Interactive Advertising Bureau

Mobile Rich-media Ad Interface Definitions (MRAID) v.1.0

Final Release
October 20, 2011

Table of Contents

Contributors

The IAB MRAID Working Group includes representatives from the following companies:

24/7 Real Media, Inc.
4INFO
AccuWeather.com
AdMarvel
AdMeld
Adobe Systems Inc.
CBS Interactive
Celtra
comScore
Crisp Media
Dow Jones & Company
FreeWheel
Goldspot Media
Google
IDG
ImServices Group
inMobi
Jumptap
Medialets
MediaMind
Microsoft Advertising
Mixpo
NBC Universal Digital Media
Nexage
PointRoll
PricewaterhouseCoopers
Rhythm NewMedia
Sprout
TargetSpot
The New York Times Co.
The Weather Channel
Time Inc.
Turner Broadcasting System, Inc.
Yahoo!, Inc.

Acknowledgement

The IAB acknowledges the contributors to the ORMMA.org API project, which provided a starting point for this document. ORMMA.org is a group of industry thought leaders who have worked together since Spring 2010 to develop and test a complete and versatile mobile rich media ad API.

Adam Schuetz, AdMarvel
Dennis Doughty, Jumptap
Jon Badenell, The Weather Channel
Nathan Carver, Crisp Media
Neal Karasic, Jumptap
Philippe Laporte, Goldspot Media
Robert Hedin, The Weather Channel
Todd Pasternack, Pointroll
Wook Chung, Google
Xavier Facon, Crisp Media

About MRAID

The Interactive Advertising Bureau ("IAB"), its members and other significant contributors joined together to create this document, a standard interface specification for mobile rich media ads. The goal of the Mobile Rich-media Ad Interface Definition (MRAID) project is to address known interoperability issues between publisher mobile applications, different ad servers and different rich media platforms.

IAB Contact Information

Joe Laszlo, Deputy Director, IAB Mobile Marketing Center of Excellence, mobile@iab.net

Executive Summary

As rich media display advertising in mobile applications and on the mobile web has become more popular over the last several years, various innovative companies have accepted the challenge of creating an ecosystem for mobile ad serving. Innovation in mobile rich media ad serving has led to many exciting possibilities for content publishers and advertisers, but it has also created inefficiencies that often delay and inhibit the optimal monetization of content.

Simplifying the process for designers of ad creatives significantly increases the likeliness that agencies will leverage mobile into their media buys. Advertisers want to review compelling creative, approve it and decide to buy a specific inventory of mobile media, regardless of which device platform, application, or technology is used to display the media.

General SDK Requirements for Supporting MRAID

This section details the requirements on an in-app ad-serving SDK that is MRAID compatible.

It is expected that an implementation would be in two parts. The first part defines a native container for rich media ads to display in apps and the second part defines a JavaScript controller for ad creatives to interact with. The native container encapsulates an HTML and JavaScript enabled Web browser view, such as iOS's UIWebView, and the controller serves as a bridge that can integrate HTML-based ads with the native capabilities. Actual implementations may vary.

When planning, key design considerations are:

- Access to the device's native features (orientation, location, acceleration, etc.)
- On and off-line Ad viewing and metrics
- Industry standard Ad development (HTML and JavaScript)
- Progressive complexity (simple things are simple, complex things are possible but harder)

Technical Audience

The specifications are technical by nature, but are not intended to limit innovation. This document is intended for Publishers or SDK vendors and addresses the needs of the Ad Designers.

Native Application Developer

There are no requirements in this specification for app developers. They should follow the instructions provided by their SDK developer for integrating ads into their application.

SDK Developer

SDK builders have a number of responsibilities outside this recommendation. (See "out-of-scope.") As mentioned, it is expected that the SDK developer will provide two interfaces to implement these recommendations: a container for the native developer to integrate via the SDK and a controller for the ad developer to use directly.

This document outlines the requirements of the controller needed by the ad developer. It is the intention of the writers that these concepts can be managed with a facade layer for existing SDKs.

Ad Developer
There are no creative requirements in this document for ad designers and developers besides the use of web standards. Ad developers who use the methods in this specification can provide consumers with a rich media experience across platforms and publishers.

It is important for ad designers to recognize that calls to the native device must be asynchronous by design. For most web developers, this is analogous to AJAX programming.

Out of Scope

Each SDK provides unique features sets to developers. This document outlines a minimum set of features for interoperability and does not define features that may also be part of an SDK such as

- Retrieving the ad from Ad Server, Ad Network, or local resources
- Reporting
- IDE integrations
- Security / Privacy
- Internationalization
- Error reporting
- Logging
- Billing and payments
- Ad dimensions and ad behavior
- Downloading of assets to the local file system for caching or off-line use

Of course, the SDK developer must implement the ability to render web content in the area intended for the ad unit. For most environments, this capability is already available as a web view component although the developer may have to develop additional functions to support these specifications.

It is the intent of the writers that SDK vendors are not limited to delivering only the features outlined in the API. They should continue to innovate and present features that differentiate them in the marketplace.

Standard Web Technologies

For interoperability, only web compatible languages should be used for markup and scripting languages. This document assumes HTML/JavaScript/CSS. The ad designer should be able to develop and test the ad unit in a web browser. If designers use tags, styles and functions which are compatible with only one browser (such as CSS3 on WebKit), then the ad should be targeted to compatible devices.

When newer web standards can provide consistency, ad designers are encouraged to use them. This may include protocols like sms: and tel:, as well as some widely implemented portions of the as-yet unfinished HTML5 specification. Designers need to be aware that in these cases, the expected protocols and implementations may not be truly interoperable across all devices and platforms.

Ad Server Requirements

The ad server used to traffic rich media ads should support HTML ads with JavaScript.

Requirements for Ad Rendering

Display of HTML Ads – Ad View Container

An MRAID-compatible SDK must display any HTML ad. Ad designers that are not concerned with rich media or accessing native features can simply provide simple HTML for display in the application.

The SDK should invoke an HTML with JavaScript rendering engine for rendering ads. In this document, that engine will be called the "web viewer". As possible, the web viewer should incorporate the capabilities of the device web browser. For example, iOS developers may use UIWebView. A given App view can have one or more Ad View Containers that will all act independently of one another.

Requirements for Ad Developers

Display Control for Rich Media Ads – Ad Controller

Additional creative requirements will register to use the MRAID API on an as-needed basis. This supports the concept of progressive complexity.

So, the ad designer is in control of the ad display, but uses the MRAID API when the ad needs to communicate with the SDK, or, in future MRAID releases, the device's native layer. The internal interaction is hidden from both the Ad developer and the App developer.

An ad that does not utilize any device features does not need to use the MRAID API at all. Some of the things an ad uses MRAID's API for are:

- Capturing user actions for:
    - Opening an embedded Web browser
    - Expanding an ad that grows from a banner to a larger size
    - Clicking within an Ad triggering an action
    - Storing metrics when off-line for transmission when the device is back on-line

- Moving or resizing the Container and Web View for:
    - Modal take-over of the entire display
    - Modal or modeless "fly-out" of the Ad from the original Container bounds

Future versions of MRAID will extend the capabilities of the API, to enable rich media ads to:

- Query the SDK for supported features, such as:
    - Accelerometer
    - Compass
    - GPS
    - Specific gestures
    - Whether the device is currently on or off-line
    - Etc.
- Register JavaScript Event Listener functions to be called by the Container for:
    - Accelerometer readings
    - Touch events
    - Gestures
    - Etc.

