESS )
{
       fprintf(stderr, "Could not create canvas\n");
       goto error;
}

/*
** Get the type of the object we just created.  Note that we
** can safely down-cast the canvas (d2fcnv *) to a generic object
** (d2fob *).
**
** The following section describes all the d2fob* functions in
** more detail.  This function simply gets the object type given
** an object pointer.
*/
if ( d2fobqt_QueryType(ctx, (d2fob *)cnv1, &otyp) != D2FS_SUCCESS )
{
       fprintf(stderr, "Could not query type\n");
       goto error;
}

/*
** Make sure it’s really a canvas
*/
if ( otyp == D2FFO_CANVAS )
       fprintf(stdout, "Yep, it’s a canvas!\n");
else
       fprintf(stderr, "This should never happen!\n");

/*
** Set the background color of the canvas using the type-specific
** convenience macro
*/
if ( d2fcnvs_back_color(ctx, cnv1, (text *)"blue") != D2FS_SUCCESS )
{
       fprintf(stderr, "Could not set canvas background\n");
       goto error;
}

/*
** Set the foreground color of the canvas using the GENERIC
** convenience macro
*/
if ( d2fobs_fore_color(ctx, (d2fob *)cnv1, (text *)"red")
       != D2FS_SUCCESS )
{
       fprintf(stderr, "Could not set canvas foreground\n");
       goto error;
}

/*
```

```
** Get the background color of the canvas without using the
** convenience macro
*/
if ( d2fcnvgt_GetTextProp(ctx, cnv1, D2FP_BACK_COLOR, &color)
        != D2FS_SUCCESS )
{
       fprintf(stderr, "Could not get canvas foreground\n");
       goto error;
}
printf ("canvas background color = %s\n", color);
free((dvoid *)color);

/*
** Get the foreground color of the canvas without using the
** convenience macro, in a generic way
*/
if ( d2fobgt_GetTextProp(ctx, (d2fob *)cnv1, D2FP_FORE_COLOR,
        &color) != D2FS_SUCCESS )
{
       fprintf(stderr, "Could not get canvas foreground\n");
       goto error;
}
printf ("canvas foreground color = %s\n", color);
free((dvoid *)color);

/*
** Create a data block object owned by form object
*/
if ( d2fblkcr_Create(ctx, (d2fob *)form, &blk1, (text *)"BLK1")
       != D2FS_SUCCESS )
{
       fprintf(stderr, "Could not create block\n");
       goto error;
}

/*
** Create an item object owned by block object
*/
if ( d2fitmcr_Create(ctx, (d2fob *)blk1, &itm1, (text *)"ITM1")
       != D2FS_SUCCESS )
{
       fprintf(stderr, "Could not create item\n");
       goto error;
}

/*
** Put the item on the canvas, and give it a reasonable size
*/
if ( d2fitms_cnv_obj(ctx, itm1, (d2fob *)cnv1) != D2FS_SUCCESS ||
     d2fitms_x_pos(ctx, itm1, 10) != D2FS_SUCCESS ||
     d2fitms_y_pos(ctx, itm1, 10) != D2FS_SUCCESS ||
     d2fitms_width(ctx, itm1, 100) != D2FS_SUCCESS ||
     d2fitms_height(ctx, itm1, 20) != D2FS_SUCCESS )
{
       fprintf(stderr, "Could not initialize item properties\n");
       goto error;
}

/*
** Get the background color of the item
*/
if ( d2fitmg_back_color(ctx, itm1, &color) != D2FS_SUCCESS )
{
       fprintf(stderr, "Could not get item foreground\n");
       goto error;
```

```
}
printf ("item foreground color = %s\n", color ? color : (text *)"NULL");
if ( color ) free((dvoid *)color);

/*
** Get the width of the item using the generic macro
*/
if ( d2fobg_width(ctx, (d2fob *)itm1, &width) != D2FS_SUCCESS )
{
      fprintf(stderr, "Could not get item width\n");
      goto error;
}
printf ("item width = %d\n", width);

/*
** Get the name of the item's canvas.  This property is a quick
** shortcut to getting the D2FP_CNV_OBJ property, and then ask
** the returned object for its name.
*/
if ( d2fitmg_cnv_nam(ctx, itm1, &name) != D2FS_SUCCESS )
{
      fprintf(stderr, "Could not get item's canvas name\n");
      goto error;
}
printf ("item's canvas name = %s\n", name);

/*
** ERROR: Try to get the item's background color using the
** canvas macro.  The object knows what type it is, which makes
** this an illegal operation.
**
** >>> This call is both a compile-time and run-time error.
*/
if ( d2fcnvg_back_color(ctx, itm1, &color) != D2FS_SUCCESS )
{
      printf("Expected error: can't treat an item as a canvas\n");
}

/*
** Duplicate the canvas object, first using the type-specific function
*/
if ( d2fcnvdu_Duplicate(ctx, (d2fob *)form, cnv1, &cnv2, (text *)"CNV2")
      != D2FS_SUCCESS )
{
      fprintf(stderr, "Could not duplicate canvas (2)\n");
      goto error;
}

/*
** Duplicate the canvas object again, now using the generic function
*/
if ( d2fobdu_Duplicate(ctx, (d2fob *)form, (d2fob *)cnv1,
      (d2fob **)&cnv3, (text *)"CNV3") != D2FS_SUCCESS )
{
      fprintf(stderr, "Could not duplicate canvas (3)\n");
      goto error;
}

/*
** Save the form to the filesystem.  Try loading the form in the
** Form Builder to see what we created!
*/
if ( d2ffmdsv_Save(ctx, form, (text *)"MOD1.fmb", FALSE)
      != D2FS_SUCCESS )
{
```

```
            fprintf(stderr, "Could not save the form\n");
            goto error;
    }

error:
    /*
    ** Destroy the form module object if it was created
    */
    if ( form )
    {
            (void) d2ffmdde_Destroy(ctx, form);
    }

    /*
    ** Destroy the context if it was created
    */
    if ( ctx )
    {
            (void) d2fctxde_Destroy(ctx);
    }

    return 0;
}
```

BASIC FORMS API FUNCTIONS

This section describes the most common functions used in Forms API programs. In this section, we will describe functions for:

- Form Object Manipulation Functions
- Object Metadata
- Enumerated Property Values
- Property Metadata
- Forms API Context

FORM OBJECT MANIPULATION FUNCTIONS

Take another look at the generic object header file `d2fob.h` that was mentioned above. In this file, prototypes for the following functions are defined:

```
d2fobcr_Create   (d2fctx *ctx, d2fob *owner, d2fob **ppd2fob, text *name,
                               d2fotyp objtyp)
d2fobde_Destroy  (d2fctx *ctx, d2fob *pd2fob)
d2fobdu_Duplicate(d2fctx *ctx, d2fob *new_owner, d2fob *pd2fob_src,
                               d2fob **ppd2fob_dst, text *new_name)
d2fobre_Replicate(d2fctx *ctx, d2fob *new_owner, d2fob *pd2fob_src,
                               d2fob **ppd2fob_dst, text *new_name)
d2fobmv_Move     (d2fctx *ctx, d2fob *pd2fob, d2fob *pd2fob_nxt)
d2fobqt_QueryType(d2fctx *ctx, d2fob *pd2fob, d2fotyp *objtyp)
```

```
d2fobfo_FindObj  (d2fctx *ctx, d2fob *owner, text *name, d2fotyp objtyp,
                               d2fob **ppd2fob)
```

Here we'll discuss these functions for manipulating Forms objects in detail.

The function **`d2fobcr_Create(ctx, owner, &newobj, name, objtyp)`** Creates and returns a new object of the specified type, with the specified owner object, and with the specified name. This operation is the same as creating a new object in the Form Builder's Object Navigator. For example, calling this function with:

```
d2fobcr_Create(ctx, pd2fblk, &pd2fitm, (text *)"NEWITEM", D2FFO_ITEM)
```

would create a new item object called "NEWITEM" owned by the specified block object. You could then proceed to set property values for the new object. As in the Form Builder development environment, all objects (except module objects) are *owned* by exactly one other object. For example, block objects are owned by form module objects, and item objects are in turn owned by block objects. Some objects, like triggers, can be owned by more than one type of object (triggers can be owned by modules, blocks, items, radio buttons, and property classes). Appendix A lists the full object hierarchy.

The function **`d2fobde_Destroy(ctx, pd2fob)`** destroys the specified object and all its properties. The object is detached from its owner object and ceases to exist. Calling this function has the same effect as deleting the object in the Object Navigator. Destroying a module object (a form module, menu module, object library, or PL/SQL library) is the same as closing the module in the Object Navigator.

The function **`d2fobdu_Duplicate(ctx, new_owner, pd2fob_src, &pd2fob_dst, new_name)`** duplicates the `pd2fob_src` object, creating a new object in `pd2fob_dst` with the given name and given owner. The new object is the same as the original object in that it has the same property values. However, the subclassing information is discarded, and the inherited property values "flattened" into local values in the new object. Contrast this with the `d2fobre_Replicate()` function which preserves the subclassing information and inheritance pointers. We'll talk more about subclassing in a later section, but the difference between duplicating and replicating is worth mentioning here.

Similar to the Duplicate function, **`d2fobre_Replicate(ctx, new_owner, pd2fob_src, &pd2fob_dst, new_name)`** creates a new object with a the given name and given owning object. The new object is an exact copy of the original object, with all the same property values and subclassing relations. This function returns `D2FS_FAIL` if the system cannot duplicate the entire object. Compare it to the d2fobdu_Duplicate() function which flattens the property values so that all property values become "local" to the new object.

The function **d2fobmv_Move(ctx, pd2fob, pd2fob_nxt)** reorders an object with respect to its siblings. This is similar to dragging and dropping the object in the Form Builder Object Navigator, except that this function requires that the object retain the same owner (in other words, the owner of pd2fob and pd2fob_nxt must be the same). Use a null pointer for pd2fob_nxt to move the object to the end of the list. If pd2fob and pd2fob_nxt do not share the same owner, or do not have the same type, the function returns with the status D2FS_WRONGPARENT or D2FS_WRONGOBJ.

To determine the type of an object, use the function **d2fobqt_QueryType(ctx, pd2fob, &objtyp)**. This function places into the objtyp parameter the type of the object, one of the D2FFO_ constants. Note that once an object is created, its type cannot be changed. Also, objects such as items and canvases have the concept of a "subtype" but this is merely an ordinary object property. For example, item objects have a property called D2FP_ITEM_TYPE that tells whether the item is a push button, text field, check box, etc. All those objects are still of type D2FFO_ITEM.

The function **d2fobfo_FindObj(ctx, owner, name, objtyp, &pd2fob)** can find an object by name given an owner and type. For example, making this function call:

```
d2fobfo_FindObj(ctx, pd2fblk, (text *)"NEWITEM", D2FFO_ITEM, &pd2fitm)
```

will find the item called "NEWITEM" belonging to the given block. The found item will be returned in the final parameter. If the object cannot be found, the function returns with status D2FS_OBJNOTFOUND.

Type-specific Functions

There are type-specific counterparts of all of these. For example, in d2falt.h, the following functions are defined:

```
d2faltcr_Create()
d2faltde_Destroy()
d2faltdu_Duplicate()
d2faltre_Replicate()
d2faltmv_Move()
d2faltqt_QueryType()
d2faltfo_FindObj()
```

The advantage of the type-specific versions is that they do both compile-time and run-time type checking. If an object of the wrong type is passed in at runtime, they return an error of type

`D2FS_TYPEMISMATCH`. We recommend you use the type-specific versions where possible. For example, if you're explicitly creating an LOV object, use `d2flovcr_Create()`.

OBJECT METADATA FUNCTIONS

On occasion, you will want to get information about an object *type* rather than about an object itself. In some sense, this is information about the information that describes an object. For that reason, we call this "object metadata". This section describes these functions that can be used to access the Forms API object metadata:

```
d2fobgcn_GetConstName (d2fctx *ctx, d2fotyp objtyp, text **objname)
d2fobgcv_GetConstValue(d2fctx *ctx, text *objname, d2fotyp *objtyp)
d2fobhp_HasProp       (d2fctx *ctx, d2fob * pd2fobj, ub2 pnum)
```

`The function` **`d2fobgcn_GetConstName(ctx, objtyp, &objname)`** translates from a `D2FFO_` object type to a human-readable string describing the object type. For example, calling:

```
text    *objname;
d2fobgcn_GetConstName(ctx, D2FFO_CANVAS, &objname)
```

would cause `objname` to point to a string containing "CANVAS". The returned string does not have to be freed, as it is a static string in a static internal table.

Inversely, the function **`d2fobgcv_GetConstValue(ctx, objname, &objtyp)`** translates from the a human readable string to the `D2FFO_` object type constant. For example, calling:

```
d2fotyp  objtyp;
d2fobgcv_GetConstValue(ctx, "CANVAS", &objtyp)
```

would set `objtyp` to `D2FFO_CANVAS`.

Finally, the function **`d2fobhp_HasProp(ctx, pd2fobj, proptyp)`** returns `D2FS_YES` or `D2FS_NO` depending on whether the specified property type 'applies' to the given object. In other words, it tells whether the property can be assigned to the given object. For example, the following line of code:

```
d2fobhp_HasProp(ctx, pd2falt, D2FP_ALT_STY)
```

returns `D2FS_YES` because alert objects have the Alert Style property, but the following line of code:

```
d2fobhp_HasProp(ctx, pd2falt, D2FP_DML_ARY_SIZ)
```

returns `D2FS_NO` because the alerts do **not** have the DML array size property.

ENUMERATED PROPERTY VALUES

In the Form Builder Property Palette, many properties are edited using a poplist that restricts the range of possible property values to a small enumerated list of values. For example, Form Builder users know that they can set the Alert Style property to Stop, Caution, or Note. We call these **enumerated** property values because these are the only legal property values for a particular property.

Internally, these properties are stored using ordinary numbers, just like conventional number-valued properties such as height, width, and number-of-records-displayed. As expected, the Forms API provides numerical constants that correspond to each of the enumerated values. These constants all begin with `D2FC_`, and are defined `d2fdef.h`. For example, the Alert Style constants are defined in `d2fdef.h` as follows:

```
/*
** Alert style (D2FP_ALT_STY)
** [ALT]
*/
#define D2FC_ALST_STOP            0
#define D2FC_ALST_CAUTION         1
#define D2FC_ALST_NOTE            2
```

You can see from the comment that these constants apply to the property `D2FP_ALT_STY`. In Form Builder, the Alert Style property displays in the Property Inspector using a poplist that shows exactly these three choices. When you call a function or macro that returns an Alert Style value, it will always be one of these three values. The following lines of code show how the enumerated constants are used:

```
d2fstatus  status;
number     alert_style;

/* Get the alert style and print a message */
status = d2faltg_alt_sty(ctx, alert_obj, &alert_style)
if ( status == D2FS_SUCCESS )
{
    printf ("The alert style is: ");

    switch ( alert_style )
    {
        case D2FC_ALST_STOP:
            printf("Stop\n ");
            break;
        case D2FC_ALST_CAUTION:
            printf("Caution\n ");
            break;
        case D2FC_ALST_NOTE:
            printf("Note\n ");
            break;
        default:
            printf("???\n ");
            break;
    }
}

/* Set the alert style to Caution */
status = d2falts_alt_sty(ctx, alert_obj, D2FC_ALST_CAUTION);
```

Below, we'll see a function called `d2fprgvn_GetValueName()` that converts between the `D2FC_` constants and the actual human-readable string used in the Property Palette.

PROPERTY METADATA FUNCTIONS

Just as there are functions that allow you to access the Forms API object metadata, there are functions for accessing the Forms API *property* metadata.

