Puppeteer

Inherits From: Object

Declared In: include/Puppeteer.h

Class Description

Puppeteer objects are used to remotely control other applications. They allow events to be posted to the target application just as if the events were performed by a human using the mouse and keyboard. In this way, programs can be easily written to provide, for example, command line access to NeXTSTEP applications. Puppeteer is similar to the journaling facility described in **NXJournaler** except that the events are not recorded; instead they are specified programatically.

A Puppeteer object is typically used as follows:

```
/*
 * Allocate a puppeteer object to control the Terminal application.
 */
puppet = [Puppeteer connectToApp:"Terminal" launch:YES];
if (!puppet) {
    fprintf(stderr, "Could not connect to Terminal\n");
    exit(1);
   }

/*
 * Attach the strings. The app will then be ready to accept events from us.
 * It will be unable to respond to real events until a releaseStrings is
 * performed.
```

```
 */
[puppet attachStrings];

/*
 * Create a new window by posting command-n.
 */
[puppet postKeyCode:'n' window:NX_KEYWINDOW flags:NX_COMMANDMASK];

/*
 * Output a string to the new terminal window.
 */
[puppet postKeyboardString:"echo hello world\n" window:NX_KEYWINDOW flags:0];

/*
   * Release strings. This is necessary for the app to continue to respond to
   * real user events.
   */
[puppet releaseStrings];

[puppet free];
```

The attach/release strings combination may be repeated many times before the object is freed, but there must always be a final releaseStrings, otherwise the application will no longer respond to real user events.

Instance Variables

char ***appName**;
id **appSpeaker**;
id **journalSpeaker**;
port_t **appPort**;
BOOL **enabled**;
int **pid**;
int **context**;

appName	The application name.
appSpeaker	The main puppet speaker.
journalSpeaker	The puppet's journal speaker.
appPort	Port used by appSpeaker.
enabled	YES if strings are attached.
pid	The application's pid.
context	The application's postscript context.

Method Types

Allocating and initializing a new Puppeteer instance
+ connectToApp:launch:

Initializing a new Puppeteer instance
- connect:launch:

Taking control
- attachStrings
- releaseStrings

Posting events
- postEvent:
- postKeyboardString:flags:
- postKeyboardEvent:window:flags:charCode:
- postKeyCode:window:flags:
- postMouseEvent:window:flags:x:y:click:
- postSingleClick:window:flags:x:y:
- postDoubleClick:window:flags:x:y:

- postTripleClick:window:flags:x:y:
- postActivate:
- dragWindow:deltaX:deltaY:
- ping

Obtaining window information
- windowList
- windowCount
- windowForPseudoNumber:
- keyWindow
- mainWindow
- mainMenu

Obtaining other information
- getPid
- getContext
- appSpeaker

Class Methods

connectToApp:launch:

+ **connectToApp:**(const char *)*theName*
  launch:(BOOL)*launch*

Return an instance of puppeteer connected to the given app, or nil on failure. This is the normal way to allocate a Puppeteer instance. If *launch* is YES the application will be launched if it is not already running.

See also: - **connect**:**launch:**

Instance Methods

appSpeaker

- **appSpeaker**

Returns the speaker that communicates with the main application Listener. This may be used to send standard speaker/listener messages such as **unhide**.

attachStrings

- **attachStrings**

Prepares the Puppeteer object for sending events. This must be called prior to posting events and a matching **releaseStrings** must be given when the user has finished sending events. Any number of **attachStrings/releaseStrings** combinations may be used before the object is freed.

connect:launch:

- (BOOL)**connect:**(const char *)*theName*
  launch:(BOOL)*launch*

Return an instance of puppeteer connected to the given app, or nil on failure. This is used by +**connectToApp**:**launch:** to initialize a Puppeteer instance. If *launch* is YES the application will be launched if it is not already running. Returns YES if connection is successful.

See also: + **connectToApp**:**launch:**

dragWindow:deltaX:deltaY:

- **dragWindow:**(int)*winNumber*
  deltaX:(double)*x*
  deltaY:(double)*y*

Drags the window *winNumber* by the specified offset. *winNumber* may be either a local postscript window number or a pseudo window number such as NX_KEYWINDOW.

getContext

- (int)**getContext**

Returns the remote application's postscript context.

getPid

- (int)**getPid**

Returns the remote application's process id.

keyWindow

- **keyWindow**

Returns a **WindowInfo** object corresponding to the remote application's key window (NX_KEYWINDOW).

mainMenu

- **mainMenu**

Returns a **WindowInfo** object corresponding to the remote application's main menu window (NX_MAINMENU).

mainWindow

- **mainWindow**

Returns a **WindowInfo** object corresponding to the remote application's main window (NX_MAINWINDOW).