Lifecycle Examples

Simple Ad Lifecycle Example

In the simplest example, an application developer adds an MRAID-compatible View to their application UI either programmatically or with an interface builder.

When the app developer wants their app to display an ad, they rely on the SDK to retrieve an HTML ad. The View then displays the resulting HTML. If the ad does not make use of the MRAID API, then it will behave as a normal HTML ad and any links will open in the device's default web browser. This fallback functionality allows fixed sized, non-interactive web creatives to be used without modification.

Rich Media Ad Lifecycle Example

In a more complex example, the Ad Designer uses the JavaScript API to take communicate with the native layer and interact with features of the device and OS. An initial ad displays first as a small shim with a static background or a "Loading..." message.

The ad may use local assets if they are available or request that assets be downloaded and executed locally.

As an example, when the user touches the ad, JavaScript uses the MRAID API to notify the app (via the SDK) that the ad is expanding so that it can stop anything that the user will not be able

to interact with. The SDK then resizes the web view to take up the entire screen of the device or the full size of the expanded ad. The SDK reserves a space at the top right corner of the expanded ad for an SDK-provided close control, and will either supply the close indicator or, if the ad specifies, will allow the ad to supply the indicator in creative.

When the user is done with the expanded ad, they click a close button that causes the ad to unregister the event listeners, resize the ad to its original size, display the ad's banner state, and notify the app that it can resume. If the device was off-line when the user interaction took place, any metrics called are cached by the SDK until the device is on-line again, and then the tracking calls are sent.

The case of an interstitial ad is very similar. The ad can use the MRAID API to query the SDK as to whether it is visible onscreen or not, waiting until it is on before it takes other actions. As with an expandable, the SDK reserves a space at the top right corner of the expanded ad for an SDK-provided close control, and will either supply the close indicator or, if the ad specifies, will allow the ad to supply the indicator in creative. When the user is done with the interstitial, they can tap the close button, which in this case changes the ad's state to "hidden," unregisters any event listeners, and notifies the app to resume.

MRAID Versions

The adoption of MRAID throughout the ad community is a high priority and essential for the success of mobile rich media across platforms. For this reason, IAB will release the full feature set of MRAID in versions. This will allow SDK vendors to meet the compliance standards of the MRAID API in a consistent way and prevent possible fragmentation inherit in implementing only a portion of the standard.

Version 1

The methods and events identified in this document provide a minimum level of requirements for rich media ads, primarily to display HTML ads that can change size in a fixed container.

Although Version 1 addresses a minimum level of functionality, the standards of MRAID remain high. In this and particularly in future versions of the API, the IAB focuses on

- **High interoperability** – ads developed to run in one MRAID container can run on MRAID containers of multiple platforms and operating systems.

- **Graceful degradation** – ads developed to take advantage of all the MRAID features also have the capacity to downgrade gracefully as needed. This will be especially important as gaining access to device functionalities becomes part of MRAID's scope in the future.

- **Progressive complexity** – ad design using the API should be simple, adding complexity only as necessary.

For examples of ads that can be developed using the MRAID Version 1 API, please see the addendum.

Future Versions

Also in an addendum, this document outlines a number of features that are currently under discussion by the IAB MRAID working group. The final specification and timing for inclusion in a future version is subject to change. In general, the IAB intends for future versions of the MRAID specification to be agile and flexible in order to meet the practical needs of the mobile advertising industry.

Interface Requirements and Definitions

This page outlines all the methods and events that ad developers will have access to.

Methods

- addEventListener
- close
- expand
- getExpandProperties
- getPlacementType
- getState
- getVersion
- isViewable
- open
- removeEventListener
- setExpandProperties
- useCustomClose

Events

- error
- ready
- stateChange
- viewableChange

Identification

It is required that ads identify themselves as being MRAID compliant. This is done by adding an MRAID script reference at the top of the creative before any MRAID functions are referenced in the creative delivered to an ad server. In other words, the MRAID identification script reference must be discoverable as soon as possible by any MRAID-compliant container or SDK.

MRAID script reference

The MRAID comment follows HTML Javascript syntax so that both fully formed web pages and HTML fragments can be identified as MRAID ads.

```
<script src="mraid.js"></script>
```

Initialization

The Controller encapsulates and abstracts all interactions between the web layer and the native layer and identifies the container as compatible with these specifications. Ad designers must include the JavaScript identification reference for MRAID, but the actual JavaScript libraries are supplied by the container, and it is the responsibility of the SDK to ensure they are available to the ad in a timely fashion after the script reference is made.

The following summarizes step-by-step the actions that the ad and SDK take in the initial loading of the ad and the injection of MRAID.

1. Ad identifies itself as MRAID as early as possible with MRAID script tag.
   <script src="mraid.js"></script>
2. SDK/MRAID-compatible Container
   a. Optionally detects the script call
   b. Always provides the MRAID JavaScript bridge for MRAID ads
   c. Provides HTML display space with an MRAID State = “loading”
3. Ad listens for “ready” event with mraid.addEventListener('ready')
4. SDK/Container
   a. Completes loading ad into HTML display space
   b. Finishes any additional initialization for the JavaScript bridge and native layers
   c. Fires the MRAID “ready” event
   d. Changes the MRAID state to “default”
5. Ad's "ready" event listener is triggered and ad continues to execute JavaScript
6. In some conditions, ad may need to see if the MRAID state was updated to "default" before the listener could be registered. In this case, the ad should use mraid.getState() and then continue to execute JavaScript on success.

ready event

The ready event triggers when the SDK is fully loaded, initialized, and ready for any calls from the ad creative.

It is the responsibility of the MRAID-compliant SDK to prepare the API methods before the ad creative is loaded. This prevents a condition where the ad cannot register to listen for the ready event because the API methods are unavailable. While the SDK may load all of MRAID at once, at a minimum the SDK must be prepared to support the getState and addEventListener capabilities as early as possible in the ad loading process; otherwise there will be no way for the ad to register for the ready event. In the event that the SDK may still need more time to initialize settings or prepare additional features, ready should only fire when the SDK is completely prepared for any MRAID request.

The ad should always attempt to wait for the ready event before executing any rich media operations. Because of timing issues, such as the ready event firing before the ad has registered to listen, ad designers should use the ready event in conjunction with the getState() method.

Example

```
function showMyAd() {
 ...
}

if (mraid.getState() === 'loading') {
    mraid.addEventListener('ready', showMyAd);
} else {
    showMyAd();
}
```

"ready"

parameters:

- none

side effects:

- MRAID JavaScript library available to ad unit

return values:

- none

event triggered:

- none

getVersion method

The getVersion method allows the ad to confirm a basic feature set before display. This version number must correspond with the MRAID version specification and not the vendor's SDK version.