```
d2fprgt_GetType       (d2fctx *ctx, ub2 pnum)
d2fprgn_GetName       (d2fctx *ctx, ub2 pnum, text **pname)
d2fprgvn_GetValueName (d2fctx *ctx, ub2 pnum, number val, text **vname)
d2fprgcv_GetConstValue(d2fctx *ctx, text *pcname, ub2 *pnum)
d2fprgcn_GetConstName (d2fctx *ctx, ub2 pnum, text **pcname)
```

The function **`d2fprgt_GetType(ctx, pnum)`** returns the storage class of the property type. For example, `d2fprgt_GetType(ctx, D2FP_ALT_STY)` returns `D2FP_TYP_NUMBER` because the Alert Style property is stored as a number in the property list.

The function **`d2fprgn_GetName(ctx, pnum, &pname)`** translates from the property type to a human-readable string describing the property type. This is the same string that appears in the left-hand column of the Form Builder Property Palette. For example, `d2fprgn_GetName(d2fctx, D2FP_ALT_STY, &pname)` puts "Alert Style" in the `pname` parameter.

Next, the function **`d2fprgvn_GetValueName(ctx, pnum, val, &vname)`** returns a human-readable name for the enumerated `D2FC_` constant. This is the same string that appears in the poplists in the Form Builder Property Palette when editing an enumerated property. For example:

```
d2fprgvn_GetValueName(ctx, D2FP_ALT_STY, D2FC_ALST_CAUTION, &vname)
```

puts "Caution" in the `vname` parameter. This function is useful when printing out the properties of an object in human-readable form. In fact, we can use this function to simplify the code example above:

```
d2fstatus  status;
number     alert_style;

/* get the alert style and print a message */
status = d2faltg_alt_sty(ctx, alert_obj, &alert_style)
if ( status == D2FS_SUCCESS )
{
    text   *vname;
```

```
        /* Get the string corresponding to the alert style value */
        status = d2fprgvn_GetValueName(ctx, D2FP_ALT_STY, alert_style, &vname);
        if ( status == D2FS_SUCCESS )
            printf ("The alert style is: %s", vname);
    }

    /* set the alert style to Caution */
    status = d2falts_alt_sty(ctx, alert_obj, D2FC_ALST_CAUTION);
```

This new version is more readable and generic, and has fewer hard coded strings.

The final two functions are only really useful when doing advanced parsing of form module files. The first function, **`d2fprgcv_GetConstValue(ctx, pcname, &pnum)`**, converts the property type from the unique internal string descriptor to a proper property type. For example:

```
    d2fprgcv_GetConstValue(ctx, "ALT_STY", &pnum)
```

puts `D2FP_ALT_STY` in the `pnum` parameter. The inverse function, **`d2fprgcn_GetConstName (ctx, pnum, &pcname)`**, converts the property type from an ID to the unique internal string. Using the same example:

```
    d2fprgcn_GetConstName(ctx, D2FP_ALT_STY , &pname)
```

puts `"ALT_STY"` in the `pname` parameter.

FORMS API CONTEXT FUNCTIONS

The header file `d2fctx.h` defines several functions, which do not apply to specific objects, but rather defines something of a global context in which the Forms API executes. As we said earlier, all functions take the Forms API context as their first argument, so the Forms API context is the first object created, and the final object destroyed. The functions in `d2fctx.h` are:

```
    d2fctxcr_Create        (d2fctx **pctx, d2fctxa *ctx_attr)
    d2fctxde_Destroy       (d2fctx *ctx)
    d2fctxsa_SetAttributes (d2fctx *ctx, d2fctxa *ctx_attr)
    d2fctxga_GetAttributes (d2fctx *ctx, d2fctxa *ctx_attr)
    d2fctxcn_Connect       (d2fctx *ctx, text *con_str, dvoid *phstdef)
    d2fctxdc_Disconnect    (d2fctx *ctx)
    d2fctxbv_BuilderVersion(d2fctx *ctx, number *version)
    d2fctxcf_ConvertFile   (d2fctx *ctx, text *filename, d2fotyp modtyp,
                            number direction)
    d2fctxbi_BuiltIns      (d2fctx *ctx, text ****pparr)
```

As we saw in the very first example program, the Forms API context is created by calling **`d2fctxcr_Create(&ctx, &ctx_attr)`**. Typically, this is the first function call in a Forms API program. The second argument to this function is a pointer to an attribute structure. The first member of this structure is `mask_d2fctxa`, which must be initialized to either zero or a bitwise "or" of the `D2FCTXA*` constants. For each specified constant, the corresponding structure members must be filled in. For example, when the mask is set to D2FCTXACDATA, the field cdata_d2fctxa must point to a valid memory address. Refer to the d2fctx.h file for the set of context attribute options.

The final function call in a Forms API program is usually **d2fctxde_Destroy(ctx)**. This function destroys the Forms API context. After this call, no further Forms API calls are possible.

After the Forms API context has been created, you may still access and in some cases modify the context's attributes. To set attributes, use **d2fctxsa_SetAttributes(ctx, &ctx_attr)**. Remember to initialize the mask_d2fctxa structure member with the attributes actually being set. Note that not all Forms API attributes are settable (only those marked with 'S' in the header file are settable). To get the context's attributes, use **d2fctxga_GetAttributes(ctx, &ctx_attr)**. Remember to initialize the mask_d2fctxa structure member with the attributes you actually want to retrieve.

As in the Form Builder, a database connection is required for many operations, such as loading and saving forms in a database, parsing record group queries, accessing stored procedures, etc. To connect to a database, use the function **d2fctxcn_Connect(ctx, conn_string, phstdef)**. This establishes a database connection given a connect string (conn_string) of the usual form username/password@database. Alternatively, you may directly supply an Oracle 'hstdef' pointer for the connection, if you have connected to the Oracle database without using the Forms API. To disconnect from the database, use the function **d2fctxdc_Disconnect(ctx)**.

The function **d2fctxbv_BuilderVersion(ctx, &vernum)** returns the version of the Forms API currently running. The format of the version number is a decimal number of the form 12334455, where 1 is the first digit, 2 is the second digit, 33 is the third digit, 44 is the fourth digit, and 55 is the fifth digit. For example, a return value of 60052902 indicates version 6.0.5.29.2. This is the same version number shown in the Form Builder when choosing the menu option Help→About Form Builder.

The Form Builder allows you to convert files from binary .fmb files to ASCII .fmt files. The benefit of .fmt files is that they are sometimes more easily stored in source control systems. To do this using the Forms API, use the function **d2fctxcf_ConvertFile(ctx, filename, modtyp, direction)**, which converts an .fmb file to an .fmt file or vice-versa. The filename parameter is the name of the file on disk, modtyp is one of the module object types (for example, D2FFO_FORM_MODULE), and direction is either BINTOTEXT or TEXTTOBIN, as defined in d2fctx.h.

The final function defined in d2fctx.h is **d2fctxbi_BuiltIns(ctx, pparr)**. This utility function allocates and returns an array-of-an-array (that is, a table) of strings listing each of the PL/SQL Built-in functions, organized by package name. Each row of the table is an array of strings; the first string in each row is the package name and the remaining strings in that row are the PL/SQL Built-ins in that package. This routine allocates memory for each string, so it should only be called once in order to avoid memory leaks.

EXAMPLE 3: USING API CONTEXT

This demonstrates several metadata and forms context concepts, including routines for connecting and disconnecting from a database, and printing the version number. It creates a canvas and a block with an item, duplicates the canvas twice, then saves the form to the file system. It also gets and sets various properties using both the generic and type-specific macros.

EXAMPLE 3 – CONTEXT.C

```
#include <stdio.h>
#include <malloc.h>

#include <d2fctx.h>
#include <d2ferr.h>
#include <d2ffmd.h>
#include <d2fcnv.h>
#include <d2fblk.h>
#include <d2fitm.h>
#include <d2fob.h>

/*
** This program illustrates the various Forms Context functions.
*/
main(int argc, char *argv[])
{
   d2fctxa       attr;
   d2fctx       *ctx = (d2fctx *)0;
   d2ffmd       *form = (d2ffmd *)0;
   d2fcnv       *cnv1;
   d2fcnv       *cnv2;
   d2fcnv       *cnv3;
   d2fblk       *blk1;
   d2fitm       *itm1;
   d2fotyp       otyp;
   number        width;
   text         *color;
   text         *name;
   number        version;

   /*
   ** Check arguments – the user must supply a connection string
   */
   if ( argc != 2 )
   {
         fprintf(stderr, "USAGE: %s <username>/<password>[@database]\n",
                 argv[0]);
         goto error;
   }
```

```
/*
** Create the Forms API context
*/
attr.mask_d2fctxa = 0;
if ( d2fctxcr_Create(&ctx, &attr) != D2FS_SUCCESS )
{
      fprintf(stderr, "Could not create the context\n");
      goto error;
}

/*
** Connect to a database
*/
if ( d2fctxcn_Connect(ctx, ((text *)argv[1], (dvoid *)0 )
      != D2FS_SUCCESS )
{
      fprintf(stderr, "Could not connect to the database\n");
      goto error;
}

/*
** Get the version of the Forms API
*/
if ( d2fctxbv_BuilderVersion(ctx, &version) != D2FS_SUCCESS )
{
      fprintf(stderr, "Could not get the version number\n");
      goto error;
}

/*
** Print out the version of the Forms API
*/
printf ("Running Version %d.%d.%d.%d.%d (%d) of the Forms API\n",
          version/10000000 % 10, version/1000000 % 10,
          version/10000 % 100, version/100 %100,
          version % 100, version);

/*
** Create a new form module
*/
if ( d2ffmdcr_Create(ctx, &form, (text *)"MOD1") != D2FS_SUCCESS )
{
      fprintf(stderr, "Could not create module\n");
      goto error;
}

/*
** Create a canvas owned by form object, using the type-specific
** function
*/
if ( d2fcnvcr_Create(ctx, (d2fob *)form, &cnv1, (text *)"CNV1")
      != D2FS_SUCCESS )
{
      fprintf(stderr, "Could not create canvas\n");
      goto error;
}

/*
** Get the type of the object we just created.  Note that we
** can safely down-cast the canvas (d2fcnv *) to a generic object
** (d2fob *).
*/
if ( d2fobqt_QueryType(ctx, (d2fob *)cnv1, &otyp) != D2FS_SUCCESS )
{
        fprintf(stderr, "Could not query type\n");
        goto error;
}

/*
** Make sure it’s really a canvas
*/
if ( otyp == D2FFO_CANVAS )
        fprintf(stdout, "Yep, it’s a canvas!\n");
else
        fprintf(stderr, "This should never happen!\n");

/*
** Set the background color of the canvas using the type-specific
** convenience macro
*/
if ( d2fcnvs_back_color(ctx, cnv1, (text *)"blue") != D2FS_SUCCESS )
{
        fprintf(stderr, "Could not set canvas background\n");
        goto error;
}

/*
** Set the foreground color of the canvas using the GENERIC
** convenience macro
*/
if ( d2fobs_fore_color(ctx, (d2fob *)cnv1, (text *)"red")
        != D2FS_SUCCESS )
{
        fprintf(stderr, "Could not set canvas foreground\n");
        goto error;
}

/*
** Get the background color of the canvas without using the
** convenience macro
*/
if ( d2fcnvgt_GetTextProp(ctx, cnv1, D2FP_BACK_COLOR, &color)
         != D2FS_SUCCESS )
{
        fprintf(stderr, "Could not get canvas foreground\n");
        goto error;
}
printf ("canvas background color = %s\n", color);
free((dvoid *)color);

