Structure of a Program Using UIL and MRM

The User Interface Language (UIL) allows an application developer to separate the specification of particular widget hierarchies from the application source code. The application defines widgets and their characteristics in a text file, which the developer compiles into a User Interface Definition (UID) file in binary format. At run time the application, using Motif Resource Manager (MRM) routines, retrieves the widget descriptions from the binary file, and MRM creates the widgets from these descriptions. The application defines callback procedures and interacts with the widgets as if it were using the Motif toolkit alone.

UIL offers several advantages over toolkit–only applications:

- UIL enforces the separation of the user interface specification from the application.
- A developer can change the interface by editing and recompiling a text file without recompiling and relinking the application program.
- The UIL compiler generates warnings for errors that the developer otherwise would not discover until running the program, if then. For example, the UIL compiler checks the spelling of resource names.
- The toolkit may handle large databases more efficiently when they are represented as UID files rather than resource files.

An application that uses UIL has two separate components: the UIL file and the application program.

The UIL file consists mainly of definitions of the application's widget hierarchy. The declaration for each widget typically includes the following components:

- Widget type
- Widget children
- Initial resource values
- Declarations for callback procedures

The UIL file can also define values for data such as compound strings, colors, and icons.

The structure of the application program is similar to that of a toolkit–only program. The chief difference is that, instead of explicitly creating each widget, the program uses MRM routines to retrieve widget definitions from the UID file and to create the widgets themselves. The program might also use MRM routines to retrieve data values defined in the UIL file. An application program using UIL must take the following actions:

- Include the required header files
- Initialize the Intrinsics
- Initialize MRM
- Open the UID file
- Register the names of callback procedures and values of identifiers specified in the UID file
- Retrieve and create widgets and data defined in the UID file
- Close the UID file
- Define callback procedures
- Make the widgets visible
- Enter the event loop

Structure of a UIL Module

A UIL module is a block of declarations and definitions for the values, procedures, literals, and objects that make up a user interface specification. Each UIL file contains either one complete module or, if the file is to be included in another UIL file, at least one complete top−level construct within a module.

Each module has the following structure:

- A *module* clause
- Zero or more declarations for the module as a whole
- Zero or more *include* directives
- Zero or more *value* declarations
- Zero or more *identifier* declarations
- Zero or more *procedure* declarations
- Zero or more *object* declarations
- Zero or more *list* declarations
- An *end module* clause

This section discusses the components of a UIL module, but it does not describe the UIL syntax in detail. For more information, see the **UIL(5X)** reference page in the *Motif Programmer's Reference*.

The module Clause

Each module begins with the declaration *module name*. The keyword *module* must be in lowercase.

Module−Level Declarations

Several optional declarations at the beginning of the module modify characteristics of the module as a whole:

names

The *names* declaration specifies whether names in the UIL file are stored in a case−sensitive or case−insensitive way. The following declaration, the default, means that names are stored as they appear in the UIL file, and all UIL keywords must be in lowercase:

```
names = case_sensitive
```

The following declaration means that all names are stored in uppercase, and UIL keywords can be in uppercase, lowercase, or mixed case:

```
names = case_insensitive
```

The entire *names* declaration itself must be in lowercase, and it affects only the part of the module that follows it.

character_set

The *character_set* clause declares the default character set for strings and compound strings specified in the module by double quotes ("*string*"). If this clause is not present, UIL derives the default character set from the language environment in which the UIL file is compiled. This does not affect the character set of strings specified in the module by single quotes ('*string*'). UIL derives the character set of these strings from the language environment in which the UIL file is compiled. The character set in this clause must be either a keyword representing one of the character sets UIL knows about or a character set returned by the *character_set* function.

objects

The *objects* clause specifies whether UIL should define objects of the specified types as widgets or gadgets. For example, this declaration specifies that UIL should define objects of type **XmPushButton** to be gadgets:

```
objects = { XmPushButton = gadget; }
```

A declaration for an individual object can override this specification.

The include Directive

The *include* directive includes the contents of a file in the current module. The directive consists of the keywords *include file* followed by a string representing the filename. If the filename has a full directory specification, UIL searches that directory for the file. Otherwise, UIL searches the directory of the main UIL source file and then the directory of the current UIL source file. The *-I* option to the **uil** command adds a directory to the search list.