`getVersion() -> String`

parameters:

- none

return values:

- String – the MRAID specification that this SDK is certified against. For the current version of MRAID, getVersion() will return "1.0"

Preloading and Initial Display

It is up to the ad developer to provide simple HTML, such as an <img> tag, for the initial display of their ad while other assets are loaded in the background. This HTML will be displayed in the Container while JavaScript uses the Controller to request and invoke additional capabilities. Ultimately, the initial HTML display may be completely replaced by a rich media ad once all assets are ready, depending on the creative requirements.

Event Handling

Event handling is a key concept of this recommendation. Communicating between the web layer and native layer is asynchronous by nature. Through event handling, the ad designer is able to listen for particular actions and respond to those actions on an as-needed basis. These specifications advocate broadcast-style events to support the broadest range of features/flexibility with the greatest consistency.

The controller exposes these methods.

addEventListener method

Use this method to subscribe a specific handler method to a specific event. In this way, multiple listeners can subscribe to a specific event, and a single listener can handle multiple events. Additional listeners are part of the MRAID roadmap. For the current version, the events are:

value	**description**
ready	report initialize complete
error	report error has occurred
stateChange	report state changes
viewableChange	report viewable changes

`addEventListener(event, listener)`

parameters:

- event – string, name of event to listen for
- listener – function, function name (or anonymous function) to execute

return values:

- none

side effects:

- In future versions of MRAID, registering listeners for device features may power up sensors in the device that will reduce battery life.

removeEventListener method

Use this method to unsubscribe a specific handler method from a specific event. Event listeners should always be removed when they are no longer useful to avoid errors. If no listener function is provided, then all functions listening to the event will be removed.

```
removeEventListener(event, listener)
```

parameters:

- event – string, name of event
- listener – function, function name (or anonymous function) to be removed

return values:

- none

events triggered:

- none

Error Handling

When an error in the SDK occurs, the "error" event is thrown with diagnostic information about the event. Any number of listeners can monitor for errors of different types and respond as needed.

error event

This event is thrown whenever an SDK error occurs. The event contains a description of the error that occurred and, when appropriate, the name of the action that resulted in the error (in the absence of an associated action, the action parameter is null). JavaScript errors remain the full responsibility of the ad designer.

```
“error” -> function(message, action)
```

parameters:

- message: String, description of the type of error
- action: String, name of action that caused error

triggered by:

- anything that goes wrong

Controlling Ad Display

Besides the initial display, the ad developer may have a number of reasons to control the display.

- An application may load views in the background to help with latency issues so that an ad is requested, but not visible to the user.
- The ad may expand beyond the default size over the application content.
- The ad may return to the default size once user interaction is complete.

getState method**, stateChange** event

Each ad container (or Web View) has a state that is one of the following:

value	description
loading	the SDK is not yet ready for interactions with the Controller
default	the initial position and size of the ad container as placed by the application and SDK
expanded	the ad container has expanded to cover the application content at the top of the view hierarchy
hidden	the ad container no longer displays the ad

The getState method returns the current state of the ad container, returning whether the ad container is in its default, fixed position or is in an expanded, larger position.

The stateChange event fires when the state is changed programmatically by the ad or by the environment. This event is thrown when the Ad View changes between default, expanded, and hidden states as the result of an expand() or a close(). The SDK may also close an ad as the result of a user or system action, such as resuming from background.

The effect on state from calling expand() and close() are defined in this table.

state	**expand()**	**close()**
loading	no effect	no effect
default	state changed to "expanded"	state changed to "hidden"
expanded	no effect	state changed to "default"
hidden	no effect	no effect

`getState() -> String`

parameters:

- none

return values:

- String: "loading", "default", "expanded", or "hidden"

related events:

- stateChange

```
"stateChange" -> function(state)
```

parameters:

- state - String, either "loading", "default", "expanded", or "hidden"

triggered by:

- expand, close, or the app

getPlacementType() method

For efficiency, ad designers sometimes flight a single piece of creative in both banner and interstitial placements. So that the creative can be aware of its placement, and therefore potentially behave differently, each ad container has a placement type determining whether the ad is being displayed inline with content (i.e. a banner) or as an interstitial overlaid content (e.g. during a content transition). The SDK returns the value of the placement to creative so that creative can behave differently as necessary. The SDK does not determine whether a banner is an expandable (the creative does) and thus does not return a separate type for expandable.

value	description
inline	the ad placement is inline with content (i.e. a banner) in the display
interstitial	the ad placement is over laid on top of content

```
getPlacementType() -> String
```

parameters:

- none

return values:

- String: "inline", "interstitial"

related events:

- none

isViewable method**, viewableChange** event

In addition to the state of the ad container, it is possible that the container is loaded off-screen as part of an application's buffer to help provide a smooth user experience. This is especially prevalent in apps that employ scrolling views or in ads that display interstitials, for example between levels of a game.

The isViewable method returns whether the ad container is currently on or off the screen. The viewableChange event fires when the ad moves from on-screen to off-screen and vice versa.

In any situation where an ad may be loaded offscreen, it is a good practice for the ad to check on its viewable state and/or register for viewableChange before taking any action.

`isViewable() -> boolean`

parameters:

- none

return values:

- boolean - true: container is on-screen and viewable by the user; false: container is off-screen and not viewable

related events:

- viewableChange

`“viewableChange” -> function(boolean)`

parameters:

- boolean - true: container is on-screen and viewable by the user; false: container is off-screen and not viewable

triggered by:

- a change in the application view controller

expand method

The expand method will cause an existing Web View (for one-part creatives) or a new Web View (for two-part creatives) to open at the highest z-order in the view hierarchy. The expanded view can either contain a new HTML document if a URL is specified, or it can reuse the same document that was in the default position. While an ad is in an expanded state, the default position will generally be obscured or inaccessible to the viewer, so the default position should take no action while the expanded state is available. Thus a complete implementation allows for ad designers to use one-part ads (where the banner and panel are part of one creative) and two-part ads (where the banner and panel are separate HTML creatives).

The expand method may change the size of the ad container, and will move state from "default" to "expanded" and fire the stateChange event. Calling expand more than once is permissible, but has no effect on state (which remains “expanded”).

An expanded view may cover all available screen area even though the ad creative may not (e.g. via a transparent or opaque overlay), or it may cover only a partial screen area. Issues of ad modality are left to the SDK and/or the app developer’s implementation of the SDK. At a minimum, however, the SDK should prevent new ads from loading during the expand state so that the user can complete any desired interactions with the ad creative without interruption. Other application-specific difficulties such as poorly built apps with multiple window objects, or

timers that change the content z-order, must be considered by SDK vendors when implementing the expand method.

An expanded view must provide an end-user with the ability to close the expanded creative. These requirements are discussed further in the description of closing expandable and interstitial ads, below.

Placement of the expanded ad on screen, especially when the expanded view can be placed in multiple locations, is left to the SDK and/or the app developer. For full-screen expands, all MRAID compliant SDKs will grant the full device screen space and will position the ad so it is fully visible. When the ad size is greater than the screen size, the SDK will center the ad vertically and horizontally–i.e., position the ad such that the center of the ad (midpoint top-to-bottom and midpoint left-to-right) is located at the center of the device screen. The SDK will size the Webview to be identical to the screen size of the device, causing outlying areas to be cropped.

When the expand method is called without the URL parameter, the current web view will be reused, simplifying reporting and ad creation. The original creative is not reloaded and no additional impressions are recorded. Implementing this definition allows for one-part creatives.