/*
** Get the foreground color of the canvas without using the
** convenience macro, in a generic way
*/
if ( d2fobgt_GetTextProp(ctx, (d2fob *)cnv1, D2FP_FORE_COLOR,
         &color) != D2FS_SUCCESS )
{
        fprintf(stderr, "Could not get canvas foreground\n");
        goto error;
}
printf ("canvas foreground color = %s\n", color);
free((dvoid *)color);

/*
** Create a data block object owned by form object
*/
if ( d2fblkcr_Create(ctx, (d2fob *)form, &blk1, (text *)"BLK1")
        != D2FS_SUCCESS )
{
        fprintf(stderr, "Could not create block\n");
        goto error;
}

/*
** Create an item object owned by block object
*/
if ( d2fitmcr_Create(ctx, (d2fob *)blk1, &itm1, (text *)"ITM1")
        != D2FS_SUCCESS )
{
        fprintf(stderr, "Could not create item\n");
        goto error;
}

/*
** Put the item on the canvas, and give it a reasonable size
*/
if ( d2fitms_cnv_obj(ctx, itm1, (d2fob *)cnv1) != D2FS_SUCCESS ||
     d2fitms_x_pos(ctx, itm1, 10) != D2FS_SUCCESS ||
     d2fitms_y_pos(ctx, itm1, 10) != D2FS_SUCCESS ||
     d2fitms_width(ctx, itm1, 100) != D2FS_SUCCESS ||
     d2fitms_height(ctx, itm1, 20) != D2FS_SUCCESS )
{
        fprintf(stderr, "Could not initialize item properties\n");
        goto error;
}

/*
** Get the background color of the item
*/
if ( d2fitmg_back_color(ctx, itm1, &color) != D2FS_SUCCESS )
{
        fprintf(stderr, "Could not get item foreground\n");
        goto error;
}
printf ("item foreground color = %s\n", color ? color : (text *)"NULL");
if ( color ) free((dvoid *)color);

/*
** Get the width of the item using the generic macro
*/
if ( d2fobg_width(ctx, (d2fob *)itm1, &width) != D2FS_SUCCESS )
{
        fprintf(stderr, "Could not get item width\n");
        goto error;
}
printf ("item width = %d\n", width);

/*
** Get the name of the item’s canvas.  This property is a quick
** shortcut to getting the D2FP_CNV_OBJ property, and then ask-
** the returned object for its name.
*/
if ( d2fitmg_cnv_nam(ctx, itm1, &name) != D2FS_SUCCESS )
{
        fprintf(stderr, "Could not get item’s canvas name\n");
        goto error;
}
printf ("item’s canvas name = %s\n", name);

/*
** ERROR: Try to get the item’s background color using the
** canvas macro.  The object knows what type it is, which makes
** this an illegal operation.
**
```

```
    ** >>> This call is both a compile-time and run-time error.
    */
    if ( d2fcnvg_back_color(ctx, itm1, &color) != D2FS_SUCCESS )
    {
          printf("Expected error: can’t treat an item as a canvas\n");
    }

    /*
    ** Duplicate the canvas object using the type-specific function
    */
    if ( d2fcnvdu_Duplicate(ctx, (d2fob *)form, cnv1, &cnv2, (text *)"CNV2")
          != D2FS_SUCCESS )
    {
          fprintf(stderr, "Could not duplicate canvas (2)\n");
          goto error;
    }

    /*
    ** Duplicate the canvas object using the generic function
    */
    if ( d2fobdu_Duplicate(ctx, (d2fob *)form, (d2fob *)cnv1,
          (d2fob **)&cnv3, (text *)"CNV3") != D2FS_SUCCESS )
    {
          fprintf(stderr, "Could not duplicate canvas (3)\n");
          goto error;
    }

    /*
    ** Save the form to the filesystem.  Try loading the form in the
    ** Form Builder to see what we created!
    */
    if ( d2ffmdsv_Save(ctx, form, (text *)"MOD1.fmb", FALSE)
          != D2FS_SUCCESS )
    {
          fprintf(stderr, "Could not save the form\n");
          goto error;
    }

error:
    /*
    ** Destroy the form module object if it was created
    */
    if ( form )
    {
          (void) d2ffmdde_Destroy(ctx, form);
    }

    /*
    ** Disconnect from the database, and destroy the context if it was
    ** created
    */
    if ( ctx )
    {
          (void) d2fctxdc_Disconnect(ctx);

          (void) d2fctxde_Destroy(ctx);
    }

    return 0;
}
```

EXAMPLE 4: WORKING WITH GENERIC OBJECTS

This example program loads a module from disk, then loops over every object in the module, printing the name and type of the object as well as the entire list of property values. It's essentially a very rudimentary version of the "object list report" in the Form Builder development environment. The example illustrates how to:

- Work with generic objects
- Use the object and property metadata functions
- Work with the object ownership hierarchy

EXAMPLE 4 – TRAVERSE.C

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>

#include <d2ferr.h>
#include <d2fctx.h>
#include <d2ffmd.h>
#include <d2fob.h>
#include <d2fpr.h>

int traverseObjects(d2fctx *ctx, d2fob *obj, int level);

/*
** This program takes a list of file
*/
int main(int argc, char *argv[])
{
    d2fctxa       attr;
    d2fctx       *ctx;    /* FAPI context */
    d2fob        *v_obj;  /* form module object */

    /*
    ** Check the arguments
    */
    if ( argc != 2 )
    {
          fprintf(stderr, "USAGE: %s <filename>\n", argv[0]);
          exit(1);
    }

    /*
    ** Create a context
    */
    attr.mask_d2fctxa = 0;
    if ( d2fctxcr_Create(&ctx, &attr) != D2FS_SUCCESS )
    {
          fprintf(stderr, "Could not create the context\n");
          exit(1);
    }

    /*
    ** load the form module
    */
    if ( d2ffmdld_Load( ctx, &v_obj, (text*)argv[1], FALSE) != D2FS_SUCCESS )
    {
          fprintf(stderr, "Could not load module %s\n", argv[1]);
          exit(1);
    }

    /*
    ** traverse all of the objects in the form module object
    */
    traverseObjects( ctx, v_obj, 0 );

    /*
    ** destroy the form module object
    */
    d2ffmdde_Destroy(ctx, v_obj);

    /*
    ** destroy the context
    */
    d2fctxde_Destroy(ctx);

    return 0;
}


/*
** This routine loops over each object and prints out its properties.
*/
int traverseObjects(d2fctx* ctx, d2fob* p_obj, int level)
{
    int         i;
    text       *v_name;
    text       *v_typ_name;
    d2fotyp     v_obj_typ;
    d2fpnum     prop_num;
    text       *v_prop_name;
    d2fob      *v_subobj;
    d2fob      *v_owner;

    /*
    ** This flag just ensures that Object properties are processed
    ** after number, boolean and text properties.
    */
    boolean     v_processing_objects = FALSE;

    /*
    ** Get object type.
    */
    if ( d2fobqt_QueryType(ctx, p_obj, &v_obj_typ) != D2FS_SUCCESS )
    {
          return (D2FS_FAIL);
    }

    /*
    ** If the object type is bogus, then this is something we're not
    ** supposed to be messing with, so we won't.
    */
    if ( v_obj_typ > D2FFO_MAX )
    {
          return (D2FS_SUCCESS);
    }

    /*
    ** Make sure object is named.  We shouldn't be messing with any
    ** object that doesn’t have a name.
    */
    if ( d2fobhp_HasProp(ctx, p_obj, D2FP_NAME) != D2FS_YES )
    {
          return D2FS_SUCCESS;
    }

    /*
    ** Print some leading spaces (indentation).
    */
    putchar(’\n’);
    for (i=0; i<3*level; i++) putchar(’ ’);

    /*
    ** Print the object’s name.
    */
    d2fobg_name(ctx, p_obj, &v_name);
    printf("OBJ NAME = %s", v_name ? v_name : (text *)"NULL");

    /*
    ** Print the object’s type.
    */
    if ( d2fobgcn_GetConstName(ctx, v_obj_typ, &v_typ_name) == D2FS_SUCCESS )
    {
          printf(" (TYPE = %s)", v_typ_name ? v_typ_name : (text *)"NULL");
    }
    printf("\n");

loop_begin:

    /*
    ** Walk through all properties.
    */
    for ( prop_num = D2FP_MIN; prop_num <= D2FP_MAX; prop_num++ )
    {
          /*
          ** If the object doesn’t have this property, or we can’t get
          ** the name of the property, or if the property has no name,
          ** then skip to the end of the loop.
          */
          if ( d2fobhp_HasProp(ctx, p_obj, prop_num) != D2FS_YES ||
               d2fprgn_GetName(ctx, prop_num, &v_prop_name)
                  != D2FS_SUCCESS || v_prop_name == (text *)0 )
          {
                continue;
          }

          /*
          ** Exclude these property types.  These properties
          ** we don’t want to print.
          */
          switch (prop_num)
          {
          case D2FP_FRST_NAVIGATION_BLK_OBJ :
          case D2FP_NXT_NAVIGATION_BLK_OBJ :
          case D2FP_PREV_NAVIGATION_BLK_OBJ :
          case D2FP_OBJ_GRP_CHILD_REAL_OBJ :
          case D2FP_OG_CHILD :
          case D2FP_SOURCE :
          case D2FP_DIRTY_INFO :
          case D2FP_ACCESS_KEY_STRID   :
          case D2FP_ALT_MSG_STRID  :
          case D2FP_BLK_DSCRP_STRID  :
          case D2FP_BTM_TTL_STRID  :
          case D2FP_BTN_1_LBL_STRID  :
```

```
case D2FP_BTN_2_LBL_STRID  :
case D2FP_BTN_3_LBL_STRID  :
case D2FP_FAIL_MSG_STRID :
case D2FP_FRAME_TTL_STRID :
case D2FP_HIGHEST_VAL_STRID :
case D2FP_HINT_STRID :
case D2FP_HLP_DSCRP_STRID :
case D2FP_INIT_VAL_STRID :
case D2FP_KBRD_ACC_STRID :
case D2FP_KBRD_HLP_TXT_STRID :
case D2FP_LABEL_STRID :
case D2FP_LOWEST_VAL_STRID :
case D2FP_MINIMIZE_TTL_STRID :
case D2FP_MNU_PARAM_INIT_VAL_STRID :
case D2FP_PARAM_INIT_VAL_STRID :
case D2FP_PRMPT_STRID :
case D2FP_SUB_TTL_STRID :
case D2FP_TEXT_STRID :
case D2FP_TITLE_STRID :
case D2FP_TOOLTIP_STRID :
case D2FP_PERSIST_CLIENT_INFO :
case D2FP_PAR_FLPATH :
case D2FP_PAR_MODSTR :
case D2FP_PAR_MODTYP :
case D2FP_PAR_MODULE :
case D2FP_PAR_NAM :
case D2FP_PAR_FLNAM :
case D2FP_PAR_SL1OBJ_NAM :
case D2FP_PAR_SL1OBJ_TYP :
case D2FP_PAR_SL2OBJ_NAM :
case D2FP_PAR_SL2OBJ_TYP :
case D2FP_PAR_TYP :
case D2FP_SUBCL_SUBOBJ :
case D2FP_SUBCL_OBJGRP :
       continue;

default:
       break;
}

/*
** Switch based on property storage TYPE.
*/
switch ( d2fprgt_GetType(ctx, prop_num) )
{
case D2FP_TYP_BOOLEAN:
       if ( !v_processing_objects )
       {
              boolean v_value;
              d2fobgb_GetBoolProp(ctx, p_obj, prop_num, &v_value);
              for (i=0; i<3*level; i++) putchar(' ');
              printf("%-30.30s = (b) %d\n", v_prop_name, v_value);
       }
       break;

case D2FP_TYP_NUMBER:
       if ( !v_processing_objects )
       {
              number v_value;
              text  *v_str;

              d2fobgn_GetNumProp(ctx, p_obj, prop_num, &v_value);
              for (i=0; i<3*level; i++) putchar(' ');
              printf("%-30.30s = (n) ", v_prop_name);
```

```
                /* If GetValueName()succeeds, print as a string */
                if ( d2fprgvn_GetValueName(ctx, prop_num, v_value,
                         &v_str) == D2FS_SUCCESS )
                        printf("%s\n", v_str);
                else
                        printf("%d\n", v_value);
        }
        break;

case D2FP_TYP_TEXT:
        if ( !v_processing_objects )
        {
                text* v_value;
                d2fobgt_GetTextProp(ctx, p_obj, prop_num, &v_value);
                for (i=0; i<3*level; i++) putchar(’ ’);
                printf("%-30.30s = (t) %s\n", v_prop_name,
                                v_value ? v_value : (text *)"NULL");
                if ( v_value )
                        free(v_value);
        }
        break;

case D2FP_TYP_OBJECT :
        /*
        ** Process object properties last (after number, text,
        ** etc.).
        */
        if (!v_processing_objects)
        {
                break;
        }

        /*
        ** Get the subobject pointed to by this property.
        */
        if ( d2fobgo_GetObjProp(ctx, p_obj, prop_num, &v_subobj)
                != D2FS_SUCCESS )
        {
                return ( D2FS_FAIL );
        }

        /*
        ** Continue if there are no subobjects.
        */
        if (v_subobj == (d2fob *)0)
        {
                break;
        }

        /*
        ** Get the owner of the subobject.
        */
        if ( d2fobg_owner(ctx, v_subobj, &v_owner) != D2FS_SUCCESS )
        {
                return (D2FS_FAIL);
        }

        /*
        * If the owner of the subobject isn’t the original object,
        * then it’s not really a subobject, so skip it.  This
        * keeps use from examining next, previous, source, the
        * canvas of an item, etc.
        */
        if (p_obj != v_owner)
        {
```

```
                     break;
                }

                /*
                ** Print the name of the object property
                */
                for (i=0; i<3*level; i++) putchar(' ');
                printf("%-30.30s = (object) ...\n", v_prop_name);

                /*
                ** Recursively process the subobject and all its siblings.
                */
                while (v_subobj != (d2fob *)0)
                {
                     traverseObjects(ctx, v_subobj, level + 1);

                     if ( d2fobg_next(ctx, v_subobj, &v_subobj)
                          != D2FS_SUCCESS)
                     {
                          return (D2FS_FAIL);
                     }
                }
                break;

           default:
                break;

           }
      }


      /*
      ** Go through the loop once more, this time looking for object-
      ** valued properties.
      */
      if ( !v_processing_objects )
      {
           v_processing_objects = TRUE;
           goto loop_begin;
      }

      return (D2FS_SUCCESS);
 }
```

WRITING ROBUST FORMS API PROGRAMS

In this section, we'll learn techniques for writing high quality, robust Forms API programs. Two of the most important ways to do this are discussed here:

- Handling Errors and Return Codes
- Managing Memory

HANDLING ERRORS AND RETURN CODES

Nearly all the Forms API functions return a status code of the following type:

```
d2fstatus
```

All the status codes are constants beginning with `D2FS_`. They are defined and documented in the file `d2ferr.h`. If a function successfully completes, it generally returns `D2FS_SUCCESS`. Otherwise, if a function fails for any reason, it returns `D2FS_FAIL` or one of the other more specific error codes, such as `D2FS_BADPROP` or `D2FS_NULLOBJ`. Some functions, such as `d2fobhp_HasProp()`, return either `D2FS_YES` or `D2FS_NO` when completing successfully, or one of the error messages if an error has occurred.

