# Mobile Rich Media Ad Interface Definition (MRAID)

VERSION 3.0

JULY 2017

# Mobile Rich Media Ad Interface Definition (MRAID) Version 3.0

MRAID Version 3.0 (MRAID 3.0) has important new features to improve the user's ad experience. The new MRAID enables the ad to measure viewability and audibility, detect MRAID environment, and get location data to present user with the best possible experience. In addition, there is guidance on pre-loading ads and understanding of ad readiness to display an ad to the user. The Video Player Ad Interface Definition (VPAID) addendum to MRAID is now fully included in the MRAID specification.

This MRAID version has been developed by the IAB Tech Lab Mobile Rich Media Ad Interface Definition (MRAID) Working Group.

## About IAB Tech Lab

The IAB Technology Laboratory is an independent, international, research and development consortium charged with producing and helping companies implement global industry technical standards. Comprised of digital publishers and ad technology firms, as well as marketers, agencies, and other companies with interests in the interactive marketing arena, the IAB Tech Lab's goal is to reduce friction associated with the digital advertising and marketing supply chain, while contributing to the safe and secure growth of the industry. Learn more about IAB Tech Lab here.

**This document is on the IAB website at: https://www.iab.com/mraid**

The following IAB member companies participated in the above working group:

AccuWeather.com
AdColony
Adform
AdGear Technologies, Inc.
Adobe
Adsiduous Media
ADTECH
AdTheorent
ADVR
AerServ
Alliance for Audited Media (AAM)
Amazon
AOL
BabyCenter
Bazaarvoice
Bonzai
Cedato Technologies Ltd
Celtra
Conversant Media
Cyber Communications Inc.
Digital Advertising Consortium Inc.
Dominion Enterprises
DoubleVerify
Flashtalking
FOX Networks Group
FreeWheel
Google
Gruuv Interactive
Hulu
IAB
Improve Digital International B.V.
Innovid
Integral Ad Science
Leaf Group
Liquidus
LogoBar Enterprises
MGID
Microsoft Advertising
MoPub/ Twitter Inc.
mPlatform
NinthDecimal
PadSquad
Pixalate
RhythmOne
Shazam
Sizmek
Taboola
The Media Trust Company
The New York Times Company
Thinknear by Telenav
Time Inc.
Tremor Video
Turner Broadcasting System
Vdopia
Vertebrae
Verve
ViralGains
Visible Measures
Westwood One
White Ops
Yahoo
Yieldmo
YuMe
Zynga

© 2016 IAB Technology Laboratory

# Table of Contents

Executive summary ....7
Audience ....7

1 Introduction .... 8

1.1 Definitions ....8
1.2 Scope ....9
1.3 How MRAID Works ....9
1.4 Versions ....10
1.4.1 Updates in MRAID 3.0 ....11

2 Overview .... 12

2.1 Web Technologies Support ....12
2.1.1 Ad Server ....13
2.1.2 Ad Rendering ....13
2.2 Ad Control ....13
2.3 Interface ....14
2.4 Offline Requests and Metrics ....15
2.5 DAA Ad Marker Implementation ....16

3 Initialization and Set-up .... 16

3.1 Initialization Overview ....16
3.1.1 Checking that MRAID Ad is Loaded ....17
3.1.2 Declaring MRAID Environment Details ....17
3.1.3 Identification ....19
3.1.4 Implementation of MRAID Events ....20
3.1.5 Loading and Showing Interstitial Ads ....24
3.1.6 Using iframes ....24
3.1.7 Viewport and Default Container Set-Up ....24
3.1.8 Standard Image for Initial Display ....25
3.1.9 Event Handling ....25

4 Features and Operation .... 25

4.1 Viewability ....25
4.1.1 Polling Rates and Event Thresholds ....26
4.1.2 Implementation Considerations ....27

**4.2 Ad Controls for Display ........ 29**
4.2.1 Ad States and How They're Changed ........ 29
4.2.2 Checking Position and Size of the Screen and Ad ........ 30
4.2.3 Changing the Size of an Ad ........ 31
4.2.4 Differences between open(), expand(), and resize() ........ 31

**5 MRAID Methods ........ 34**

**5.1 getVersion() ........ 34**
**5.2 addEventListener() ........ 34**
**5.3 removeEventListener() ........ 35**
**5.4 open() ........ 35**
**5.5 close() ........ 36**
**5.6 unload() ........ 38**
**5.7 useCustomClose() (deprecated) ........ 38**
**5.8 expand() ........ 39**
**5.9 isViewable() (deprecated) ........ 41**
**5.10 playVideo() ........ 41**
**5.11 resize() ........ 41**
**5.12 storePicture() ........ 42**
**5.13 createCalendarEvent() ........ 43**
**5.14 VPAID methods ........ 44**
5.14.1 initVpaid() ........ 44
5.14.2 vpaidObject.subscribe() ........ 44
5.14.3 vpaidObject.startAd() ........ 44
5.14.4 vpaidObject.unsubscribe() ........ 44
5.14.5 vpaidObject.getAdDuration() ........ 44
5.14.6 vpaidObject.getAdRemainingTime() ........ 45

**6 Properties ........ 46**

**6.1 supports ........ 46**
**6.2 getPlacementType ........ 47**
**6.3 get/set orientationProperties ........ 47**
**6.4 getCurrentAppOrientation ........ 49**
**6.5 getCurrentPosition ........ 50**
**6.6 getDefaultPosition ........ 50**
**6.7 getState ........ 51**
**6.8 get/set expandProperties ........ 51**
**6.9 getMaxSize ........ 53**
**6.10 getScreenSize ........ 53**
**6.11 get/set resizeProperties ........ 54**

6.12 getLocation ... 56

7 Events ... 57

7.1 Error ... 57
7.2 ready ... 58
7.3 sizeChange ... 59
7.4 stateChange ... 59
7.5 exposureChange ... 60
7.6 audioVolumeChange ... 62
7.7 viewableChange (deprecated) ... 65

8 Working with Device Features ... 65

8.1 Device Orientation ... 65
8.2 Store a picture ... 66
8.3 Calendar Events ... 67
8.4 Video ... 68

9 VPAID Events and Methods ... 69

9.1 VPAID Interaction in MRAID Ads ... 69
9.1.1 Initializing VPAID in the MRAID Context ... 70
9.1.2 Sending and Receiving VPAID events ... 71
9.1.3 If VPAID Is Not Supported ... 71
9.1.4 VPAID AdClickThru Event ... 72
9.1.5 VPAID AdPaused, AdPlaying Events ... 72
9.1.6 Support for Auto-Start Video ... 72
9.2 Clickthrough Behavior and Viewability ... 73
9.3 Counting Impressions ... 74

10 Glossary of Terminology ... 75

11 Appendix: W3C CalendarEvent Interface ... 77

## Executive summary

MRAID is an acronym for Mobile Rich Media Ad Interface Definition. It enables indirect communication between an ad and the mobile app so the ad can execute a rich, interactive experience.

Without MRAID, ad developers would have to design interactive ads specifically for each proprietary system into which it would serve. The cost of such development would be prohibitive for brands to effectively advertise in mobile apps.

MRAID offers a library of calls that the ad can use to communicate with an MRAID-compliant app. As more mobile apps are equipped to handle MRAID ads, ad developers can count on a more predictable ad experience.

MRAID 3.0 offers updates that improve mobile ad execution with features that help track viewability, deliver clarity on initialization and ad readiness, and integrate with IAB VPAID (Video Player Ad Interface Definition) for ads with interactive video, and other incremental upgrades.

To make use of these upgrades, ad developers must adopt the new features to support relevant ad functions, and app developers must upgrade their mobile apps to support these new features or look to their MRAID SDK providers for an update.

As the use of mobile increases, so does the demand for higher quality ads. Buyers want improved tracking, while app publishers want improved ad experience. Implementing and/or upgrading to MRAID 3.0 offers support for both improved tracking and better performance for higher quality ads.

## Audience

MRAID is a protocol that ad designers and ad developers use to develop ads capable of interacting with an MRAID-compliant app. App developers or their SDK providers must provide a technical component to the app capable of setting up a container and responding to MRAID calls the ad makes.

Though this specification offers technical guidance for ad developers and app publishers or their SDK providers, anyone working in ad ops for mobile ad campaigns should be familiar with this document.

# 1 Introduction

MRAID specifies an API that enables interaction between a rich ad experience and the native mobile app into which it is served. The MRAID implementation for an app, usually offered as an SDK, initiates a container where the ad will display and a controller that the ad uses to interface with the app container.

An MRAID-compliant ad calls functions, provides information, and behaves according to this specification. An MRAID-compliant native app provides information, responds to MRAID calls, and behaves as defined in this specification.

## 1.1 Definitions

MRAID involves moving parts that work together to produce an interactive ad experience. In order to clearly represent these moving parts, the following keywords are defined as they are used in this document.

**host:** The component of a mobile app that provides the space (container) to display ads and the services (controller) that ads can access via the MRAID API. The host may be implemented as an SDK or it could be an inherent feature of the app.