When the expand method is called with the URL parameter, a new web view will be used. Implementing this definition allows for two-part creatives.

`expand([URL])`

parameters:

- URL (optional): The URL for the document to be displayed in a new overlay view. If null, the body of the current ad will be used in the current webview.

return values:

- none

events triggered:

stateChange

Controlling expandProperties

The expand properties object is intended to provide additional features to ad designers. In MRAID v.1.0, expand properties that can be set by the ad designer are limited to the width and height of the ad creative in pixels, and whether the creative is supplying its own close indicator. The expandProperties are held in a JSON object that can be written and read by the ad.

```
expandProperties object = {
 “width” : integer,
```

```
 “height” : integer,
 “useCustomClose” : boolean,
 “isModal” : boolean (read only)
}
```

properties:

- width : integer – width of creative in pixels, default is full screen width
- height : integer – height of creative in pixels, default is full screen height. Note that when getting the expand properties before setting them, the values for width and height will reflect the actual values of the screen. This will allow ad designers who want to use application or device values to adjust as necessary.

- useCustomClose : boolean – true, SDK will stop showing default close graphic and rely on ad creative's custom close indicator; false (default), SDK will display the default close graphic. This property has exactly the same function as the useCustomClose method (described below), and is provided as a convenience for creators of expandable ads.
- isModal : boolean – true, the SDK is providing a modal container for the expanded ad; false, the SDK is not providing a modal container for the expanded ad; this property is read-only and cannot be set by the ad designer

getExpandProperties method

The getExpandProperties method returns the whole JSON expandProperties object.

Use this method to get the properties for expanding an ad.

```
getExpandProperties() -> JSON
```

parameters:

- none

return values:

- { ... } - this object contains the expand properties

events triggered:

- none

setExpandProperties method

The setExpandProperties method sets the whole JSON object.

```
setExpandProperties(properties)
```

Use this method to set the ad's expand properties, in particular the maximum width and height of the ad creative.

parameters:

- properties: JSON { ... } - this object contains the width and height of the expanded ad. For more info see properties object.

return values:

- none

events triggered:

- none

Closing Expandable and Interstitial Ads

An MRAID-compliant SDK must provide an end-user with the ability to close an expanded or interstitial ad. This is a requirement to ensure that users are always able to return to the publisher content even if an ad has an error. The ad designer may optionally provide additional design elements to close the expanded or interstitial view via the close() method, described below.

MRAID requires the location reserved for the close control be a 50x50 clickable area in the top-right corner of the ad. Reserving this location provides consistency for ad designers running campaigns across apps and rich media vendors. The default design of the SDK-controlled close indicator is left to the vendor/app publisher. Ad designers may optionally choose to provide the indicator for the SDK-supplied close capability, although the ad designer may not move that capability from the SDK's specified location. If the ad designer builds the close indicator into the creative they must specify so via the useCustomClose() method, or as a convenience by setting useCustomClose in the expandProperties() object. If the ad designer does not provide its own close indicator graphic within the creative, the SDK will supply its default close indicator. This SDK-supplied clickable area will be placed at the highest z-order possible, and must always be available to the end user.

If expand was used with a URL parameter (e.g., a two-part ad), then closing the ad must display the original content. If the SDK suspended the app when the ad changed to the expand state, then the SDK should notify the app to resume.

If the expanded or interstitial ad view was closed using the SDK-supplied close control, then the stateChange event is still fired and the app still notified to resume. Expanded ads must always listen for the stateChange event and adjust as necessary.

close method

The close method will cause the ad webview to downgrade its state. It will also fire the stateChange event. For ads in an expanded state, the close() method moves to a default state. For ads in a default state, the close() method moves to a hidden state. This method may be used by ad designers as an addition to the SDK-enforced close ability.

close()

parameters:

- none

return values:

- none

event triggered:

- stateChange

useCustomClose method

Although MRAID requires all implementing SDKs to provide a clickable area with a default "close" indicator graphic, it is possible for ad creators to use their own designs for the close indicator.

This method serves as a convenience method to the expand property of the same name. Setting the property or calling this method both have the same effect and can be used interchangeably. If an ad sets useCustomClose via both expand properties AND this method, whichever is invoked later will override the earlier setting. They signal the SDK to stop using the default close indicator.

For expanded ads, the designer does not need to call this method and would normally set the useCustomClose property in setExpandProperties().

For a stand-alone interstitial where there is no call to expand(), but there is still a close() requirement, the ad designer should call this method as early as possible.

Ad designers should be clear that an MRAID-compliant SDK is required to show the default close indicator until the useCustomClose method is called and/or the property is set.

useCustomClose(boolean)

parameters:

- true – ad creative supplies its own designs for the close area
- false – SDK default image should be displayed for the close area

return values:

- none

events triggered:

- none

Opening an Embedded Browser

If the ad wants to open an external mobile web site, or micro site, from an MRAID ad, it can call the open method which will open an embedded browser window in the application.

open method

The open method will display an embedded browser window in the application that loads an external URL. On device platforms that do not allow an embedded browser, the open method invokes the native browser with the external URL.

Note: This should be used only for external web pages that are not MRAID ads. The displayed page will not load the app's MRAID-compliant SDK and so the close method will not have any effect on the embedded browser. It can only be closed by the user selecting the close control for the window, which is implementation specific.

Use this method to open an HTML browser to an external web page. This may launch an external browser, depending on the SDK implementation. To place the ad over content, use the expand() method instead.

The native browser controls – back, forward, refresh, close – will always be present. For reporting, open should always be used for click through actions.

open(URL)

parameters:

- URL - String, the URL of the web page

return values:

- None

Addendum 1 – Candidates for Future Versions

This addendum outlines additional methods and events currently under discussion by the IAB's MRAID working group for possible inclusion in future versions on the MRAID API. **None of the methods or events on pages 25-44 is a part of MRAID 1.0, and all will require discussion by the IAB working group before they are included in future MRAID versions.** The IAB expects to begin work on MRAID 2.0 with the working group immediately following the completion of MRAID 1.0.

Methods

- createEvent
- getDefaultPosition
- getExpandProperties
- getKeyboard
- getHeading
- getLocation
- getMaxSize
- getNetwork
- getOrientation
- getScreenSize
- getShakeProperties
- getSize
- getTilt
- hide
- makeCall
- openMap
- playAudio
- playVideo
- request
- resize
- sendMail
- sendSMS
- setExpandProperties
- setShakeProperties
- show
- storePicture
- supports

Events

- headingChange
- keyboardChange
- locationChange
- networkChange
- orientationChange
- response
- screenChange
- shake
- sizeChange
- tiltChange

Resizing an Ad

The open() method is intended for click-throughs where an entire web site is loaded in a micro browser. The expand() method is intended for ads that expand to cover the content of the application.

In addition to these, the resize() method is intended for ads that grow (or shrink) and the application content is pushed out of the way (or fills in the empty space).

There are a number of supporting methods for understanding the placement of size of the ad that must be implemented with resize().

resize method

The resize method will cause the existing Web View to resize itself within the current view hierarchy using the existing HTML document.