For example, if you want to retrieve the bevel of an item, you would write the following code:

```
d2fstatus  status;
number     bevel;

status = d2fitmg_bevel(ctx, itemobj, &bevel)
```

If `status` is `D2FS_SUCCESS`, then you can be assured that the `bevel` variable contains the bevel property value for the item object. If it returns something else, then you cannot assume that `bevel` refers to anything valid; in fact, it is probably uninitialized memory.

Note: we *strongly* recommended that you check the status codes returned by every Forms API call. Ignoring the status codes will result in hard-to-debug memory-related problems, as you may be referring to uninitialized memory or corrupted objects.

Managing Memory

By default, the Forms API uses the standard `malloc()`, `realloc()`, and `free()` functions to manage user memory. If desired, you can instruct the Forms API to use your own memory management routines. You can do this when creating the Forms API context by setting the `D2FCTXAMCALLS` attribute mask and providing function pointers for `malloc`, `realloc`, and `free` when calling `d2fctxcr_Create()`. The signature of the user-supplied functions is the same as `malloc/realloc/free`, except that the `d2fctx` (context) is passed in as an additional (first) argument. If your memory management routines need to access any non-global variables, you can use the “client data” field of the context to store a pointer to any arbitrary memory address. For example:

```
/* function prototypes */
void *mymalloc (d2fctx *ctx, size_t size);
void *myrealloc(d2fctx *ctx, void *ptr, size_t newsize);
void  myfree   (d2fctx *ctx, void *ptr);

/* main function */
main()
{
  void     clientptr;
  d2fctx  *ctx;
  d2fctxa  attr;

  /* set up some client-defined information */
  clientptr = InitMyApplication();

  /* set up the attribute mask for creating the context */
  attr.mask_d2fctxa = D2FCTXAMCALLS | D2FCTXACDATA;
  attr.cdata_d2fctxa = clientptr;
  attr.d2fmalc_d2fctxa = mymalloc;     /* type = d2fmalc */
  attr.d2fmfre_d2fctxa = myfree;       /* type = d2fmrlc */
  attr.d2fmrlc_d2fctxa = myrealloc;    /* type = d2fmfre */

  /* create the context */
  d2fctxcr_Create(&ctx, &attr);

  . . .
}

/* my malloc routine */
void *mymalloc (d2fctx *ctx, size_t size)
{
  d2fctxa  attr;

  /* get the client data from the FAPI context */
  attr.mask_d2fctxa = D2FCTXACDATA;
  if ( d2fctxga_GetAttributes(ctx, &attr) != D2FS_SUCCESS )
     return ( (void *)0 );

  /* call the REAL client malloc with client's own context handle */
  return ( MyRealMalloc(attr.cdata_d2fctxa, size);
}
```

In most cases, you will not want to provide your own routines, and only want to accept the default.

An important memory-related issue that you should be aware of is that when you "get" text-valued property values such as color name, label, or title, the string that returns is a *copy* of the string that is stored internally. When you are finished using the string, you need to `free()` it. If you do not free the string, you will cause a memory leak. Similarly, when "setting" a text-valued property, the API makes a copy of the string before storing it in the object.

By contrast, when "getting" blob-valued properties, the returned value is *not* a copy of the value stored internally; it is a pointer to the same piece of memory. These values should not be explicitly freed by the client; the Forms API will free the memory when the object is destroyed or the property value is changed or defaulted.

For number-valued, Boolean-valued, and object-valued properties, there are no memory issues because no memory is allocated when getting or setting these property types.

OBJECT SUBCLASSING USING THE FORMS API

In this section, we'll learn how to use object subclassing in the Forms API. This section contains two major sections:

- Basic subclassing
- Advanced subclassing

BASIC SUBCLASSING

First a little background on subclassing in Oracle Forms. Each Forms object has a list of properties, and each property can either be unspecified (that is, default) or specified (that is, has a local value). Almost every object can also have something called a "subclassing source" which is a reference to another object. If an object has a subclassing source, then the object will pick up ("inherit") values for *unspecified* properties from the subclassing source object. If the property is unspecified in the subclassing source, then Form Builder will look in *that* object's subclassing source (and so on) until it finds an object that does not have a subclassing source, at which point the system default value is returned. An object's subclassing source must be either another object of the same type, or a property class object.

In the Form Builder, you can set an object's subclassing source using the "Subclass Info" dialog. In the Forms API, a function is provided that allows you to set or remove an object's subclassing source. There are also functions for querying whether properties are specified, unspecified, inherited, etc.

Take another look at the generic object header file `d2fob.h`. In this file, prototypes for the following functions are defined:

```
d2fobsc_SubClass       (d2fctx *ctx, d2fob *pd2fob, d2fob *parent,
                                                    boolean keep_path)
d2fobis_IsSubclassed   (d2fctx *ctx, d2fob *pd2fob)
d2fobip_InheritProp    (d2fctx *ctx, d2fob *pd2fob, ub2 pnum)
d2fobii_IspropInherited(d2fctx *ctx, d2fob *pd2fob, ub2 pnum)
d2fobid_IspropDefault  (d2fctx *ctx, d2fob *pd2fob, ub2 pnum)
```

The function **`d2fobsc_SubClass(ctx, pd2fob, parent, keep_path)`** changes the subclassing source of an object to the specified `parent` object. This is the same as setting the Subclass Information property in the Form Builder's Property Palette. As in the Form Builder, an important side effect of this function is that local property values in the child object will be removed (that is, made unspecified, or defaulted) for all properties which are locally defined (that is, specified) on the parent object. The `keep_path` argument indicates whether the system should refer to the parent object's module by filename alone or by path+filename. The recommended choice is `FALSE` for this argument in most cases.

You can find out whether an object is subclassed by calling **`d2fobis_IsSubclassed(ctx, pd2fob)`**. This function returns `D2FS_YES` or `D2FS_NO` depending on whether the object has a subclassing source object from which it may inherit property values.

The function **`d2fobip_InheritProp(ctx, pd2fob, pnum)`** removes any local value for the specified property type `pnum`, putting the property value back to its default or "unspecified" state. This effectively forces the object to inherit the property value from its subclassing source, if it has one. If it does not have a subclassing source, then the property value is inherited from the system default value.

Use the function **`d2fobid_IspropDefault(ctx, pd2fob, pnum)`** to find out whether or not an object has a local (that is, "specified") value for the given property. If the object has a local value, then the function returns `D2FS_NO` since the property is *not* default-valued. If the object does not have a local value, meaning the object is either inheriting a value or picking up the system default value, then this function returns `D2FS_YES`.

The function **`d2fobii_IspropInherited(ctx, pd2fob, pnum)`** is similar to `d2fobid_IspropDefault()`, but slightly different. This function returns `D2FS_YES` or `D2FS_NO` depending on whether the given object is actually inheriting the specified property from a subclassing source object. It's different because it only returns `D2FS_YES` when the given object has no local value, and is inheriting a non-default value from its subclassing source. By contrast, `d2fobid_IspropDefault()` will always return `D2FS_YES` when the object has no local value, whether or not the object is inheriting from another object.

ADVANCED SUBCLASSING

This section describes a few advanced topics related to object subclassing.

Three special Forms properties have been defined which allow you to directly access and manipulate the internal bits of the subclassing information structure:

- `D2FP_PAR_FLNAM`
- `D2FP_PAR_FLPATH`
- `D2FP_PAR_MODSTR`

Together, these three properties uniquely identify a specific file either in the database or on the filesystem. The identified file is where the object's subclassing source object lives. If the name and path are null, that implies that the subclassing source is in the same module as the child (that is, a local reference).

The `D2FP_PAR_FLNAM` property of an object contains the filename of the module where the subclassing source resides. If the subclassing source is in the same module, then a null string is returned. The `D2FP_PAR_FLPATH` property of an object contains the pathname of the module where the subclassing source can be found. If the subclassing source is in the same module, then a null string is returned. Lastly, the `D2FP_PAR_MODSTR` property of an object tells whether the object's subclassing source is stored either on the filesystem or in the database.

Once the subclassing source object's module file is identified, up to four pairs of additional properties identify the actual source object within that file. These properties are:

- `D2FP_PAR_MODULE,      D2FP_PAR_MODTYP`
- `D2FP_PAR_NAM,         D2FP_PAR_TYP`
- `D2FP_PAR_SL1OBJ_NAM, D2FP_PAR_SL1OBJ_TYP`
- `D2FP_PAR_SL2OBJ_NAM, D2FP_PAR_SL2OBJ_TYP`

Each pair of properties represents a "source level." A source level is an object name and type, with each level becoming more specific, and lower in the object hierarchy. Together, the source levels pinpoint a precise object in the source module. For example, an item's subclassing information consists of three levels: the first (module) level might be a module object called `MODULE1`, the second level could be a block called `BLOCK2`, and the third level could be an item called `ITEM3`—thus identifying a specific item source object. Note that objects directly owned by the module only require two levels, while low-level objects such as radio buttons and some triggers require as many as four levels.

The `D2FP_PAR_MODULE` and `D2FP_PAR_MODTYP` properties identify the module name and type. The `D2FP_PAR_NAM` and `D2FP_PAR_TYP` properties identify the next level (a child of the module), and the `D2FP_PAR_SL1OBJ_NAM` and `D2FP_PAR_SL1OBJ_TYP` properties identify a child of the child of the module. Finally, the `D2FP_PAR_SL2OBJ_NAM` and `D2FP_PAR_SL2OBJ_TYP` properties identify a third-level descendant object relative to the module.

Each of these properties is both gettable and settable. If you use these low-level functions to *set* the subclassing source of an object (as opposed to calling the more convenient `d2fobsc_SubClass()`), you must call:

```
d2fobra_Reattach(d2fctx *ctx, d2fob *pd2fob)
```

to adjust the given object's actual source pointer to reflect changes made to the individual internal subclassing properties. This is intended for hooking up subclassing relations after the source information has been directly modified using one of the above low-level functions.

EXAMPLE 5: USING SUBCLASSING

This example demonstrates subclassing concepts. In the code below, we create a form containing two canvas objects. The background color is set on the first canvas. Then, the second canvas is subclassed from the first canvas, thereby inheriting its color property. The example demonstrates how property values are propagated from a subclassing source object, and how property values may be overridden with local values. Lastly, we see how the `D2FP_PAR_MODULE` property contains the name of the module where an object's subclassing source object resides.