Included files are useful for definitions common to more than one UIL module. In conjunction with the *-I* option to **uil**, they are also useful in internationalizing applications. Localized definitions for strings, font lists, and the like can reside in files included from different directories depending on language environment. In this case, the *include* directives should not specify the directories; instead, you can use the *-I* option to **uil** to compile files for different language environments without editing or duplicating UIL files.

The value Declaration

The *value* clause defines one or more names and associates them with values. The names can stand for values elsewhere in the module.

The specification for each value is either a literal expression or a call to a UIL function that generates a value. Each value has a UIL type that depends on the representation of the literal or the type of value returned by the UIL function. For more information on UIL types, literals, and functions, see the **UIL(5X)** reference page in the *Motif Programmer's Reference*.

By default, the names and their associated values are private to the module. The *value* declaration can also export a value to other modules or import a value from another module. For each name declared to be imported, MRM assigns the value from the corresponding exported declaration at run time.

In this example, the value *id_1* is exported:

```
value
    id_1: exported 1;
    label_1: compound_string('Off');
```

Another module can use the value *id_1* as follows:

```
value
    id_1: imported integer;
```

The identifier Declaration

An *identifier* clause declares one or more names that can appear elsewhere in the module. At run time, MRM assigns values to these names from data defined in the application program. The application uses the **MrmRegisterNames** or **MrmRegisterNamesInHierarchy** routine to establish the correspondence between UIL identifier names and application–defined data. The UIL compiler performs no type checking on identifiers.

The following example identifies names for x and y values that the application defines at run time:

```
identifier
    app_x_value;
    app_y_value;
```

The procedure Declaration

A *procedure* clause declares names of callback procedures or of creation routines for user–defined widgets. The application program itself defines the actual procedures. As with identifiers, the application must use **MrmRegisterNames** or **MrmRegisterNamesInHierarchy** to associate the procedure names with the actual procedures at run time.

For a callback procedure, the *procedure* declaration can also specify the type of data represented by the second argument (the application data pointer) to the callback routine:

```
procedure
    toggle_cb (integer);
    push_button_cb (integer);
```

The object Declaration

An *object* clause defines a widget or gadget and assigns a name that can stand for the object elsewhere in the UIL module. As with values, an object definition by default is private to the UIL module, but the *object* clause can declare it to be exported or imported. In addition to the UIL name, the *object* clause specifies the object's type and a list (enclosed in braces) that can define children, initial resource values, and callback

procedures.

Object Type

The object type specification is a keyword that is usually the same as the name of the corresponding toolkit widget class. For example, the type keyword for a MainWindow is **XmMainWindow** and for a PushButton is **XmPushButton**. UIL also allows type specifications that correspond to toolkit convenience routines for creating some kinds of specialized widgets, including menus, dialogs, ScrolledList, and ScrolledText. For example, the keyword **XmPulldownMenu** specifies a PulldownMenu, and the keyword **XmPromptDialog** specifies a PromptDialog.

The *object* clause can also specify that the object is to be either a widget or a gadget, overriding the default specified by the *objects* clause. For example, the following defines a PushButtonGadget:

```
object
    pb: XmPushButton gadget {};
```

Alternately, an *object* clause can specify a gadget by using the gadget class name (for example, **XmPushButtonGadget**) as the type specification.

Children

An *object* clause can specify the children of a composite widget. This specification appears inside the object list section and consists of the keyword *controls* followed by a list of child declarations. The declaration for each child consists of an object type and, usually, a name that refers to the definition for the child widget in its own *object* clause. Instead of a name for the child, the declaration can contain an entire local definition for the child widget in the form of an object list section. The child declaration can optionally begin with the keyword *managed* or *unmanaged*, which specifies whether or not MRM should manage the child after creating it. The default is to manage the child.

Some manager widgets automatically create children. For example, MainWindow creates three separators to separate its main components. The *controls* list can contain declarations for these children so that the UIL file can specify resource values for them. The declaration for an automatically created child begins with a specification of the name of the child, formed by prepending **Xm_** to the actual name of the child widget. The names of automatically created children are documented in the reference pages for the manager widgets in the *Motif Programmer's Reference*.