The resize method will move the state from "default" to "resized" and fire the stateChanged event. If the state is not "default" then there is no effect.

The SDK must notify the app developer that the all view windows should resize as part of a resize() call.

Note: In most cases, expand is the correct behavior for rich media ads. The resize method should only be used in views where application content can be moved around within the existing view hierarchy, for example an ad cell in a table view that can grow and push the cells above and below apart. Designers should use the getMaxSize method before calling resize.

Use this method to resize the main ad view to the desired size. The views place in the view hierarch will not change, so the effect on other views is up to the app developer. To place the ad over content, use the expand() method instead.

```
resize(width, height)
```

parameters:

- width: Number: the width in pixels
- height: Number: the height in pixels

return values:

- none

events triggered:

- sizeChange, stateChange

side effects:

- changes state

getSize method
The getSize method will return the current size of the ad.

```
getSize() -> JSON
```

parameters:

- none

return value:

- JSON - {width, height}

related events:

- none

getMaxSize method
The getMaxSize method returns the maximum size an ad can resize to. This value defaults to the size of the screen but can be overridden by the app developer in native code. If an ad tries to resize larger than maxSize, then an error is thrown.

Use this method to return the maximum size an ad can grow to using the resize() method. This may be set by the developer, or be the size of the parent view.

```
getMaxSize() -> JSON
```

parameters:

- none

return value:

- JSON, {width, height} - the maximum width and height the view can grow to

related events:

- none

sizeChange event
The sizeChange event fires when the ads size within the app UI changes. This can be the result of an orientation change of the device or calls to the resize or expand methods.

This event is thrown when the display state of the web viewer changes.

```
"sizeChange" -> function(width, height)
```

parameters:

- width - Number: the width of the view
- height - Number: the height of the view

triggered by:

- a change in the view size as the result of a resize, expand, close, orientation, or the app after registering a "size" event listener.

getDefaultPosition method
The getDefaultPosition method returns the position and size of the default ad view regardless of what state the calling view is in.

Use this method to get the location and size of the default ad view.

```
getDefaultPosition() -> JSON
```

parameters:

- none

return values:

- JSON - {x, y, width, height}

getScreenSize method
The getScreenSize method returns the current size of the device screen. To receive updates on screen size changes use addEventListener for "screen" events.

Use this method to get the maximum pixel width and height of the application. This may be less than the actual device screen size, depending on the publisher. Although point width (pt) is preferred over pixel width (px) in the ad design because of device screens with different DPI specs, it is still essential for the designer to know how many pixels are on the screen..

```
getScreenSize() -> JSON
```

parameters:

- none

return values:

- {width, height}

related event:

- screenChange

screenChange event
The screenChange event fires when the devices screen size changes, usually as the result of an orientation change.

This event is thrown when the device screen size changes.

```
"screenChange" -> function(width,height)
```

parameters:

- width - Number: the width of the screen
- height - Number: the height of the screen

triggered by:

- a change in the device orientation after registering a "orientation" event listener.

Displaying an Ad

In cases where the rich media ad needs additional time to prepare resources or interactions, the designer may wish that the ad is hidden until it is ready. This means that the app view dedicated to the ad display should not be on screen until a show() method is called. Similarly, the designer may want the add to completely disappear from the screen – including the display area – when a user closes the ad. The hide() method is intended to remove the ad's web view from the view hierarchy.

show method

The show method will cause a hidden ad to become visible in the default position. The transition effect is managed by the individual SDK.

The show method will move the state from "hidden" to "default" and fire the stateChange event. If the state is not "hidden" there is no effect.

This method has no return value and is executed asynchronously (so always listen for a result event before taking action instead of assuming the change has occurred).

`show()`

parameters:

- none

return values:

- none

side effects:

- changes the state value

event triggered:

- stateChange

hide method

The hide method will cause ads in their default state to hide themselves and hide the view they are in.

The hide method will move the state from "default" to "hidden" and fire the stateChanged event. If the state is not "default" then there is no effect.

Use this method to hide the web viewer. The method has no return value and is executed asynchronously (so always listen for a result event before taking action instead of assuming the change has occurred).

`hide()`

parameters:

- none

return values:

- none

side effects:

- changes state

Controlling expandProperties

The expand() method can support additional features beyond managing a full-screen view that covers content. The expand properties object is intended to provide those features to ad designers.

When an ad calls the expand method, the way the ad expands depends on the expandProperties. The expandProperties are held in a JSON object that can be written and read by the ad.

At a minimum, the following properties should be supported.

```
properties object properties = {
 "useBackground" : "true|false",
 "backgroundColor" : "#rrggbb",
 "backgroundOpacity" : "n.n",
 "lockOrientation" : "true|false"
}
```

"useBackground"

"useBackground" should contain a boolean value (true/false) indicating the presence of a background. If "useBackground" is not specified in the properties object, a value of false is assumed.

"backgroundColor"

"backgroundColor" is a standard numeric RGB value (most logically expressed in hexadecimal with two digits each for red, green, and blue).

"backgroundOpacity"

"backgroundOpacity" is a number between 0 and 1 inclusively (ranging from 0 equaling fully transparent to 1 equaling fully opaque). If either "backgroundColor" or "backgroundOpacity" is not specified in the properties object, values of 0xffffff and 1.0 respectively are assumed.

"lockOrientation"

The "lockOrientation" property is a boolean value (true/false) and if it is not specified in the properties object a value of false is assumed.

getExpandProperties method

The getExpandProperties method returns the whole JSON object.

Use this method to get the properties for expanding an ad.

```
getExpandProperties() -> JSON
```

parameters:

- none

return values:

- { ... } - this object contains all the web viewer properties besides dimension that are supported by the SDK vendor, for more info see properties object

events triggered:

- none

setExpandProperties method

The setExpandProperties method sets the whole JSON object.

```
setExpandProperties(properties)
```

Use this method to set the ad's expand properties.

parameters:

- properties: JSON { ... } - this object contains any number of properties, such as transition, that might be used by the SDK when presenting the full screen web viewer. For more info see properties object.

return values:

- none

events triggered:

Offline Requests and Metrics

Rich Media Ads that can work while the device is without network connectivity need the ability to store and later forward metrics about how and when users interact with the ad.

While the following request method and response event are provided for greater flexibility in offline ads, their use is not confined to offline. An ad designer can use the request/response pair in an online state to provide Ajax style updates, for example.

request method

The request method makes an HTTP request when the device has network connectivity and caches the request for later transmission when the device is offline.

The method executes asynchronously, but returns a Boolean value of false to facilitate use in anchor tags. There is also an option explicitly for metrics tracking that will cache requests offline and execute them whenever the device reconnects. The display parameter supports the following values:

value	description
ignore	the response is ignored
proxy	the response is cached if the device is off-line and proxied when connectivity returns

```
request(uri, display) -> false
```

parameters:

- URL - string, the fully qualified URL of the page or call to action asset
- display - string, the display style for the call to action

event triggered:

- response

return values:

- false

response event

The response event is fired when a request method completes and provides the response if desired.