EXAMPLE 5 – SUBCLASSING.C

```
#include <stdio.h>
#include <malloc.h>

#include <d2fctx.h>
#include <d2ferr.h>
#include <d2ffmd.h>
#include <d2fcnv.h>
#include <d2fblk.h>
#include <d2fitm.h>
#include <d2fob.h>

/*
** This program illustrates the use of subclasing in the Forms API
*/
main(int argc, char *argv[])
{
   d2fctxa       attr;
   d2fctx       *ctx = (d2fctx *)0;
   d2ffmd       *form = (d2ffmd *)0;
   d2fcnv       *cnv1;
   d2fcnv       *cnv2;
   text         *color;
   text         *parmod;

   /*
   ** Create the Forms API context
   */
   attr.mask_d2fctxa = 0;
   if ( d2fctxcr_Create(&ctx, &attr) != D2FS_SUCCESS )
   {
         fprintf(stderr, "Could not create the context\n");
         goto error;
   }

   /*
   ** Create a new form module
   */
```

```
if ( d2ffmdcr_Create(ctx, &form, (text *)"MOD1") != D2FS_SUCCESS )
{
      fprintf(stderr, "Could not create module\n");
      goto error;
}

/*
** Create a canvas owned by form object
*/
if ( d2fcnvcr_Create(ctx, (d2fob *)form, &cnv1, (text *)"CNV1")
      != D2FS_SUCCESS )
{
      fprintf(stderr, "Could not create canvas 1\n");
      goto error;
}

/*
** Set the background color of the canvas
*/
if ( d2fcnvs_back_color(ctx, cnv1, (text *)"blue") != D2FS_SUCCESS )
{
      fprintf(stderr, "Could not set canvas background\n");
      goto error;
}

/*
** Create a another canvas owned by form object
*/
if ( d2fcnvcr_Create(ctx, (d2fob *)form, &cnv2, (text *)"CNV2")
      != D2FS_SUCCESS )
{
      fprintf(stderr, "Could not create canvas 2\n");
      goto error;
}

/*
** Make cnv1 the subclassing source of cnv2
*/
if ( d2fobsc_SubClass(ctx, cnv2, cnv1, FALSE) != D2FS_SUCCESS )
{
      fprintf(stderr, "Could not subclass canvas\n");
      goto error;
}

/*
** Find out whether cnv2 is now subclassed (it should be!)
*/
if ( d2fobis_IsSubclassed(ctx, cnv2) == D2FS_YES )
      printf("Cnv2 is subclassed (correct)\n");
else
      printf("Cnv2 is NOT subclassed (error)\n");


/*
** Is the background color being inherited?
*/
if ( d2fobii_IspropInherited(ctx, cnv2, D2FP_BACK_COLOR) == D2FS_YES )
      printf("Cnv2 background color is inherited (correct)\n");
else
      printf("Cnv2 background color is NOT inherited (error)\n");

/*
** Get the background color of cnv2 (it should be blue since
** the property is being inherited)
*/
```

```
if ( d2fcnvg_back_color(ctx, cnv2, &color) != D2FS_SUCCESS )
{
       fprintf(stderr, "Could not get background color\n");
       goto error;
}
printf ("Cnv2 color = %s (blue)\n", color ? color : (text *)"NULL");
if ( color ) free((dvoid *)color);

/*
** Set the background color of cnv1
*/
if ( d2fcnvs_back_color(ctx, cnv1, (text *)"red") != D2FS_SUCCESS )
{
       fprintf(stderr, "Could not set canvas background\n");
       goto error;
}

/*
** Get the background color of cnv2 (should be red this time)
*/
if ( d2fcnvg_back_color(ctx, cnv2, &color) != D2FS_SUCCESS )
{
       fprintf(stderr, "Could not get background color\n");
       goto error;
}
printf ("Cnv2 color = %s (red)\n", color ? color : (text *)"NULL");
if ( color ) free((dvoid *)color);

/*
** Override cnv2 color with "green"
*/
if ( d2fcnvs_back_color(ctx, cnv2, (text *)"green") != D2FS_SUCCESS )
{
       fprintf(stderr, "Could not set canvas background\n");
       goto error;
}

/*
** Get the background color of cnv2 (should be green this time)
*/
if ( d2fcnvg_back_color(ctx, cnv2, &color) != D2FS_SUCCESS )
{
       fprintf(stderr, "Could not get background color\n");
       goto error;
}
printf ("Cnv2 color = %s\n", color ? color : (text *)"NULL");
if ( color ) free((dvoid *)color);

/*
** Is the background color being inherited?  Since we're overriding
** the property, it won't be inherited this time.
*/
if ( d2fobii_IspropInherited(ctx, cnv2, D2FP_BACK_COLOR) == D2FS_NO )
       printf("Cnv2 background color is NOT inherited (correct)\n");
else
       printf("Cnv2 background color is inherited (error)\n");

/*
** Re-inherit the property from cnv1
*/
if ( d2fobip_InheritProp(ctx, cnv2, D2FP_BACK_COLOR) != D2FS_SUCCESS )
{
       fprintf(stderr, "Could not get reinherit property\n");
       goto error;
}
```

```
    /*
    ** Lastly, get the PAR_MODULE property to print the module
    ** where cnv2's subclassing source can be found.
    */
    if ( d2fobg_par_module(ctx, cnv2, &parmod) != D2FS_SUCCESS )
    {
          fprintf(stderr, "Could not get parent module\n");
          goto error;
    }
    printf ("Parent Module = %s\n", parmod ? parmod : (text *)"NULL");
    if ( parmod ) free((dvoid *)parmod);


error:
    /*
    ** Destroy the form module object if it was created
    */
    if ( form )
    {
          (void) d2ffmdde_Destroy(ctx, form);
    }

    /*
    ** Disconnect from the database, and destroy the context if it was
    ** created
    */
    if ( ctx )
    {
          (void) d2fctxde_Destroy(ctx);
    }

    return 0;
}
```

TYPE-SPECIFIC FUNCTIONS

All the functions we've seen so far have applied to objects of nearly all types. In addition, all the properties we've seen so far have been simple in the sense that they are standard text-valued properties, number-valued properties, etc. In this section, we'll look at some functions that apply *only* to specific object types. They're needed in order to deal with complex properties, and also to implement behavior in Form Builder that doesn't directly involve getting and setting ordinary properties.

For example, only module objects can be loaded and saved, and only PL/SQL libraries can be attached and detached. Other objects have complex properties that require special functions to deal with them. In the Form Builder, these properties are edited using special dialogs; in the Forms API, they are edited using these special functions. Some examples are: a list item's "list elements" property, an image graphic's image data, and a menu module's list of menu roles. Here we'll describe special functions for:

- File operations

- Object Libraries
- PL/SQL Libraries
- List Item objects
- Font objects
- Image objects
- Record group objects
- Relation objects
- Menu role properties

FILE OPERATIONS

As we have seen, form, menu, and object library module objects (which correlate to `.fmb`, `.mmb`, and `.olb` files, respectively) are each represented by a normal object (`d2ffmd`, `d2fmmd`, and `d2folb`) with a standard set of properties. In addition to the standard object functions, the following functions apply to these types of objects:

```
d2fxxxld_Load       (ctx, d2fob **pd2fxxx, text *filename, boolean db)
d2fxxxsv_Save       (ctx, d2fob *pd2fxxx, text *filename, boolean db)
d2fxxxfv_FileVersion(ctx, text *filename, boolean db, &version)
d2fxxxdl_Delete     (ctx, text *filename, boolean db)
```

where *xxx* is `fmd`, `mmd`, or `olb`. Getting the file version does not actually load the form into memory, and no module object is created. As in the very first example, the `db` parameter indicates whether to access the module from either the file system or the database, and `filename` indicates the file name. You can only Delete a file from the database.

In addition, form and menu modules can be compiled into their executable equivalents:

```
d2fxxxcf_CompileFile(ctx, pd2fxxx)
```

This creates an `.fmx` or `.mmx` file on disk. Object libraries cannot be compiled.

PL/SQL library modules can be loaded using `d2flibld_Load()`, but they cannot be saved or compiled, and no interface exists for determining the file version. Therefore, PL/SQL libraries are read-only in the Forms API.

OBJECT LIBRARIES

In Form Builder, Object Libraries store and organize objects to be reused in various other form or menu modules. The Forms API provides a set of functions that allow you to build and modify object libraries:

```
d2folbao_AddObj      (d2fctx *ctx, d2folb *pd2folb, d2folt *pd2folt,
                      d2fob *pd2fob, d2fob **ppd2fob, boolean replace )
d2folbro_RemoveObj   (d2fctx *ctx, d2folb *po2olb, d2fob *pd2fob )
d2folbf2_Findobjbypos(d2fctx *ctx, d2folb *pd2folb, number pos,
                      d2fob  **ppd2fob)
d2folbsd_SetDesc     (d2fctx *ctx, d2folb *pd2folb, d2fob *pd2fob,
                      text    *desc)
d2folbgd_GetDesc     (d2fctx *ctx, d2folb *pd2folb, d2fob *pd2fob,
                      text  **desc)
d2folbot_ObjTabname  (d2fctx *ctx, d2folb *pd2folb, d2fob *pd2fob,
                      text  **tname)
```

Objects can be added to an object library using **`d2folbao_AddObj(ctx, pd2folb, pd2folt, pd2fob, &ppd2fob, replace)`**. This places a *copy* of an object into the specified object library, on the specified object library tab page. If an object with the same name and type already exists, it will be replaced if `replace` is `TRUE`, or else the function will return with `D2FS_OBJNOTUNIQUE` if `replace` is `FALSE`. If the function succeeds, the new copy of the object will be returned into `ppd2fob`.

The function **`d2folbro_RemoveObj(ctx, pd2folb, pd2fob)`** removes an object from an object library.

The function **`d2folbf2_Findobjbypos(ctx, pd2folb, index, &ppd2fob)`** returns the object (into `ppd2fob`) in the object library identified by `index`. The object library tab page is ignored here. This is a convenient way to iterate through all the objects in an object library.

Similarly, **`d2foltf2_Findobjbypos(ctx, pd2folt, index, &pret_obj)`** returns the object from a particular tab page of the object library identified by `index`. This is a convenient way to iterate through all the objects on a specific tab page of an object library

The Forms API function **`d2folbss_SetSmartclass(ctx, pd2folb, pd2fob, state)`** marks or unmarks the specified object as a smartclassed object. Smartclassing is a feature of the Form Builder development environment. A smartclassed object is identified by a special symbol in the Object Library editor. Smartclassed objects appear in the "smart class" popup submenu in the rest of the Form Builder; and they provide a quick way to set the subclassing source of an object in the Form Builder.

The function **`d2folbis_IsSmartclassed(ctx, pd2folb, pd2fob)`** returns `D2FS_YES` or `D2FS_NO` depending if the specified object is marked as a smartclassed object.

The functions **`d2folbgd_GetDesc(ctx, pd2folb, pd2fob, &desc)`** and **`d2folbsd_SetDesc (ctx, pd2folb, pd2fob, desc)`** get and set the description string associated with a particular object library object. This description appears in Form Builder in a special pane of the object library editor. It is associated with the object library object itself.

Lastly, the function **`d2folbot_ObjTabname(ctx, pd2folb, pd2fob, &tabname)`** returns the name of the object library *tab* on which the given object library object `pd2fob` appears.

PL/SQL LIBRARIES

You can attach and detach PL/SQL libraries to and from Form or Menu modules using the following functions:

```
d2falbat_Attach(ctx, parent, &ppd2falb, db, name)
d2falbdt_Detach(ctx, pd2falb)
```

For the "attach" function, the parameter `db` indicates where the library is stored (either in the database or on the filesystem). The `name` parameter indicates the file name (either in the database or on the filesystem). If the library is successfully attached, the `ppd2falb` OUT parameter will be populated.

The "detach" function simply detaches a given attached library from a Form or Menu module.

LIST ITEM OBJECTS

In the Form Builder, item objects that have an item type of "list" (that is, `D2FC_ITTY_LS`) have a property called "Elements in List". The list elements are edited using a special dialog that lets you add, remove, and modify the individual list elements. In the Forms API, three functions are provided which allow you to similarly access list elements:

```
d2fitmile_InsertListElem(d2fctx *ctx, d2fitm *pd2fitm, number index,
                         text *label, text *value)
d2fitmdle_DeleteListElem(d2fctx *ctx, d2fitm *pd2fitm, number index)
d2fitmgle_GetListElem   (d2fctx *ctx, d2fitm *pd2fitm, number index,
                         text **label, text **value)
```

For all functions, the `pd2fitm` item argument must be of type `D2FC_ITTY_LS`; otherwise, the function returns an error. As in the Form Builder, each list element has both a label and a value. List elements are identified by the parameter `index`; the first list element has an index of 1. If an invalid index is passed, the functions return a `D2FS_INVALIDINDEX` error.

The function `d2fitmile_InsertListElem()` creates a new list element at position `index`, shifting subsequent list elements down. It requires both a label and a value. The function `d2fitmdle_DeleteListElem()` deletes the list element identified by `index`, shifting the remaining list elements back up. Finally, the function `d2fitmgle_GetListElem()` returns the label and value for the list element identified by `index`.

FONT OBJECTS

A font object (`d2ffnt`) is a normal object (`d2fob`) with a normal set of properties. The object, however, does not appear in the Form Builder. It is a Forms API object that represents an abstraction of a collection of individual font-related properties.

Two special functions are associated with the font object:

```
d2ffntex_Extract(ctx, pd2ffnt, pd2fob, vat_typ)
d2ffntap_Apply(ctx, pd2ffnt, pd2fob, vat_typ)
```

The `d2ffntex_Extract()` function builds a font object out of the font-related properties of another object, such as an item object. The `vat_typ` argument is one of the `D2FC_VATY_` constants. Importantly, the font object must already have been created. After calling `d2ffntex_Extract()`, you may manipulate the font object without changing the font properties of the object from which it was "extracted". Typically, this function is used in conjunction with the `d2ffntap_Apply()` function; you can extract a font object from one object and apply it to another object, in order to force two objects to have the same font. Note that the `pd2fob` object must have the correct font-related properties; otherwise the function will return `D2FS_DONTHAVE`. If some of the font properties in `pd2fob` are default, then those same properties will be made default in the font object.

The `d2ffntap_Apply()` function "applies" the contents of the font object to the specific object. This is a shortcut to setting the individual font properties one-by-one. The properties of the `pd2ffnt` font object passed in can be populated either by calling `d2ffntex_Extract()` on some other object, or by manually setting the font-related properties of the font object itself.

IMAGE OBJECTS

In the Form Builder layout editor, background graphics (boilerplate) images are created by importing a file from the filesystem. To support this in the Forms API, a function is called as follows:

```
d2fgraim_importImage(d2fctx *ctx, d2fgra *pd2fgra, text *filename, format)
```

This functions allows you to import an image file into a graphics image object. The graphics object `pd2fgra` must already have been created, and must have a graphics type `D2FC_GRTY_IMAGE`. The `format` parameter is a `D2FC_IMFM_` constant. This allows you to import images from a variety of file formats, including BMP, GIF, JPEG, and so on.

RECORD GROUP OBJECTS

The query property for record groups is normally set using the `d2frcgs_rec_grp_qry()` macro. When this is called, the query is parsed using the SQL parser, and record group column specification objects are created for each column in the query. In some cases, you may not want to parse the SQL and create the column spec objects. In this case, you can call:

```
d2frcgs_qry_noparse(d2fctx *ctx, d2frcg *pd2frcg, text *query)
```

This special function sets the query property without parsing or creating the column spec objects.