Following is an example of specifications for child widgets:

```
object
    main_win: XmMainWindow {
        controls {
            XmMenuBar main_menu;
            Xm_Separator1 { arguments
                                {
                                 XmNseparatorType = XmSHADOW_ETCHED_OUT;
                                 XmNmargin = 5;
                                };
                          };
            XmScrolledText text_win;
        };
    };
```

In general, a child widget can be of any type the Motif toolkit allows for a child of the parent widget. In some cases, the type of the child differs from the Motif toolkit class. For example, dialogs and menus require shells as their parents, but in UIL a dialog or menu is declared to be a direct child of its parent, with no intervening shell. MRM creates the shell at run time. In this way, UIL and MRM act like the Motif convenience routines for creating dialogs and menus.

Some widget hierarchies in UIL are slightly different from the corresponding hierarchies in the toolkit. For example, in UIL a PulldownMenu in an OptionMenu is described as a child of the OptionMenu, not of the OptionMenu's parent as it is in the toolkit. In a PulldownMenu system from a MenuBar or a PopupMenu, each PulldownMenu is a child of the associated CascadeButton, not of the CascadeButton's parent as it is in the toolkit. For more information, see Chapter 6.

Resource Values

An *object* clause can specify resource values for MRM to pass to the widget's creation function. This specification appears inside the object list section and consists of the keyword *arguments* followed by a list of resource declarations. The declaration for each resource consists of the name

of the resource as in the toolkit (for example, **XmNheight**) followed by = (equals sign) and a value for the resource. The type of the value must be of the proper UIL type for that resource. For information on the required UIL type for each resource, please refer to the *Motif Programmer's Reference*.

Following is an example of specifications for initial resource values:

```
object
    main_win: XmScrolledText {
        arguments {
            XmNrows = 10;
            XmNwordWrap = true;
            XmNbackground = color('red');
        };
    };
```

In some cases, UIL provides a value for a resource related to a resource that appears in a specification. For example, if a specification contains a value for **XmNitems** in a List, UIL provides the appropriate value for **XmNitemCount**. For resources of type, dimension, or position, you can specify units by using the syntax described in the **XmNunitType** resource of the **XmPrimitive** reference page.

Callback Procedures

An *object* clause can specify procedures to appear in callback lists for the object. This specification appears inside the object list section and consists of the keyword *callbacks* followed by a list of callback list declarations. The declaration for each callback list consists of the name of the callback resource as in the toolkit (for example, **XmNactivateCallback**) followed by = (equals sign) and a value specification for the resource.

In addition to appropriate toolkit resources, the specification can include the special callback list name **MrmNcreateCallback**. MRM invokes callback procedures on this list when it creates the widget. These procedures provide a means for the application to identify the widget ID of a widget created by MRM.

The value specification can be one of two forms:

- If the callback list contains only one procedure, the specification consists of the keyword *procedure* followed by the procedure name and, optionally, a value in parentheses for the application data argument to the procedure.
- If the callback list contains more than one procedure, the specification consists of the keyword *procedures* followed by a list of procedure specifications. Each specification consists of the procedure name and, optionally, a value in parentheses for the application data argument to the procedure.

The UIL compiler issues a warning if a procedure specification contains an application data argument whose type does not match the argument type in the corresponding *procedure* declaration.

The application uses the **MrmRegisterNames** routine or the **MrmRegisterNamesInHierarchy** routine to establish the correspondence between UIL procedure names and the application–defined procedures.

Following is an example of specifications for a callback list:

```
object
    pb: XmPushButton {
        callbacks {
            XmNactivateCallback =
                procedure pb_activate_cb (pb_ident);
        };
    };
```

The list Declaration

A *list* clause defines one or more lists of specifications for resources, callbacks, procedures, or widget children. Each list has a symbolic name that the application can use to refer to the list elsewhere in the UIL file, usually in an *object* declaration. The main use for this clause is to define lists of specifications that are common to more than one object definition.

A *list* clause consists of the keyword *list* followed by one or more list specifications. Each list specification contains the name, type, and contents of the list. Following are the four kinds of lists:

- A list of resources consists of the keyword *arguments* followed by a list of resource specifications.
- A list of callbacks consists of the keyword *callbacks* followed by a list of callback specifications.
- A list of procedures consists of the keyword *procedures* followed by a list of procedure specifications.