This event is thrown when a request action with a display type of "proxy" returns a response.

```
“response” -> function(uri, response)
```

parameters:

- uri: String, the URI of the original request action

- response: String, the full body of the response

triggered by:

- a request() method call returning

Access to Native Features

Using standard web technologies in the ad design, and relying on a web viewer in the app to render ads supports many presentation needs. But for truly rich media advertising, there must also be support for device features – many of which are normally only available to application developers. Devices offer a wide array of functionality beyond simple display of ad content for rich media ads. The Controller provides a layer of abstraction between the ad designer and the device to greater enable cross-device media creation.

Dynamic properties require an event listener strategy. Using the addEventListener method allows the ad developer to access all native features the SDK supports.

Knowing what features are available to listen for, and what the event names are requires a naming convention. For each feature a Controller supports, the getter method is "get"+feature name and the event name is feature name+"Change". For example, if a Controller supports the device's native location capabilities, the supported feature is "location", to get the location the ad developer would call "getLocation", and to listen for changes the ad developer would addEventListener for "locationChange".

As devices differentiate, or hardware vendors innovate, additional native features can be added using the same naming convention.

supports method

The supports method allows the ad to interrogate the SDK for support of specific device features.

The controller should support as many of the following features as is possible for a given device.

value	**description**
network	the device can report on its network connectivity and connectivity changes
keyboard	the device uses a soft keyboard that impacts display
orientation	the device can report on its orientation and orientation changes
screen	the device can report on the screen size
heading	the device can report on the compass direction it is pointing
location	the device can report on its location
shake	the device can report on being shaken

sms	the device can send an SMS message
tilt	the device can report on its tilt and tilt changes
phone	the device can make a phone call
email	the device can compose an email
calendar	the device can create a calendar entry
camera	the device can take a still picture image

`supports(feature) -> Boolean`

parameters:

- String, name of feature

return values:

- Boolean – true, the feature is supported and getter and events are available; false, the feature is not supported on this device

Working with the Device's Physical Characteristics

Most devices have several different kinds of sensors that can report on various physical characteristics of the device, such as its location, the direction it is pointing, its orientation, and its motion.

It's important to know that requesting a device's hardware features impacts physical properties such as battery life and available memory. Ad designers should only request native features on an as-needed basis and use removeEventListener when the feature is no longer needed.

getHeading method

The getHeading method alone returns the last compass heading of the device. The heading may be unknown or out-of date.

To activate the compass and receive updates, use addEventListener for "headingChange" events.

Use this method to get the most recent compass direction of the current vertical axis of the device. To receive events when the a change occurs, register an event listener for "headingChange" events. Values are:

value	description
-1	no heading known
0-359	compass direction in degrees

`getHeading() -> Number`

parameters:

- none

return value:

- Number, the degrees

related event:

- headingChange

headingChange event

The headingChange event fires when the devices compass direction changes.

This event is thrown when the devices compass direction changes.

`"headingChange" -> function(heading)`

parameters:

- heading: Number, compass heading in degrees or -1

triggered by:

- a change in the device heading after the compass has been activated by registering a "heading" event listener.

getLocation method

The getLocation method alone returns the last location reading and accuracy of the device. The location may be unknown or out-of-date.

To activate the location system and receive updates use addEventListener for "locationChange" events.

Use this method to get the most recent location reading from the device. To receive events when the a change occurs, register an event listener for "locationChange" events.

`getLocation() -> JSON`

parameters:

- none

return value:

- JSON, {lat, lon, acc} - the latitude, longitude, and accuracy of the reading or null

related event:

- locationChange

locationChange event

The locationChange event fires when the devices location changes.

This event is thrown when the device has successfully geolocated itself.

`“locationChange” -> function(lat, lon, acc)`

parameters:

- lat: Number, latitude value of device
- lon: Number, longitude value of device
- acc: Number, accuracy of the reading

triggered by:

- a change in the device heading after the GPS has been activated by registering a "location" event listener.

getOrientation method

The getOrientation method returns the current device orientation.

To receive updates on orientation changes use addEventListener for "orientationChange" events.

Use this method to get the most recent orientation of the device. To receive events when a change occurs, register an event listener for "orientationChange" events. Possible results include:

value	description
-1	device orientation unknown
0	0 degrees (portrait)
90	90 degrees (tilted clockwise to landscape)
180	180 degrees (portrait upside down)
270	270 degrees (tilted counter-clockwise to landscape)

`getOrientation() -> Number`

parameters:

- none

return values:

- Number

related event:

- orientationChange

orientationChange event

The orientationChange event fires when the device is rotated or tilted to a new orientation.

This event is thrown when the application screen orientation changes.

```
"orientationChange" -> function(orientation)
```

parameters:

- orientation - Integer, degrees from upright portrait view

triggered by:

- a change in the device orientation after registering a "orientation" event listener.

getTilt method

The getTilt method returns the last reported device tilt readings in 3 dimensions.

To receive updates on tilt changes use addEventListened for "tiltChange" events.

This method returns the last reading of the devices 3 dimensional tilt.

```
getTilt() -> JSON
```

parameters:

- none

return values:

- JSON - {x, y, z}

related events:

- tiltChange

tiltChange event

The tiltChange event fires when the devices 3 dimensional tilt values change.

This event is thrown when the device has successfully determined its spacial orientation.

```
"tiltChange" -> function(x, y, z)
```

parameters:

- x,y,z – Numbers, the x, y, and z axis values in radians

triggered by:

- a change in the device tilt after the accelerometer is activated by registering a "tilt" event listener.

getShakeProperties method

The getShakeProperties method returns the current thresholds that define a shake gesture. The defaults should be sufficient in most cases, but setShakeProperties is available as required.

Use this method to retrieve the current shake properties.

```
getShakeProperties() -> JSON
```

parameters:

- none

return values:

- {interval, intensity}

related events:

- shake

shake event

The shake event fires when the device is shaken within the thresholds of the current shake properties.

This event is thrown when the device accelerometer detects that the device has been "shaken" as defined by the getShake parameters. Because the ad developer may register one listener for a soft shake and another listener for a harder shake, this event provides threshold and time parameters

```
"shake" -> function()
```

parameters:

- none

triggered by:

- The device if a shake gesture is detected after registering a "shake" event listener.

setShakeProperties method

The setShakeProperties will set ad specific thresholds for what is interpreted by the SDK as a shake gesture. The default values should be sufficient in most cases and ad designers are not required (nor encouraged) to use setShakeProperties.

Use this method to set the shake properties. This method rarely needs to be called as supported devices have default settings.

```
setShakeProperties(properties)
```

parameters:

- properties: JSON { intensity, interval }

return values:

- none

events triggered:

- none

side effects:

- none

getKeyboard method

On devices that have a virtual keyboard, the display of the keyboard can affect the ad display.

The keyboardChange event fires when the virtual keyboard opens or closes.

Use this method to determine if the virtual keyboard is present on the screen. The boolean result is true if the keyboard is present and false if it is hidden or not applicable.

```
getKeyboard() -> Boolean
```

return value:

- Boolean - the virtual keyboard is present

related event:

- keyboardChange

keyboardChange event

This event is thrown when the software keyboard is opened or closed for text entry in an ad.

```
"keyboardChange" -> function(open)
```

parameters:

- boolean, open - whether the keyboard is open

triggered by:

- a change in the state of the virtual keyboard after registering a "keyboard" event listener.

