General Design Notes on the Motif Toolkit Interface

June 17, 2001

1 Data Transport

1.1 Packet format

- **Header:**

<table>
<tr><td>32 bits</td><td>serial number</td><td rowspan="4">This header takes 12 bytes</td></tr>
<tr><td>16 bits</td><td>sequence position</td></tr>
<tr><td>16 bits</td><td>sequence length</td></tr>
<tr><td>32 bits</td><td>packet length (including header)</td></tr>
</table>

- **Data:**

  (packet_length - 12) bytes of information

- Packets have a fixed maximum size (4k).
- Packets a grouped together to form random length messages. The sequence length refers to how many packets comprise the message, and each packet is tagged with its position in that sequence.
- All packets in the same message have the same serial number.
- Messages are built up as their constituent packets arrive. It should be possible to interleave the packets of different messages and still have the individual messages be constructed properly.
- It is tacitly assumed that packets arrive in their proper sequence order.
- A packet with a sequence position/length field denoting [0 of 0] is a cancellation packet. The message having that serial number should be discarded.

1.1.1 Data format

Each data entry in a message is represented as:

8	bits	type tag
24	bits	immediate data
	rest	other data (if necessary)

2 Greeting Protocol

When a Lisp process first establishes a connection to the server, it sends a 16 bit quantity which represents "1" to it. The server using this to decide whether to byte swap words when sending them to Lisp. The general policy is that all data is presented to the Lisp process in the order that Lisp uses.

Following the byte swapping information, the Lisp process sends an initial message which contains:

- A string giving the target X display name
- A string for the application name
- A string for the application class

3 Request Protocol

Request format:

16	bits	request opcode
8	bits	request flags (0=nothing, 1=require confirm)
8	bits	argument count (unused)

At the moment, the request flags field is used only to indicate whether the Lisp client desires a confirmation message when the request is finished processing. If the request returns any values, this counts as the confirmation. Otherwise, an empty confirmation message will be sent.

Server reply format:

32	bits	response tag
	rest	return data (if any)

The response tag can have the following values:

TAG	MEANING
CONFIRM_REPLY	confirmation (for synchronization)
VALUES_REPLY	return values from a request
CALLBACK_REPLY	a widget callback has been invoked
EVENT_REPLY	an X event handler has been invoked
ERROR_REPLY	an error has occurred
WARNING_REPLY	a non-fatal problem has occurred
PROTOCOL_REPLY	a protocol callback has been invoked

4 Object Representations

4.1 Data format in message

Accelerators	32	bit	integer ID
Atom	32	bit	Atom ID
Boolean	24	bit	immediate data
Color	24	bit	immediate data (Red value) followed by 2 16 bit words for Green and Blue
Colormap	32	bit	Colormap XID
Compound Strings	32	bit	address
Cursor	32	bit	Cursor XID
Enumeration	24	bit	immediate integer
Font	32	bit	Font XID
Font List	32	bit	integer ID
Function	24	bit	immediate token
Int	32	bit	integer
List	24	bit	immediate data (length) followed by each element recorded in order
Pixmap	32	bit	Pixmap XID
Short	24	bit	immediate integer
(1) Strings	24	bit	immediate data (length of string including `'\0'`) followed by string data padded to end on a word boundary ... *or*...
(2) Strings	24	bit	immediate token (for common strings)
Translations	32	bit	integer ID
Widgets	32	bit	integer ID
Window	32	bit	Window XID

For objects such as translations, widgets, accelerators, font lists, and compound strings, the 32 bit ID is just the address of the object in the C server process. They are represented in Lisp by structures which encapsulate their ID's and provide them with Lisp data types (other than simply INTEGER).

5 Information in widget structure

- integer ID for identifying the widget to the C server
- widget class keyword (e.g. :FORM, :PUSH-BUTTON-GADGET, :UNKNOWN)
- parent widget
- list of (known) children
- USER-DATA slot for programmer use
- list of active callback lists
- list of active protocol lists
- list of active event handlers

The last three are for internal use in cleaning up Lisp state on widget destruction

6 Callback handlers

A callback handler is defined as:

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

The WIDGET argument is the widget for which the callback is being invoked.
The CLIENT-DATA &rest argument allows the programmer to pass an arbitrary number of Lisp objects to the callback procedure[1].
The CALL-DATA argument provides the information passed by Motif regarding the reason for the callback and any other relevant information.
The XEvent which generated the event may be accessed by:

```
(with-callback-event (event call-data)
   ....)
```

Action procedures are used in translation tables as:

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

Action procedures may access their event information by:

[1] **Note:** this deviates from CLM and Motif in C.

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

Where callback data is passed in structures, `XEvents` are represented as aliens. This is because `XEvents` are rather large. This saves the consing of large structures for each event processed.

Actions to be taken after the callback handler terminates the server's callback loop can be registered by:

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

7 Structure of the Server

When the server process is started, it establishes standard sockets for clients to connect to it and waits for incoming connections. When a client connects to the server, the server will fork a new process (unless `-nofork` was specified on the command line) to deal with incoming requests from the client. The result of this is that each logical application has its own dedicated request server. This prevents event handling in one application from blocking event dispatching in another.

Each request server is essentially an event loop. It waits for an event to occur, and dispatches that event to the appropriate handlers. If the event represents input available on the client connection, it reads the message off the stream and executes the corresponding request. If the event is an X event or a Motif callback, relevant information about that event is packed into a message and sent to the Lisp client. After sending the event notification, the server will enter a callback event loop to allow processing of requests from the client's callback procedure. However, during the callback event loop, only input events from the client will be processed; all other events will be deferred until the callback is terminated.

