Section 5 - UIL File Format

This page describes the format and contents of each reference page in Section 5, which covers the UIL file format.

Name

Section – a brief description of the file section.

Syntax

This section describes the syntax for the section of the UIL file. Anything in constant width type should be typed exactly as shown. Items in italics are expressions that should be replaced by actual names and values when you write a UIL file. Anything enclosed in brackets is optional. An ellipsis (...) means that the previous expression can be repeated multiple times and a vertical bar (|) means to select one of a set of choices.

Description

This section provides an overview of the particular section in the UIL module and it explains the syntax that is expected for the section. A UIL source file, also known as a UIL module, describes the user interface for an application. It consists of a module name, optional module settings, optional include directives, zero or more sections that describe all or part of a user interface, and an end module statement. The module specifies the widgets used in the interface, as well as the resources and callbacks of these widgets. UIL gives you the ability to use variables, procedures, lists, and objects to describe the interface.

A major portion of a UIL module is the sections that describe the user interface. They are the value section, for defining and declaring variables; the procedure section, for declaring callback routines; the identifier section, for declaring values registered by the application at run-time; the list section, for defining lists of procedures, resources, callbacks, and widgets; and the object section, for defining the widgets, their resources, and the widget hier-archy.

In this section, we provide reference pages for each section of a UIL source file, as well as for the overall module structure and the include directive. Figure 5-1 shows an example of a UIL module that contains all of these sections.

UIL Syntax

Symbols and identifiers in a UIL module must be separated by whitespace or punctuation characters in order to be recognized by the UIL compiler. Like C, no other restrictions are placed on the formatting of a UIL module, although the maximum line length accepted by the compiler is 132 characters.

Comments in UIL can take two different forms: single-line and multi-line. A single-line comment begins with a exclamation point (!) and continues to the end of the line. A multi-line comment begins with the characters /* and ends with the characters */. Since the UIL compiler suspends normal parsing within comments, they cannot be nested.

Values, identifiers, procedures, lists, and objects are declared or defined with programmer-assigned names. Names can be composed of upper and lowercase characters from A to Z, the digits from 0 to 9, and the underscore (_) and dollar sign ($) characters. Names may be up to 31 characters in length; they cannot begin with a digit. The names option, which is described on the module reference page, affects the case sensitivity of names. The UIL compiler maintains a single name-space for all UIL keywords and programmer-defined names. This means that the name of each value, identifier, procedure, list, and object must be unique.

UIL Keywords

UIL keywords are categorized into reserved and unreserved keywords. Reserved keywords cannot be redefined by the programmer, while unreserved keywords can be used as programmer-defined names. In general, you should avoid redefining unreserved keywords because it can lead to confusion and programming errors. UIL uses the following reserved and unreserved keywords:

