Interactive Advertising Bureau

Mobile Rich-media Ad Interface Definitions (MRAID) v.2.0

Released September 27, 2012

Table of Contents

Contributors

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

24/7 Real Media, Inc.
AccuWeather.com
AdMarvel
AdMeld
ADTECH
Adobe Systems Inc.
AOL
CBS Interactive
Celtra
Crisp Media
Dow Jones & Company
ESPN
FreeWheel
Goldspot Media
Google
Greystripe
IDG
inMobi
Innovid
Jumptap
Medialets
MediaMind
Microsoft Advertising
Mixpo
Mocean Mobile
NBC Universal Digital Media
New York Times Co.
Nexage
Pandora
PointRoll
Rhythm NewMedia
Spongecell
Sprout
TargetSpot
Time Inc.
Turner Broadcasting System, Inc.
Univision
The Weather Channel
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. Contributors to ORMMA at the point the IAB launched the MRAID project included:

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, Senior 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 mobile in-app 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.

Definitions

The following terms are used throughout the MRAID specification.

Ad View/Container: The constrained area which displays the ad creative. Publishers either place the Ad Container within the content (for inline placements) or over the content (for interstitial placements) and present the ad creative. The container provides the area on the screen, the MRAID controller, and the web-based view for the ad to display. Ad Containers are usually, though not necessarily, provided by SDKs. An app may contain multiple Ad Containers from a single SDK.

Close Event Region: The close event region is a tappable area on the ad creative that will cause the ad to return to its default state (in the case of an expandable/resizeable ad) or be removed from the screen (in the case of an interstitial).

Close Indicator: The close indicator is the visual cue to the user as to the location of the close event region.

Controller: The JavaScript code that provides ad designers access to MRAID methods and events. The ad creative uses the controller to perform advertising-related interactions with the Ad Container, and, indirectly, with the application and the device.

Density-Independent Pixels: All length values passed between the container and the creative through the MRAID API are in density-independent pixels.
Density-independent pixels are an abstraction from physical screen pixels meant to simplify application and content development across devices of different screen densities.

Using density-independent pixels means that, for example, retina display iPhones and older iPhones will return the same dimensions/measures, despite having different numbers of physical pixels. 1 density-independent pixel corresponds rougly 1/160 of an inch (1 device pixel on a device with roughly 160 DPI).

On iOS, these should map to "points"; on Android, to "density-independent pixels".

Note: One density-independent pixel will match 1 CSS pixel only if the viewport scale is 1.0. To map between CSS pixels and density-independent pixels, the creative should use the following formula:

css_pixels * viewport_scale = density_independent_pixels

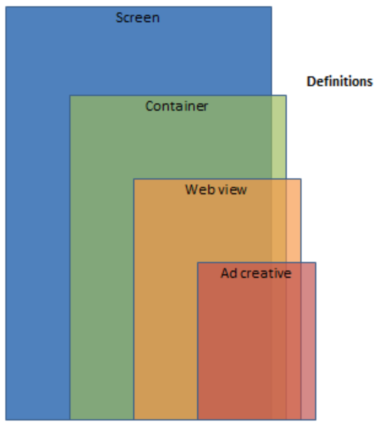
Inline Ad: An ad that appears onscreen accompanied by other kinds of content, e.g., a banner on a web page or in an app.

Interstitial Ad: A full page modal ad that displays on top of content – a "roadblock" or "overlay." The ad must be dismissed for the user to return to the publisher content. Such ads can appear between levels of a game, or before or after a video clip or other dynamic content. (An ad that is in-between pages and swipes into view like in many magazine apps, is considered an inline ad under MRAID.)

Physical Pixels: The actual pixels on a device screen. For example, a retina-display iPhone measures 960x640 physical pixels. MRAID API length values are always calculated in density-independent pixels (defined above) NOT physical pixels.

SDK: Sofware Development Kit. The reusable piece of code (library) integrated into publisher apps to enable advertisements/Ad Containers. An SDK, by itself, is not a visual component.

Web View: The HTML-based viewer that displays the ad creative. The web view is used to perform rendering of HTML- and Javascript-enabled ads.



General Requirements for Supporting MRAID

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

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 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.) in a consistent manner, where allowed by the app publisher/ad seller.
- 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 designer to use directly.

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

Ad Designer

There are no creative requirements in this document for ad designers besides the use of web standards. Ad designers 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.

Viewport and Default Container Set-Up

Creative designers should be aware of the need to understand and potentially override the default settings of the web view in which they are running. They should do this by querying an MRAID container just as they would query a web page to understand the environment there. These settings include height and width of the container, scale, and whether the user can change the scale of the container.

While MRAID does not establish any new parameters or controls over the web view, it is recommended that the creative should check and adjust the parameters, since the author of said creative might wish to set them differently from the default container settings..

Out of Scope

Each MRAID implementation 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[1]
- 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 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. These other features must be implemented outside the MRAID namespace.

An ad that uses SDK-specific features in addition to MRAID features would not necessarily be an MRAID ad anymore in the sense of working across all MRAID-compliant SDKs.

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 container must display any HTML ad. The container should invoke an HTML with JavaScript rendering engine for rendering ads. In this document, that engine will be called the "web view". Whenever possible, the web view should incorporate the capabilities of

[1] That is, MRAID does not define ad sizes (dimensions) or how ads should move or change in response to user interaction.

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 Designers

Display Control for Rich Media Ads – Ad Controller

An ad designer that expects his/her ad to make use of MRAID must indicate that by invoking the mraid.js script as soon as possible as the ad loads. This signals the SDK to inject the MRAID javascript into the creative.

MRAID remains in the background, leaving the ad designer in control of the ad display, but is available so that the creative can use the MRAID API when/if the ad needs to access MRAID features and functionalities. The internal interaction between the creative and the rich media SDK is hidden from both the Ad designer and the App developer.

An ad that does not utilize any device features does not need to use the MRAID API at all. However if the ad does not invoke MRAID, it will get the SDK's default solution. Some of the things an ad uses MRAID's API for are:

- Opening an embedded web browser
- Detecting whether the ad is viewable or not
- Expanding an ad that grows from a banner to a larger size
- Clicking within an ad triggering an action

Ad designers are encouraged to rely on MRAID's capabilities to achieve the above effects.