**MRAID Implementation:** The features of the host that provide MRAID functionality to ads. This includes JavaScript properties through which the ad can detect the presence of MRAID, enable MRAID, and invoke MRAID services.

**SDK:** Acronym for Software Development Kit. In the case of MRAID, the SDK refers to the implementation code and instructions that providers offer to mobile app publishers.

**SDK provider:** For the purposes of this document, an SDK provider is a mobile ad tech services provider that develops the MRAID implementation code and instructions for mobile app publishers.

**ad container:** The constrained area in an app reserved for ad display. App publishers might arrange the ad container within the app content (inline) or reserve the container for display over the content (interstitial). Multiple containers may be available in the app.

**webview:** Technology of a mobile OS that displays web content and executes JavaScript. MRAID does not require the use of a webview, but in a typical MRAID host, the ad container contains a webview, where the ad is the top-level HTML document within the Webview.

**native layer:** The technical layer in which communication and operation takes place with regard to the native app.

**ad:** For the purpose of this document, an ad includes all the creative, library references, code, and other files, including any MRAID functions, used to display an MRAID ad in the mobile environment.

## 1.2 Scope

Each MRAID implementation offers unique feature sets to developers. This document explains the setup, initiation, functions, properties, events, and expected behaviors in response to these features. However, some operational details are excluded. Examples of features that are out-of-scope for this document include:

- 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

SDK providers must include the ability to render web content in the area intended for the ad unit. For most environments, this capability is already available as a webview but the developer may have to develop additional functions to support these specifications.

MRAID is not limited to features described in this document. Developers are encouraged to innovate and include features that differentiate them in the marketplace. However, in order to maintain an interoperable baseline of features, additional feature sets must be implemented outside the MRAID namespace. Awareness of extra feature sets and integration may be required to support functionality.

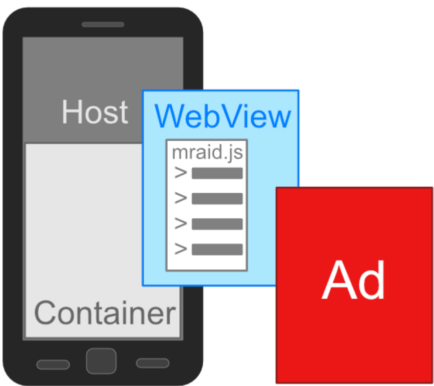
## 1.3 How MRAID Works

As a protocol between two systems, each system must be equipped to follow the call and response expectations. In the most simplistic sense of MRAID, the two systems involved are the ad and the mobile app. One system makes a call to the other and the other responds. However, the ad and app don't actually communicate directly.

Mobile ad campaign services (vendors) build the components for interaction to minimize the impact on mobile app developers. Ad technologist is a person or a platform that can implement MRAID specific functionality. Ad technologist may also help ad designer or ad developer use MRAID to implement interactive features. An ad designer or ad developer produces the ad with all the intended functionality, and the ad technologist uses scripts to standardize the functionality to MRAID specifications. The vendor may also provide some recommendations for technical designs that simplify the process.

Likewise, on the app side, a vendor develops the components for interaction within a container that a simple JavaScript tag initiates. The container is where the ad can be executed using standardized protocols for communication in a browser-like environment, or webview. The app may initiate a "listener" in order to track interactions or respond to certain events the ad sends.

The following image illustrates the working parts of an MRAID system.




## 1.4 Versions

Maintaining full backwards compatibility in MRAID is one of the goals of this project. In the MRAID 3.0 update, the Working Group maintained the six key goals established in version 2.0:

- **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 is especially important as MRAID continues to include device access in version updates.
- **Progressive complexity:** ad design using the API should be simple, adding complexity only as necessary.
- **Consistent means for ad operation:** MRAID provides ads a consistent means of communication regarding their need to expand and open an

app's embedded browser (or browser if an embedded browser does not exist).
- **Consistent exit controls:** MRAID ads will always have a consistent control that allows users to exit the ad experience and return to the app or content.
- **Flexibility for publishers:** although MRAID-compliant SDKs must support all MRAID capabilities, app publishers or 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 1.0**
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 2.0**
MRAID 2.0 extended the capabilities of MRAID 1.0 to give ad designers more control over expandable ads using the `resize()` method. Version 2.0 also offered the `supports()` feature as a standard way to query a device regarding certain capabilities. Other features included: consistent handling of video creative, adding an entry to the device calendar, and storing an image in the device photo roll.

**MRAID Video Addendum**
After the release of version 2.0, the working group drafted an addendum for making use of IAB's Video Player-Ad Interface Definition (VPAID) in the MRAID context. The Video Addendum provides guidance on how to initiate VPAID and how the host can track those interactions using select VPAID events.

### 1.4.1 Updates in MRAID 3.0

MRAID 3.0 comes with an overhaul to the documentation, integration of the VPAID addendum to MRAID 2.0, viewability changes, and other features designed to enhance the rich mobile ad experience. The updates in MRAID 3.0 are as follows:

- **Viewability:** The measures of viewability must consider factors beyond whether the current webview is in view. In MRAID 3.0, `isViewable()` is deprecated but remains in this version for backward compatibility. Instead, recommendations include practices that use `exposureChange` event to better communicate the visibility of the ad.
- **MRAID detection and initialization:** Previous MRAID versions were ambiguous about how the ad would initially detect whether it was running in an MRAID-compliant webview. Updates in MRAID 3.0 introduce

MRAID_ENV object to facilitate the early passing of version and other relevant attributes at initialization time.

- **Revised MRAID events implementation:** This version provides guidance for proper communication of states between the host and the ad and unambiguous implementation of sequence of events.
- **Audibility measurement:** A new event introduced for detecting whether audio can be heard and when volume changes.
- **Location:** A new addition to `supports` features to indicate whether access to location is enabled, and if so, provide information about the location.
- **Pre-loading and ad readiness:** Previous versions of MRAID lacked guidance on communicating whether the ad's assets were loaded and ready to display, sometimes causing blank screens to display and poor user experience. MRAID 3.0 offers guidance for the host on how to check for ad readiness before display and for how to pre-load interstitial ads appropriately.
- **unload method:** A new method introduced to provide a graceful exit mechanism for the ad in cases of runtime exceptions. It enables the ad to indicate to host to dismiss the webview when ad does not want to continue being shown to the user.
- **Two-part ads deprecated**: While two-part ads may still be used and existing features from MRAID 2.0 continue to support these ads, new features added to MRAID 3.0 are not designed to support two-part ads.
- **VPAID Events**: An addendum to MRAID 2.0 introduced support for initiating VPAID and reporting certain events. MRAID 3.0 integrates this addendum and offers optional compliance for the host to support ads developed using both MRAID and VPAID.
- **`useCustomClose():`** This method is being deprecated in MRAID 3.0.

# 2 Overview

MRAID is designed to take advantage of standardized web technologies for use in native apps. This overview provides background information on how these technologies are used in MRAID operations.

## 2.1 Web Technologies Support

For interoperability, MRAID should only use web-compatible languages for markup and scripting languages. While MRAID is code-agnostic, the assumption is that HTML, JavaScript, and CSS are used. The ad designer should be able to develop and test the ad unit in a web browser. If designers use tags, styles, and functions compatible with only one browser (such as CSS3 on WebKit), then the ad should be targeted to compatible devices.

When newer web standards provide consistency across browsers, ad designers are encouraged to use them. Examples of such protocols may include sms: and tel:

Since the publication of MRAID 2.0, HTML5 has been released and should be used as much as possible. Ad designers should be aware that despite increased consistency, expected protocols and implementations might still lack true interoperability across all devices and platforms.

### 2.1.1 Ad Server

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

### 2.1.2 Ad Rendering

An MRAID-compatible app must be capable of displaying any HTML ad. The process involves invoking an HTML file that includes a JavaScript tag used to initiate a rendering engine for executing ads. In this HTML file, that rendering engine is the "webview".

Whenever possible, the webview should incorporate the capabilities of the device web browser. For example, iOS developers may use `WKWebView`. Mobile apps may initiate multiple webviews that each act independently of one another. An ad may be built of multiple components that display in separate webviews.

## 2.2 Ad Control

Ad designers that expect ads to make use of MRAID must invoke the `mraid.js` script before the ad is loaded (see section 3 on initialization). Once called, the ad container can inject the MRAID JavaScript into the ad file.

The host then remains in the background, leaving the ad designer in control of ad display. When the ad needs to interact with the app, the host is made available to handle the interaction. This interaction between the ad and MRAID can be managed with little to no impact on either the ad designer or the app developer.

An ad that does not use any device features does not need to use MRAID, but, without MRAID, the host handles the ad as a simple display ad.