- list of widget children consists of the keyword *controls* followed by a list of specifications for the children.

In each case, the form of the list is the same as that of the corresponding clause of an *object* declaration.

Following is an example of a *list* declaration:

```
list
    pb_activate_procs: procedures {
        pb_ac_proc_1 ();
        pb_ac_proc_2 ();
    };

list
    pb_callbacks: callbacks {
        XmNactivateCallback = pb_activate_procs;
        XmNarmCallback = procedure pb_arm_proc ();
    };

list
    pb_args: arguments {
        XmNheight = 10;
        XmNbackground = color('red');
    };

object
    pb_1: XmPushButton {
        arguments {
            arguments pb_args;
            XmNlabelString = pb_label_1;
        };
        callbacks pb_callbacks;
     };

object
    pb_2: XmPushButton {
        arguments {
            arguments pb_args;
            XmNlabelString = pb_label_2;
        };
        callbacks pb_callbacks;
     };

list
    menu_items: controls {
        XmPushButton pb_1;
        XmPushButton pb_2;
    };

object
    menu_1: XmPulldownMenu {
        controls menu_items;
    };
```

The end module Clause

Each UIL module must end with an *end module* clause.

Structure of a Program Using MRM

The following sections describe the structural elements of an application program that uses the Motif Resource Manager.

Including Header Files

An application that uses MRM must include all the header files it would need if it did not use MRM. These include `<Xm/Xm.h>`, header files specific to each widget the program uses, and any header files needed by Motif routines. In addition, the application must include the file `<Mrm/MrmPublic.h>`. This file contains definitions that the MRM routines need.

Following is an example of including header files for an application that uses only a Text widget and MRM:

```
#include <Mrm/MrmPublic.h>
#include <Xm/Xm.h>
#include <Xm/Text.h>
```

Initializing the Intrinsics

An application initializes the Intrinsics as in any other program, usually by calling **XtAppInitialize**. The application must call **XtDisplayInitialize** either directly or indirectly before opening any UID files.

Initializing MRM

An application that uses MRM must initialize MRM by calling **MrmInitialize** before fetching any widgets from UID files. It is a good idea to call **MrmInitialize** before using any other MRM routines.

Opening UID Files

After initializing MRM and the Intrinsics, an application uses **MrmOpenHierarchyPerDisplay** to find and open one or more UID files that contain the widget definitions and other information to be loaded.

> **Note:** The UID format is portable only across same−size architecture machines.

MrmOpenHierarchyPerDisplay uses search paths in much the same way **XtDisplayInitialize** uses them to build the initial resource database. One argument to **MrmOpenHierarchyPerDisplay** is a list of UID filenames, each of which represents either a full pathname or a name to be substituted in a file search path. The search path comes from the **UIDPATH** environment variable or, if **UIDPATH** is not set, from a series of default paths. **MrmOpenHierarchyPerDisplay** calls **XtResolvePathname** to search these paths. When it uses a search path, **MrmOpenHierarchyPerDisplay** looks for files first by using a suffix of *.uid* and then by using a NULL suffix.

As with the initial resource database, UID files can reside in different directories depending on the language environment. The search paths can include these substitutions, as well as others recognized by **XtResolvePathname**:

- %*N* is replaced by the class name of the application.
- %*L* is replaced by the display's language specification.
- %*l* is replaced by the language part of the language specification.
- %*U* is replaced by the current filename from the list of filenames passed as an argument to **MrmOpenHierarchyPerDisplay**.

MrmOpenHierarchyPerDisplay returns an ID that identifies the list of open UID files for subsequent calls to routines that load data from the files. On each request to load data, MRM searches the list of files in order. This ordered list of open files is the UID hierarchy. The program can retrieve data from the hierarchy until it calls **MrmCloseHierarchy**.