Special Media Assets

Rich Media Ads can access two special asset types that allow them to take a screenshot of the device's screen and take a picture with the device's camera. Calling the addAsset method with a URL of "mraid://screenshot" will take a screenshot and save it to the specified alias. Calling the addAsset method with a URL of "mraid://photo" will open the devices camera interface and save a photo it to the specified alias.

storePicture method

The storePicture method will place a picture in the device's photo album. The picture may be local or retrieved from the Internet.

This method will store the image or other media type specified by the URL.

```
storePicture(URL)
```

parameter:

- URL -String: the URL to the image or other media asset

Working with Device Connectivity

getNetwork method

The getNetwork method returns the current network connection type for the device. To receive updates on network changes use addEventListener for "network" events.

Use this method to identify the most recent network status of the device. To receive events when a change occurs, register an event listener for "networkChange" events. Possible results include:

value	description
offline	no network connection
wifi	network using a wifi antennae
cell	network using a cellular antennae (such as 3G)
unknown	network connection in unknown state

`getNetwork() -> String`

parameters:

- none

return values:

- String

related event:

- networkChange

networkChange event

The networkChange event fires when the devices network connectivity changes.

This event is thrown when the device network connection changes, such as loosing or acquiring an Internet connection. The connection type values will vary depending on the device and carrier.

`“networkChange” -> function(online, connection)`

parameters:

- online: Boolean, true – device is connected to the Internet, false – device cannot access the Internet
- connection: String, description of connection type such as none, wifi, or cell

triggered by:

- a change in the state of the network after registering a "network" event listener.

Working with Native Applications

Web protocols such as sms: or mailto: allow ad designers to include applications into their rich media ads by simply using URLs. These protocols are not always consistent across devices.

This specification recommends using methods instead to ensure that the ad and current application are politely suspended while the user takes action.

createEvent method
The createEvent method opens the device UI to create a new calendar event with default values provided. The ad is suspended while the UI is open.

Use this method to create a new event in the devices calendar.

```
createEvent(date, title, body)
```

parameters:

- date - Date, the date and time of the event
- title - String, the title of the event
- body - String, the body of the event

return value:

- none

makeCall method
The makeCall method opens the device UI for making a phone call to a specified number. The ad is suspended while the UI is open.

Use this method to make a phone call on from the device to the number provided. This is similar to a tel:// protocol, but the SDK will attempt to suspend the application.

```
makeCall(number)
```

parameters:

- number - String: the phone number

return values:

- none

sendMail method
The sendMail method opens the device UI for sending an email message with the content provided. The ad is suspended while the UI is open.

Use this method to compose an email message on the device. This is similar to the mailto:// protocol, but the SDK will attempt to suspend the application.

```
sendMail(recipient, subject, body)
```

parameters:

- recipient - String, the email address for the message
- subject - String, the subject line of the message
- body - String, the body of the message

return value:

- none

sendSMS method
The sendSMS method opens the device UI for sending an SMS message with the content provided. The ad is suspended while the UI is open.

Use this method to compose a SMS message on the device. This is similar to the sms:// protocol, but the SDK will attempt to suspend the application.

`sendSMS(recipient, body)`

parameters:

- recipient - String, the email address for the message
- body - String, the body of the message

return value:

- none

playAudio method
Use this method to play a audio on the device. This may launch an external player, depending on the SDK implementation. To place the audio with the content, set the inline property. For the most part, properties follow the HTML5 audio tag conventions.

Controls:

property	values	description
autoplay	autoplay	include if audio should play immediately
controls	controls	include if native player controls should be visible
loop	loop	include if video should start over again after finishing
inline	inline	include if audio should be included with ad content
startStyle	normal/fullscreen	set to fullscreen if audio should start playing in native full screen mode – user may still use controls to change size, default is normal
stopStyle	normal/exit	set to exit if audio player should exit after the audio stops, default is normal

`playAudio(URL, properties)`

parameters:

- URL - String, the URL of the audio or audio stream
- properties - JSON:, list of the properties for native player

return values:

- none

playVideo method
Use this method to play a video on the device. This may launch an external player, depending on the SDK implementation. To place the video over content, set the inline property. For the most part, properties follow the HTML5 video tag conventions.

Controls:

property	values	description
audio	muted	include if audio track should be muted
autoplay	autoplay	include if video should play immediately
controls	controls	include if native player controls should be visible
loop	loop	include if video should start over again after finishing
inline	{top, left}	provide the top and left coordinates in pixels if video should play inline (or on top of) ad content
width	(pixels)	pixel width of video, required for inline
height	(pixels)	pixel height of video, required for inline
startStyle	normal/fullscreen	set to fullscreen if video should start playing in native full screen mode – user may still use controls to change size, default is normal
stopStyle	normal/exit	set to exit if video player should exit after the video stops, default is normal

`playVideo(URL, properties)`

parameters:

- URL - String, the URL of the video or video stream
- properties - JSON:, list of the properties for native player

return values:

- none

openMap method
Use this method to open a native map with the Point of Interest (POI) parameter formatted according to the Google Maps standard (see http://mapki.com/wiki/Google_Map_Parameters)

`openMap(POI, fullscreen)`

parameters:

- POI - String, Google Maps-formatted argument. The parameter must describe a point on a map, not, for example, driving directions
- fullscreen - boolean, whether the map displays within the current View or within a new View that takes up the whole screen

return values:

- none

Handling Call-to-Action Events

A rich media ad implements multiple call-to-action events beyond the click to microsite. These events may be executed as anchor links or scripted functions. This means an SDK cannot just listen for clicks in the browser. It must support programmatic clicks as well.

Hyperlinks

Rich media ads can have HTML hyperlinks in them, but the ad developer needs to be careful about using them. Loading a new web page in the ad view that is not written to the MRAID spec can leave the ad, and possibly the app, in an unusable state. Additionally, some devices override or implement certain URLs in their own applications, such as mail, maps, calendar, SMS, and phone calls. MRAID provides methods for those functions in the Level-2 specification.

Addendum 2 – Ad Examples and Code Samples

This addendum contains sample JavaScript for rich media expandable and interstitial ads created using MRAID 1.0. Full versions of both examples, including javascript and graphics assets are available as a zip file at www.iab.net/mraid.