Lifecycle Examples

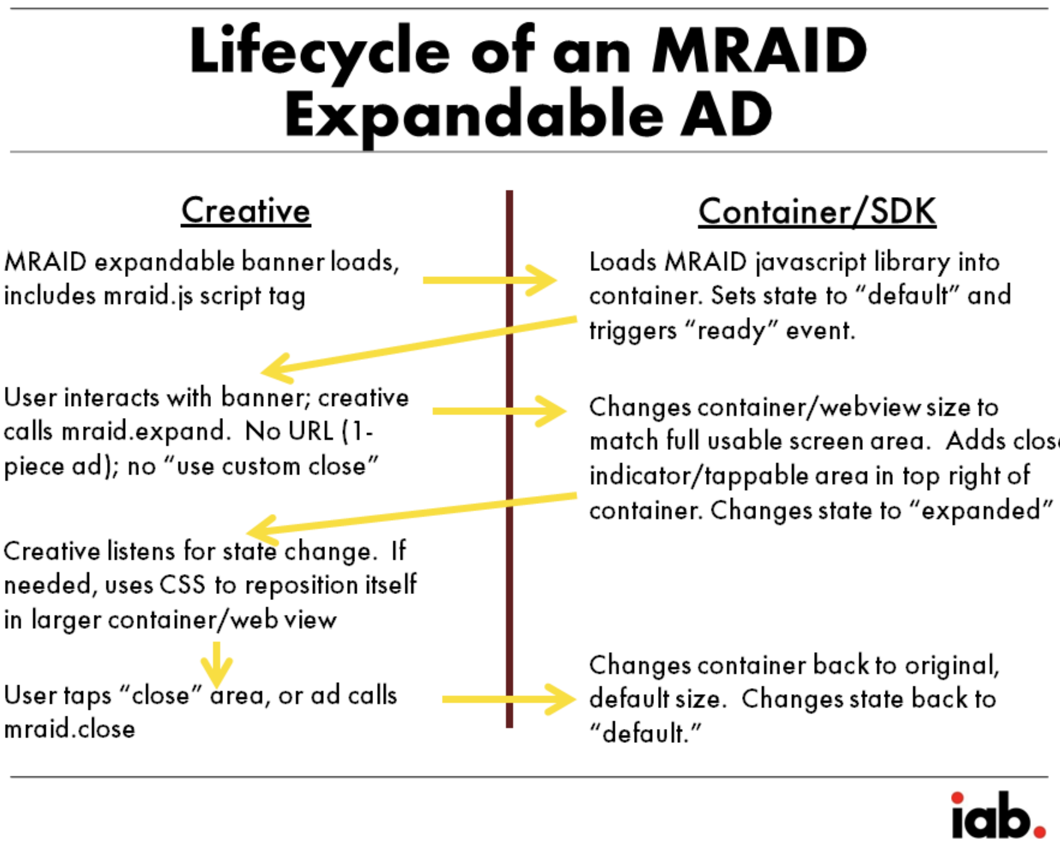
Simple Ad Lifecycle Example

Non-rich-media ads (e.g., basic banners) can optionally invoke MRAID. If the ad does not invoke MRAID via the mraid.js script tag, then it will behave however the application/SDK normally handles such ads.

Ad designers may wish for such simple HTML ads to invoke mraid.js, if they want to use the MRAID-standard container rather than the SDK's default container. In that case, the ad designer should make sure to use mraid.open() for any links to ensure consistent behavior.

Lifecycle of an MRAID Expandable Ad Example

In a rich media ad lifecycle example, the Ad Designer uses the JavaScript API to communicate with the native layer and interact with features of the device and OS.


Lifecycle of an MRAID Expandable AD
Creative
Container/SDK
MRAID expandable banner loads, includes mraid.js script tag
Loads MRAID javascript library into container. Sets state to "default" and triggers "ready" event.
User interacts with banner; creative calls mraid.expand. No URL (1-piece ad); no "use custom close"
Changes container/webview size to match full usable screen area. Adds close indicator/tappable area in top right of container. Changes state to "expanded"
Creative listens for state change. If needed, uses CSS to reposition itself in larger container/web view
User taps "close" area, or ad calls mraid.close
Changes container back to original, default size. Changes state back to "default."
iab.

As an example, when the user touches the ad, the ad uses the MRAID API to request that the ad can expand. The SDK should (though this is not part of the MRAID specification) notify the app 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 container reserves a space at the top right corner of the expanded ad container for an MRAID-enforced close event region, 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 resize to its original size, display the ad's banner state, and notify the app that it can resume.

Lifecycle of an MRAID Interstitial Ad

The case of an interstitial ad is very similar. The ad can use the MRAID API to query the container as to whether it is visible onscreen or not, waiting until it is on before it takes other actions. As with an expandable, the container reserves a space at the top right corner of the expanded ad for an MRAID-enforced close event region, 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 advertising across platforms. For this reason, IAB is releasing 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.

Maintaining full backwards compatibility in MRAID is a key goal of this project. An MRAID 2.0-compliant SDK should be able to run an MRAID 1.0 ad with no problems whatsoever, and an MRAID 1.0-compliant SDK should be able to handle the MRAID 1.0-compliant features of an MRAID 2.0 ad.

In establishing both versions of MRAID, the IAB and its MRAID working group have focused on six key goals:

- **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.
- **Consistent means for ads to change size and/or open new pages, preferably in an embedded browser** –MRAID provides ads a consistent way to communicate with rich media SDKs regarding their need to expand, and open an app's embedded browser (or in the native browser if an embedded browser does not exist).
- **Consistent means for a consumer to exit an ad** – MRAID ads will always have a consistent control by which a user can indicate that they wish to exit out of the ad experience and return to the app/content they were in.
- **Flexibility for publishers** – although MRAID-compliant SDKs must support all MRAID capabilities, app publishers/ad sellers are free to allow or disallow ads that make use of the features MRAID enables. That is, MRAID enables rich media ad features, but does not dictate that all sellers of rich media ads must support all those features.

Version 1

The methods and events included in MRAID Version 1 provide a minimum level of requirements for rich media ads, primarily to display HTML ads that can change size in a fixed container (e.g., expand from banner to larger/full screen size), and interstitial ads.

Version 2

MRAID Version 2 extends the capabilities of MRAID Version 1 to give ad designers more control over expandable ads, and provides a new method, resize() that permits more subtle and interesting size changes in ad creatives as well.

In addition, MRAID v.2 provides a standard way to query a device regarding certain capabilities, offers consistent handling of video creative, and addresses two native capabilities not well implemented by HTML5 at present: adding an entry to the device calendar and storing an image in the device photo roll.

For examples of ads that can be developed using the MRAID Version 2 API, please see the addendum.

Interface Requirements and Definitions

This list outlines all the methods and events that ad designers will have access to under MRAID v2.0. Methods and events new in MRAID v2.0 (e.g., that were not in MRAID v1) are indicated by an asterisk below.

Methods

- addEventListener
- createCalendarEvent*
- close
- expand
- getCurrentPosition*
- getDefaultPosition*
- getExpandProperties
- getMaxSize*
- getPlacementType
- getResizeProperties*
- getScreenSize*
- getState
- getVersion
- isViewable
- open
- playVideo*
- removeEventListener
- resize
- setExpandProperties
- setResizeProperties*
- storePicture*
- supports*
- useCustomClose

Events

- error
- ready
- sizeChange*
- stateChange
- viewableChange

Identification

It is required that ads identify themselves as being MRAID compliant. This is done by adding an MRAID script reference as soon as possible the creative and well before any MRAID functions are referenced in the creative. In other words, the MRAID identification script reference must be identifiable as soon as possible by any MRAID-compliant container or SDK.