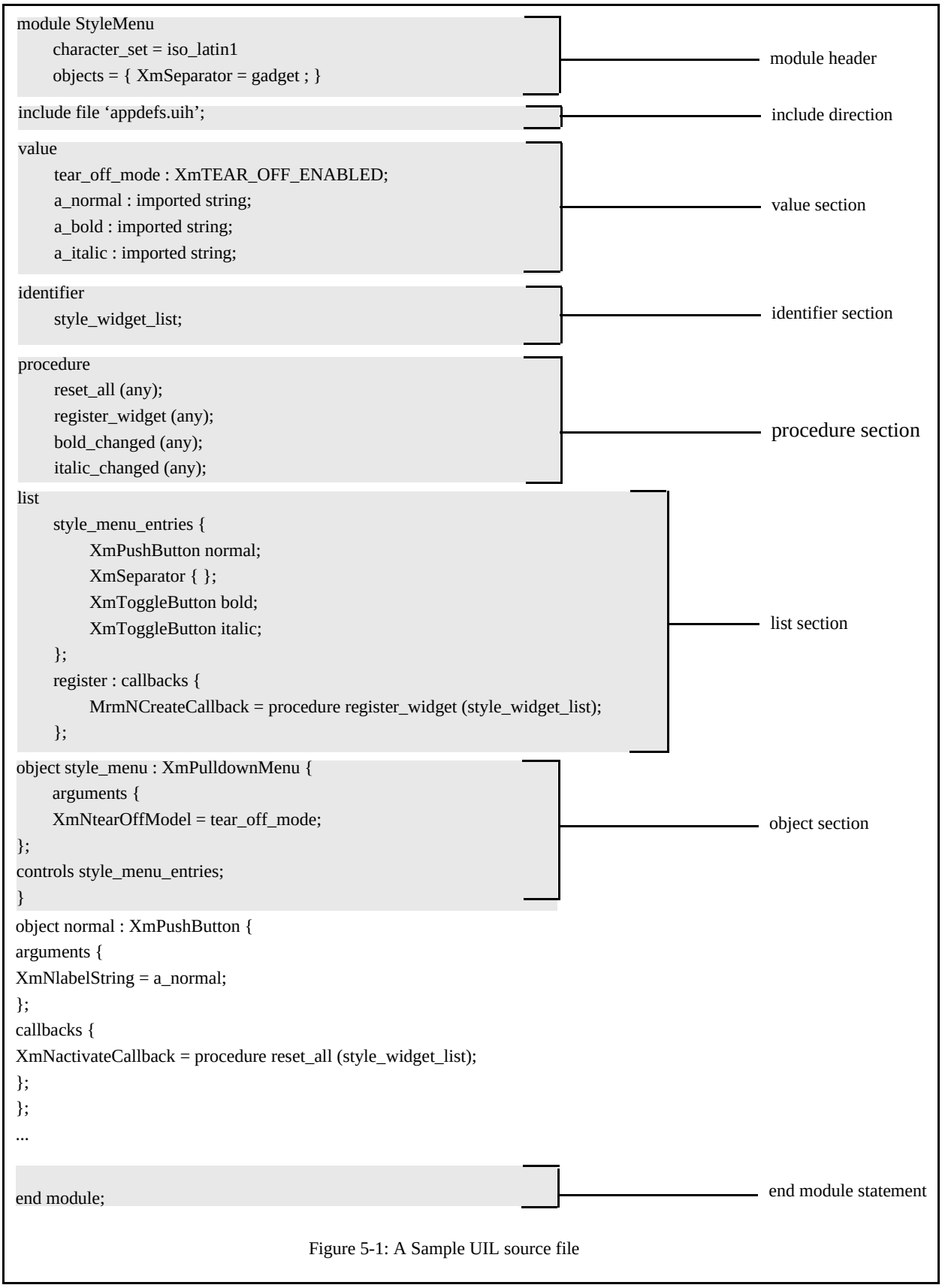

Figure 5-1: A Sample UIL source file

Type	Reserved Keywords
General	module, end, widget, gadget
Section and list name	arguments, callbacks, controls, identifier, include, list, object, procedure, procedures, value
Storage classes	exported, private
Boolean constants	on, off, true, false

Type	Unreserved Keywords
Resource names	XmNaccelerators, XmNactivateCallback, et al.
Character set names	iso_latin1, iso_greek, et al.
Enumerated values	XmATTACH_FORM, XmSHADOW_ETCHED_IN, et al.
Widget class names	XmPushButton, XmSeparator, et al.
Option names and values	background, case_insensitive, case_sensitive, file, foreground, imported, managed, names, objects, right_to_left, unmanaged, user_defined
Type names	any, argument, asciz_table, asciz_string_table, boolean, character_set, color, color_table, compound_string, compound_string_table, float, font, font_table, fontset, icon, integer, integer_table, keysym, reason, rgb, single_float, string, string_table, translation_table, wide_character, xbitmapfile

Usage

This section provides less formal information about the section: how you might want to use it in a UIL module and things to watch out for.

Example

This section provides examples of the use of the section in a UIL module.

See Also

This section refers you to related functions, UIL file format sections, and UIL data types. The numbers in parentheses following each reference refer to the sections of this book in which they are found.

Name

identifier – run-time variable declaration section.

Syntax

identifier *identifier_name;*
[...]

Description

The *identifier* section contains variable declarations that are registered at run-time by the application with `MrmRegisterNames`() or `MrmRegisterNamesInHierarchy`(). The section begins with the UIL keyword *identifier*, followed by a list of names separated by semicolons.

Usage

A value declared as an identifier can be assigned to a named variable in a value section, it can be passed as the parameter to a callback procedure, or it can be assigned to a resource in the arguments section of an object definition. An identifier value cannot be used in an expression or as part of a complex literal type definition. An identifier value does not have any type associated with it, so it can be passed as a parameter to any callback that can take an argument or it can be assigned to any resource, regardless of the type of parameter or resource expected.

Example

```
...
identifier
    display_name;
    highlight_color;

value
    alias : display_name;

procedure
    highlight (color);

object label : XmLabel {
    arguments {
        XmNlabelString : display_name;
    }
    callbacks {
        XmNfocusInCallback = procedure highlight (highlight_color);
    };
}
...
```

See Also

`MrmRegisterNames`(3), `MrmRegisterNamesInHierarchy`(3), `procedure`(5), `object`(5).

Name

include – include file directive.

Syntax

include file *'file_name';*

Description

The *include* directive tells the UIL compiler to suspend parsing of the current file and switch to the specified file. Parsing of the original file resumes after the end of the included file has been reached. Include directives may be nested, which means that an included file can contain include directives.