Some of the features an ad may take advantage of in the MRAID API include:

- Opening an embedded web browser
- Detecting exposure change and ad interaction
- Expanding an ad from a banner to a larger size
- Triggering an action in response to tapping, shaking, location, and other user-engagement activities

Ad designers are encouraged to rely on MRAID's capabilities to achieve the above effects. Even simple display ads must consider the use of MRAID for consistent hyperlink behavior.

## 2.3 Interface

Ad designers have access to the following methods, properties, and events:

| Methods | Description |
|---|---|
| getVersion | ad checks the version of the MRAID implementation that the host is using. |
| addEventListener | ad registers a listener for a specified event |
| removeEventListener | ad removes a listener for a specified event |
| open | ad specifies a URL to be opened in a new webview |
| close | ad calls to downgrade the state ad container |
| useCustomClose | deprecated in MRAID 3.0. This method call will be ignored by MRAID 3.0 compliant hosts |
| unload | ad calls to dismiss or remove the webview because it cannot load or render the assets. Host can either remove the webview, replace with another document or refresh with a new ad call |
| expand | ad requests ad container expansion |
| isViewable (deprecated) | ad queries the host about the on-screen status of the ad container |
| playVideo | ad requests video play in native player |
| resize | ad requests ad container size change to accommodate new ad size |
| storePicture | ad requests prompt to user about storing a picture on the device |
| createCalendarEvent | ad request prompt to the user for adding an event to the device calendar |
| VPAID Methods | a collection of methods used to manage a VPAID video ad in the context of MRAID<br>• initVpaid<br>• vpaidObject.subscribe<br>• vpaidObject.startAd<br>• vpaidObject.unsubscribe<br>• vpaidObject.getAdDuration<br>• vpaidObject.getAdRemainingTime |
| **Properties** | |
| supports | ad requests details on features the host supports |
| getPlacementType | ad requests confirmation of either an inline or interstitial placement |

| | |
|---|---|
| getOrientationProperties | ad requests details on landscape or portrait orientation |
| setOrientationProperties | ad specifies preferences for allowing or locking orientation, if supported, for ad display |
| getCurrentAppOrientation | ad requests current orientation of the app |
| getCurrentPosition | ad requests current coordinates of the ad container |
| getDefaultPosition | ad requests default coordinates of the ad container |
| getState | ad requests current state of the ad container: loading, default, expanded, resized, hidden |
| getExpandProperties | ad requests current expand properties |
| setExpandProperties | ad specifies new expand properties |
| getMaxSize | ad request max ad container dimensions available |
| getScreenSize | ad requests dimensions of device screen |
| getResizeProperties | ad requests current dimensions of the ad container in its resized state |
| setResizeProperties | ad specifies dimensions for resizing the ad container |
| getLocation | ad requests current coordinates of the device |
| **Events** | |
| error | host reports an error |
| ready | host reports that MRAID libraries are loaded |
| sizeChange | host reports that ad container dimensions have changed |
| stateChange | host reports that the state of the ad container has changed |
| exposureChange | host reports that the percentage of ad container exposure has changed |
| audioVolumeChange | host reports a change in volume |
| viewableChange (deprecated) | host reports a change in ad container viewability |

## 2.4 Offline Requests and Metrics

Rich Media Ads that 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 integrate with common APIs to facilitate storing and forwarding ad metrics such as impressions, views, and other ad activity; however, details for how these metrics are measured or how they are reported is out of the scope for MRAID.

## 2.5 DAA Ad Marker Implementation

The Digital Advertising Alliance (DAA) establishes principles for user privacy in digital advertising. To align with these principles as they apply to the mobile environment, the DAA has developed implementation guidelines that involve the placement of a marker, which offers users information about the ad and the option to opt out.

The DAA Ad Marker Implementation Guidelines for Mobile was released April 2014. Enforcement of privacy principles in the mobile environment began on September 1, 2015.

While no updates have been added to MRAID 3.0, MRAID has always had the ability to support the display of an icon overlay and open a window that can provide users with more information about the ad. Ad developers and mobile vendors should review Interest-Based Advertising (IBA) Principles for mobile ads as part of the ad development and delivery strategy.

# 3 Initialization and Set-up

MRAID enables complex ad interactions in a native app environment. This interaction requires that the host initiate a container where the ad executes and displays. The following sections describe the initiation process.

## 3.1 Initialization Overview

MRAID governs interactions between the ad and app using an MRAID implementation that initiates a container for ad display. Ad designers must include a script request for `mraid.js`, but the host actually supplies the JavaScript libraries using a webview.

The webview must ensure that the appropriate JavaScript libraries are made available to the ad as soon as possible after the `mraid.js` reference is made. The webview confirms that the libraries are ready by sending 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. Host establishes webview for ad execution.
2. Host makes the MRAID_ENV object available before the ad is loaded so that the ad can identify MRAID compliance.
3. Ad identifies MRAID compliance by invoking the MRAID script tag before initial ad load is complete. See section 3.1.1 for details.
4. (Optional) Host detects the MRAID script call.

5. Host provides MRAID JavaScript bridge for ads.
6. Host provides limited MRAID object with MRAID state ='loading' and the ability to query state.
7. When ad uses createElement for mraid.js, ad must wait for mraid.js to finish loading before accessing the mraid object. Ad is required to ensure `mraid.js` is available. Ad must use the ready event to determine full `mraid.js` availability.
8. Ad checks for `mraid.getState()=='loading'`, if true, then ad listens for `ready` event using `mraid.addEventListener('ready')`
9. Host loads MRAID library into the webview
    a. Changes MRAID state to "default" and send `stateChange` event.
    b. Fires the MRAID `ready` event.
10. Event listener for the ad captures the ready event and can access MRAID features as needed.

### 3.1.1 Checking that MRAID Ad is Loaded

The ad is considered loaded when the document inside the webview containing the ad is parsed and all of its sub-resources (including ad assets) have finished loading. At that time, `document.readyState` is set to `complete` and the `load` event is emitted on the `window` object.

This check may not be required for all types of ads e.g. small size banner ads may not require this. But this check must be performed for all large sized ads e.g. interstitials or ads that require heavy assets to be loaded before rendering.

The host can detect when the webview and all of its components including all ad assets has completely loaded by checking for the following:

- On Android, use the `onPageFinished()` handler
- On iOS* inspect the document within the webview by polling for the `document.readyState` until it is set to `complete` and the `load` event is emitted on the `window` object.

In iOS the host can also use `webViewDidFinishLoad()`, but this event can sometimes be triggered too soon. The `document.readyState` must also be checked to verify ad load on iOS.

When there is no ad loaded for any reason, it is required that the ad be able to communicate to the SDK that there is no ad present. This must be done using the `mraid.unload()` method

### 3.1.2 Declaring MRAID Environment Details

In previous versions of MRAID, guidance was given for the ad to identify as an MRAID ad (using `mraid.js`) as soon as possible, but the ad had no access to

information about the container into which it would load. In cases where the ad vendor offers both an MRAID version and a non-MRAID version, the ad server needs to gather details about the environment so that it doesn't waste valuable resources trying to call `mraid.js` for an environment that can't support it.

An MRAID 3.0 ad container provides an MRAID_ENV object that enables the ad to verify whether the container is MRAID-compliant along with other information about the environment, such as MRAID version, SDK version, and other key details the ad needs to operate more efficiently. The MRAID_ENV object is available even when the ad does not request access to MRAID by loading the mraid.js script.

The MRAID_ENV object declares specific information about the MRAID environment and the SDK container and makes that information available to the creative immediately upon load. The ad can use these details to deliver a better user experience and improve analytics.

The following script is an example of what the MRAID_ENV might look like:

```
<script>
window.MRAID_ENV = {
 version: '3.0',
 sdk: 'SDK Name',
 sdkVersion: '1.0.0',
 appId: 'com.iab.myapp',
 ifa: '01234567-89ab-cdef-0123-456789abcdef',
 limitAdTracking: true,
 coppa: false
</script>
```

**MRAID_ENV Attributes:**
The following attributes are used in the MRAID_ENV object that the host provides to the ad when requested.

| Attribute | Description |
|---|---|
| **version*** | (string) The version of the MRAID spec implemented by this SDK. It must equal the same value returned by the getVersion method of mraid interface. |
| **sdk*** | (string) The name of the SDK running this webview. |
| **sdkVersion*** | (string) The SDK version string. String may be left empty if no version is available. |
| **appId** | (string) The package name or application ID of the app running this ad. Usually referred to as the bundle id |
| **ifa** | (string) The user identifier for advertising purposes. For iOS, this must be the Identifier for Advertising (IDFA). For Android, this must be the Google Advertising ID (AID). |

| | |
|---|---|
| **limitAdTracking** | (Boolean, required if IFA is set) `true` if limit ad tracking is enabled, `False` otherwise |
| **coppa** | (Boolean, required for ads intended for children) `true` for child-directed, `false` otherwise |

*required

For ease of implementation, unused optional fields may be omitted from the object entirely, or provided with default values (empty string for string types, 0 for number types).