RELATION OBJECTS

Relation objects define master/detail relationships between two blocks. The relation object itself defines the join between the two blocks, as well as various runtime behaviors, but the execution logic that coordinates the master and detail blocks is embedded in a number of triggers and program units that are automatically created when a relation is built using the Form Builder.

After setting specific properties on relations using the Forms API (for example, relation name, detail block name, join condition, delete style), you may choose to regenerate the triggers that drive the relation at runtime. This happens automatically when changing these properties in the Form Builder, so it is likely that the Forms API developer will want to mimic that behavior. You may also want to update the relation after creating a new relation, and setting its properties.

The function for "updating" the relation is:

```
d2frelup_Update(d2fctx *ctx, d2frel *pd2frel).
```

All rules in the Form Builder regarding user-modifiable sections of the generated triggers also apply to the Forms API. These rules are described in the online help.

MENU ROLE PROPERTIES

In the Form Builder, menu module objects have a property called Module Roles, and menu items have a property called Item Roles. Both are edited using a special dialog invoked from the Property Palette. In the Module Roles dialog, you can add, remove, and edit menu roles. In the Item Roles dialog, you can associate one or more of those roles with a particular menu item.

In the Forms API, several functions allow you to similarly manipulate menu roles. The functions for defining the Module Roles are as follows:

```
d2fmmdar_AddRole(ctx, pd2fmmd, index, role_name);
d2fmmdrr_RemoveRole(ctx, pd2fmmd, index);
d2fmmdgr_GetRole(ctx, pd2fmmd, index, &role_name);
```

The last function allows you to iterate through all the roles. You may use the macro `d2fmmdg_role_count()` to find the total number of roles in the menu module.

The functions for associating a role with a menu item are:

```
d2fmniar_AddRole(ctx, pd2fmni, index, role_name);
d2fmnirr_RemoveRole(ctx, pd2fmni, index);
d2fmnigr_GetRole(ctx, pd2fmni, index, &role_name);
```

As before, `d2fmnigr_GetRole()` allows you to iterate through all the roles attached to the menu item. The macro `d2fmnig_role_count()` can be used to find the total number of roles associated with the menu item.

THREE CORRECTED FORMS API EXAMPLES

The three examples in the Form Builder online help (topic "About the Open API") contained errors. The programs have been corrected, tested, and are presented below.

CORRECTED EXAMPLE 1 – USING SUBCLASSING

```
/*
** This example determines if the Form Builder object is a subclassed object
** and sets the file path of the parent to NULL if the object is subclassed.
** This sample only processes the following object types:  form-level
** triggers, alerts, blocks, items, item-level triggers, radio buttons, and
** block-level triggers. Use a similar method to process other object types.
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <d2fctx.h>
#include <d2ffmd.h>
```

```
#include <d2fblk.h>
#include <d2fitm.h>
#include <d2falt.h>
#include <d2ftrg.h>
#include <d2frdb.h>

int main(int argc, char *argv[])
{
    d2fctx   *v_ctx;
    d2fctxa   v_ctx_attr;
    d2ffmd   *v_fmd;
    d2fblk   *v_blk;
    d2fitm   *v_itm;
    d2falt   *v_alt;
    d2ftrg   *v_trg;
    d2frdb   *v_rdb;
    text     *v_form_name;
    d2fstatus status;

    /* Get the form name from the command line. */
    v_form_name = (text *)argv[1];

    /* Initialize the attribute mask and create the Forms API context */
    v_ctx_attr.mask_d2fctxa = 0;
    status = d2fctxcr_Create(&v_ctx, &v_ctx_attr);

    /* Load the form */
    status = d2ffmdld_Load(v_ctx, &v_fmd, v_form_name, FALSE) ;
    if ( status == D2FS_SUCCESS )
    {
        /* Process Form Level Trigger Objects */
        for ( status = d2ffmdg_trigger(v_ctx,v_fmd,&v_trg);
              v_trg != NULL;
              status = d2ftrgg_next(v_ctx,v_trg,&v_trg) )
        {
            if ( d2ftrgis_IsSubclassed(v_ctx,v_trg) == D2FS_YES )
            {
                d2ftrgs_par_flpath(v_ctx,v_trg,NULL);
            }
        }

        /* Process Alert Objects */
        for ( status = d2ffmdg_alert(v_ctx,v_fmd,&v_alt);
              v_alt != NULL;
              status = d2faltg_next(v_ctx,v_alt,&v_alt) )
        {
            if ( d2faltis_IsSubclassed(v_ctx,v_alt) == D2FS_YES )
            {
                d2falts_par_flpath(v_ctx,v_alt,NULL);
            }
        }

        /* Process Block Objects */
        for ( status = d2ffmdg_block(v_ctx,v_fmd,&v_blk);
              v_blk != NULL;
              status = d2fblkg_next(v_ctx,v_blk,&v_blk) )
        {
            if ( d2fblkis_IsSubclassed(v_ctx,v_blk) == D2FS_YES )
            {
                d2fblks_par_flpath(v_ctx,v_blk,NULL);
            }

            /* Process Item Objects */
            for ( status = d2fblkg_item(v_ctx,v_blk,&v_itm);
                  v_itm != NULL;
                status = d2fitmg_next(v_ctx,v_itm,&v_itm) )
            {
                if ( d2fitmis_IsSubclassed(v_ctx,v_itm) == D2FS_YES )
                {
                    d2fitms_par_flpath(v_ctx,v_itm,NULL);
                }

                /* Process Item Level Trigger Objects */
                for ( status = d2fitmg_trigger(v_ctx,v_itm,&v_trg);
                      v_trg != NULL;
                      status = d2ftrgg_next(v_ctx,v_trg,&v_trg) )
                {
                    if ( d2ftrgis_IsSubclassed(v_ctx,v_trg)==D2FS_YES )
                    {
                        d2ftrgs_par_flpath(v_ctx,v_trg,NULL);
                        printf("item trigger is Subclassed\n");
                    }
                    else if ( d2ftrgis_IsSubclassed(v_ctx, v_trg)==D2FS_NO )
                    {
                        printf("item trigger is NOT Subclassed\n");
                    }
                }

                /* Process Radio Button Objects */
                for ( status = d2fitmg_rad_but(v_ctx,v_itm,&v_rdb);
                      v_rdb != NULL;
                      status = d2frdbg_next(v_ctx,v_rdb,&v_rdb) )
                {
                    if ( d2frdbis_IsSubclassed(v_ctx,v_rdb) == D2FS_YES )
                    {
                        d2frdbs_par_flpath(v_ctx,v_rdb,NULL);
                        printf("radio button is Subclassed\n");
                    }
                    else if ( d2frdbis_IsSubclassed(v_ctx, v_rdb)==D2FS_NO )
                    {
                        printf("radio button is NOT Subclassed\n");
                    }
                }
            }

            /* Process Block Level Trigger Objects */
            for ( status = d2fblkg_trigger(v_ctx,v_blk,&v_trg);
                  v_trg != NULL;
                  status = d2ftrgg_next(v_ctx,v_trg,&v_trg) )
            {
                if ( d2ftrgis_IsSubclassed(v_ctx,v_trg) == D2FS_YES )
                {
                    d2ftrgs_par_flpath(v_ctx,v_trg,NULL);
                    printf("block trigger is Subclassed\n");
                }
                else if ( d2ftrgis_IsSubclassed(v_ctx, v_trg) == D2FS_NO )
                {
                    printf("block trigger is NOT Subclassed\n");
                }
            }
        }

        /* Save the form */
        d2ffmdsv_Save(v_ctx, v_fmd, (text *)0, FALSE);

        /* Generate the forms executable (fmx) */
        d2ffmdcf_CompileFile(v_ctx, v_fmd );
    }

    /* Destroy the API Context */
    d2fctxde_Destroy(v_ctx);

    return 0;
}
```