If an include file is specified an absolute pathname, which means that it begins with a slash (/), the compiler looks for the file in that specific location. Otherwise, the compiler tries to locate the file by searching in one or more directories. The directory that contains the UIL source file specified on the command line is searched first. (This directory may or may not be the same as the directory the compiler was invoked from.) If the file is not found there, the compiler searches any directories specified on the command line with the -I option in the order that they were specified. Next, the compiler searches the */usr/include* directory. Finally, if the specified file cannot be found, the compiler generates an error message and exits.

When an *include* directive is encountered, the UIL compiler ends the current section. Therefore, an include file must specifically use one of the section name keywords to begin a new section.

Usage

Include files are used to break up modules into more manageable pieces or to provide a common place for definitions and declarations that are shared by several modules. Include files should not be used for defining strings. Strings should be defined in a separate UIL module and loaded at run-time as part of an Mrm hierarchy. The `MrmOpenHierarchyPerDisplay`() reference page explains how different UID files can be loaded based on the LANG environment variable. String declarations, however, are suitable for placement in an include file.

A UIL module can include a maximum of 99 files. This is not a nesting limit, but a limit on the total number of files that can be included. Because the UIL compiler maintains an open file descriptor for each included file, even after it has been included, the limit may be less than 99 due to operating-system imposed limits. If the UIL compiler tries to include a file and the maximum number of open file descriptors have been used, the compiler prints an error and exits. If this situation occurs, you should reduce the number of files included or increase the maximum number of open file descriptors.

If the string containing the include filename is missing a closing quotation mark, or if extraneous characters precede or follow the string, the UIL compiler may generate many strange errors.

Example

From *callbacks.uih*:

```
procedure
    save();
    save_as (string);
    open (string);
    select (integer_table);
    quit();
```

From *edit_window.uil*:

```
module edit_window
! Include callback definitions
include file 'callbacks.uih';
...
end module;
```

See Also

`MrmOpenHierarchyPerDisplay`(3), `uil`(4), `module`(5).

Name

list - list definition section.

Syntax

```
list
    list_name :
        arguments {
            argument_name = value_expression; | arguments arguments_list_name;
            [...]
        }; |
    list_name :
        callbacks {
            reason_name = procedure procedure_name [ ( [ value_expression ] ) ]; |
            reason_name = procedures {
                procedure_name [ ( [ value_expression ] ) ];
                [...]
            }; |
            reason_name = procedures procedure_list_name; | callbacks callbacks_list_name;
            [...]
        }; |
    list_name :
        controls {
            [ managed | unmanaged ] object_class object_name; |
            [ managed | unmanaged ] object_class [ widget | gadget ] { [ attributes ] }; |
            [ managed | unmanaged ] user_defined procedure creation_procedure {[ attributes ] }; |
            auto_created_object_name { [ attributes ] }; |
            controls controls_list_name;
            [...]
        }; |
    list_name :
        procedures {
            procedure_name [ ( [ value_expression ] ) ]; | procedures procedures_list_name;
            [...]
        };
    [...]
```

Description

The *list* section is used to define lists of resources, callbacks, procedures, or controls that can be used when setting attributes of a widget defined in an object section. Each list definition consists of a list name followed by a colon, a list type, and a list of items of that type separated by semicolons. Each item can be a single item (resource, callback, procedure, or widget) or a list of that type of item. When a list contains another list, the result is the same as if the items in the included list were specified directly in the including list. The storage class of lists is limited to private. Unlike variables and objects, lists cannot be exported, imported, or retrieved by an application at run-time.

The type of a list determines the type and the format of the items it contains. UIL allows the following types of lists: *arguments*, *callbacks*, *controls*, and *procedures*. The format of the items in arguments, callbacks, and controls lists is the same as the format for the corresponding subsection in an object definition. The exact syntax is described in the *object* section reference page.

The procedures list type exists to allow the specification of a list of procedures for a single callback. Each routine in a procedures list is invoked by the specified callback. A procedures list is specified by the symbol *procedures*, followed by a list of procedures declared elsewhere in the module. An individual procedure is specified with the name of the procedure and an argument specification consistent with the routine's declaration. The order in which routines in a procedures list are invoked is not specified by

the Xt Intrinsics. If you need to have several procedures called in a particular order, you should register a single callback that calls the procedures in that order.

Like many values in UIL, a list can be specified directly in the arguments, callbacks, or controls subsection or as a callback procedures list. An inline list is specified by the type of the list, followed by a list of items of that type.