The optional fields must be set to default values or unset, and *must only be provided given explicit opt-in from the app publisher*. This opt-in may be global for the app, or on an impression by impression basis (for example, if a publisher supports both direct sold and anonymous inventory).

In the case of a 2-part ad, the host only makes the MRAID_ENV object available to the initial ad. The second part of the 2-part ad must use details the ad received in the first part if the second part needs those details.

### 3.1.3 Identification

To make use of MRAID, the ad must request the `mraid.js` script while the HTML ad is being loaded as described in section 3.1.1.

Requesting `mraid.js` can be done in one of three ways:

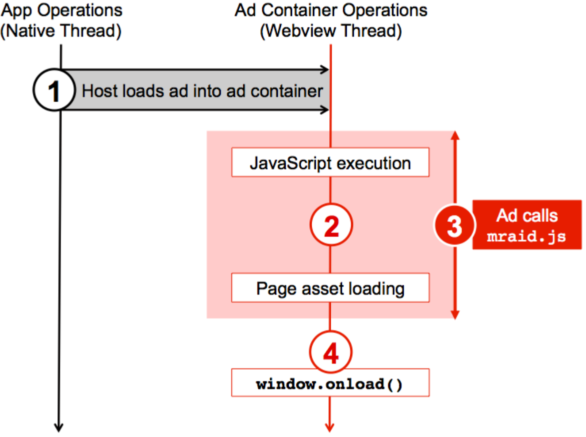
- Use the tag `<script src="mraid.js"></script>` in the HTML of the creative.
- Use `document.write()` of the script tag from JavaScript code placed in the creative.
- Use DOM element insertion of the script tag from JavaScript code placed in the creative.

In the above cases, the script tag can be injected dynamically using JavaScript code. The script tag must be injected before ad load is complete. See section 3.1.1 regarding ad load details.

MRAID sample ads demonstrate where the script tag should be positioned. See www.iab.com/mraid for resources on MRAID implementation.

Ad designers must avoid using the string `mraid.js` for any purpose other than to identify the use of MRAID. Using `mraid.js` more than once may lead to multiple injections of MRAID libraries, contributing to file size and slowing down ad performance.

The following diagram illustrates the interval for when `mraid.js` may be requested:




1. The host app loads the ad into an ad container.
2. Within the ad container, the host executes in-page and synchronously loaded JavaScript along with any other page assets.
3. The ad must call `mraid.js` before the ad container is completely loaded.
4. The ad should listen to `window.load()event` to confirm container load is complete

Non-MRAID ads should be able to operate in an MRAID webview with or without calling `mraid.js`. One reason a simple ad might use MRAID is to provide consistent behavior for hyperlink interaction. When MRAID is used, all hyperlinks must use `mraid.open()`.

### 3.1.4 Implementation of MRAID Events

In MRAID, user interaction can trigger a series of events to happen. To ease the burden of ad creation, MRAID 3.0 mandates that these chained events are fired after container manipulation as a result of either user interaction or creative action.

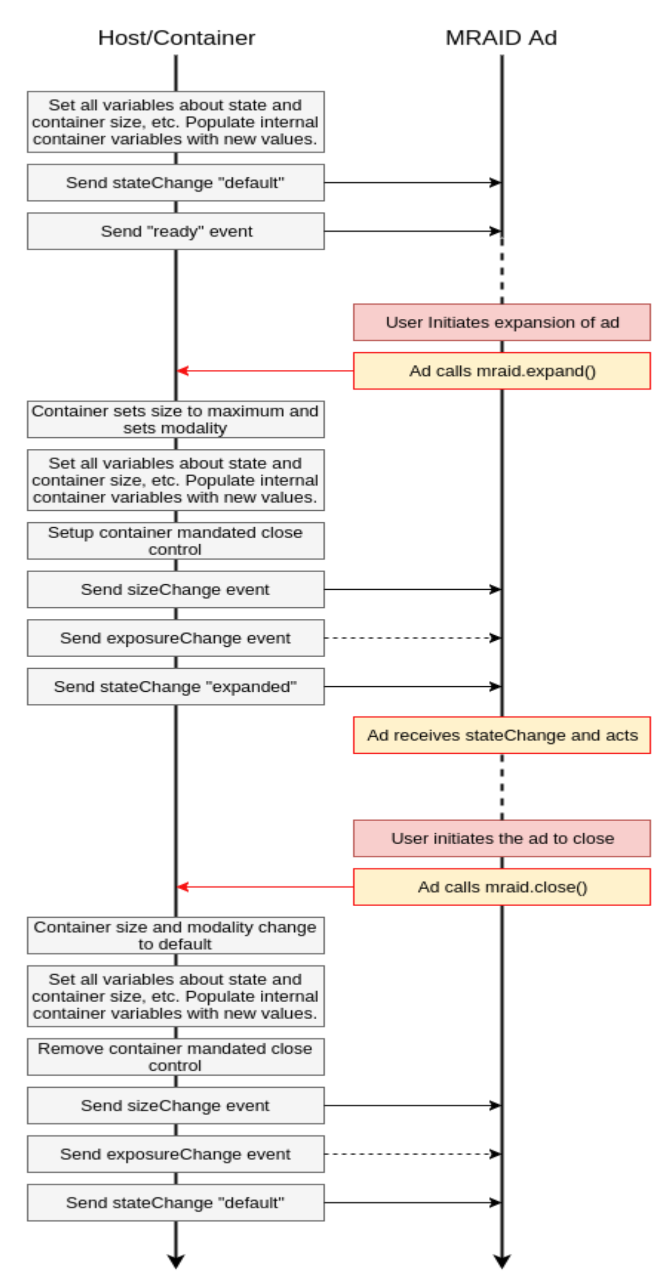
In general, the container must perform all needed container size changes or value changes before sending any event. This allows an ad to only need to watch a single event instead of watching multiple events.

For example in the case of expandable ads, an ad would require `stateChange`, `exposureChange`, and a `sizeChange` events to perform the expanded creative execution. In previous versions of MRAID, an ad was forced to not only check that the state has changed to "expanded" but was also required to check that the container size actually changed with the `sizeChange` event.

In MRAID 3.0, the container must perform all the needed changes and then fire chained events. In the expandable example, the events `stateChange`,

`exposureChange` and `sizeChange` will all fire. This allows the ad to only need to watch the event it expects and needs i.e. `stateChange`.
Expandable and interstitial examples are explained in more detail in this section.

**Expandable Ad**
In a typical use case for MRAID, the ad starts as a banner and expands to a full screen experience upon user initiation.



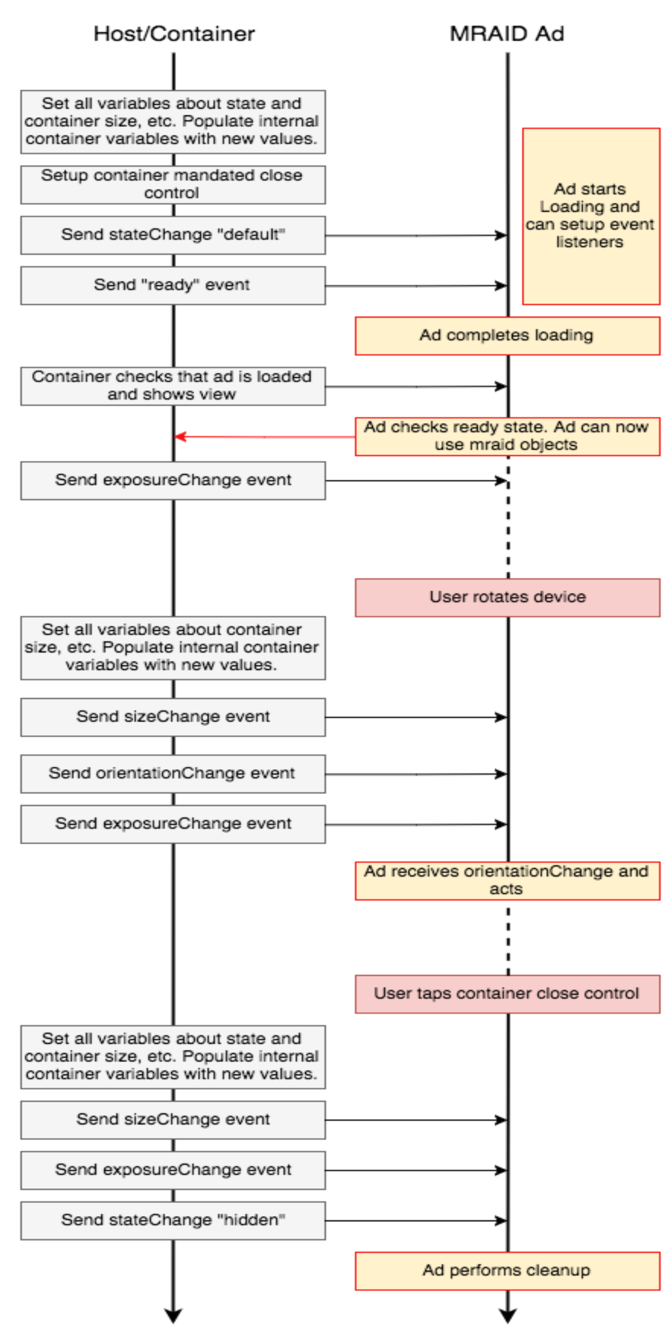
When the user taps the ad, the ad uses the MRAID javascript layer to request an expansion by calling `mraid.expand()`. MRAID calls to request ad expansion.