CORRECTED EXAMPLE 2 – CREATING A MASTER-DETAIL FORM

```
/*
** This example creates a master-detail form based on the dept and
** emp database tables owned by the user scott.  The master contains
** the following fields: empno, ename, job, sal, and deptno. The
** detail contains the following fields: deptno, dname, and loc.
** The join condition is deptno.
**
** ### This example needs to have ERROR CHECKING for each D2F* call.
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <d2fctx.h>
#include <d2ffmd.h>
#include <d2fwin.h>
#include <d2fcnv.h>
#include <d2fblk.h>
#include <d2fitm.h>
#include <d2frel.h>

int main()
{
    d2fctx   *v_ctx;
    d2fctxa   v_ctx_attr;
    d2ffmd   *v_fmd;
    d2fwin   *v_win;
    d2fcnv   *v_cnv;
    d2fblk   *v_emp_blk;
    d2fblk   *v_dept_blk;
    d2fitm   *v_Emp_empno_itm;
    d2fitm   *v_Emp_ename_itm;
    d2fitm   *v_Emp_job_itm;
    d2fitm   *v_Emp_sal_itm;
    d2fitm   *v_Emp_deptno_itm;
    d2fitm   *v_Dept_deptno_itm;
    d2fitm   *v_Dept_dname_itm;
    d2fitm   *v_Dept_loc_itm;
    d2frel   *v_rel;
    d2fstatus status;

    /* Initialize the attribute mask and create the Forms API context */
    v_ctx_attr.mask_d2fctxa = 0;
    d2fctxcr_Create(&v_ctx, &v_ctx_attr);

    /* Connect to a database */
    d2fctxcn_Connect(v_ctx, (text*)"scott/tiger", (dvoid *)0);

    /* Create the form */
    d2ffmdcr_Create(v_ctx, &v_fmd, "MYFORM");

    /* Create a window */
    d2fwincr_Create(v_ctx,v_fmd,&v_win,(text*)"MYWIN");

    /* Create a canvas */
    d2fcnvcr_Create(v_ctx, v_fmd, &v_cnv, (text*)"MYCANVAS");
```

```
/* Set viewport width */
d2fcnvs_vprt_wid(v_ctx, v_cnv, 512);
/* Set viewport height */
d2fcnvs_vprt_hgt(v_ctx, v_cnv, 403);
/* Set window */
d2fcnvs_wnd_obj(v_ctx, v_cnv, v_win);
/* Set viewport X-position */
d2fcnvs_vprt_x_pos(v_ctx, v_cnv, 0);
/* Set viewport Y-position */
d2fcnvs_vprt_y_pos(v_ctx, v_cnv, 0);
/* Set width */
d2fcnvs_width(v_ctx, v_cnv, 538);
/* Set height */
d2fcnvs_height(v_ctx, v_cnv, 403);

/* Create an EMP block */
d2fblkcr_Create(v_ctx, v_fmd, &v_emp_blk, (text*)"EMP");
/* Set to database block */
d2fblks_db_blk(v_ctx, v_emp_blk, TRUE);
/* Set query data source to Table */
d2fblks_qry_dat_src_typ(v_ctx, v_emp_blk, D2FC_QRDA_TABLE);
/* Set query data source name to EMP table */
d2fblks_qry_dat_src_nam(v_ctx, v_emp_blk, (text*)"EMP");
/* Set DML data source type to Table */
d2fblks_dml_dat_typ(v_ctx, v_emp_blk, D2FC_DMDA_TABLE);
/* Set DML data source name to EMP table */
d2fblks_dml_dat_nam(v_ctx, v_emp_blk, (text*)"EMP");

/* Create EMP.EMPNO item */
d2fitmcr_Create(v_ctx, v_emp_blk, &v_Emp_empno_itm, (text*)"EMPNO");
/* Set item type */
d2fitms_itm_typ(v_ctx, v_Emp_empno_itm, D2FC_ITTY_TI);
/* Set Enable property */
d2fitms_enabled(v_ctx, v_Emp_empno_itm, TRUE);
/* Set item (keyboard) navigable property */
d2fitms_kbrd_navigable(v_ctx, v_Emp_empno_itm, TRUE);
/* Set item Data Type property */
d2fitms_dat_typ(v_ctx, v_Emp_empno_itm, D2FC_DATY_NUMBER);
/* Set item Max Length property */
d2fitms_max_len(v_ctx, v_Emp_empno_itm, 6);
/* Set item Required property */
d2fitms_required(v_ctx, v_Emp_empno_itm, TRUE);
/* Set Distance Between Records property */
d2fitms_dist_btwn_recs(v_ctx, v_Emp_empno_itm, 0);
/* Set Database block(Database Item) property */
d2fitms_db_itm(v_ctx, v_Emp_empno_itm, TRUE);
/* Set Query Allowed */
d2fitms_qry_allowed(v_ctx, v_Emp_empno_itm, TRUE);
/* Set Query Length */
d2fitms_qry_len(v_ctx, v_Emp_empno_itm, 6);
/* Set Update Allowed */
d2fitms_updt_allowed(v_ctx, v_Emp_empno_itm, TRUE);
/* Set Item Displayed (Visible) */
d2fitms_visible(v_ctx, v_Emp_empno_itm, TRUE);
/* Set Item Canvas property */
d2fitms_cnv_obj(v_ctx, v_Emp_empno_itm, v_cnv);
/* Set Item X-position */
d2fitms_x_pos(v_ctx, v_Emp_empno_itm, 32);
/* Set Item Y-position */
d2fitms_y_pos(v_ctx, v_Emp_empno_itm, 50);
/* Set Item Width */
d2fitms_width(v_ctx, v_Emp_empno_itm, 51);
/* Set Item Height */
d2fitms_height(v_ctx, v_Emp_empno_itm, 17);
/* Set Item Bevel */
```

```
d2fitms_bevel(v_ctx, v_Emp_empno_itm, D2FC_BEST_LOWERED);
/* Set item Hint */
d2fitms_hint(v_ctx, v_Emp_empno_itm, (text*)"Enter value for :EMPNO");

/* Create EMP.ENAME item */
d2fitmcr_Create(v_ctx, v_emp_blk, &v_Emp_ename_itm, (text*)"ENAME");
/* Set item type */
d2fitms_itm_typ(v_ctx, v_Emp_ename_itm, D2FC_ITTY_TI);
/* Set Enable property */
d2fitms_enabled(v_ctx, v_Emp_ename_itm, TRUE);
/* Set item (keyboard) navigable property */
d2fitms_kbrd_navigable(v_ctx, v_Emp_ename_itm, TRUE);
/* Set item Data Type property */
d2fitms_dat_typ(v_ctx, v_Emp_ename_itm, D2FC_DATY_CHAR);
/* Set item Max Length property */
d2fitms_max_len(v_ctx, v_Emp_ename_itm, 10);
/* Set Distance Between Records property */
d2fitms_dist_btwn_recs(v_ctx, v_Emp_ename_itm, 0);
/* Set Database block(Database Item) property */
d2fitms_db_itm(v_ctx, v_Emp_ename_itm, TRUE);
/* Set Query Allowed */
d2fitms_qry_allowed(v_ctx, v_Emp_ename_itm, TRUE);
/* Set Query Length */
d2fitms_qry_len(v_ctx, v_Emp_ename_itm, 10);
/* Set Update Allowed */
d2fitms_updt_allowed(v_ctx, v_Emp_ename_itm, TRUE);
/* Set Item Displayed (Visible) */
d2fitms_visible(v_ctx, v_Emp_ename_itm, TRUE);
/* Set Item Canvas property */
d2fitms_cnv_obj(v_ctx, v_Emp_ename_itm, v_cnv);
/* Set Item X-position */
d2fitms_x_pos(v_ctx, v_Emp_ename_itm, 83);
/* Set Item Y-position */
d2fitms_y_pos(v_ctx, v_Emp_ename_itm, 50);
/* Set Item Width */
d2fitms_width(v_ctx, v_Emp_ename_itm, 77);
/* Set Item Height */
d2fitms_height(v_ctx, v_Emp_ename_itm, 17);
/* Set Item Bevel */
d2fitms_bevel(v_ctx, v_Emp_ename_itm, D2FC_BEST_LOWERED);
/* Set item Hint */
d2fitms_hint(v_ctx, v_Emp_ename_itm, (text*)"Enter value for :ENAME");

/* Create EMP.JOB item */
d2fitmcr_Create(v_ctx, v_emp_blk, &v_Emp_job_itm, (text*)"JOB");
/* Set item type */
d2fitms_itm_typ(v_ctx, v_Emp_job_itm, D2FC_ITTY_TI);
/* Set Enable property */
d2fitms_enabled(v_ctx, v_Emp_job_itm, TRUE);
/* Set item (keyboard) navigable property */
d2fitms_kbrd_navigable(v_ctx, v_Emp_job_itm, TRUE);
/* Set item Data Type property */
d2fitms_dat_typ(v_ctx, v_Emp_job_itm, D2FC_DATY_CHAR);
/* Set item Max Length property */
d2fitms_max_len(v_ctx, v_Emp_job_itm, 9);
/* Set Distance Between Records property */
d2fitms_dist_btwn_recs(v_ctx, v_Emp_job_itm, 0);
/* Set Database block(Database Item) property */
d2fitms_db_itm(v_ctx, v_Emp_job_itm, TRUE);
/* Set Query Allowed */
d2fitms_qry_allowed(v_ctx, v_Emp_job_itm, TRUE);
/* Set Query Length */
d2fitms_qry_len(v_ctx, v_Emp_job_itm, 9);
/* Set Update Allowed */
d2fitms_updt_allowed(v_ctx, v_Emp_job_itm, TRUE);
```

```
/* Set Item Displayed (Visible) */
d2fitms_visible(v_ctx, v_Emp_job_itm, TRUE);
/* Set Item Canvas property */
d2fitms_cnv_obj(v_ctx, v_Emp_job_itm, v_cnv);
/* Set Item X-position */
d2fitms_x_pos(v_ctx, v_Emp_job_itm, 160);
/* Set Item Y-position */
d2fitms_y_pos(v_ctx, v_Emp_job_itm, 50);
/* Set Item Width */
d2fitms_width(v_ctx, v_Emp_job_itm, 70);
/* Set Item Height */
d2fitms_height(v_ctx, v_Emp_job_itm, 17);
/* Set Item Bevel */
d2fitms_bevel(v_ctx, v_Emp_job_itm, D2FC_BEST_LOWERED);
/* Set item Hint */
d2fitms_hint(v_ctx, v_Emp_job_itm, (text*)"Enter value for :JOB");

/* Create EMP.SAL item */
d2fitmcr_Create(v_ctx, v_emp_blk, &v_Emp_sal_itm, (text*)"SAL");
/* Set item type */
d2fitms_itm_typ(v_ctx, v_Emp_sal_itm, D2FC_ITTY_TI);
/* Set Enable property */
d2fitms_enabled(v_ctx, v_Emp_sal_itm, TRUE);
/* Set item (keyboard) navigable property */
d2fitms_kbrd_navigable(v_ctx, v_Emp_sal_itm, TRUE);
/* Set item Data Type property */
d2fitms_dat_typ(v_ctx, v_Emp_sal_itm, D2FC_DATY_NUMBER);
/* Set item Max Length property */
d2fitms_max_len(v_ctx, v_Emp_sal_itm, 9);
/* Set Distance Between Records property */
d2fitms_dist_btwn_recs(v_ctx, v_Emp_sal_itm, 0);
/* Set Database block(Database Item) property */
d2fitms_db_itm(v_ctx, v_Emp_sal_itm, TRUE);
/* Set Query Allowed */
d2fitms_qry_allowed(v_ctx, v_Emp_sal_itm, TRUE);
/* Set Query Length */
d2fitms_qry_len(v_ctx, v_Emp_sal_itm, 9);
/* Set Update Allowed */
d2fitms_updt_allowed(v_ctx, v_Emp_sal_itm, TRUE);
/* Set Item Displayed (Visible) */
d2fitms_visible(v_ctx, v_Emp_sal_itm, TRUE);
/* Set Item Canvas property */
d2fitms_cnv_obj(v_ctx, v_Emp_sal_itm, v_cnv);
/* Set Item X-position */
d2fitms_x_pos(v_ctx, v_Emp_sal_itm, 352);
/* Set Item Y-position */
d2fitms_y_pos(v_ctx, v_Emp_sal_itm, 50);
/* Set Item Width */
d2fitms_width(v_ctx, v_Emp_sal_itm, 70);
/* Set Item Height */
d2fitms_height(v_ctx, v_Emp_sal_itm, 17);
/* Set Item Bevel */
d2fitms_bevel(v_ctx, v_Emp_sal_itm, D2FC_BEST_LOWERED);
/* Set item Hint */
d2fitms_hint(v_ctx, v_Emp_sal_itm, (text*)"Enter value for :SAL");

/* Create EMP.DEPTNO item */
d2fitmcr_Create(v_ctx, v_emp_blk, &v_Emp_deptno_itm, (text*)"DEPTNO");
/* Set item type */
d2fitms_itm_typ(v_ctx, v_Emp_deptno_itm, D2FC_ITTY_TI);
/* Set Enable property */
d2fitms_enabled(v_ctx, v_Emp_deptno_itm, TRUE);
/* Set item (keyboard) navigable property */
d2fitms_kbrd_navigable(v_ctx, v_Emp_deptno_itm, TRUE);
/* Set item Data Type property */
```

```
d2fitms_dat_typ(v_ctx, v_Emp_deptno_itm, D2FC_DATY_NUMBER);
/* Set item Max Length property */
d2fitms_max_len(v_ctx, v_Emp_deptno_itm, 4);
/* Set item Required property */
d2fitms_required(v_ctx, v_Emp_deptno_itm, TRUE);
/* Set Distance Between Records property */
d2fitms_dist_btwn_recs(v_ctx, v_Emp_deptno_itm, 0);
/* Set Database block(Database Item) property */
d2fitms_db_itm(v_ctx, v_Emp_deptno_itm, TRUE);
/* Set Query Allowed */
d2fitms_qry_allowed(v_ctx, v_Emp_deptno_itm, TRUE);
/* Set Query Length */
d2fitms_qry_len(v_ctx, v_Emp_deptno_itm, 4);
/* Set Update Allowed */
d2fitms_updt_allowed(v_ctx, v_Emp_deptno_itm, TRUE);
/* Set Item Displayed (Visible) */
d2fitms_visible(v_ctx, v_Emp_deptno_itm, TRUE);
/* Set Item Canvas property */
d2fitms_cnv_obj(v_ctx, v_Emp_deptno_itm, v_cnv);
/* Set Item X-position */
d2fitms_x_pos(v_ctx, v_Emp_deptno_itm, 493);
/* Set Item Y-position */
d2fitms_y_pos(v_ctx, v_Emp_deptno_itm, 50);
/* Set Item Width */
d2fitms_width(v_ctx, v_Emp_deptno_itm, 30);
/* Set Item Height */
d2fitms_height(v_ctx, v_Emp_deptno_itm, 17);
/* Set Item Bevel */
d2fitms_bevel(v_ctx, v_Emp_deptno_itm, D2FC_BEST_LOWERED);
/* Set item Hint */
d2fitms_hint(v_ctx, v_Emp_deptno_itm, (text*)"Enter value for :DEPTNO");

/* Create a DEPT block */
d2fblkcr_Create(v_ctx, v_fmd, &v_dept_blk, (text*)"DEPT");
/* Set to database block */
d2fblks_db_blk(v_ctx, v_dept_blk, TRUE);
/* Set query data source to Table */
d2fblks_qry_dat_src_typ(v_ctx, v_dept_blk, D2FC_QRDA_TABLE);
/* Set query data source name to EMP table */
d2fblks_qry_dat_src_nam(v_ctx, v_dept_blk, (text*)"DEPT");
/* Set DML data source type to Table */
d2fblks_dml_dat_typ(v_ctx, v_dept_blk, D2FC_DMDA_TABLE);
/* Set DML data source name to EMP table */
d2fblks_dml_dat_nam(v_ctx, v_dept_blk, (text*)"DEPT");

/* Create DEPT.DEPTNO item */
d2fitmcr_Create(v_ctx, v_dept_blk, &v_Dept_deptno_itm, (text*)"DEPTNO");
/* Set item type */
d2fitms_itm_typ(v_ctx, v_Dept_deptno_itm, D2FC_ITTY_TI);
/* Set Enable property */
d2fitms_enabled(v_ctx, v_Dept_deptno_itm, TRUE);
/* Set item (keyboard) navigable property */
d2fitms_kbrd_navigable(v_ctx, v_Dept_deptno_itm, TRUE);
/* Set item Data Type property */
d2fitms_dat_typ(v_ctx, v_Dept_deptno_itm, D2FC_DATY_NUMBER);
/* Set item Max Length property */
d2fitms_max_len(v_ctx, v_Dept_deptno_itm, 4);
/* Set item Required property */
d2fitms_required(v_ctx, v_Dept_deptno_itm, TRUE);
/* Set Distance Between Records property */
d2fitms_dist_btwn_recs(v_ctx, v_Dept_deptno_itm, 0);
/* Set Database block(Database Item) property */
d2fitms_db_itm(v_ctx, v_Dept_deptno_itm, TRUE);
/* Set Query Allowed */
d2fitms_qry_allowed(v_ctx, v_Dept_deptno_itm, TRUE);
```

```
/* Set Query Length */
d2fitms_qry_len(v_ctx, v_Dept_deptno_itm, 4);
/* Set Update Allowed */
d2fitms_updt_allowed(v_ctx, v_Dept_deptno_itm, TRUE);
/* Set Item Displayed (Visible) */
d2fitms_visible(v_ctx, v_Dept_deptno_itm, TRUE);
/* Set Item Canvas property */
d2fitms_cnv_obj(v_ctx, v_Dept_deptno_itm, v_cnv);
/* Set Item X-position */
d2fitms_x_pos(v_ctx, v_Dept_deptno_itm, 32);
/* Set Item Y-position */
d2fitms_y_pos(v_ctx, v_Dept_deptno_itm, 151);
/* Set Item Width */
d2fitms_width(v_ctx, v_Dept_deptno_itm, 38);
/* Set Item Height */
d2fitms_height(v_ctx, v_Dept_deptno_itm, 17);
/* Set Item Bevel */
d2fitms_bevel(v_ctx, v_Dept_deptno_itm, D2FC_BEST_LOWERED);
/* Set item Hint */
d2fitms_hint(v_ctx, v_Dept_deptno_itm, (text*)"Enter val for :DEPTNO");

/* Create DEPT.DNAME item */
d2fitmcr_Create(v_ctx, v_dept_blk, &v_Dept_dname_itm, (text*)"DNAME");
/* Set item type */
d2fitms_itm_typ(v_ctx, v_Dept_dname_itm, D2FC_ITTY_TI);
/* Set Enable property */
d2fitms_enabled(v_ctx, v_Dept_dname_itm, TRUE);
/* Set item (keyboard) navigable property */
d2fitms_kbrd_navigable(v_ctx, v_Dept_dname_itm, TRUE);
/* Set item Data Type property */
d2fitms_dat_typ(v_ctx, v_Dept_dname_itm, D2FC_DATY_CHAR);
/* Set item Max Length property */
d2fitms_max_len(v_ctx, v_Dept_dname_itm, 14);
/* Set Distance Between Records property */
d2fitms_dist_btwn_recs(v_ctx, v_Dept_dname_itm, 0);
/* Set Database block(Database Item) property */
d2fitms_db_itm(v_ctx, v_Dept_dname_itm, TRUE);
/* Set Query Allowed */
d2fitms_qry_allowed(v_ctx, v_Dept_dname_itm, TRUE);
/* Set Query Length */
d2fitms_qry_len(v_ctx, v_Dept_dname_itm, 14);
/* Set Update Allowed */
d2fitms_updt_allowed(v_ctx, v_Dept_dname_itm, TRUE);
/* Set Item Displayed (Visible) */
d2fitms_visible(v_ctx, v_Dept_dname_itm, TRUE);
/* Set Item Canvas property */
d2fitms_cnv_obj(v_ctx, v_Dept_dname_itm, v_cnv);
/* Set Item X-position */
d2fitms_x_pos(v_ctx, v_Dept_dname_itm, 70);
/* Set Item Y-position */
d2fitms_y_pos(v_ctx, v_Dept_dname_itm, 151);
/* Set Item Width */
d2fitms_width(v_ctx, v_Dept_dname_itm, 102);
/* Set Item Height */
d2fitms_height(v_ctx, v_Dept_dname_itm, 17);
/* Set Item Bevel */
d2fitms_bevel(v_ctx, v_Dept_dname_itm, D2FC_BEST_LOWERED);
/* Set item Hint */
d2fitms_hint(v_ctx, v_Dept_dname_itm, (text*)"Enter value for :DNAME");

/* Create DEPT.LOC item */
d2fitmcr_Create(v_ctx, v_dept_blk, &v_Dept_loc_itm, (text*)"LOC");
/* Set item type */
d2fitms_itm_typ(v_ctx, v_Dept_loc_itm, D2FC_ITTY_TI);
/* Set Enable property */
```

```
    d2fitms_enabled(v_ctx, v_Dept_loc_itm, TRUE);
    /* Set item (keyboard) navigable property */
    d2fitms_kbrd_navigable(v_ctx, v_Dept_loc_itm, TRUE);
    /* Set item Data Type property */
    d2fitms_dat_typ(v_ctx, v_Dept_loc_itm, D2FC_DATY_CHAR);
    /* Set item Max Length property */
    d2fitms_max_len(v_ctx, v_Dept_loc_itm, 13);
    /* Set Distance Between Records property */
    d2fitms_dist_btwn_recs(v_ctx, v_Dept_loc_itm, 0);
    /* Set Database block(Database Item) property */
    d2fitms_db_itm(v_ctx, v_Dept_loc_itm, TRUE);
    /* Set Query Allowed */
    d2fitms_qry_allowed(v_ctx, v_Dept_loc_itm, TRUE);
    /* Set Query Length */
    d2fitms_qry_len(v_ctx, v_Dept_loc_itm, 13);
    /* Set Update Allowed */
    d2fitms_updt_allowed(v_ctx, v_Dept_loc_itm, TRUE);
    /* Set Item Displayed (Visible) */
    d2fitms_visible(v_ctx, v_Dept_loc_itm, TRUE);
    /* Set Item Canvas property */
    d2fitms_cnv_obj(v_ctx, v_Dept_loc_itm, v_cnv);
    /* Set Item X-position */
    d2fitms_x_pos(v_ctx, v_Dept_loc_itm, 173);
    /* Set Item Y-position */
    d2fitms_y_pos(v_ctx, v_Dept_loc_itm, 151);
    /* Set Item Width */
    d2fitms_width(v_ctx, v_Dept_loc_itm, 96);
    /* Set Item Height */
    d2fitms_height(v_ctx, v_Dept_loc_itm, 17);
    /* Set Item Bevel */
    d2fitms_bevel(v_ctx, v_Dept_loc_itm, D2FC_BEST_LOWERED);
    /* Set item Hint */
    d2fitms_hint(v_ctx, v_Dept_loc_itm, (text*)"Enter value for :LOC");

    /* Create Relation between EMP and DEPT */
    d2frelcr_Create(v_ctx, (d2fob *)v_dept_blk, &v_rel, (text*)"DEPT_EMP");
    /* Set Relation Detail block */
    d2frels_detail_blk(v_ctx, v_rel, (text *)"EMP");
    /* Set Master Deletes property */
    d2frels_del_rec(v_ctx, v_rel, D2FC_DERE_NON_ISOLATED);
    /* Set Deferred property */
    d2frels_deferred(v_ctx, v_rel, FALSE);
    /* Set Auto Query property */
    d2frels_auto_qry(v_ctx, v_rel, FALSE);
    /* Set Prevent Masterless property */
    d2frels_prvnt_mstrless_ops(v_ctx, v_rel, FALSE);
    /* Set Join Condition property */
    d2frels_join_cond(v_ctx, v_rel, (text*)"DEPTNO");
    /* Instantiate Relation: creates master-detail triggers */
    d2frelup_Update(v_ctx, v_rel);

    /* Save Form */
    d2ffmdsv_Save(v_ctx, v_fmd, (text*)0, FALSE);

    /* Compile Form */
    d2ffmdcf_CompileFile(v_ctx, v_fmd);

    /* Destroy Context */
    d2fctxde_Destroy(v_ctx);
}
```