Following is an example of a call to **MrmOpenHierarchyPerDisplay**. The example initializes MRM and the Intrinsics, opens a UID hierarchy, and closes the hierarchy.

```
int main(int argc, char **argv)
{
    Widget         app_shell;
    XtAppContext   app;
```

```
    static String  file_names[] = { "app_1", "app_2" };
    MrmHierarchy   hierarchy_id;

    app_shell = XtAppInitialize(&app, "Example",
        (XrmOptionDescList) NULL, 0, (Cardinal *) &argc, argv,
        (String *) NULL, (ArgList) NULL, 0);
    MrmInitialize();
    switch (MrmOpenHierarchyPerDisplay(XtDisplay(app_shell),
             (MrmCount) XtNumber(file_names), file_names,
             (MrmOsOpenParamPtr *) NULL, &hierarchy_id)) {
    case MrmSUCCESS:
        if (MrmCloseHierarchy(hierarchy_id) == MrmSUCCESS) {
            exit 0;
        } else {
            fprintf(stderr,
                    "Unable to close UID hierarchy.\n");
            exit 1;
        }
    case MrmNOT_FOUND:
        fprintf(stderr, "Unable to open UID files.\n");
        exit 1;
    default:
        fprintf(stderr, "Unable to open UID hierarchy.\n");
        exit 1;
    }
}
```

Registering Callbacks and Identifiers

The application must register the names of all callback procedures and identifiers defined in the UIL files. Registering the names associates the symbolic names in the UIL files with procedures and data defined in the program. **MrmRegisterNames** and **MrmRegisterNamesInHierarchy** accomplish this task. Names registered by **MrmRegisterNames** are global to all UID hierarchies, whereas names registered by **MrmRegisterNamesInHierarchy** are local to a particular hierarchy. When MRM looks up the program–defined value associated with a name in a given hierarchy, it searches first for an association local to the hierarchy and then for a global association.

Following is an example using **MrmRegisterNames**:

```
    void PBActivateCB_1(Widget pb, XtPointer app_data,
        XtPointer widget_data);
    void PBActivateCB_2(Widget pb, XtPointer app_data,
        XtPointer widget_data);
    void PBArmCB(Widget pb, XtPointer app_data,
        XtPointer widget_data);

    static MrmRegisterArg cb_list[] = {
        { "pb_ac_proc_1",   (XtPointer) PBActivateCB_1 },
        { "pb_ac_proc_2",   (XtPointer) PBActivateCB_2 },
        { "pb_arm_proc",    (XtPointer) PBArmCB }
    };

    ...
        if (MrmRegisterNames(cb_list,
                             (MrmCount) XtNumber(cb_list))
             == MrmSUCCESS) {
            ...
        } else {
            ...
        }
```

Fetching Information from UID Files

MRM can fetch the following information from UID files:

- Named widgets, defined by *object* clauses, and their descendants. Use **MrmFetchWidget** or **MrmFetchWidgetOverride**.
- Named color literals, defined by *color* or *rbg* functions and appearing in *value* clauses. Use **MrmFetchColorLiteral**.
- Named icon literals, defined by *icon* functions and appearing in *value* clauses. Use **MrmFetchIconLiteral**.
- Other named literals appearing in *value* clauses. Use **MrmFetchLiteral** or **MrmFetchSetValues**.

MRM can fetch literals appearing in *value* clauses only if they are defined as *exported.*

After creating a top−level shell, using **XtAppInitialize** or **XtAppCreateShell**, the application can use **MrmFetchWidget** to fetch the child of the top−level shell and its descendants. For each widget in the tree, **MrmFetchWidget** does the following:

- Calls the appropriate widget creation routine, passing it the initial resource values defined in the *arguments* specification in the *object* clause
- Adds the callback routines defined in the *callbacks* specification of the *object* clause
- Calls any **MrmNcreateCallback** callbacks
- Manages all child widgets unless they are defined to be *unmanaged*

The application does not have to fetch all widgets at the beginning of the program. To create widgets such as menus and dialogs as needed, the application can call **MrmFetchWidget** at any time.

The application can fetch the same widget definition more than once. MRM creates a new widget each time, essentially using the UIL definition as a template. **MrmFetchWidgetOverride** is useful here, as it allows the application to override the initial resource values specified in the UIL file.