The container notifies the app that the ad is expanding so that it can stop anything that prevents user interaction. The container then resizes, and the webview reserves a space at the top right corner for the close event region and adds a close button.

When the user taps the close button or if the ad calls `mraid.close()`, the ad resizes to its original size and the container notifies the app that it may resume normal functionality.

The `exposureChange` event normally fires, but if the ad never registered a listener, the SDK may opt to not fire the event to save on computational resources.

**Interstitial Ad**
Interstitial ads operate a lot like expandable ads except that instead of resizing after the close function is tapped, an interstitial ad state is changed to 'hidden.' Since the ad is not called upon again, any registered event listeners are unregistered.




© 2016 IAB Technology Laboratory

### 3.1.5 Loading and Showing Interstitial Ads

Sometimes an interstitial ad is shown to the user before all the assets are fully loaded. The result is a partially loaded or blank ad. In such instances, the poor user experience may cause the user to dismiss the ad before it renders. To prevent display of partially loaded ads, the host must wait for the ad to load completely before display is initiated.

The ad must load in a webview that is off screen or hidden until all assets are loaded. See section 3.1.1 for identifying when the ad is loaded.

The host must load and display the interstitial ad in the same webview. The host must not load the ad into a temporary webview and then load it again inside a new on-screen webview to show it. Doing so will cause the ad to be reloaded, which may result in over-counting of ad requests and pixel trackers, and could result in additional per-ad serving costs for dynamic ads.

### 3.1.6 Using iframes

Because of the potential use of multiple platforms used to serve an ad, sometimes an MRAID ad is a container that serves another MRAID ad inside of an iframe. The result is a series of nested iframes that may be generated before the ad is delivered.

When an MRAID ad is contained within a nested iframe, accessing the native MRAID implementation is technically difficult or impossible. MRAID cannot directly support an ad within nested iframes. In cases where an ad wishes for nested iframes to access MRAID functionality, the outermost frame of the ad will need to provide its own mechanism for accessing MRAID through nested iframes.

A solution for working in nested iframes is outside the scope of the MRAID specification.

### 3.1.7 Viewport and Default Container Set-Up

An MRAID ad needs to know the default settings of the container set up for ad execution and be able to override default settings to suit proper ad execution. To find out about the container's default settings, the ad can query the container the same way it would query a webpage. Default settings for MRAID include: width and height of the container, scale, and whether the user can scale the container.

While MRAID does not establish any new parameters or controls over the container, the ad must check and adjust the parameters for initial display as needed.

### 3.1.8 Standard Image for Initial Display

The ad designer is responsible for providing a standard image to use for initial display. Access to this initial asset can be provided using an `<img>` tag, which is displayed while other assets are loaded in the background. Any interactive elements of the ad can replace the simple image once all additional assets are ready.

### 3.1.9 Event Handling

Event handling is key to MRAID functionality. Communicating between the web layer and native layer is asynchronous by nature. Through event handling, the ad is able to listen for particular events and respond to those events as needed. MRAID advocates broadcast-style events to support the broadest range of features and flexibility with the greatest consistency.

# 4 Features and Operation

MRAID enables rich media interactions for ads in mobile, allowing features such as expansion, resizing, and rich metric tracking. In order for smooth operation for all that MRAID has to offer, developers must understand the order of operation for each interaction and expected responses in both the ad and the app.

Details on these features and their operation are described in this section. Initialization and set-up are described in section 3. Specific methods, properties, and events are described in detail in sections 5-7. Details on working with device features at the native level, such as creating a calendar event and storing a picture are described in section 8. Integration with IAB VPAID for operation and tracking in the native video player is covered in section 9.

## 4.1 Viewability

When an ad is displayed in a web browser, JavaScript code in the ad inspects properties of the DOM hierarchy and gathers geometric measurements to determine the viewability of the ad. However, that technique does not work for ads displayed in a native app container. The JavaScript code cannot determine the measurements relative to the native UI, which is required when calculating whether the webview is visible to the user.

A more complete report of viewability involves monitoring shifts in exposure along with looking at user interaction. Instead of tracking viewability of the active webview, MRAID 3.0 introduces the `exposureChange` events. See section 7.5 for details on the use of this event.

The update to viewability related changes in MRAID 3.0 are guided by the following principles:

- **Performance:** minimize the impact of measurement on the end-user experience, including animation smoothness and battery life.
- **Minimal impact:** make only additions required to provide better insight into container viewability and nothing more.
- **Future-proof:** avoid specifying explicit thresholds in order to support upcoming measurement standards and provide maximum creative flexibility.
- **Backwards compatible:** containers that are currently MRAID compliant must remain compliant, i.e. the meaning of existing APIs should not be redefined.

Viewability is a measurement that determines whether the ad is in a position where the user has the opportunity to see the ad on their device.

In 2016, the Media Rating Council (MRC) released guidelines for measuring mobile ad viewability. These guidelines should be used to develop strategies for tracking viewable ad impressions in mobile web and mobile in-app experiences. Guidelines for using the MRAID 3.0 event for `exposureChange` are aligned with MRC guidelines for mobile ad measurement.

The `isViewable()` method and `viewableChange` event are deprecated in MRAID 3.0. These features are retained in this version for backward compatibility.

### 4.1.1 Polling Rates and Event Thresholds

Consistent with the update principle to maintain performance, event processing for the `exposureChange` event in MRAID 3.0 must not interfere with smooth UI animations or reduce battery life.

The MRAID 3.0 implementation on the host platform must coalesce updates to avoid excessive messaging to JavaScript. The combined updates must report the `exposureChange` event no later than 200ms after a change in the exposed area of the ad. Where polling is used, the view hierarchy must be viewed frequently enough to catch changes in exposure. Polling must occur at least 5 times per second (every 200 ms).

### Sample Usage

The following process represents a compliant execution of the `exposureChange` event:

1. The ad container registers a listener for the `exposureChange` event.

   For example:

   ```
   mraid.addEventListener('exposureChange',
   handleExposureChange);
   ```

2. The `exposureChange` event calls the ad's event handler and compares the container rectangle with the size of the ad creative.

   For example,

   ```
   function handleExposureChange(exposedPercentage,
   visibleRectangle, occlusionRectangles
   ) {
         if (exposedPercentage >= 50.0) {
         // log visible time ...
         }
   }
   ```

3. When the ad has been sampled long enough with visible area to meet the viewability threshold, the ad must be considered viewable. If no further measurement is needed, remove the `exposureChange` listener.

   For example:

   ```
   mraid.removeEventListener('exposureChange',
   handleExposureChange);
   ```

### 4.1.2 Implementation Considerations

Users and application developers want the best performance for their applications. Ad measurement should not degrade the operation of an app. MRAID implementations of viewability must take the following into consideration:

**Avoid time-consuming JavaScript work**
Applications with webviews that display ads run on two processor threads: one thread for the native UI and a second thread for the JavaScript in the webview. Avoid time-consuming JavaScript work while the native thread is waiting.

For example, the iOS WKWebview uses the Objective-C method:

`evaluateJavaScript:completionHandler`

This method does not have the side effect of blocking UI thread. This is recommended to be used instead of webview.

**Use low-latency APIs**
Native platforms include APIs that provide low-latency notification of UI changes that affect viewability. Implementations of MRAID's new `exposureChange` event must use such notifications, particularly when low polling rates or back-off are in effect. An example of such an event is use of the `ViewTreeObserver` event on Android.

**Prevent blocking operations**
Do not perform blocking operations on the OS application lifecycle callbacks.

On iOS, these include the following `UIApplicationDelegate` methods:

- `applicationWillResignActive`
- `applicationDidEnterBackground`
- `applicationWillEnterForeground`
- `applicationDidBecomeActive`

The corresponding `NSNotificationCenter` events also add to blocking operations.

On Android, blocking operations include the following Activity methods:

- `onPause`
- `onStop`
- `onStart`
- `onResume`

The OS will terminate an application that does not return in a few seconds from these methods being called.

**Enqueue events without blocking native operations**
Sometimes the JavaScript of an ad can be buggy or even malicious. When the code takes too long to execute, subsequent events can be delayed or blocked. The MRAID implementation must avoid blocking the native thread when enqueueing events. Whether in the native thread or in the webview, enqueued events must only be passed when previous MRAID events listeners have returned.

## 4.2 Ad Controls for Display

Besides 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.

Sections 4.2.1 to 4.2.4 explain the controls available to the ad for advanced ad display, such with expanding ads and interstitials.