Usage

A list can be used to group collections of resources, callbacks, and widget children that are common to several object definitions. To specify more than one procedure for a single callback, you must use a list. A simple style/behavior hierarchy can be specified by using nested list definitions, as the example below illustrates. If a resource or callback setting occurs more than once in an arguments or callbacks list, the last occurrence has precedence over earlier occurrences. This feature allows you to define a list that includes settings from another list but overrides some of the settings. The UIL compiler issues an informational message about multiple occurrences, but the messages can be turned off by using the -w compiler option.

Example

```
...
! Declare procedures used below.
procedure
    shift();
    floor_it();
    armed();
    ready();

list
    ! Declare some lists to implement widget styles.
    base_style : arguments {
        ! This list contains individual elements only.
        XmNforeground = default_foreground;
        XmNbackground = default_background;
    };
    button_style : arguments {
        ! Include another list in this list.
        arguments base_style;
        XmNfontList = font ('*helvetica-bold-r-normal-*-120-100-100*');
    };

! Declare a list of procedures to be set on an individual callback.
list
    super_button_activate : procedures {
        shift();
        floor_it();
    };
list
    super_button_callbacks : callbacks {
        XmNactivateCallback = procedures super_button_activate;
        ! Set the arm callback to an inline list of procedures.
        XmNarmCallback = procedures {
            armed();
            ready();
        };
    };

object
    super_button : XmPushButton {
```

```
            arguments {
                ! Use arguments in button_style list and add one of our own.
                arguments button_style;
                XmNarmColor = color ('yellow');
            };
            callbacks super_button_callbacks;
        };
...
```

See Also

`object`(5), `procedure`(5).

Name

module – module structure.

Syntax

```
module module_name
    [ names = [ case_insensitive | case_sensitive ] ]
    [ character_set = character_set ]
    [ objects = { widget_name = gadget | widget; [...] } ]
    [ [ include_directive ] | [ value_section ] | [ procedure_section ] |
        [ identifier_section ] | [ list_section ] | [ object_section ] ]
[...]
end module;
```

Description

A UIL module must begin with the keyword module, followed by the name of the module. You may name a module anything you like, as long as it is a valid UIL identifier. The name of a module is defined as a symbol in the compiler's symbol table, and therefore may not be a UIL reserved keyword. In addition, the name of the module cannot be used as the name of an object, variable, identifier, widget, or procedure elsewhere in the module. Option settings for the module are specified following the module statement. There are three different options that you can set: the case sensitivity of the module, the default character set, and the default object variant.

The *names* option specifies the case sensitivity of keywords and symbols in the UIL module. The syntax of this option is the keyword names, followed by either *case_sensitive* or *case_insensitive*. The default is *case_sensitive*, which means that all keywords must be lowercase and the case of symbols is significant. If *case_insensitive* is specified, keywords may be in upper, lower, or mixed case, and all programmer-defined values, procedures, identifiers, and objects are stored as uppercase in the UID file. For example, the three symbols JellyBean, jellybean, and JELLYBEAN are considered different symbols when names are *case_sensitive*, but are considered the same symbol when names are *case_insensitive*. If this option is specified, it must be the first option after the module name and must be specified in lowercase only.

The *character_set* option specifies the character set used for compound_string, font, and fontset values that are not defined with an explicit character set. The syntax of this option is the keyword *character_set*, followed by the name of a built-in character set. (See the character_set reference page for a list of the built-in character sets.) A user-defined character set cannot be used for this option. If this option is not specified, the default character set is determined from the codeset portion of the LANG environment variable if it is set, or XmFALLBACK_CHARSET otherwise. Setting this option overrides the LANG environment variable and turns off localized string parsing specified by the -s compiler option. When the character_set defaults to XmFALLBACK_CHARSET, the UIL compiler may use ISO8859-1 as the character set, even if the value has been changed by the vendor. Therefore, you should specify a character set explicitly instead of relying on the value of XmFALLBACK_CHARSET.

The *objects* option specifies whether the widget or gadget variant is used by default for CascadeButton, Label, PushButton, Separator, and ToggleButton objects. The syntax of the option is the keyword *objects*, followed by a list of object-specific settings. Like all lists in UIL, each setting is separated by a semicolon and enclosed by curly braces. Each object setting is the name of one of the classes listed above, followed by either widget or gadget. The default value for all of the classes is widget. You can override these settings when you define a specific object by adding widget or gadget after the object class name.

UIL also supports a version option setting, which consists of the string version, followed by a string representing the version of the module. This option is obsolete in Motif 1.2 and is retained for backward compatibility. You may encounter this setting in older UIL source files but you should not use it in new ones. The version string is stored in the UID file, but is not used by Mrm and cannot be accessed by the

application. To make a version identifier that is accessible by the application through Mrm, you can store a version value in an exported UIL variable.