MRAID script reference
The MRAID tag follows HTML Javascript syntax so that both fully formed web pages and HTML fragments can be identified as MRAID ads. mraid.js will be included as a script in the document either using an HTML tag or as javascript. MRAID sample ads (see www.iab.net/mraid) illustrate where the script tag should be positioned.

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

While MRAID ads need to identify themselves as such via the mraid.js script in a timely fashion so that the container can inject the MRAID libraries, ad designers should avoid using the string "mraid.js" for any other purpose in an ad creative, as doing so may lead containers/SDKs to mistakenly inject multiple copies of the MRAID libraries.

Initialization

MRAID governs interactions between the ad and the container 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 container to ensure they are available to the ad in a timely fashion after the script reference is made, and to signal as such by firing the ready event.

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

1. Ad identifies itself as MRAID as early as possible by invoking 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 limited MRAID object with an MRAID State = "loading" and the ability to query the state
3. If the ad uses createElement, needs to wait for mraid.js to finish loading
4. If MRAID State="loading" then ad listens for "ready" event with mraid.addEventListener('ready')
5. SDK/Container finishes initializing MRAID library into the webview

   a. Changes the MRAID state to "default" and the StateChange Event is triggered

   b. Fires the MRAID "ready" event

6. Ad's "ready" event listener is triggered and ad JavaScriptcan now use the MRAID APIs

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

It is the responsibility of the MRAID-compliant container 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 container may load all of MRAID at once, at a minimum the container 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 container may still need more time to initialize settings or prepare additional features, ready should only fire when the container 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 (e.g., 1.0 or 2.0) and not the vendor's SDK version.

```
getVersion() -> String
```

parameters:

- none

return values:

- String – the MRAID version that this SDK is certified against by the IAB, or that this SDK is compliant with. For example, for the current version of MRAID, getVersion() will return "2.0."

Initial Display

It is up to the ad designer 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 MRAID. Communicating between the web layer and native layer is asynchronous by nature. Through event handling, the ad designer is able to listen for particular events and respond to those events on an as-needed basis. MRAID advocates 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. An ad may register for more than one listener at a time, and it is required that ads be permitted to do so. The events supported by MRAID v.2 are:

value	**description**
ready	report initialize complete
error	report error has occurred

stateChange	report state changes
viewableChange	report viewable changes
sizeChange	report a change in size of the ad

`addEventListener(event, listener)`

parameters:

- event – string, name of event to listen for
- listener – function to execute

return values:

- none

side effects:

- *none*

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 specified, then all functions listening to the event will be removed.

`removeEventListener(event, listener)`

parameters:

- event – string, name of event
- listener – function to be removed

return values:

- none

events triggered:

- none

Error Handling

When an error in the container occurs, the "error" event is triggered 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 triggered whenever a container 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

Ad designers should note that errors can be handled on either a synchronous or an asynchronous basis by the SDK/container.

While the "message" part of the error event is not defined by the MRAID specification and mainly intended for pre-flight debugging of creative, the "action" part of the error is always the name of the method that the ad tried to use that led to the error. In principle, any MRAID method may trigger an error, so ad designers using an error event listener should listen for the following as potential error actions:

- addEventListener
- createCalendarEvent
- close
- expand
- getCurrentPosition
- getDefaultPosition
- getExpandProperties
- getMaxSize
- getPlacementType
- getResizeProperties
- getScreenSize
- getState
- getVersion
- isViewable
- open
- playVideo
- removeEventListener
- resize
- setExpandProperties
- setResizeProperties
- storePicture
- supports
- useCustomClose

While any MRAID method may lead to an error, in practice using resize, adding an image to a device's photo album or adding an event to a calendar are the most likely MRAID methods to generate errors. Ad designers using those methods should be particularly diligent about adding an error listener to check whether an error occurs on a resize, storePhoto, or createCalendarEvent action, so that the ad creative can potentially take a different action.

Controlling Ad Display

Besides the initial display, the ad designer 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 Webview) has a state that is one of the following:

value	description
loading	the container is not yet ready for interactions with the MRAID implementation
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
resized	the ad container has changed size via MRAID 2.0's resize() method
hidden	the state an interstitial ad transitions to when closed. Where supported, the state a banner ad transitions to when closed

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 or resized, larger position, or hidden.

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

Any MRAID ad can have only one state at a time. In the case of two-part expandable ads, this requirement means there is only one state for both web views. For as long as the expanded view is onscreen, using getState() will return "expanded."

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

How to read: the initial state of the creative is in the left-most column; how the state changes when an MRAID method is used can be found by looking down the column for that method. So for example, if the ad's state is "expanded" and the close() method is used, the ad's state changes to "default."

Initial state	**expand()**	**resize()**	**close()**
loading	no effect	no effect	no effect
default	For a banner, state changed to "expanded" For an interstitial, no effect	For a banner, state changed to "resized" For an interstitial, no effect	For a banner, state changed to "hidden" (if supported by SDK/container) For an interstitial, state changed to "hidden"
expanded	no effect (state remains "expanded")	triggers an error; state remains "expanded"	state changed to "default"
resized	state changed to "expanded"	state changed to "resized" (that is, an event listener will hear a new stateChange event, even though the state is still "resized" after the event fires)	state changed to "default"
hidden	no effect	no effect	no effect

In the case of a two-piece expandable, the new, expanded web view starts in the "loading" state briefly until MRAID is available, upon which the "ready" event is fired and the state of the ad then transitions to "expanded." The banner (the first piece) of the two-piece ad also changes its state, from "default" to "expanded."

For an interstitial ad, the web view goes from "loading" to "default," and when the interstitial is closed, the state changes to "hidden."

`getState() -> String`

parameters:

- none

return values:

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

related events:

- stateChange

`"stateChange" -> function(state)`

parameters:

- state - String, either "loading", "default", "expanded", “resized”, 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 container returns the value of the placement to creative so that creative can behave differently as necessary. The container 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 default ad placement is inline with content in the display (i.e. a banner)
interstitial	the ad placement is over laid on top of content

getPlacementType should always return the placement that it initially displayed in. That is, in the case of two-part expandables, the second, expanded part should also see “inline” if it does a getPlacementType.

```
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 interstitial ads, 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.

For a two-piece expandable ad, when the ad state is expanded, isViewable will return an answer based on the viewability of the expanded piece of the ad.

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.

Note that MRAID does not define a minimum threshold percentage or number of pixels of the ad that must be onscreen to constitute "viewable."[2]

`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

Below is an example of an ad that takes into account both "Ready" and "isViewable" before taking action.