### 4.2.1 Ad States and How They're Changed

Each webview used to execute the ad has a state that is one of the following:

- **loading:** the webview 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 using the `resize()` method
- **hidden:** the state of an interstitial ad when closed. If supported, a banner ad may also be hidden when closed

The webview state is changed when calling: `expand()`, `resize()`,`close()`or `unload()`. The effect of calling these methods is outlined in the following table.

| Initial state | expand() | resize() | close() | unload() |
|---|---|---|---|---|
| **loading** | no change | no change | no change | Not Applicable* |
| **default (banner)** | state change to “expanded” | state change to “resized” | state change to “hidden” (if supported) | Not Applicable |
| **default (interstitial)** | no change | no change | state change to "hidden" | Not Applicable |
| **expanded** | no change (state remains “expanded”) | triggers an error; state remains “expanded” | state change to “default” | Not Applicable |

| | | | | |
|---|---|---|---|---|
| **resized** | state change to “expanded” | state remains as "resized" but with updated value | state change to “default” | Not Applicable |
| **hidden** | no change | no change | no change | Not Applicable |

***Not Applicable since `unload()` method is used when the ad does not want to be shown to the user. In this case the host can either dismiss or remove the webview, replace it with another document or refresh it with another ad.**

Ads with a two-part expansion and ads that are interstitials flow slightly different state change paths:

**Two-Part Expandable Ads**

Two-part expandable ads use two different webviews where ad components may expand independently of each other. Since MRAID ads carry one state at a time, two-part expandable ads carry the state of expanded for as long as the expanded view is onscreen.

The new, expanded webview starts in the “loading” state until MRAID is available. When the “ready” event is fired, the state of the ad transitions to “expanded.” The banner, the first part of the two-part ad also changes its state, from “default” to “expanded.”

**Interstitial Ads**

For an interstitial ad, the webview goes from “loading” to “default,” and when the interstitial is closed, the state changes to “hidden.”

The `getState()` method reports the current state that was last sent using the `stateChange` event. These features are further described in sections 6.7 and 7.4, respectively.

### 4.2.2 Checking Position and Size of the Screen and Ad

MRAID 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.

See the following sections for details:

- 6.5 getCurrentPosition
- 6.6 getDefaultPosition
- 6.10 getScreenSize
- 6.9 getMaxSize
- 7.3 sizeChange

### 4.2.3 Changing the Size of an Ad

MRAID v2 includes three distinct ways for an ad to change its size: `open()`, `expand()`, or `resize()`.

The simplest method for size change is to use the `open()` method. Intended for opening hyperlinks, this method can also open a new webview to display a new component of the ad in a different size. This format for an ad is called a two-part expandable ad.

The `expand()` method is used to expand ads in a fairly simple, straightforward way that covers the content of the application.

The `resize()` method is used for ads that grow or shrink in more subtle ways that take place within a dialogue of app operation. This method offers designers more freedom and control; however, additional methods and listeners are required for both the ad and the app or webview to react appropriately in different placements.

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

Although these methods are related, they promote an approach of progressive complexity. Distinguishing between `open(), expand() and resize()` helps ad designers choose the best method for their needs.

All these methods must be user initiated based on IAB New Ad Portfolio definition. User initiation is defined as discrete user action e.g. click or tap.

Ad must ensure that these methods are called only when initiated by the user. Ad must not initiate these requests without user action.

**open()**

- Lowest common denominator
- Used for advertiser landing pages or microsites
- Opens a new url in the device's default 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 creative
- MRAID-enforced tap-to-close area in fixed (top right) location
- Relative alignment for creative
- Check screen size (getScreenSize) before expanding

**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 creative 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
- Check max size allowed (`getMaxSize`) before resizing
- Host must not refuse `resize()` request from the ad

The following table summarizes key 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 creative** | n/a | Y | N |
| **One-part creative** | n/a | Y | Y |
| **Aligned to screen** | n/a | Y | N |
| **Background provided for small creative** | 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 |

When the expanded creative is smaller than full screen, calling expand() causes the webview to blank out, cover, or otherwise obscure the underlying app making

it very clear that the expanded ad is modal in nature. Modal, partial-screen expansions are not allowed in MRAID, but partial-screen, non-modal expansions can be created using the `resize()` method.

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

# 5 MRAID Methods

MRAID methods initiate a function, usually having to do with ad operation that requires some adjustment of the ad container or transfer of information. For example, calling `expand()` prompts the SDK to stop app operation and expand the ad container. Some methods involve checking or updating properties (section 6) and using an event (section 7) to report or confirm certain actions.

## 5.1 getVersion()

The ad calls the `getVersion()` method to query the host about which MRAID version the host supports. The host returns a version number string ("3.0" for MRAID 3.0). The version number indicates the version of MRAID that the host supports (1.0, 2.0, or 3.0, etc.), NOT the version of the vendor's SDK.

| Syntax | `getVersion()` |
|---|---|
| Parameters | None |
| Return Values | A String that indicates the MRAID version with which the SDK is compliant (not the version of the SDK). For example, if version 5.2 of the SDK is compliant with MRAID 3.0, then `getVersion()` returns "3.0." |
| Related Event | None |

Note: the MRAID and SDK versions are offered in the MRAID_ENV object discussed in section 3.1.1.

## 5.2 addEventListener()

The ad calls `addEventListener()` to register a specific listener for a specific event. The ad may register more than one listener, each to support listening for separate event. The host dispatches an event to all registered listeners for each specific event that occurs. The ad may also register a single listener to multiple events instead of a listener for each event.

The events supported in MRAID 3.0 are:

- **ready:** report initialize complete
- **error:** report error has occurred
- **stateChange:** report state changes
- **exposureChange:** report change in exposure
- **viewableChange (deprecated):** report viewable changes
- **sizeChange:** report viewable changes

| Syntax | `addEventListener(event, listener)` |
|---|---|
| Parameters | event: a string for the name of the event to listen for<br>listener: function to execute |
| Return Values | None |
| Related Event | None |

## 5.3 removeEventListener()

When the ad no longer needs notification of a particular event, `removeEventListener()` is used to unregister to that event. To avoid errors, event listeners must always be removed when they are no longer needed. If no listener function is specified in the in the `listener` attribute for the call, then all functions listening to the event will be removed.

| Syntax | `removeEventListener(event, listener)` |
|---|---|
| Parameters | event: a string for the name of the event to remove<br>listener: function to be removed |
| Return Values | None |
| Related Event | None |

## 5.4 open()

The ad can call the `open()` method to prompt the host to open an external mobile website in a browser window that is the default browser on the user's device The purpose of this method is to handle clickthroughs in the ad. All MRAID ads must handle clickthroughs using the `open()` method.

| **General Implementation Note** | The `open()` method must only be used for external web pages that are not MRAID ads. The displayed page resulting from calling `open()` cannot load a second instance of the MRAID implementation, which means that the `close()` method is inoperable in the opened browser. The opened window can only be closed using whatever close control is implemented as part of the opened browser. |
|---|---|

The native browser controls – back, forward, refresh, close – are always present in the opened window. SDK providers consider the `open()` function as a reportable event.

| Syntax | `open(URL)` |
|---|---|
| Parameters | URL: a string for the URL of the webpage to be opened |
| Return Values | None |
| Related Event | 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 webview, or the content could open a separate browser window and load the indicated HTML link there.

The open method must always use the native device or OS behavior or user setting for opening a URL from an MRAID ad. This will ensure experience that the user expects and enable necessary device controls user needs to navigate to and away from the external links.

## 5.5 close()

The ad uses `close()` to downgrade the container state. The host responds by changing the state depending on the current state using the `stateChange` event described in section 7.4.

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. These ad states and the state that results from calling `close()` are described in section 4.2.1.

For ads in a default state, ad developers must avoid using `mraid.close()` . To inform the container that the ad needs to be dismissed, use `mraid.unload()`.

If an ad uses multiple `resize()` calls or a `resize()` followed by the `expand()` call, `close()` returns the container state to 'default.' Calling `close()` in these instances does NOT simply undo the most recently called `resize()` or `expand()`.

| Syntax | `close()` |
|---|---|
| Parameters | None |

| Return Values | None |
|---|---|
| Related Event | StateChange (section 7.4) |

**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 an area on the ad creative that users can tap to close the ad or collapse it back to its default state. The host provides the required close event region in a creative-specified location for all MRAID resizable ads.
- Close Indicator: The close indicator is the visual cue that identifies the close event region for the user. For resized ads, the ad supplies a close indicator graphic, while the host provides the close event region.

The host must always include a 50x50 density independent pixel close event region. Recommended position is the top right corner of the container provided for the ad. When a user taps this indicator, the ad returns to its default state. The recommended format is a "X" button for the close indicator.

`useCustomClose()` method is being deprecated and the ad must not provide its own custom close indicator. The close indicator is always provided by the host. This ensures consistent experience across apps for expanded or resized ads.