Following is a simple example using **MrmFetchWidget** to create the main widget hierarchy for an application:

```
int main(int argc, char **argv)
{
    Widget         app_shell, top_level;
    XtAppContext   app;
    static String  file_names[] = { "app_1", "app_2" };
    MrmHierarchy   hierarchy_id;
    MrmType        top_level_class;

    MrmInitialize();
    app_shell = XtAppInitialize(&app, "Example",
        (XrmOptionDescList) NULL, 0, (Cardinal *) &argc, argv,
        (String *) NULL, (ArgList) NULL, 0);
    switch (MrmOpenHierarchyPerDisplay(XtDisplay(app_shell),
             (MrmCount) XtNumber(file_names), file_names,
             (MrmOsOpenParamPtr *) NULL, &hierarchy_id)) {
    case MrmSUCCESS:
        if (MrmFetchWidget(hierarchy_id, "top_level",
              app_shell, &top_level, &top_level_class)
                                          != MrmSUCCESS) {
            fprintf(stderr,
                     "Unable to fetch top-level widget.\n");
        }
        if (MrmCloseHierarchy(hierarchy_id) == MrmSUCCESS) {
            exit 0;
        } else {
            fprintf(stderr,
                     "Unable to close UID hierarchy.\n");
            exit 1;
        }
    case MrmNOT_FOUND:
        fprintf(stderr, "Unable to open UID files.\n");
        exit 1;
    default:
        fprintf(stderr, "Unable to open UID hierarchy.\n");
        exit 1;
    }
}
```

Closing the UID File

MrmCloseHierarchy closes all files in the specified UID hierarchy. The application can close and reopen a hierarchy, but usually it does not close a hierarchy until it is finished reading data from the UID files. When the application uses multiple hierarchies, operating system limits on the number of open files may make it necessary to close one hierarchy before opening another.

Defining Callback Procedures

An application that uses MRM defines callback procedures in the same way as an application that uses only the toolkit. For callbacks delared in UIL files, the application must use **MrmRegisterNames** or **MrmRegisterNamesInHierarchy** to associate the UIL callback procedure names with the actual procedures defined in the program.

An application can create widgets, such as dialogs and PopupMenus, as the program needs them. If these widgets are defined in UIL files, a callback procedure can call **MrmFetchWidget** to fetch them from UID files.

Making Widgets Visible

MrmFetchWidget never manages the widget the application is fetching. It does manage all other widgets in the tree whose root is the widget being fetched, except for widgets declared *unmanaged* in the UIL file. **MrmFetchWidget** does not realize any widgets in the tree.

The application must manage any unmanaged widgets created by **MrmFetchWidget**, and it must realize all widgets it wants to make visible. In the simple case where the application fetches the entire widget hierarchy at the beginning of the program, it typically manages the widget it fetches and then realizes the top–level shell:

```
int main(int argc, char **argv)
{
 Widget         app_shell, top_level;
 XtAppContext   app;
 static String  file_names[] = { "app_1", "app_2" };
 MrmHierarchy   hierarchy_id;
 MrmType        top_level_class;

    MrmInitialize();
    app_shell = XtAppInitialize(&app, "Example",
        (XrmOptionDescList) NULL, 0, (Cardinal *) &argc, argv,
        (String *) NULL, (ArgList) NULL, 0);
    switch (MrmOpenHierarchyPerDisplay(XtDisplay(app_shell),
             (MrmCount) XtNumber(file_names), file_names,
             (MrmOsOpenParamPtr *) NULL, &hierarchy_id)) {
    case MrmSUCCESS:
        if (MrmFetchWidget(hierarchy_id, "top_level", app_shell,
              &top_level, &top_level_class) == MrmSUCCESS) {
            XtManageChild(top_level);
            XtRealizeWidget(app_shell);
        } else {
            fprintf(stderr,
                     "Unable to fetch top-level widget.\n");
        }
        if (MrmCloseHierarchy(hierarchy_id) == MrmSUCCESS) {
            exit 0;
        } else {
            fprintf(stderr,
                      "Unable to close UID hierarchy.\n");
            exit 1;
        }
    case MrmNOT_FOUND:
        fprintf(stderr, "Unable to open UID files.\n");
        exit 1;
    default:
        fprintf(stderr, "Unable to open UID hierarchy.\n");
```

```
            exit 1;
        }
    }
```

Entering the Event Loop

As with toolkit applications that do not use MRM, a program using MRM typically calls **XtAppMainLoop** to enter the event loop after realizing the top–level shell.