Expandable Ad

```
/*
 * Description: The loader for a rich media (video + expandable) ad
using the MRAID api. This ad has just a base image (w:316px h:728px)
and a video (w:316px h:728px)
 * clicking on which causes the ad to expand (w:949px h:728px). The
expanded layer has a close button (w:24px h:24px) at the top right
corner that is used with customClose
 * Author: Aditya Kalro
 * Company: Yahoo!
 */

document.write("<script src=\"mraid.js\"></script>");

/*
 * Checking for the state of the mraid client library and subscribing
to the ready event if necessary
 * When the client library is ready call the showAd method to render
the ad
 */
if (mraid.getState() != 'ready') {
        console.log("MRAID Ad: adding event listener for ready");
        mraid.addEventListener('ready', showAd);
} else {
        showAd();
}

/*
 * The showAd method registers event listeners for the mraid events and
renders
 * the base ad (simple image)
 */
function showAd() {
        basePath = "http://localhost:8666/yahoo.ads.mraid_richmedia/";
        registerMraidHandlers(mraid, basePath);
        renderBaseAd(mraid, basePath);
        /*
         * set the expand properties to use the custom close method
since the ad
         * renders it's own close button in the expanded layer
         */
```

```
	mraid.setExpandProperties({
		useCustomClose : true
	});
};

/*
 * Add a listener to the stateChange event to figure out what state the
client
 * listener is in and whether to render the rich functionality or not
 */
function registerMraidHandlers(mraid, basePath) {
	mraid.addEventListener("stateChange", function(state) {
		switch (state) {
		// Event trigger when the ad-container goes offscreen
		case "hidden":
			removeOverlayLayer();
			break;
		// Event trigger when the ad-container is onscreen
		case "default":
			renderOverlayLayer(mraid, basePath);
			break;
		}
	});
}

/*
 * Render the base image of the ad (this is what is rendered in the
hidden
 * state).
 */
function renderBaseAd(mraid, basePath) {
	var imageURL = basePath + "assets/mraid_column_static.jpg";
	console.log("rendering base ad");
	var baseImage = document.createElement("img");
	baseImage.setAttribute("id", "base_img");
	baseImage.src = imageURL;
	baseImage.setAttribute("style", "border:0px; width:316px;
height:728px;");
	document.appendChild(baseImage);

}

function resolveVideoPath(path, cb) {
	console.log("in resolveVideoPath");
	cb(path);
}

/*
 * Rendering the rich functionality of the ad (is called when the ad
comes
 * onscreen) This renders the richmedia overlay div and adds a video
into it.
 * Clicking on the div results in it's expansion (ad_expand method)
 */
function renderOverlayLayer(mraid, basePath) {
```

```
        console.log("Rendering overlay layer");
        var overlayContainer = document.createElement("div");
        overlayContainer.setAttribute("id", "overlayContainer_mraid");
        var bkgImagePath = basePath +
"assets/mraid_main_bg_949x728.jpg";
        overlayContainer
                        .setAttribute(
                                        "style",

        "width:949px;height:728px;position:absolute;right:-
633px;top:0px;z-index:1000;display:none;background:url("
                                                        + bkgImagePath +
")");
        document.appendChild(overlayContainer);
        resolveVideoPath(
                        basePath + "assets/mraid_column4.mp4",
                        function(videoPath) {
                                var closeButtonPath = basePath +
"assets/close-button.png";
                                overlayContainer.innerHTML = "<video
width='316px' height='728px' loop=\"true\" id = \"video_elem\" src='"
                                                        + videoPath
                                                        + "'></video>"
                                                        + "<img
style='position:absolute;right:10px;top:10px;' src='"
                                                        + closeButtonPath + "'
width='24px' height='24px'/>";
                                var videoPlayer =
document.getElementById("video_elem");

// Make sure video is playing to avoid blink as video
// player is loading
        videoPlayer.addEventListener("playing", function() {
                console.log("Video has started playing");
                overlayContainer.style.display = "block";
                videoPlayer.style.display
        }, false);

        videoPlayer.addEventListener("play", function() {
                console.log("Video has started playing");
                overlayContainer.style.display = "block";
        }, false);

        videoPlayer.addEventListener("click", function() {
                ad_expand(mraid, videoPlayer, overlayContainer);
        }, false);

        var closeButton =
overlayContainer.getElementsByTagName("img")[0];
        closeButton.addEventListener("click", function() {
                ad_close(mraid, videoPlayer, overlayContainer);
        }, false);

        videoPlayer.play();
});
```

```
}

/*
 * The handler for closing the expanded layer of the ad. The
mraid.close method
 * is used here to make sure that the ad publishing app knows about the
change
 * in the state of the ad.
 */
function ad_close(mraid, videoPlayer, overlayContainer) {
        overlayContainer.style.webkitTransition = "-webkit-transform
250ms ease-in-out";
        overlayContainer.style.webkitTransform = "translateX(0)";
        overlayContainer.addEventListener("webkitTransitionEnd",
function() {
                videoPlayer.style.display = "block";

        overlayContainer.removeEventListener("webkitTransitionEnd",
                                arguments.callee, false);
        }, false);
        // Notifies the content of the ad closing
        mraid.close();

}

/*
 * The handler for expanding the ad. The mraid.expand method is used
here to
 * make sure that the ad publishing app knows about the change in the
state of
 * the ad.
 */

function ad_expand(mraid, videoPlayer, overlayContainer) {
        // Notifies the content of the ad taking over the screen
        mraid.expand();
        videoPlayer.style.display = "none";
        overlayContainer.style.webkitTransition = "-webkit-transform
250ms ease-in-out";
        overlayContainer.style.webkitTransform = "translateX(-633px)";
}

/*
 * Removes the overlay layer when the ad is going offscreen
 */
function removeOverlayLayer() {
        var overlayContainer =
document.getElementById("overlayContainer_mraid");
        mraid.close();
}
```

Interstitial Ad

```
/*
 * Description: The loader for a static (basic clickable) ad using the
MRAID api. This ad has just one image w:316px h:728px that clicks
 * through to a landing page (www.livestand.com)
 * Author: Aditya Kalro
 * Company: Yahoo!
 */

document.write("<script src=\"mraid.js\"></script>");

/*
 * Checking for the state of the mraid client library and subscribing
to the ready event if necessary
 * When the client library is ready call the showAd method to render
the ad
 */
if (mraid.getState() != 'ready') {
        console.log("MRAID Ad: adding event listener for ready");
        mraid.addEventListener('ready', showAd);
} else {
        showAd();
}

/*
 * The showAd method registers event listeners for the mraid events and
renders
 * the base ad (simple image)
 */
function showAd() {
        basePath = "http://localhost:8666/yahoo.ads.mraid_static/";
        registerMraidHandlers(mraid, basePath);
        renderBaseAd(mraid, basePath);
        /*
         * set the expand properties to use the custom close method
since the ad
         * renders its own close button in the expanded layer
         */
        mraid.setExpandProperties({
                useCustomClose : true
        });
};

/*
 * Add a listener to the stateChange event to figure out what state the
client
 * listener is in and whether to render the rich functionality or not
 */
function registerMraidHandlers(mraid, basePath) {
        mraid.addEventListener("stateChange", function(state) {
                switch (state) {
                // Event trigger when the ad-container goes offscreen
                case "hidden":
                        break;
```

```
                // Event trigger when the ad-container is onscreen
                case "default":
                        // This is where the impression beacon (if any)
should be fired
                        break;
                }
        });
}

/*
 * Render the basic ad (an image with wrapped in an anchor element)
 */
function renderBaseAd(mraid, basePath) {
        var landingPage = "http://www.yahoo.com";
        var imageURL = basePath + "assets/mraid_column_example.jpg",
adImage = "<img width='316px' height='728px' border=0 src='"
                        + imageURL + "'/>";
        var anchor = "<a id=\"base_image_example\" href=\"" +
landingPage + "\"> "
                        + adImage + "</a>";
        document.write(anchor);
        var anchorElement =
document.getElementById("base_image_example");
        anchorElement.onclick = function() {
                console.log("Clicking on anchorElement = " +
anchorElement.href);
                mraid.open(anchorElement.href);
                return false;
        };
}
```