The bulk of a UIL module is the sections that describe the user interface, which occur after the module name and optional module settings. Briefly, the sections are the value section, for defining and declaring variables; the procedure section, for declaring callback routines; the identifier section, for declaring values registered by the application at run-time; the list section, for defining lists of procedures, resources, callbacks, or widgets; and the object section, for defining the widgets and their resources, and the widget hierarchy. Each section is described completely in a separate reference page.

Every UIL module must end with the end module statement, which is simply the string end module followed by a semicolon (;). A final newline is required after the end module statement or the UIL compiler generates an error message stating that the line is too long.

Example

```
module print_panel
    names = case_insensitive
    character_set = iso_latin1
    objects = { XmPushButton = gadget; }
! sections
...
end module;
```

See Also

`identifier`(5), `include`(5), `list`(5), `object`(5), `procedure`(5), `value`(5), `character_set`(6), `compound_string`(6), `font`(6), `fontset`(6), `font_table`(6), `string`(6).

Name

object – widget declaration and definition section.

Syntax

object *object_name : imported object_type;* or

object *object_name :* [exported | private]
object_type [widget | gadget] |
user_defined procedure *creation_procedure*
{ [
arguments {
arguments *argument_list_name;* |
argument_name = value_expression;
[...]
}; |
callbacks {
callbacks *callback_list_name;* |
reason_name = procedure procedure_name [([*value_expression*])]; |
reason_name = procedures {
procedure_name [([*value_expression*])];
[...]
}; |
reason_name = procedures procedures_list_name;
[...]
}; |
controls {
controls *controls_list_name;* |
[managed | unmanaged] *object_class object_name;* |
[managed | unmanaged] *object_class* [widget | gadget] { [*attributes*] }; |
[managed | unmanaged] user_defined procedure *creation_procedure {* [*attributes*] }; |
auto_created_object_name { [*attributes*] };
[...]
};
[...]]
};

Description

The *object* section is used to declare or define the objects that compose the user interface of an application. These objects can be either widgets or gadgets and are created at run-time with the routines `MrmFetchWidget`() and `MrmFetchWidgetOverride`(). Both built-in Motif widgets and user-defined widgets can be defined in an object section. In addition, in Motif 2.0 and later, the section is used to define special pseudo-objects which represent the XmRendition, XmRenderTable, XmTab, and XmTabList resource types.

An object declaration informs the UIL compiler about an object that is defined in another UIL module. A declaration consists of the object name followed by a colon, the keyword *imported*, and the type of the imported widget.

An object definition consists of an object name followed by a colon, an optional storage class, a built-in widget class name or used-defined creation procedure, and a list of attributes. An object's attributes may include resource settings, callbacks, and a list of the object's children. The storage class may be either *private* or *exported*. The default storage class is *exported*. Widgets defined as private are not prevented from being retrieved directly with Mrm, but you can still declare widgets as *private* to indicate that they should not be retrieved directly.

When defining an instance of a built-in widget, the name of a Motif class (such as XmPushButton or XmMessageDialog) follows the optional storage class. A class that has both widget and gadget variants can be followed by *widget* or *gadget* to indicate which variant is used. The default variant is widget, unless gadget is specified in the *objects* setting in the UIL module header. For gadget variants, the UIL compiler also allows the name Gadget to be appended directly to the widget class name (as in XmPush-ButtonGadget). This syntax is inconsistent, however, so you should avoid using it.

When defining an instance of a user-defined widget, the optional storage class is followed by the string *user_defined procedure* and the name of a widget creation procedure. The procedure must be declared in a *procedure* section elsewhere in the module. It must also be registered by the application at run-time with `MrmRegisterClass`() before the widget is retrieved. The C prototype of a creation procedure is described in the `MrmRegisterClass`() manual page in Section 3, *Mrm Functions*.

The remainder of an object definition consists of three optional subsections that define the widget's resources, callbacks, and children. The subsections are enclosed by curly braces, which must be present, even when none of the subsections are specified. Each subsection consists of the name of the subsection followed by the name of a list defined in a list section or a list of items enclosed by curly braces. The arguments subsection specifies resource settings, the *callbacks* subsection specifies callback procedures, and the *controls* section specifies child widgets. In Motif 2.0 and later, the controls section also specifies XmTab and XmRendition constituents of the XmTabList and XmRenderTable pseudo-objects.

Arguments

The *arguments* subsection, if present, specifies one or more resource settings and/or resource lists. A list is specified with the symbol arguments, followed by the name of an arguments list defined elsewhere in the module. Resource settings are of the form *resourceName = value*. The resource name may be built-in or user-defined. (See the argument reference page in Section 6, *UIL Data Types*, for information about creating user-defined resource names.) If the same resource is set more than once in a widget's arguments section, the last occurrence of the setting is used and the UIL compiler issues an informational message.

