CMUCL Motif Toolkit

June 17, 2001

1 Naming conventions

In general, names in the Lisp Motif interface are derived directly from the C original. The following rules apply:

1. Drop `Xt`" and `Xm` (also `XmN`, `XmC`, etc.) prefixes
2. Separate words by dashes (`-`) rather than capitalization
3. Resource names and enumeration values are given as keywords
4. Replace underscores (`_`) with dashes (`-`)

Examples:

`XtCreateWidget`	$\Longrightarrow$	`create-widget`
`XmNlabelString`	$\Longrightarrow$	`:label-string`
`XmVERTICAL`	$\Longrightarrow$	`:vertical`

Some exceptions:
Compound string functions (`XmString`...) are prefixed by `compound-string-` rather than `string-` in Lisp.

Functions or resources, with the exception of the `compound-string-xxx` functions, which require compound string arguments, may be given Lisp `SIMPLE-STRING`s instead.

The arguments to functions are typically the same as the C Motif equivalents. Some exceptions are:

- Widget creation functions have a `&rest` arg for resource values.
- Functions which take a string table/length pair in C only take a list of strings in Lisp.
- Registering functions such as `ADD-CALLBACK` use a `&rest` arg for registering an arbitrary number of `client-data` items.

2 Starting things up

The Motif toolkit interface is divided into two parts. First, there is a server process written in C which provides an RPC interface to Motif functions. The other half is a Lisp package which connects to the server and makes requests on the user's behalf. The Motif interface is exported from the `TOOLKIT` (nickname `XT`) package.

2.1 Variables controlling connections

`*DEFAULT-SERVER-HOST*` A string naming the machine where the Motif server is to be found. The default is NIL, which causes a connection to be made using a Unix domain socket on the local machine. Any other name must be a valid machine name, and the client will connect using Internet domain sockets.

`*DEFAULT-DISPLAY*` Determines the display on which to open windows. The default value of NIL instructs the system to consult the `DISPLAY` environment variable. Any other value must be a string naming a valid X display.

`*DEFAULT-TIMEOUT-INTERVAL*` An integer specifying how many seconds the Lisp process will wait for input before assuming that the connection to the server has timed out.

2.2 Handling Connections

`OPEN-MOTIF-CONNECTION (hostname xdisplay-name app-name app-class)` Opens a connection to a server on the named host and opens a display connection to the named X display. The `app-name` and `app-class` are for defining the application name and class for use in resource specifications. An optional process-id argument can be passed if a local server process has already been created. This returns a MOTIF-CONNECTION object.

`CLOSE-MOTIF-CONNECTION (connection)` This closes a toolkit connection which was created by OPEN-MOTIF-CONNECTION.

`*MOTIF-CONNECTION*` Bound in contexts such as callback handlers to the currently active toolkit connection.

`*X-DISPLAY*` Bound in contexts such as callback handlers to the currently active CLX display.

`WITH-MOTIF-CONNECTION ((connection) &body forms)` This macro establishes the necessary context for invoking toolkit functions outside of callback/event handlers.

`WITH-CLX-REQUESTS (&body forms)` Macro that ensures that all CLX requests made within its body will be flushed to the X server before proceeding so that Motif functions may use the results.

`RUN-MOTIF-APPLICATION (init-function)` This is the standard CLM entry point for creating a Motif application. The init-function argument will be called to create and realize the interface. It returns the created MOTIF-CONNECTION object. Available keyword arguments are:

`:init-args`	list of arguments to pass to init-function
`:application-class`	application class (default `"Lisp"`)
`:application-name`	application name (default `"lisp"`)
`:server-host`	name of Motif server to connect to
`:display`	name of X display to connect to

`QUIT-APPLICATION ()` This is the standard function for closing down a Motif application. You can call it within your callbacks to terminate the application.

3 The Server

The C server is run by the `motifd` program. This will create both Inet and Unix sockets for the Lisp client to connect to. By default, the Inet and Unix sockets will be specific to the user.

When a Lisp client connects to the server, it forks a copy of itself. Thus each Lisp application has an exclusive connection to a single C server process. To terminate the server, just `Ĉ` it.

Switches to change behavior:

`-global`	Sockets created for use by everyone rather than being user-specific.
`-local`	No Inet socket is created and the Unix socket is process-specific
`-noinet`	Instructs the server not to create an Inet socket.
`-nounix`	Instructs the server not to create a Unix socket.
`-nofork`	Will keep the server from forking when connections are made. This is useful when debugging the server or when you want the server to die when the application terminates.
`-trace`	Will spit out lots of stuff about what the server is doing. This is only for debugging purposes.

Typically, users do not need to be concerned with server switches since, by default, servers are created automatically by your Lisp process. However, if you wish to share servers, or use servers across the network, you will need to run the server manually.

4 Widget creation

`CREATE-APPLICATION-SHELL (&rest resources)` Creates the `applicationShell` widget for a new Motif application.

`CREATE-WIDGET, CREATE-MANAGED-WIDGET (name class parent &rest resources)` These create new widgets. `CREATE-WIDGET` does not automatically manage the created widget, while `CREATE-MANAGED-WIDGET` does.

`CREATE-<widget_class> (parent name &rest resources)` Convenience function which creates a new widget of class `<widget_class>`. For instance, `CREATE-FORM` will create a new `XmForm` widget.

`*CONVENIENCE-AUTO-MANAGE*` Controls whether convenience functions automatically manage the widgets they create. The default is NIL.