The host may also decide to follow the native app experience for closing or dismissing an ad. This may be done to preserve the native user experience of navigating content in a specific app – e.g. in some apps content navigation is done by using swipes across full screen content or by tapping on left or right of the ad. The same behavior may be used for ads that appear in between content for closing the ads. This does not apply to expandable ads or resize ads. They must have host provided close indicator.

For MRAID 2.0 or older version ads in MRAID 3.0 containers, `useCustomClose()` requests will be ignored by the host

A resized ad must position itself such that the entire close event region appears onscreen. If the host detects that resizing will result in the close event region being off screen, the host must return an error and ignore the resize call, leaving the ad in its current state. This requirement also means that a resized ad must be at least 50x50 density independent pixels, to ensure there is room on the resized creative for the close event region.

The close control is mandatory and provided by the host in all cases.

On Android OS, user action for ‘back’ button must be interpreted as close method by the ad.

## 5.6 unload()

The key purpose of this method is to allow the ad to communicate to the host that the ad must no longer be shown to the user. The ad can request the host to completely dismiss the ad by using the `unload()` method. The host responds by dismissing the ad and then either removing the webview or replacing it with another document or refreshing it with a new ad.

The ad may use this method anytime in its lifecycle when it determines it no longer wants to continue being shown to the user and it is not desirable to return to default state.

The ad should use this method when it encounters errors or runtime exception that will not allow it to render properly to the user. Some examples of situations when an ad may decide to use `unload()` method:

- During initialization, it can encounter communication error with the ad server or it is not able to load all required assets properly or it determines that it does not want to be shown in the environment it is being served
- During later stages in the lifecycle, it may encounter errors in loading a required asset or resource and determine that it does not want to continue being shown to the user
- A pre-loaded ad may determine it does not want to be shown to the user.

| Syntax | `unload()` |
|---|---|
| Parameters | None |
| Return Values | None |
| Related Event | StateChange (section 7.4) |

`unload()` provides a graceful exit mechanism for the ad and the host without causing errors on screen or blank screen being shown to the user.

## 5.7 useCustomClose() (deprecated)

This method is being deprecated in MRAID 3.0

## 5.8 expand()

The ad uses `expand()` to request support for an ad expansion experience, either by changing the width and height of the current webview (one-part creative) or by opening a new webview at the highest level in an expanded size (two-part creative). 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.

This option enables the ad designer to provide expandable ads as either a one-part ad (banner and panel as one creative) or a two part ad (banner and panel are separate HTML creative).

> **One-Part Creative (no URL specified)**
> When the ad calls `expand()` without specifying a URL, the host expands within the existing webview. This approach simplifies ad design and reporting because the original creative is not reloaded and no additional impressions are recorded.
>
> **Two-Part Creative (URL specified)**
> When the ad calls `expand()` with a URL specified, the host opens a new webview. The URL supplied must be a complete HTML page not a snippet or fragment. If the second-part ad creative requires the use of MRAID, the ad must request mraid.js for the new webview. Whether or not the expanded part of the ad requests mraid.js, the host must supply the close event region. If ad specifies a close indicator graphic in `expandProperties`, then the host will use the supplied graphic to indicate close. Otherwise, the host also provides the close indicator graphic.

While an ad is in an expanded state, the default position is obscured or inaccessible to the viewer, so the default position must take no action while the expanded state is available

**Host response**
The host responds to an `expand()` call by changing the state from 'default' to 'expanded' using the `stateChange` event described in section 7.4. The host ignores multiple calls to expand and retains the state status of 'expand.'

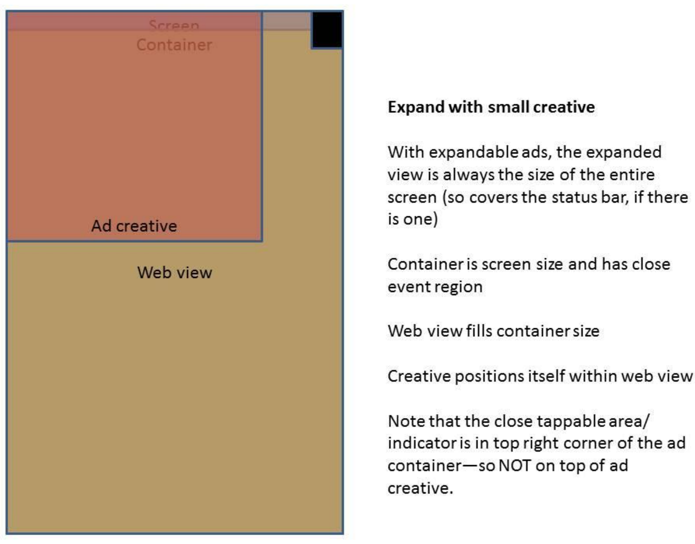
The host also expands the container to cover all available screen area even though the expanded creative may be smaller than the screen area. Any area of the ad container not filled with ad creative may be transparent or opaque. Expanded ads are always modal, and the container must prevent new ads from loading during the expanded state.

**Potential Issues**
Complications possible with an expanded ad may involve multiple window objects that prevent operation in the expanded state, or a timer that changes the content z-order (layer position). These complex app environments may interfere

with the user's ability to close the ad. The SDK developers must consider these complications and adjust accordingly for specific MRAID implementations.

**Placement of the Expanded Creative**
The ad designer decides where to place the expanded ad creative on screen, especially when the expanded view can be placed in multiple locations. For full-screen expands, all MRAID implementations offer full screen view for the ad and position the ad so it is fully visible.

When the expanded creative size is greater or smaller than the screen size of the device, the host adjusts the container to the maximum size the app and device allow. The host may not scale the ad to fit the screen. Instead, the ad may position the expanded creative as desired within the expanded container using CSS.




| Syntax | `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 |
| Related Event | StateChange |

## 5.9 isViewable() (deprecated)

The `isViewable()` method is deprecated in MRAID 3.0 and will be removed in future versions of MRAID. This feature remains in MRAID 3.0 for the purposes of backward compatibility.

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()` returns an answer based on the viewability of the expanded ad.

In any situation where an ad may be loaded off screen, the ad must 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."

| Syntax | `isViewable()` |
|---|---|
| Parameters | None |
| Return Values | true: container is on-screen and viewable by the user according to established values<br>false: container is off-screen and not viewable |
| Related Event | viewableChange |

## 5.10 playVideo()

The ad uses `playVideo()` when a video component of the creative needs to play in the device's native player. To play the video inline rather than in the native player, the ad designer must use HTML5 video tags instead.

| Syntax | `playVideo(URI)` |
|---|---|
| Parameters | URI: String, the URI of the video or video stream |
| Return Values | None |
| Related Event | None |

## 5.11 resize()

Calling `resize()` is a request for a container size change that accommodates the creative size change. Resize is used for a succession of changes or a size

change that is less than full screen size and that doesn't interfere with app operation.

Before calling `resize()`, the ad must specify the desired width and height of the resize using `setResizeProperties()`. Calling `resize()` before `setResizeProperties` will result in an error.

After calling resize, the host responds by adjusting the ad container to the ad's desired size.

Resize operates at a higher z-index than the app content so that it does not push or reposition app content. If an app wishes to support content-shifting ads, like "push-downs," the app must implement app repositioning features to support such functionality.

**Resulting Events**
If the action is executed successfully, the host sends two events: `stateChange` (section 7.4) and `sizeChange` (section 7.3). The `stateChange` is updated from 'default' to 'resized.' If the ad state was 'expanded' before calling `resize()`, the host sends the `error` event discussed in section 7.1. The `sizeChange` event is sent with the new width and height of the resized container.

| **General Implementation Note** | Use `expand()` instead of `resize()` for ad creative that expands to full-screen (or larger) size. Since a full-screen or larger expansion covers app content, the modal nature of `expand()` prompts stopping other app operations. Resize always results in a non-modal size change, and some portion of the app must always remain visible to the end user. |
| --- | --- |

| Syntax | `resize()` |
| --- | --- |
| Parameters | None |
| Return Values | None |
| Related Event | sizeChange<br>stateChange |

## 5.12 storePicture()

An ad that offers users the option to store an image, `storePicture()` can be used (if supported), to place an image in the device's photo album. The image may be part of the ad or retrieved from a server.

The ad can query the host for device support of `storePicture` using the `supports` method described in section 6.1. If `storePicture` is not supported, the ad must refrain from calling `storePicture()`.

If supported, the host responds by initiating a control to ask the user's permission to store the image. The control is a modal OS-level handler that offers a confirm/cancel option for the user. If the device lacks such a handler, the host reports a false value in the supports object for the `storePicture` feature.

The MRAID implementation must support adding a picture using an HTTP redirect (for tracking purposes); however, the implementation is not required to support meta-redirects.

If the attempt to add the picture fails for any reason or is cancelled by the user, the host sends an error.

| Syntax | `storePicture(URI)` |
|---|---|
| Parameters | String: the URI to the image or other media asset |
| Return Values | None |
| Related Event | None |

## 5.13 createCalendarEvent()

If supported, the ad can use `createCalendarEvent()` to open the device calendar UI for adding an event. The host responds by populating the create calendar event sheet on the native device. If no event sheet is found, the host reports a value of 'false' for the calendar feature in the supports object.