If the widget instance being defined is from a built-in Motif widget class, the predefined resources set in the arguments section must be valid for the widget class, but any user-defined resource can be set. It can be useful to set user-defined constraint resources on a built-in widget when it is the child of a user-defined constraint widget. If the widget instance being defined is a user-defined widget, any predefined or user-defined resources can be set in its arguments section. You should take care to set resources that are valid for user-defined widgets, as the UIL compiler is unable to detect invalid resources.

The UIL compiler normally verifies that the type of a value matches the type of the resource to which it is assigned. Type checking is not possible, however, when a value is assigned to a user-defined resource of type any, or when a variable declared in an identifier section is assigned to a resource.

The type of a resource and the value assigned to it do not always have to be an exact match. The UIL compiler automatically converts certain values to the appropriate type for a resource. If a type mismatch occurs and a conversion cannot be performed, the compiler generates an error message and a UID file is not generated. The table below summarizes the supported conversions:

Value Type	Can be Assigned To
string	compound_string
asciz_string_table	compound_string_table
icon	pixmap
xbitmapfile	icon
rgb	color
font	font_list

Value Type	Can be Assigned To
fontset	font_list

When a built-in array resource is specified in the arguments subsection, the UIL compiler automatically sets the associated count resource. All but one of the built-in arrays with associated counts are XmStringTable resources; they are listed in the compound_string_table reference page in Section 6, *UIL Data Type*s. The other resource is the Text and TextField resource XmNselectionArray and its associated count resource, XmNselectionArrayCount.

Callbacks

The *callbacks* subsection, if present, specifies one or more callback settings and/or callback lists. A list is specified with the symbol *callbacks*, followed by the name of a callback list defined elsewhere in the module. A callback setting consists of the callback name, such as XmNactivateCallback, followed by an equal sign (=) and either a single procedure name or the name of a list of procedures defined elsewhere in the module. A single procedure is specified by the symbol *procedure* followed by its name, and an argument specification consistent with the procedure's declaration. A list is specified by the symbol *procedures* followed by the name of the list. If the same callback is set more than once in a widget's callbacks section, the last occurrence of the setting is used and the UIL compiler issues an informational message.

A procedure used in the callbacks section must be declared in a procedure section elsewhere in the module. It must also be registered by the application at run-time with `MrmRegisterNames`() or `MrmRegisterNamesInHierarchy`() before any widgets that reference it are created.

If the widget instance being defined is from a built-in Motif widget class, the predefined callbacks set in the callbacks section must be valid for the widget class, but any user-defined callbacks can be set. There should not be any need to set a user-defined callback on a built-in widget, however. If the widget instance being defined is a user-defined widget, any built-in or user-defined callbacks can be set in the *callbacks* section.

In addition to the standard Motif callbacks, Mrm supports the MrmNcreateCallback, which is called by Mrm when a widget is created. The prototype of an MrmNcreateCallback is the same as any other Xt callback procedure. The *call_data* passed to the callback is an XmAnyCallbackStruct.

Controls

The *controls* subsection, if present, specifies a list of children. Each entry in the list may be a list of children, an object defined elsewhere, an object defined inline, or an automatically-created child. A list is specified with the symbol *controls* followed by the name of a controls lists defined elsewhere in the module. Specify an object defined elsewhere using an optional initial state of managed or unmanaged, followed by *user_defined* or a widget class and the name of the child widget. If the same child widget occurs more than once in a widget's *controls* section, an instance of the child is created for each occurrence.

An inline object definition is similar, but the name of the child widget is replaced by a set of widget attributes. The name of the inline widget is automatically generated by the UIL compiler. Inline definitions can be used to define widget instances that have few or no attributes and that do not need to be referenced by name. You may wish to avoid inline definitions, however, since the widget name is not well-defined, which makes customization via X resources difficult. An automatically-created child is specified by the name of the child followed by an attributes list. Appendix D, *Table of UIL Objects*, lists the automatically-created children of the built-in Motif widgets. The ability to specify attributes for automatically-created children is only available in Motif 1.2 and later.

If the widget instance being defined is from a built-in Motif widget class, the children specified in the *controls* section must be valid for the widget class, but any user-defined children can be specified. If the widget instance being defined is a user-defined widget, any built-in or used-defined children can be specified in the *controls* section. The UIL compiler verifies that the children specified in the controls section are allowable children for the widget being defined. Appendix D, *Table of UIL Objects*, lists the valid children for each built-in widget class. Any children are allowed for user-defined widgets. If an

invalid child is specified in a widget's controls section, the UIL compiler generates an error and no UID file is produced.