5 Callbacks

Callbacks are registered with the `ADD-CALLBACK` function. Unlike Motif in C, an arbitrary number of client-data items can be registered with the callback. Callback functions should be defined as:

```
(defun callback-handler (widget call-data \&rest client-data) ... )
```

The passed widget is that in which the callback has occurred, and the call-data is a structure which provides more detailed information on the callback. Client-data is some number of arguments which have been registered with the callback handler. The slots of the call-data structure can be derived from the C structure name using the standard name conversion rules. For example, the call-data structure for button presses has the following slot (aside from the standard ones): click-count.

To access the X event which generated the callback, use the following:

```
(defun handler (widget call-data \&rest client-data)
  (with-callback-event (event call-data)
    ;; Use event structure here
  ))
```

Since callback procedures are processed synchronously, the Motif server will remain blocked to event handling until the callback finishes. This can be potentially troublesome, but there are two ways of dealing with this problem. The first alternative is the function `UPDATE-DISPLAY`. Invoking this function during your callback function will force the server to process any pending redraw events before continuing. The other (slightly more general) method is to register deferred actions with the callback handling mechanism. Deferred actions will be invoked after the server is released to process other events and the callback is officially terminated. Deferred actions are not invoked if the current application was destroyed as a result of the callback, since any requests to the server would refer to an application context which was no longer valid. The syntax for their usage is:

```
(with-callback-deferred-actions <forms>)
```

You may register only one set of deferred actions within the body of any particular callback procedure, as well as within event handlers and action procedures. Registering a second (or more) set of deferred actions will overwrite all previous ones.

When using deferred action procedures, care must be taken to avoid referencing invalid data. Some information available within callbacks is only valid within the body of that callback and is discarded after the callback terminates. For instance, events can only be retrieved from the call-data structure within the callback procedure. Thus the code

```
(with-callback-deferred-actions
  (with-callback-event (event call-data)
    (event-type event)))
```

is incorrect since the event will be fetched after the callback is terminated, at which point the event information will be unavailable. However, the code

```
(with-callback-event (event call-data)
  (with-callback-deferred-actions
    (event-type event)))
```

is perfectly legitimate. The event will be fetched during the callback and will be closed over in the deferred action procedure.

6 Action procedures

Action procedures can be registered in translation tables as in the following example:

```
<Key> q: Lisp(SOME-PACKAGE:MY-FUNCTION)\n
```

The generating X event can be accessed within the action handler using:

```
(with-action-event (event call-data)
  ... use event here ...
)
```

7 Event handlers

X events are also represented as structured objects with slot names which are directly translated from the C equivalent. The accessor functions are named by `<event name>-<slot name>`. Some examples:

`(event-window event)`	This applies to all events
`(event-type event)`	So does this
`(button-event-x event)`	Some button event
`(button-event-button event)`	accessors

At the moment, `XClientMessage` and `XKeyMap` events are not supported (they will be in the not too distant future).

`Provided conveniences`

Since Motif requires the use of font lists for building non-trivial compound strings, there are some Lisp functions to ease the pain of building them:

`BUILD-SIMPLE-FONT-LIST (name font-spec)` Returns a font list of with the given name associated with the given font. For example,

```
(build-simple-font-list "MyFont" "8x13")
```

`BUILD-FONT-LIST (flist-spec)` This allows for the construction of font lists with more than one font. An example:

```
(build-font-list `(("EntryFont" ,entry-font-name)
                   ("HeaderFont" ,header-font-name)
                   ("ItalicFont" ,italic-font-name)))
```

There are certain callbacks which are of general use, and standard ones are provided for the programmer's convenience. For all callbacks except `QUIT-APPLICATION-CALLBACK`, you register some number of widgets with `ADD-CALLBACK`. These will be the widgets acted upon by the callback:

`QUIT-APPLICATION-CALLBACK ()` Callback to terminate the current application.

`DESTROY-CALLBACK` Destroys all the widgets passed to it.

`MANAGE-CALLABCK` Manages all the widgets passed to it.

`UNMANAGE-CALLBACK` Unmanages all the widgets passed to it.

`POPUP-CALLBACK` Calls popup on all widgets passed to it.

`POPDOWN-CALLBACK` Calls popdown on all widgets passed to it.

8 Some random notes

- When using functions such as REMOVE-CALLBACK, the client-data passed must be EQUAL to the client-data passed to ADD-CALLBACK.
- When using REMOVE-CALLBACK, etc., the function may be supplied as either 'FUNCTION or #'FUNCTION. However, they are considered different so use the same one when adding and removing callbacks.
- You cannot directly access the XmNitems resources for List widgets and relatives. Instead, use (SET-ITEMS <widget> ....) and (GET-ITEMS <widget>).

9 Things that are missing

- Real documentation
- Support for XClientMessage and XKeyMap events
- Callback return values (e.g. XmTextCallback's)
- Ability to send strings longer than 4096 bytes.

All these things should start appearing in the next few weeks[1]

10 A brief example

```
(defun my-callback (widget call-data quit)
  (format t "Got callback on ~A~%" widget)
  (format t "Callback reason was ~A~%" (any-callback-reason call-data))
  (format t "Quit button is ~A~%" quit))

(defun test-init ()
  (let* ((shell (create-application-shell))
 (rc (create-row-column shell "rowColumn"))
 (quit (create-push-button-gadget rc "quitButton"
  :label-string "Quit"))
 (button (create-push-button-gadget rc "button"
    :highlight-on-enter t
    :shadow-thickness 0
    :label-string "This is a button")))

    (add-callback quit :activate-callback #'quit-application-callback)
```

[1] NOTE: This document was written at CMU many years ago. The "formatter" does not make any guarantees about the validity of this claim.

```
    (add-callback button :activate-callback 'my-callback quit)

    (manage-child rc)
    (manage-children quit button)
    (realize-widget shell)))

(defun test ()
  (run-motif-application 'test-init))
```