```
// Wait for the SDK to become ready
if (mraid.getState() === 'loading') {
    mraid.addEventListener('ready', onSdkReady);
} else {
    onSdkReady();
}

function onSdkReady() {
    // Wait for the ad to become viewable for the
first time
    if (mraid.isViewable()) {
        showMyAd();
    } else {
        mraid.addEventListener('viewableChange',
function(viewable) {
```

[2] The IAB currently has a project underway to establish guidelines for in-app ad measurement, potentially including a viewability threshold, is currently underway within the IAB.

```
                if (viewable) {

mraid.removeEventListener('viewableChange',
arguments.callee);
                    showMyAd();
                }
            });
        }

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

Changing the Size of an Ad

MRAID v2 includes three distinct ways for an ad to change its size. By far the simplest is to use the open() method, which is intended for click-throughs where an entire web site is loaded in a new browser window (generally this will be a browser running within the app, but it may be the device's native browser).

The expand() method is intended for ads that expand in a fairly simple, straightforward way to cover the content of the application.

In addition to these, the resize() method is intended for ads that grow or shrink in more subtle ways, in a dialogue with the app in which it is running. This method allows designers complete freedom and control – with the trade-off that additional methods and listeners are required for both the ad creative and the app/container to react appropriately in different placements.

Differences between resize(), expand(), and open()

Although these methods are related, they promote an approach of progressive complexity. That is, simple operations should be simple, but sophisticated efforts are still be possible. Understanding the distinction between resize(), expand() and open() helps ad designers choose the best method for their needs.

open()

- lowest common denominator
- used for advertiser landing pages or microsites
- opens a new URL, generally in a browser window within the app, however may open in the device's native browser
- always full screen
- no additional properties

expand()
- simple interface
- maintains ad experience
- full screen
- few additional properties
- support for one-part or two-part creatives
- MRAID-enforced tap-to-close area in fixed (top right) location
- relative alignment for creatives

resize()
- flexible interface
- continuous, non-modal ad experience
- no default values, can change to larger or smaller sizes
- additional properties and methods required
- one-part creatives only
- MRAID-enforced tap-to-close area, but ad designer can change the close area's position within the creative area.
- absolute positioning possible
- supports direction of resizing

This table summarizes the differences between these methods.

property	**open()**	**expand()**	**resize()**
modal	Y	Y	N
MRAID-enforced close control	N	Y	Y
viewer stays within ad experience	N	Y	Y
two-part creatives	n/a	Y	N
one-part creatives	n/a	Y	Y
aligned to screen	n/a	Y	N
background provided for small creatives	n/a	Y	N
size up	n/a	Y	Y
size down	n/a	N	Y
app-defined max area	n/a	N	Y
callback required to complete	n/a	N	Y
supports directionality	n/a	N	Y
creative can control position of resized ad	n/a	N	Y
app can return to default state	n/a	N	Y

In MRAID 2.0 the creation of partial-screen non-modal expansions requires using the resize() method. The deprecation of the expand property "isModal" is reflected in this usage chart. Under MRAID 2.0, calling expand() using expanded creative that is smaller than the size of the full screen requires the container web view blank out, cover, or otherwise obscure the underlying app to make it very clear to the end user that the expanded ad is modal in nature. In that sense, modal, partial screen ads are not allowed in MRAID 2.0.

	Modal	**Non-modal**
Full Screen	OK - Use expand()	*Not possible*
Partial Screen	*Not possible*	OK - Use resize()

Open: Open an External Mobile Web Site in a Browser Window

If the ad needs to open an external mobile web site, or micro site, from an MRAID ad, it can call the open method which will open a browser window to view the external HTML content. Wherever possible, this will be via an embedded browser 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 implementation. To remain within an MRAID ad experience, use the expand() method instead.

The native browser controls – back, forward, refresh, close – will always be present. For reporting, open() may be used by SDK vendors as a reportable event.

open(URL)

parameters:

- URL - String, the URL of the web page

return values:

- None

Hyperlinks

When the user clicks on an HTML hyperlink (defined by an <a href=""> tag) in an MRAID ad, there are two possibilities: the linked page could load in the existing ad web view, or the content could open a separate browser window and load the indicated HTML link there. MRAID-compliant SDKs can opt for either strategy, so ad designers should avoid using inline hyperlinks and window.location changes. mraid.open() is the appropriate and correct way for an MRAID ad to specify that a link should open a page in a separate browser. 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.

Handling Call-to-Action Events

A rich media ad implements multiple call-to-action events beyond the tap to microsite. These events may be executed as anchor links or scripted functions. This means a container or SDK cannot just listen for taps in the browser. It must support programmatic taps/clicks (e.g., window.location changes) as well.

Expand: Simple, Modal, Increase in Size of the Ad

For ad creative that changes size in a relatively simple manner, typically expanding from banner to full-screen size, the expand method provides a simple way to communicate this to the container.

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 level (e.g., at a higher z-index value than any app content) 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).

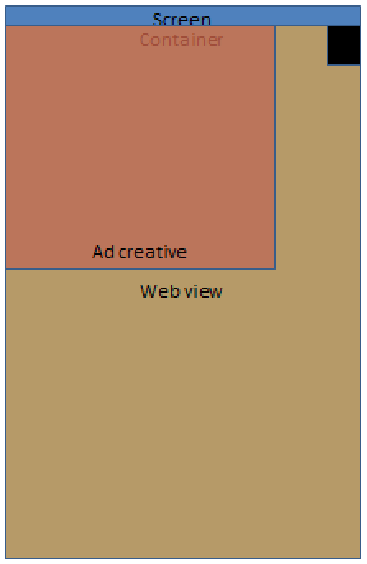
The expand method may change the size of the ad container, and will move state from "default" or "resized" to "expanded" and fire the stateChange event. In the case of both one-piece ads and two-piece ads, calling expand() multiple times will be ignored, and multiple expand calls have no effect on state (which remains "expanded").

An expanded view must cover all available screen area even though the ad creative may not (e.g. via a transparent or opaque overlay). The expanded ad is always modal, and naturally the container 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 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 ad designer. 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 or smaller than the screen size of the device, the SDK will size the web view to be identical to the maximum size allowed by the device and app. The creative will not be scaled down or up to the size of the device's screen; rather it will be up to the ad creative to position itself appropriately within the expanded web view via CSS.



Expand with small creative

In this example, status bar still shows on screen (max size is less than screen size)

Container is max size and has close event region

Web view fills container size

Creative positions itself within web view

Note that the close tappable area/ indicator is in top right corner of the ad container—so NOT on top of ad creative.

When the expand method is called without the URL parameter, the current 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 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 or a non-URL parameter is used, the body of the current ad will be used in the current webview.

return values:

- none

events triggered:

stateChange

Controlling Expand Properties

The expand properties object is intended to provide additional features to ad designers. Expand properties that can be set by the ad designer are limited to the width and height of the ad creative, and whether the creative is supplying its own close indicator. The expandProperties are held in a JavaScript object that can be written and read by the ad. Ad designers can also control the orientation of an expandable ad via orientation properties, set separately.