ping

- **ping**

Sometimes it is necessary to be sure that the application has processed all the events that have been sent to it. This method provides that mechanism. It sends a standard speaker message to the remote application to which a reply must be given. Thus this method will only return when the target app has processed all outstanding requests.

postActivate:

- **postActivate:**(BOOL)*activate*

If *activate* is YES, posts an event to the remote application making it the active application, else posts an event to deactivate the application. It is sometimes necessary for certain mouse clicks to work for the application to be the active application. One example of this is where clicking on a button in a window of a non-active application merely results in that application becoming active and the window becoming the key window. A further click is required to activate the button.

postDoubleClick:flags:x:y:

- **postDoubleClick:**(int)*window*
  flags:(int)*flags*
  x:(double)*x*
  y:(double)*y*

Posts the events representing a double mouse click to the remote application. This method is just a shorthand for four separate mouse events: a mouse down/up pair with click set to one, followed by another mouse down/up pair with click set to two. *window* specifies which window the mouse click occurs in and may be either a local postscript window number or a pseudo window number such as NX_KEYWINDOW. *flags* may be used to set event flags (see dpclient/event.h). *x* and *y* specify the coordinates of the mouse position relative to the bottom left corner of the window.

postEvent:

- **postEvent:**(NXEvent *)*event*

This is the main method through which events are posted to the remote application. See the description of the NXEvent structure for more details.

postKeyboardEvent:window:flags:charCode:

- **postKeyboardEvent:**(int)*eventType*
  window:(int)*window*
  flags:(int)*flags*
  charCode:(char)*charCode*

Posts a keyboard event. *eventType* is the keyboard event type, either NX_KEYDOWN or NX_KEYUP, *window* is the window number where the event is posted, *flags* are the event flags (eg NX_COMMANDMASK) and *charCode* is the character code.

See also: - **postKeyCode:window:flags:**

postKeyboardString:flags:

- **postKeyboardString:**(const char *)*keyString*
  flags:(int)*flags*

Posts the given string to the key window of the remote application. This method simply calls **postKeyboardEvent:window:flags:charCode** twice (key down and key up) for every character in the string.

postKeyCode:window:flags:

- **postKeyCode:**(char)*charCode*
  window:(int)*window*
  flags:(int)*flags*

Posts a pair of keyboard events (NX_KEYDOWN followed by NX_KEYUP) to represent the typing of a single character given by *charCode*. *window* is the window number where the event is posted and *flags* are the event flags (eg NX_COMMANDMASK).

postMouseEvent:window:flags:x:y:click:

- **postMouseEvent:**(int)*eventType*
  window:(int)*window*
  flags:(int)*flags*
  x:(double)*x*
  y:(double)*y*
  click:(int)*click*

Posts a mouse event to the remote application. *eventType* is the type of event (eg NX_MOUSEDOWN), *window* is the window number in which the event occurs (see descriptions above). *flags* may be used to specify any further event flags. *x* and *y* are the coordinates of the mouse event relative to the bottom left corner of the window. For a single mouse click, *click* should be set to one for the mouse down/up pair. A double mouse click consists of a two pairs, the first with *click* set to one, and the second with *click* set to two. Similarly, a triple mouse click has three down/up pairs with the last pair having a *click* value of three.

postSingleClick:flags:x:y:

- **postSingleClick:**(int)*window*
  flags:(int)*flags*
  x:(double)*x*
  y:(double)*y*

Posts the events representing a single mouse click to the remote application. This method is just a shorthand for a mouse down/up pair with click set to one. See the description of the parameters in **postDoubleClick:flags:x:y:**.

postTripleClick:flags:x:y:

- **postTripleClick:**(int)*window*
  flags:(int)*flags*
  x:(double)*x*
  y:(double)*y*

Posts the events representing a triple mouse click to the remote application. This method is just a shorthand for six separate mouse events: a mouse down/up pair with click set to one, followed by another mouse down/up pair with click set to two, followed by another pair with click set to three. See the description of the parameters in **postDoubleClick:flags:x:y:**.

releaseStrings

- **releaseStrings**

Frees the remote application so that it can continue to respond to real user events. This method **must** be called when the controlling application has completed a session of posting events.

windowCount

- (int)**windowCount**

Returns the number of windows belonging to the remote application. One use of this is to determine if a particular mouse click has resulted in a new window appearing.

windowForPseudoNumber

- **windowForPseudoNumber:**(int)*pseudoNumber*

Returns a **WindowInfo** object for the given pseudo window number (eg NX_KEYWINDOW), or nil if it can't be determined.

windowList

- **windowList**

Returns a list of the remote application's windows. A new list is created each time this method is called, and it is the caller's responsibility to free it and it's contents. Each object is of the **WindowInfo** class.