The server supports a standard means for reading and writing data objects into messages for communication with the Lisp client. For every available type of data which may be transported there are reader and writer functions. For instance, `WIDGET` is a valid type for argument data. Two functions are defined in the server: `message_read_widget()` and `message_write_widget()`. To allow for a more generalized interface to argument passing, the server defines the functions `toolkit_write_value()` and `toolkit_read_value()`. These functions are passed data and a type identifier; it is their job to look up the correct reader/writer function. Clearly, if the type of an argument is known at compile time then it is best to use the specific reader/writer functions. However, if such type information is not known at compile time, as is the case with arbitrary resource lists, the higher level `toolkit_xxx_value()` functions are the only available options.

8 Structure of the Client

...

9 Adding New Requests to the System

In order to add a new function to the toolkit interface, this new function must be declared in both C and Lisp.

Lisp provides a convenient macro interface for writing the necessary RPC stub. The form of this definition is:

```
(def-toolkit-request <C name> <Lisp name> <:confirm|:no-confirm>
  "Documentation string"
  (<arguments>)
  (<return-values>)
  <optional forms>)
```

Entries in the argument list should be of the form `(<name> <type>)`. The return value list is simply a list of types of the return value(s). Any forms supplied at the end will be executed in a context where the arguments are bound to the given names and the return value is bound to `RESULT` (if there was only one) or `FIRST`, `SECOND`, ..., `FOURTH` (for up to 4 return values). At the moment, the interface does not support any more than 4 return values. You must also specify a value for the confirmation option (`:CONFIRM` or `:NO-CONFIRM`). If you expect return values, you must specify `:CONFIRM` in order to receive them. Otherwise, you may specify `:NO-CONFIRM`. Use of `:NO-CONFIRM` allows for increased efficiency since the client will issue a request but not wait for any response. All function prototypes should be placed in the prototypes.lisp file. A few examples of request prototypes:

```
(def-toolkit-request "XtSetSensitive" set-sensitive :no-confirm
  "Sets the event sensitivity of the given widget."
  ;;
  ;; Takes two arguments:  widget and sensitivep
  ((widget widget) (sensitivep (member t nil)))
  ;;
  ;; No return values expected
  ())

(def-toolkit-request "XtIsManaged" is-managed :confirm
  "Returns a value indicating whether the specified widget is managed."
```

```
  ;;
  ;; Takes one argument: widget
  ((widget widget))
  ;;
  ;; Expects one return value (which is a boolean)
  ((member t nil)))

(def-toolkit-request "XmSelectionBoxGetChild" selection-box-get-child
                     :confirm
  "Accesses a child component of a SelectionBox widget."
  ;;
  ;; Takes two arguments: w and child
  ((w widget) (child keyword))
  ;;
  ;; Expects a return value which is a widget
  (widget)
  ;;
  ;; Now we execute some code to maintain the state of the world.
  ;; Given that this widget may be one we don't know about, we must
  ;; register it as the child of one we do know about.
  (widget-add-child w result)
  (setf (widget-type result) :unknown))
```

After adding a request prototype in Lisp, you must add the actual code to process the request to the C server code. The general form of the request function should be:

```
int R<name>(message_t message)
{
  int arg;
  ...
  toolkit_read_value(message,&arg,XtRInt);
  ...
}
```

Where `<name>` is the C name given in the request prototype above. You must also add an entry for this function in the functions.h file. An example of a standard request function is:

```
int RXtCreateWidget(message_t message)
{
  String name;
  WidgetClass class;
  Widget w,parent;
  ResourceList resources;
```

```
  toolkit_read_value(message,&name,XtRString);
  toolkit_read_value(message,&class,XtRWidgetClass);
  toolkit_read_value(message,&parent,XtRWidget);

  resources.class = class;
  resources.parent = parent;
  toolkit_read_value(message,&resources,ExtRResourceList);

  w = XtCreateWidget(name,class,parent,
     resources.args,resources.length);
  reply_with_widget(message,w);
}
```

Certain standard functions for returning arguments are provided in the file `requests.c`; `reply_with_widget()` is an example of these.

10 Summary of differences with CLM

X objects (e.g. windows, fonts, pixmaps) are represented as CLX objects rather than the home-brewed representations of CLM. As a consequence, this requires that CLX be present in the core. If this were to cause unacceptable core bloat, a skeletal CLX could be built which only supported the required functionality.

Stricter naming conventions are used, in particular for enumerated types. A value named `XmFOO_BAR` in C will be called `:foo-bar` in Lisp, consistently. Abbreviations such as `:form` (for `:attach-form`) are not allowed since they are often ambiguous. Where CLM abbreviates callback names (e.g. `XmNactivateCallback` becomes `:activate`), we do not (e.g. `:activate-callback`).

Some differently named functions which can be resolved without undo hassle.

Passing of information to callbacks and event handlers. In CLM, callback handlers are defined as:

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

The `CLIENT-DATA` argument is some arbitrary data which was stashed with the callback when it was registered by the application. The call-data represents the call-data information provided by Motif to the callback handler. Each data item of the callback information is passed as a separate argument. In our world, callback handlers are defined as:

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

The call-data is packaged into a structure and passed as a single argument and the user is allowed to register any number of items to be passed to the callback as client-data. Being able to pass several items of client-data is more convenient for the programmer and the packaging of the call-data information is more appealing than splitting it apart into separate arguments. Also, CLM only transports a limited subset of the available callback information. We transport all information. Event handlers differ in the same way. The client-data is the `&rest` arg and the event info is packaged as a single object. Accessing the generating event in a callback handler is done in the following manner:

```
(defun handler (widget call-data &rest client-data)
  (with-callback-event (event call-data)
    ;; Access slots of event such as:
    ;;  (event-window event) or
    ;;  (button-event-x event)
  ))
```