Expand properties can only be set BEFORE the ad calls expand(). Changes after the ad is in its expanded state will be ignored.

```
expandProperties object = {
 “width” : integer,
 “height” : integer,
 “useCustomClose” : boolean,
 “isModal” : boolean (read only)
}
```

properties:

- width : integer – width of creative, default is full screen width.
- height : integer – height of creative, 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, container will stop showing default close graphic and rely on ad creative’s custom close indicator; false (default), container 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 container is modal for the expanded ad; false, the container is not modal for the expanded ad; this property is read-only and cannot be set by the ad designer. Note that while this could be false in MRAID 1.0, in MRAID v2.0 will always return “true.”

getExpandProperties method
The getExpandProperties method returns the whole JavaScript Object expandProperties object.

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

```
getExpandProperties() -> JavaScript Object
```

parameters:

- none

return values:

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

events triggered:

- none

setExpandProperties method
The setExpandProperties method sets the whole JavaScript object.

```
setExpandProperties(properties)
```

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

parameters:

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

return values:

- none

events triggered:

- none

Controlling Orientation Properties

The orientation properties object is intended to provide ad designers with additional control over expandable and interstitial ads. The orientationProperties are held in a JavaScript object that can be written and read by the ad. The orientationProperties object only affects the expanded state of an expandable ad, or an interstitial ad. A banner in its default state cannot use orientationProperties to prevent the app from reorienting or force the app to switch to a different orientation layout. Resizeable ads can use orientationproperties, but they won't have any effect.

```
orientationProperties object = {
  "allowOrientationChange" : boolean,
 "forceOrientation" : "portrait|landscape|none"
}
```

- allowOrientationChange : boolean – If set to "true" then the container will permit orientation changes; if set to false, then the container will ignore orientation changes (e.g., the web view will not change even if the orientation of the device changes). Default is "true."
- forceOrientation : string – can be set to a value of "portrait," landscape," or "none." If forceOrientation is set then a view must open in the specified orientation, regardless of the orientation of the device. That is, if a user is viewing an ad in landscape mode, and taps to expand it, if the ad designer has set the forceOrientation orienation property to "portrait" then the ad will open in portrait orientation. Default is "none."

To enable finer control over ad behavior, an ad designer can change the setting of either of the orientation properties after the ad is in an expanded state. This way an ad may start in portrait but instruct the user to change orientation to play a game. The game requires tilting so no orientation changes should be allowed until the user is done. MRAID-compliant SDKs must be able to accept changes to expand properties throughout a user's interaction with an expandable ad.

For example:

```
mraid.setOrientationProperties (
{"allowOrientationChange":true} );
mraid.expand()

/* user changes to landscape, starts game */
mraid.setOrientationProperties ( {"allowOrientationChange":
false } );

/* user is done with game */
mraid.setOrientationProperties (
{"allowOrientationChange":true} );
```

getOrientationProperties method
The getOrientationProperties method returns the whole JavaScript object orientationProperties object.

Use this method to get the properties for the orientation of the expanded part of an expandable, or an interstitial ad.

```
getOrientationProperties() -> JavaScript Object
```

parameters:

- none

return values:

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

events triggered:

- none

setOrientationProperties method
The setOrientationProperties method sets the JavaScript orientationProperties object.

```
setOrientationProperties(properties)
```

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

parameters:

- properties: JavaScript Object { ... } - this object contains the values for allowOrientationChange and forceOrientation.

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 differentiates two aspects to a "close" feature:

- Close Event Region: The close event region is a tappable area on the ad creative that will cause the ad to close and return to its default state. The close event region is required and supplied by the container in the top-right corner of all MRAID expandable and interstitial ads.
- Close Indicator: The close indicator is the visual cue to the user as to the location of the close event region. By default the container will supply a close indicator superimposed on the close event region. Optionally, the creative designer can use their own close indicator graphic, in which case they can suppress the default close indicator.

MRAID requires the location reserved for the close event region be a 50x50 clickable area in the top-right corner of the ad container. Reserving this location provides consistency for ad designers running campaigns across apps and rich media vendors. The default design of the container-controlled close indicator is left to the vendor/app publisher. Ad designers may optionally choose to provide the indicator for the default close capability. 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 container will supply its default close indicator. This container-supplied tappable area will be placed at a higher level than other app or ad content, and must always be available to the end user.

For Two-Part Ads: 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 app was suspended when the ad changed to the expand state, then the app should be notified of the expansion status change.

For One-Part Ads: If the expanded or interstitial ad view was closed using the container-supplied close event region, then the stateChange event is still fired and the app still notified of the expansion status change. Expanded ads must always listen for the stateChange event and adjust as necessary.

close method
The close method will cause the ad container to downgrade its state. For ads in an expanded or resized state, the close() method moves the ad to a default state. For interstitial ads in a default state, the close() method moves to a hidden state. For banners in a default state, the effect of calling close() is deliberately left undefined by the MRAID specification. Depending on the implementation, it may be ignored, cause an error, or change the state of the banner to "hidden." As a result it is generally not recommended that ad designers use mraid.close() in a banner. This method may be used by ad designers as an addition to the MRAID-enforced close ability. It will also fire the stateChange event.

Note that if an ad employs multiple resize() calls or a resize() followed by an expand(), close() changes the creative back to its default, banner state. It does NOT simply undo the most recently called resize() or expand().

```
close()
```

parameters:
- none

return values:
- none

event triggered:
- stateChange

useCustomClose method
Although MRAID requires all implementing containers 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 container 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 requirement of an MRAID-enforced close control, the ad designer should call this method immediately after the Ready event.

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 design for the close indicator
- false – container default image should be displayed for the close indicator

return values:

- none

events triggered:

- none

Resize: Enables Sophisticated Ad Size Changes

Ad creative that needs to engage in a succession of size changes, or to change size non-modally to less-than-fullscreen size, has the ability to do so by calling resize. As with expand, the resize method operates at a higher z-index than the app content, and so is positioned above the underlying content, and so does not push or reposition the app content.

resize method
The resize method will cause the existing web view to change size using the existing HTML document. Like expand(), resize() size changes happen at highest level in the view hierarchy, and so do not automatically shift or otherwise reposition underlying content. App publishers that want to support content-shifting ads like "push-downs" can do so using resize but must implement the repositioning of app content in response to the resize independently.

The resize method will move the state from "default" to "resized" and fire the stateChange event. Resize can be called multiple times by the creative. Additional calls to resize will also trigger the stateChanged event although the state value will remain "resized." Calls to resize from an "expanded" state will trigger an error event and not change the state.

Note: resize should not be used for ad creative that expands to full-screen (or larger) size: for such creative executions expand() should always be used. Resize will always result in a non-modal size change, and some portion of the app should always remain visible to the end user.