CORRECTED EXAMPLE 3 – USING FINDOBJ

```
/*
** This example shows how the hierarchical relationship
** between forms objects is maintained in the API using
** FindObj
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <d2fctx.h>
#include <d2fob.h>
#include <d2ffmd.h>
#include <d2fblk.h>
#include <d2fitm.h>
#include <d2ftrg.h>

int main(int argc, char *argv[])
{
    d2fctx*   v_ctx;
    d2fctxa   v_ctx_attr;
    d2fob*    v_fmd;
    d2fob*    v_blk;
    d2fob*    v_itm;
    d2fob*    v_trg;
    text*     v_blk_name;
    text*     v_itm_name;
    text*     v_trg_txt;
    text*     v_form_name;
    d2fstatus status;

    /* Get the form name from the command line */
    v_form_name = (text*) argv[1];

    /* Initialize the attribute mask and create the Forms API context */
    v_ctx_attr.mask_d2fctxa = 0;
    status = d2fctxcr_Create(&v_ctx, &v_ctx_attr);
    if (status != D2FS_SUCCESS )
    {
        printf("%d: Context creation failed\n",status);
        return(status);
    }

    /* Load the form */
    status = d2ffmdld_Load(v_ctx, &v_fmd, v_form_name, FALSE) ;
    if ( status != D2FS_SUCCESS )
    {
        printf("%d: Form load failed\n",status);
        return(status);
    }

    /* find a block named "BLOCK" */
    status = d2fobfo_FindObj(v_ctx, v_fmd, (text*)"BLOCK",
                             D2FFO_BLOCK, &v_blk );
    if ( status != D2FS_SUCCESS )
    {
        printf("%d: find BLOCK failed\n",status);
        return(status);
    }

    /* this will be NULL if the object doesn’t exist */
    if ( v_blk != (d2fob*)0 )
    {
        /* find an item named "ITEM" in block "BLOCK" */
        status = d2fobfo_FindObj(v_ctx, v_blk, (text*)"ITEM",
                                 D2FFO_ITEM, &v_itm );
        if ( status != D2FS_SUCCESS )
        {
            printf("%d: find ITEM failed\n",status);
            return(status);
        }

        /* this will be NULL if the object doesn’t exist */
        if ( v_itm != (d2fob *)0 )
        {
            /* find a trigger "WHEN-NEW-ITEM-INSTANCE" */
            /* in block.item "BLOCK.ITEM"              */
            status = d2fobfo_FindObj(v_ctx, v_itm,
                                     (text*)"WHEN-NEW-ITEM-INSTANCE",
                                     D2FFO_TRIGGER, &v_trg );
            if ( status != D2FS_SUCCESS )
            {
                printf("%d: find WHEN-NEW-ITEM-INSTANCE failed\n",status);
                return(status);
            }

            /* this will be NULL if the object doesn’t exist */
            if ( v_trg != (d2fob *)0 )
            {
                /* we found the block/item/trigger; print the block.item */
                d2fobg_name(v_ctx, v_blk, &v_blk_name);
                d2fobg_name(v_ctx, v_itm, &v_itm_name);
                printf("Block.Item: %s.%s\n", v_blk_name, v_itm_name);
                free(v_blk_name);
                free(v_itm_name);

                /* print the trigger text */
                d2fobg_trg_txt(v_ctx, v_trg, &v_trg_txt);
                if ( v_trg_txt && v_trg_txt[0] )
                {
                    printf("Trigger text: %s\n", v_trg_txt);
                    free(v_trg_txt);
                }
                else
                {
                    printf("Trigger text is null\n");
                }
            }
        }
    }

    /* destroy the Forms API context */
    d2fctxde_Destroy(v_ctx);

    return(D2FS_SUCCESS);
}
```

RELATED DOCUMENTS

For more information about Oracle Forms, refer to the following documents on the Oracle Forms 6*i* CD-ROM, available in both HTML and PDF formats:

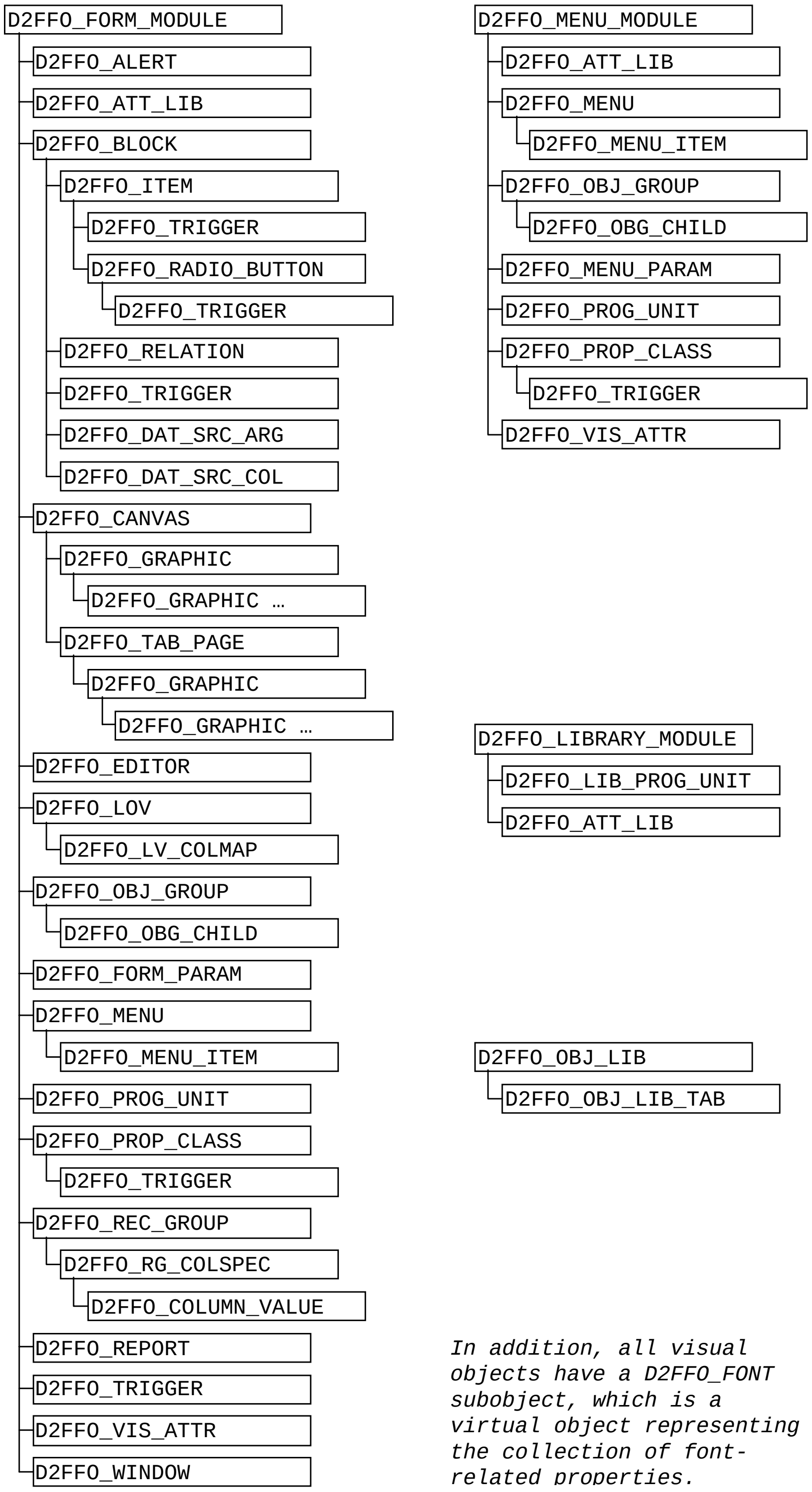
- *Oracle Forms 6i Release Notes*
- *Oracle Forms 6i Getting Started (Windows 95/NT)*
- *Oracle Forms 6i Deploying Reports to the Web*
- *Oracle Forms 6i Guidelines for Building Applications*
- *Oracle Forms 6i Form Builder Reference*
- *Deploying Form Builder Applications to the Web with Oracle Forms Server Release 6i*

The Form Builder online help (search for "Open API") contains detailed reference information about the Forms API, including lists of object types, properties, functions, and macros.

For up-to-date information about Oracle Forms and the Forms API, visit us on the web at http://www.oracle.com/tools/developer.

APPENDIX A: OBJECT HIERARCHY

This diagram shows the complete Form Builder object hierarchy.



In addition, all visual objects have a D2FFO_FONT subobject, which is a virtual object representing the collection of font-related properties.

Major Contributors:	Andrew Sefkow
	Kathryn Dumont
	Duncan Mills
	PV Dharan
	Aditi Shete
	Ken Chu
	Robert Nix

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
+1.650.506.7000
Fax +1.650.506.7200
http://www.oracle.com/