From Motif 2.0 and later, the controls section can be used to specify constituent entries in an XmRenderTable or XmTabList pseudo-object. For the XmRenderTable object, the controls section lists a set of further objects of type XmRendition. For the XmTabList object, each listed control is an object of type XmTab. The rendition and tab list objects are associated with one another by specifying an XmTabList object as an XmNtabList value within the controls section of an XmRendition object. The example given below clarifies the relationships.

Usage

A named widget can be specified as a value for a resource of type *widget*, such as the Form constraint resource XmNleftWidget, or as the argument of a callback procedure declared with a parameter of type any or widget. Prior to Motif 1.2.1, UIL does not allow the type *widget* to be used as an argument type in a procedure declaration. You can specify type any to work around this problem. Older versions of UIL may require the widget class name to precede a widget value that is assigned to a resource or used as callback parameter. Since all versions of UIL accept this syntax, you can avoid potential difficulties by always using it.

Mrm places some restrictions on the widgets that can be assigned to a resource or used as a callback parameter. The widget must be a member of the same hierarchy as the widget definition in which it is used. A widget hierarchy includes the widget named in the call to `MrmFetchWidget`() or `MrmFetchWidgetOverride`() and the widgets created in the widget tree below it. If a named widget does not exist when a reference to it is encountered, Mrm waits until all of the widgets in the hierarchy have been created and tries to resolve the name again. If a widget reference still cannot be resolved, Mrm does not set the specified resource or add the specified callback. As of Motif 1.2, Mrm does not generate a warning message when this situation occurs.

The advantage of this functionality is that, unlike in C, you do not have to worry about the creation order of a widget hierarchy when you are specifying a widget as a resource value or callback parameter. UIL also makes the creation of OptionMenus and MenuBars easier by allowing you to specify a PulldownMenu as the child of an OptionMenu or CascadeButton. The XmNsubMenuId resource of the object is automatically set to widget ID of the menu. When specified as the child of a CascadeButton, the menu is created as a child of the MenuBar that contains the button. As a convenience, you can also specify a PopupMenu as the child of any widget (but not gadget).

As of Motif 1.2, the UIL compiler does not support user-defined imported widgets. If you need to import a user-defined widget, declare it with the type of a built-in widget that is a valid child for the context where the imported widget is used.

Example

```
...

object romulus : XmPushButton gadget {
    callbacks {
        XmNactivateCallback = procedure create_Rome();
    };
};

object remus : imported XmPushButton;

object mars : XmForm {
    arguments {
        XmNbackground = color ('orange');
    };
    controls {
        ! Define a couple of children.
        XmPushButton romulus;
```

```
            unmanaged XmPushButton remus;
            ! Define an inline separator.
            XmSeparator { };
        };
};

object thing : user_defined procedure create_thing {
        ...
};

object scale : XmScale {
        controls {
            ! Set the labelString on the automatically created label.
            Xm_Title {
                arguments {
                    XmNlabelString = 'Temperature';
                    XmNrenderTable = rtable_1;
                };
            };
        };
};

! Motif 2.0 and later: pseudo-objects for rendition
! XmRenderTable objects contain rendition objects as controls
object rtable_1 : XmRenderTable {
        controls {
            XmRendition rendition_1;
        };
};

! XmRendition objects contain XmTabList objects as controls
! Note that XmNtag is not a supported argument:
! the tag is implicitly the object name
object rendition_1 : XmRendition {
        arguments {
            XmNfontName = "fixed";
            XmNunderlineType = XmDOUBLE_LINE;
        };
        controls {
            XmTabList tablist_1;
        };
};

! XmTabList objects contain XmTab objects as controls
object tablist_1 : XmTabList {
        controls {
            XmTab tab1;
            XmTab tab2;
        };
};

object tab1 : XmTab {
        arguments {
            XmNtabValue = 1.75;
            XmNunitType = XmCENTIMETERS;
            XmNoffsetModel = XmABSOLUTE;
        };
```

```
};

object tab2 : XmTab {
    arguments {
        XmNtabValue = 2.0;
        XmNunitType = XmCENTIMETERS;
        XmNoffsetModel = XmRELATIVE;
    };
};
...
```

See Also

`MrmFetchWidget`(3), `MrmFetchWidgetOverride`(3), `MrmRegisterNames`(3), `MrmRegisterNamesInHierarchy`(3), `list`(5), `procedure`(5), `value`(5), `any`(6), `argument`(6), `compound_string_table`(6), `reason`(6), `widget`(6).

Name

procedure – procedure declaration section.

Syntax

procedure *procedure_name* [([*value_type*])];
[...]