Use this method to request a resize of the default ad view to a desired size and screen position. Note that resize() relies on parameters that are stored in the resizeProperties JavaScript object. Thus the creative must set those parameters via the setResizeProperties() method BEFORE attempting to resize(). Calling resize() before setResizeProperties will result in an error.

The container will notify the app of the resize request so that the app can react to the change as appropriate. For example, a publisher integration may listen for resize() calls to implement behavior like a "push-down" ad. If the resize is valid, then the sizeChange event is fired. If the parameters are out of range, then the error event identifies the exception.

`resize()`

parameters:

- none

return values:

- none

events triggered:

- sizeChange, stateChange

side effects:

- changes state

Close Control for Resized Ads

As with expandable ads, resized ads must have a way for the person viewing the ad to return the ad to its default state. MRAID differentiates two aspects to a "close" feature:

- Close event region: The close event region is a tappable area on the ad creative that will cause the ad to close/collapse back to its default state. The close event region is required and supplied by the container in a creative-specified location for all MRAID resizable ads.
- Close Indicator: The close indicator is the visual cue to the user as to the location of the close event region. For resized ads, the container does NOT supply a close indicator superimposed on the close event region. Instead, FOR RESIZED ADS, THE CREATIVE MUST ALWAYS SUPPLY ITS OWN CLOSE INDICATOR GRAPHIC.

MRAID-compliant SDKs must therefore always supply containers with a 50x50 close event region located on the ad creative, tapping on which will return the ad to its default state. While this close event region must be present, the ad designer can specify where on the ad the control should be located. If the ad designer opts not to specify a location for the close event region then by default the container will position it at the top right corner of the resized ad container.

Unlike the case of expand(), for resize() the container will not supply a close indicator. Rather, it is expected that the ad designer will include a close indicator in the creative.

While the tappable close control is mandatory, ad designers are free to include other ways to close a resized ad, by using MRAID's close() method.

resizeProperties object

```
resizeProperties object = {
 “width” : integer,
 “height” : integer,
 “customClosePosition” : string,
 "offsetX" : integer,
 "offsetY" : integer,
 "allowOffscreen" : boolean
}
```

Notes:

- width : (required) integer – width of creative in pixels
- height : (required) integer – height of creative in pixels
- customClosePosition: (optional) string – either "top-left", "top-right", "center", "bottom-left", "bottom-right," "top-center," or "bottom-center" indicates the origin of the container-supplied close event region relative to the resized creative. If not specified or not one of these options, will default to top-right.
- offsetX is the horizontal delta from the banner's upper left-hand corner where the upper left-hand corner of the expanded region should be placed; positive integers for expand right; negative for left
- offsetY is the vertical delta from the banner's upper left-hand corner where the upper left-hand corner of the expanded region should be placed; positive integers for expand down; negative for up
- allowOffscreen tells the container whether or not it should allow the resized creative to be drawn fully/partially offscreen
  - True (default): the container should not attempt to position the resized creative
  - False: the container should try to reposition the resized creative to always fit in the getMaxSize() area

When allowOffscreen is set to False, the SDK will do its best to move the default (banner) ad container to ensure that the resized creative fits on the screen. For example, if ad is on the top of the screen, and ad wants to resize upwards by 50 pixels, then the SDK will move the default (banner) ad 50 pixels down and then execute the resize. If allowOffscreen is set to true in this case, the resized portion of the ad will extend off the top of the screen.

allowOffscreen cannot solve all positioning issues. For example, if an ad successfully resizes in landscape orientation, but then becomes larger than the size of the screen due to an orientation change to portrait, the setting of allowOffscreen to false will have no effect, as there is no way the container/SDK can successfully reposition a landscape creative to fit on a portrait screen.

getResizeProperties method

The getResizeProperties method returns the whole JavaScript object resizeProperties object.

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

`getResizeProperties()` -> JavaScript Object

parameters:

- none

return values:
- { ... } - this object contains the resize properties

events triggered:
- none

setResizeProperties method
The setResizeProperties method sets the whole JavaScript object.

```
setResizeProperties(properties)
```

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

parameters:
- properties: JavaScript Object { ... } - this object contains the width and height of the resized ad, close position, offset direction (all in density-independent pixels), and whether the ad can resize offscreen. For more info see properties object.

return values:
- none

events triggered:
- none

Resize ads should be QA tested carefully. Ads that set parameters that are impossible for the container to follow will result in an error event being triggered and the resize will not take place. For example, an error will occur if an ad sets allowOffscreen to "false" but sets the width and height of the resize to be too big to actually fit on the screen.

Checking Position and Size of the Screen and Ad
MRAID v2.0 includes several methods enabling an ad to check where and how large it is, and the maximum size it can expand to. Ad designers can use these capabilities to give their ads increased flexibility to behave differently on different devices and/or differently sized screens.

getCurrentPosition method
The getCurrentPosition method will return the current position and size of the ad view, measured in density-independent pixels.

```
getCurrentPosition() -> JavaScript Object
```

parameters:
- none

return value:

- JavaScript Object - {x, y, width, height}: x=number of density-independent pixels offset from left edge of the rectangle defining getMaxSize(); y=number of density-independent pixels offset from top of the rectangle defining getMaxSize(); width=current width of container; height=current height of container (both measured in density-independent pixels)

related events:

- none

getMaxSize method
The getMaxSize method returns the maximum size (in density-independent pixel width and height) an ad can expand or resize to. Depending on the device, the max size may differ from the full screen dimensions, due to screen area reserved for a status bar or other elements outside the app.

Use this method to return the size of the view that contains the app (which defines the maximum space the ad may expand or resize within).

```
getMaxSize() -> JavaScript Object
```

parameters:

- none

return value:

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

related events:

- none

sizeChange event
The sizeChange event fires when the ad's 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. Measurements are in density-independent pixels.

This event is triggered when the display state of the ad's web view 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, measured in density-independent pixels, regardless of what state the calling view is in.

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

`getDefaultPosition() -> JavaScript Object`

parameters:

- none

return values:

- JavaScript Object - {x, y, width, height}: x=number of density-independent pixels offset from left of getMaxSize(); y=number of density-independent pixels offset from top of getMaxSize(); width=current width of container; height=current height of container

getScreenSize method
The getScreenSize method returns the current actual pixel width and height, based on the current orientation, in density-independent pixels, of the device on which the ad is running. Note that the ScreenSize will change if the device is turned from portrait to landscape mode (and vice versa). Note also that getScreenSize will return the TOTAL size of the device screen, including area (if any) reserved by the OS for status/system bars or other functions, which cannot be overridden by the app or the ad. Designers seeking to enable creative to check how much usable screen real estate is available should use getMaxSize rather than getScreenSize.

`getScreenSize() -> JavaScript Object`

parameters:

- none

return values:

- {width, height}

related event:

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.

MRAID has the potential to provide common APIs to facilitate storing and forwarding of ad impression delivery, view, and other metrics from the app back to the ad server. However, until measurement methodologies and the metrics themselves are standardized (for example by the ongoing IAB/MMA/MRC In-App Ad Measurement Guidelines project), adding measurement functionality to MRAID would be premature.

The MRAID working group expects that this capability will be evaluated and potentially added to MRAID as part of a future release.

Access to Native Features

MRAID encourages the use of standard web technologies in ad design as much as possible for presentation needs, basic functions, and even an increasing list of the advanced ad functionalities required for truly rich media advertising. MRAID's role around access to native features is to help rich media ads discover what capabilities a device will support, and to fill in any gaps in capability not widely available, or not fully stabilized and consistent, within HTML5/Webkit.

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

An MRAID compliant SDK must be able to deliver all of these functionalities on any device that is capable of them. However, individual publisher implementations of the SDK may result in deactivating features/capabilities that conflict with publisher policies.

value	**description**
sms	the device supports using the sms: protocol to send an SMS message
tel	the device supports initiating calls using the tel: protocol
calendar	the device can create a calendar entry
storePicture	the device supports the MRAID storePicture method
inlineVideo	the device can play back video files inline rather than requiring use of the native video player

```
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. Access to most of these capabilities is standardized in HTML5 at present (or will be in the near future), and where an open standard provides access to a device feature or capability, MRAID defers to the open standard.

Device Orientation

Ad creative should be able to request the orientation of a device/web container via HTML5 with consistent results across devices and MRAID implementations. For Android implementations and iOS versions after 5.0 this happens automatically; however, earlier iOS implementations require a code tweak on the part of the SDK vendor in order to window orientation changes fire events properly.

To be compliant with MRAID 2.0, an SDK needs to deploy a code modification of this sort for pre-iOS 5.0 Apple devices. While SDK vendors can use whatever technical solution they prefer to achieve this, the following sample code offers an example of a means to address this issue.

```
-
(void)didRotateFromInterfaceOrientation:(UIInterfaceOrienta
tion) fromInterfaceOrientation {
  if (UIInterfaceOrientationIsPortrait(newOrientation) ||
      UIInterfaceOrientationIsLandscape(newOrientation)) {

      NSInteger degrees = 0;

      switch (self.interfaceOrientation) {
        case UIInterfaceOrientationPortrait:
          degrees = 0;
          break;
        case UIInterfaceOrientationLandscapeLeft:
          degrees = 90;
          break;
        case UIInterfaceOrientationLandscapeRight:
          degrees = -90;
          break;
        case UIInterfaceOrientationPortraitUpsideDown:
          degrees = 180;
          break;
        default:
          // Don't care about this orientation.
          return;
      }

      // Update the window.orientation property then
trigger
      // onorientationchange().
      NSString *javascript = [NSString stringWithFormat:
          // Create the 'window.orientation' read-only
property.
```

```
          @"window.__defineGetter__('orientation',function(
){return %i;});"
          // Dispatch the 'orientationchange' event.  This
also calls
          // 'window.onorientationchange()'.
          @"(function(){"
            @"var event = document.createEvent('Events');"
            @"event.initEvent('orientationchange', true,
false);"
            @"window.dispatchEvent(event);"
          @"})();",
          degrees];
      [self.webView
stringByEvaluatingJavaScriptFromString:javascript];
  }
}
```

Ad designers cannot rely on window.orientation to determine whether a device is in portrait or landscape mode. The value of window.orientation is intended to indicate the screen's position in relation to the "standard" orientation axis, i.e., the axis on which the values of a DeviceOrientationEvent are reported. However, that standard orientation may not be portrait (height greater than width) mode. Indeed, on widescreen Android tablets, such as the Samsung Galaxy Tab 10.1., window.orientation is set to zero when the device is in landscape (width greater than height) mode.

Ad designers should instead use the mraid.getScreenSize() method to retrieve the current width and height of the device screen.

Storing a Picture

Rich Media Ad designers may want to add a picture to the camera roll or photo album of the device they are running on. This can be handy for a number of features, including storing coupons for later redemption.

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. To ensure that the user is aware a picture is being added to the photo album, MRAID requires the SDK/container use an OS-level handler to display a modal dialog box asking that the user confirm or cancel the addition to the photo album for each image added. If the device does not have a native "add photo" confirmation handler, the SDK should treat the device as though it does not support storePicture.

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

MRAID-compliant containers will support adding a picture via an HTTP redirect (for tracking purposes); however they will not necessarily support meta redirects.

If the attempt to add the picture fails for any reason or is cancelled by the user, it will trigger an error.

`storePicture(URI)`

parameter:

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

related event:

- none

Creating Calendar Events

createCalendarEvent method

The createCalendarEvent method opens the device UI to create a new calendar event. The ad is suspended while the UI is open. To ensure the creation of a calendar event is always user initiated and authorized, MRAID-compliant containers must invoke the device's native "create calendar event" sheet, pre-populated with data supplied by the ad. Where a device does not support such a "create calendar event" sheet, the SDK should treat that device as if it does not support adding calendar events.

Calendar event data should be delivered in the form of a JavaScript object written to the W3C's calendar specification. See Appendix.

If the attempt to create the calendar event fails or is cancelled by the user, it will trigger an error.

`createCalendarEvent(parameters)`

parameters:

- parameters: JavaScript Object {...} – this object contains the parameters for the calendar entry, written according to the W3C specification for calendar entries. See Appendix.

return value:

- none

related event:

- none

For example, the following would add a calendar event for the Mayan Apocalypse/End of the World on December 21, 2012, taking place "everywhere" and starting at midnight Eastern time and ending at midnight Eastern time on December 22, 2012.