Description

The procedure section contains declarations of procedures that can be used as a callback for a widget or as a user-defined widget creation function. Procedures can also be used in a procedure list; procedure lists are used to associate more than one callback procedure with a specific callback. Procedure lists are described on the list reference page.

The procedure section begins with the UIL keyword procedure, followed by list of procedure declarations. Each declaration consists of the procedure name followed by optional parentheses enclosing an optional parameter type. Valid type names are listed in the Introduction to *[cmtr06].

Usage

A procedure declaration can be used to specify whether a procedure expects a parameter, and if so, the type of the parameter. The UIL compiler verifies that a procedure reference conforms to its declaration. If a procedure name is not followed by parentheses, the compiler does not count parameters or perform any type checking when the procedure is used. Zero arguments, or one argument of any type, can be used in the reference.

If the procedure name is followed by an empty pair of parentheses, a reference to the procedure must contain zero arguments. User-defined widget creation functions should be declared as taking no parameters, although the UIL compiler does not enforce this rule.

If the procedure name is followed by a parenthesized type name or widget class, a reference to the procedure must contain exactly one argument of the specified type or class. If the type any is specified, the reference can contain an argument of any type. Prior to Motif 1.2.1, the UIL compiler generates an error if a widget class name is specified as the type in a procedure declaration. If the parameter to a callback procedure is an imported value or an identifier that cannot be resolved at run-time, a segmentation fault may occur when the callback is called.

Because identifiers and procedures are registered in the same name space with MrmRegisterName() and MrmRegisterNamesInHierarchy(), it is possible to declare a value as a procedure in the UIL source, even though the entry that is registered may not be a procedure. An attempt to call a non-procedure value usually causes an application to crash.

Example

```
...
procedure
    exit();
    print (string);
    XawCreateForm();
    popup (XmPopupMenu);

object form : user_defined procedure XawCreateForm { };

object quit : XmPushButton {
    callbacks {
        MrmNcreateCallback = procedure print ('Hello!');
        XmNactivateCallback = procedure exit();
        XmNdestroyCallback = procedure print ('Goodbye!');
    };
};
```

See Also

MrmFetchWidget(3), MrmFetchWidgetOverride(3), MrmRegisterNames(3), MrmRegisterNamesInHierarchy(3), identifier(5), list(5), object(5).

Name

value – variable definition and declaration section.

Syntax

value *value_name :* [exported | private] *value_expression* | imported *value_type;*
[...]

Description

The *value* section contains variable definitions and declarations. A variable is defined by assigning a value to it. A variable declaration is used to inform the UIL compiler of the existence of a variable defined in another module. The value assigned to a variable may be an arithmetic or string expression, a literal value, or another variable or identifier.

A value can be declared with a storage class of *private*, *exported*, or *imported*. Values are private by default. Private and exported values consist of a named variable and the value that is assigned to it. Private values are only accessible within the module in which they are defined. An exported variable definition includes the symbol *exported* before the value assigned. Exported values are accessible in other modules and from the application, in addition to the module in which they are defined.

You can access an exported value in another module by declaring it as an imported value in the module where you want to access it. Imported value declarations consist of a named variable, the symbol *imported*, and the type of the variable. If an imported value is exported from more than one module, the value from the module that occurs first in the array passed to `MrmOpenHierarchyPerDisplay`() is used.

Values of all types can be declared as private; values of most types can be declared as exported and imported. The Introduction to Section 6, *UIL Data Types*, contains a table that summarizes the storage classes that are allowed for each type.

Usage

Variables used in an expression can be forward referenced. However, the specification of some complex literals cannot contain forward-referenced values. The UIL compiler indicates a value cannot be found in these cases. Refer to the reference page for a type to see if its literal representation can contain forward references.

Typically, the value of a variable used in an expression or in the specification of a complex literal must be accessible in the module in which it is used. As a result, in most cases you cannot use an imported variable in an expression or complex value specification. If an imported value is used in an invalid context, the UIL compiler issues an error message.

Example

```
...
! See individual type reference pages for additional examples.
value
    version     : exported 1002;
    Soothsayer  : 'Beware the ides of March.';
    ides        : 15;
    background  : imported color;
    ...
```

See Also

`MrmFetchBitmapLiteral`(3), `MrmFetchColorLiteral`(3), `MrmFetchIconLiteral`(3), `MrmFetchSetValues`(3), `argument`(6), `asciz_string_table`(6), `boolean`(6), `color`(6), `color_table`(6), `compound_string`(6), `compound_string_table`(6), `float`(6), `font`(6), `fontset`(6), `font_table`(6), `icon`(6), `integer`(6), `integer_table`(6), `keysym`(6), `reason`(6), `rgb`(6), `single_float`(6), `string`(6), `translation_table`(6), `wide_character`(6), `xbitmapfile`(6).