```
createCalendarEvent({description: "Mayan
Apocalypse/End of World", location: 'everywhere',
start: '2012-12-21T00:00-05:00, end: '2012-12-
22T00:00-05:00'})
```

Working with Video

Video on mobile devices can be played either inline (within the current web view, app, or mobile web page) or by opening a native player on the device. For many advertising applications, inline playback will be preferred: it is less disruptive to the viewer's experience, and playback within a web view enables HTML5 reporting on metrics related to how much of the creative was viewed. These metrics are generally harder to access, or unavailable, when video is viewed in the native player.

Ad designers must keep in mind that device/OS limitations may prevent inline video playback (this is notably the case with devices running Android version 2.x and earlier).

However, MRAID-compliant containers should support inline playback where possible, and permit ad designers to specify if video creative should play inline or in a separate player. Ad designers can use the "supports(inlineVideo)" method to determine whether the device running the creative will display video inline.

In order to enable inline video playback and autoplay of video, MRAID-compliant SDKs should consistently insert the any necessary enabling tags into the web view depending on operating system of the device.

For iOS devices, the following tags must be used:

- `webView.mediaPlaybackRequiresUserAction = NO;`
- `webView.allowsInlineMediaPlayback = YES;`

For Android (Honeycomb, Ice Cream Sandwich and above) devices, the SDK must invoke hardware acceleration, which is dependent on the view in question and how it is added to the WindowManager:

- `getWindow().setFlags(WindowManager.LayoutParams.FLAG_HARDWARE_ACCELERATED, WindowManager.LayoutParams.FLAG_HARDWARE_ACCELERATED);`

For Android 2.x and earlier devices, it is not possible to play video inline; the native player is always invoked by the playVideo method.

playVideo method
Use this method to play a video on the device via the device's native, external player. Note that this is purely a convenience method for the OS's existing external player, and does not imply a separate, SDK-based video player. To play video inline (on devices where that feature is supported), use HTML5 video tags.

`playVideo(URI)`

parameters:

- URI - String, the URI of the video or video stream

return values:

- none

Appendix: W3C CalenderEvent Interface

Taken from: W3C Calendar API, Sections 4.3 and 4.4
W3C Working Draft 19 April 2011
This version:
    http://www.w3.org/TR/2011/WD-calendar-api-20110419/
Latest published version:
    http://www.w3.org/TR/calendar-api/
Latest editor's draft:
    http://dev.w3.org/2009/dap/calendar/
Editors:
    Richard Tibbett, Opera Software ASA
    Suresh Chitturi, Research in Motion (RIM)

4.3 CalendarEvent interface

The CalendarEvent interface captures a calendar event object.

The current use of DOMString for dates and times is known to be insufficient for representing events with timezones. The group is working on addressing that limitation, looking at the development of TZDate object that would address this.

```
[NoInterfaceObject]
interface CalendarEvent {
    readonly attribute DOMString           id;
             attribute DOMString           description;
             attribute DOMString?          location;
             attribute DOMString?          summary;
             attribute DOMString           start;
             attribute DOMString?          end;
             attribute DOMString?          status;
             attribute DOMString?          transparency;
             attribute CalendarRepeatRule? recurrence;
             attribute DOMString?          reminder;
};
```

4.3.1 Attributes

`description` of type DOMString

A description of the event.

```
{description: "Meeting with Joe's team"}
```

No exceptions.

`end` of type DOMString, nullable

The end date and time of the event as a valid date or time string.

```
{end: '2011-03-24T10:00:00-08:00'} // Event ends on March 24,
2011 @ 6pm (UTC)
```

No exceptions.

`id` of type DOMString, readonly

A globally unique identifier for the given `CalendarEvent` object. Each `CalendarEvent` referenced from `Calendar` MUST include a non-empty `id` value.

An implementation MUST maintain this globally unique resource identifier when a calendar event is added to, or present within, a Calendar.

An implementation MAY use an IANA registered identifier format. The value can also be a non-standard format.

No exceptions.

`location` of type DOMString, nullable

A plain text description of the location of the event.

```
{location: 'Conf call #+4402000000001'}
```

No exceptions.

`recurrence` of type *CalendarRepeatRule*, nullable

The recurrence or repetition rule for this event

```
{recurrence: {frequency: 'daily'}}     // Event occurs every
day and never expires

{recurrence: {frequency: 'weekly',     // Event occurs
weekly...

daysInWeek: [2, 4],      // ...every Tuesday and Thursday
```

```
expires: '2011-06-11T12:00:00-04:00'}} // Event expires on or
before June 11, 2011 @ 4pm (UTC)

{recurrence: {frequency: 'weekly',     // Event occurs
weekly...on every Wednesday

                                       // (if we say the
'start' attribute is March 24, 2011 @ 2pm (Wednesday) as

                                       // shown above and no
daysInWeek attribute is provided)

expires: '2011-06-11T11:00:00-05:00'}} // Event expires on or
before June 11, 2011 @ 4pm (UTC)

{recurrence: {frequency: 'monthly',    // Event occurs
monthly...

daysInMonth: [-5],       // ...5 days before the end of each
month

expires: '2011-06-11T20:00:00+04:00'}} // Event expires on or
before June 11, 2011 @ 4pm (UTC)

{recurrence: {frequency: 'monthly',    // Event occurs
monthly...on the 24th day of every month

                                       // (if we say the
'start' attribute is March 24, 2011 @ 2pm as

                                       // shown above and no
daysInMonth attribute is provided)

expires: '2011-06-11T20:00:00+04:00'}} // Event expires on or
before June 11, 2011 @ 4pm (UTC)

{recurrence: {frequency: 'yearly',     // Event occurs
yearly...on the 24th day of every March

                                       // (if we say the
'start' attribute is March 24, 2011 @ 2pm as

                                       // shown above and no
daysInMonth attribute is provided)

expires: '2011-06-11T16:00:00+00:00'}} // Event expires on or
before June 11, 2011 @ 4pm (UTC)

{recurrence: {frequency: 'yearly',     // Event occurs
yearly...

daysInMonth: [24],       // ...every 24th day...

monthsInYear: [3, 6],    // ...in every March and June
```

```
expires: '2011-06-11T16:00:00Z'}} // Event expires on or
before June 11, 2011 @ 4pm (UTC)

{recurrence: {frequency: 'yearly',     // Event occurs
yearly...

daysInYear: [168],       // ...every 168th day of each year

expires: '2011-06-11T21:45:00+05:45'}} // Event expires on or
before June 11, 2011 @ 4pm (UTC)
```

No exceptions.

`reminder` of type DOMString, nullable

A reminder for the event.

This attribute can be specified as a positive valid date or time string.

, denoting a one-time reminder or as a negative value in milliseconds denoting a relative relationship to the start time of the calendar event.

A relative reminder is recommended for setting a reminder for recurrent events.

```
{reminder: '2011-03-24T13:00:00+00:00'}  // Remind ONCE on
March 24, 2011 @ 1pm (UTC)

{reminder: '-3600000'}       // Remind 1 hour before every
occurrence of this event
```

No exceptions.

`start` of type DOMString

The start date and time of the event as a valid date or time string.

```
{start: '2011-03-24T09:00-08:00'} // Event starts on March
24, 2011 @ 5pm (UTC)
```

No exceptions.

`status` of type DOMString, nullable

An indication of the user's status of the event.

This parameter may be set to one of the following constants:

```
'pending', 'tentative', 'confirmed', 'cancelled'.
```

```
{status: 'pending'} // Event is awaiting user action
```

No exceptions.

`summary` of type DOMString, nullable

A summary of the event.

```
{summary: "Agenda:\n\n\t* Introductions\n\t* AoB"}
```

No exceptions.

`transparency` of type DOMString, nullable

An indication of the display status to set for the event.

This parameter may be set to one of the following constants:

```
'transparent', 'opaque'.
```

```
{freebusy: 'transparent'} // Mark event as transparent in
Calendar
```

No exceptions.