Any ad operation is suspended while the calendar UI is open. Calendar event data must be delivered in the form of a JavaScript object written to the W3Cs calendar specification. See the appendix for details on the W3Cs calendar specification.

If the attempt to create the calendar event fails or is cancelled by the user, the host sends an error.

| Syntax | `createCalendarEvent(parameters)` |
|---|---|
| Parameters | URI: String: the URI to the image or other media asset |
| Return Values | None |
| Related Event | None |

## 5.14 VPAID methods

An MRAID ad may include video designed to report VPAID events. The following MRAID methods are used to setup VPAID events in an MRAID context, as well as obtain minimal information about the video ad being shown. See section 9 for additional details on working with VPAID in an MRAID app implementation.

### 5.14.1 initVpaid()

Once the ad has checked for VPAID support in the `mraid.supports()` method, it needs to give the container a `vpaidObject` that the container uses to communicate events and information about the video playback. The VPAID object enables the host to subscribe to VPAID events, start the ad, and call a limited set of VPAID methods. After `initVpaid()` is called, in order for video playback to begin, the container must call `vpaidObject.startAd()`. The ad must not begin video playback until `videoObject.startAd()` is called.

| Syntax | `initVpaid(vpaidObject)` |
|---|---|
| Parameters | vpaidObject – a reference to the JavaScript VPAID object present in the ad |
| Return Values | None |
| Related Event | None |

### 5.14.2 vpaidObject.subscribe()

After the ad calls `initVpaid()`, the host can subscribe to select VPAID events using `vpaidObject.subscribe()`. For a list of supported and unsupported VPAID events, see section 9.1.3.

### 5.14.3 vpaidObject.startAd()

When the host has determined that the app is ready to play the video ad, it can call `vpaidObject.startAd()` to notify the MRAID ad that the VPAID creative may now play.

### 5.14.4 vpaidObject.unsubscribe()

When the host no longer needs to listen for an event to which it is subscribed, it can call `vpaidObject.unsubscribe()`, using the event name to specify which event it no longer needs.

### 5.14.5 vpaidObject.getAdDuration()

The host uses `vpaidObject.getAdDuration()` to query the ad for total ad duration of the video ad. The ad returns the number of seconds for total duration of the ad.

#### 5.14.6 vpaidObject.getAdRemainingTime()

The host uses `vpaidObject.getAdRemainingTime()` to query the ad for total play time remaining at the time of the query. The ad returns the number of seconds remaining for ad play at the time of request.

# 6 Properties

The methods covered in this section enable the ad to query the host about certain container properties. In some cases, the ad can set certain properties to instruct the host on how to update the container to accommodate ad operation. For example, the ad must first set resize properties before calling the resize method. The host then uses the set properties to adjust the container accordingly.

## 6.1 supports

The ad can use the `supports` feature to query the host about which device-native features the app can access. Awareness of supported native features helps the ad compensate in environments where certain features are not supported.

The ad can query the host for support of the following native features:

| Feature | 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 playback HTML5 video files using the <video> tag and honors the size (width and height) specified in the video tag. This does not require the video to be played in full screen. |
| **vpaid** | the device container supports VPAID handshake with ad to communicate VPAID events discussed in section 9 |
| **location** | the device supports access to GPS coordinates |

The MRAID implementation on the app must be able to deliver all of these functionalities on any device that is capable of them, except where app publishers have deactivated features that conflict with publisher policies.

| Syntax | `supports(feature)` |
|---|---|
| Parameters | feature: String, name of feature as listed in the above table |
| Return Values | true: the feature is supported and getter and events are available<br>false: the feature is not supported on this device |
| Related Event | None |

## 6.2 getPlacementType

The ad calls `getPlacementType()` to determine whether it's being loaded in an inline placement or an interstitial.

For efficiency, ad vendors sometimes flight a single creative in both banner (inline) and interstitial placements. These ads may be designed to behave differently depending on how it's placed.

In the case of a two-part expandable ad, the second part expansion creative must also query the host about placement type.

The host returns either 'inline' or 'interstitial' as defined in the table below.

| Syntax | `getPlacementType()` |
|---|---|
| Parameters | None |
| Return Values | **inline**: the default ad placement is inline with content in the display (banner)<br>**interstitial**: the ad placement is over laid on top of content |
| Related Event | None |

## 6.3 get/set orientationProperties

The ad uses the `orientationProperties` to query the host about the current orientation of the device and set `orientationProperties` to accommodate ad display for expansions and interstitials. A banner in its default state cannot use `orientationProperties`. Resizeable ads can use `orientationProperties`, but they won’t have any effect.

The following code snippet is an example of the `orientationProperties` object:

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

}
```

The two properties offered in the orientation object are as follows:

- **allowOrientationChange : boolean**
  Set to “true,” the container will permit device-based orientation changes; set to false, the container will ignore device-based orientation changes (e.g., the webview will not change even if the orientation of the device changes). Default is “true.” At any time, the ad may request a change of its orientation by setting the forceOrientation variable, regardless of how allowOrientationChange is set.

- **forceOrientation : string**
  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, when a user taps to expand an ad in landscape mode and the `forceOrientation` property is set to 'portrait,' then the ad will open in portrait orientation regardless of the orientation of the device or previous orientation of the ad. Default is “none.”

To enable finer control over ad behavior, the ad can change the setting of either property in the orientation object 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 must be allowed until the user is done. The host must be able to accept changes to expand properties throughout a user’s interaction with an expandable ad.

The ad must set both properties together to ensure proper control of the orientation of the ad. E.g. to force change to landscape orientation, set `allowOrientationChange` to ‘false and set f`orceOrientation` to ‘landscape’

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}
);
```

The `getOrientationProperties` method prompts the host to return the whole `orientationProperties` object.

| Syntax | `getOrientationProperties()` |
|---|---|
| Parameters | None |
| Return Values | JavaScript object: contains the orientation properties |
| Related Event | None |

The `setOrientationProperties` method sets properties in the `orientationProperties` object.

| Syntax | `setOrientationProperties(properties)` |
|---|---|
| Parameters | Properties: a JavaScript object that contains the values for `allowOrientationChange` and `forceOrientation`. |
| Return Values | None |
| Related Event | None |

## 6.4 getCurrentAppOrientation

The ad calls `getCurrentAppOrientation` to query the host about the current orientation of the app.

The host returns the current orientation of the app as either "portrait" or "landscape" and whether or not the given orientation is locked. If locked, then the `forceOrientation` option in `setOrientationProperties` is not supported.

| Syntax | `getCurrentAppOrientation()` |
|---|---|
| Parameters | None |
| Return Values | JavaScript object: {orientation, locked} where:<br>● **orientation**: value may be either "portrait" or "landscape"<br>● **locked**: Boolean value indicating whether orientation is locked in current position. If "true" then `forceOrientation` in `setOrientationProperties` is not supported. |
| Related Event | None |

© 2016 IAB Technology Laboratory

## 6.5 getCurrentPosition

The ad calls `getCurrentPosition` to query the host for the current position of the ad container.

The host returns the current position and size of the ad container, measured in density-independent pixels.

| Syntax | `getCurrentPosition()` |
| --- | --- |
| Parameters | None |
| Return Values | JavaScript object: {x, y, width, height} where:<br>● x=number of density-independent pixels offset from left edge of the rectangle defining `getMaxSize()`<br>● y=number of density-independent pixels offset from top of the rectangle defining `getMaxSize()`;<br>● width=current width of container in density-independent pixels<br>● height=current height of container in density-independent pixels) |
| Related Event | None |

## 6.6 getDefaultPosition

The ad calls `getDefaultPosition` query the host about the position and size of the default ad container, measured in density-independent pixels.

The host returns values for the default position and size of the ad container regardless of what state the calling view is in.

| Syntax | `getDefaultPosition()` |
| --- | --- |
| Parameters | None |
| Return Values | JavaScript object: {x, y, width, height} where:<br>● x=number of density-independent pixels offset from left edge of the rectangle defining `getMaxSize()`<br>● y=number of density-independent pixels offset from top of the rectangle defining `getMaxSize()`;<br>● width=current width of container in density-independent pixels<br>● height=current height of container in density-independent pixels) |

| Related Event | None |
|---|---|

## 6.7 getState

At any time, the ad may use `getState` to query the host about the state of the ad container and make requests accordingly.

The host returns the current state of the ad container using values that describe whether the ad container is in its default and fixed position, in an expanded or resized state, a larger position, or hidden.

| Syntax | `getState()` |
|---|---|
| Parameters | None |
| Return Values | String: "loading", "default", "expanded", "resized," or "hidden" |
| Related Event | stateChange |

## 6.8 get/set expandProperties

The ad uses `getExpandProperties` to query the host on the current expand settings, and `setExpandProperties` to set the width and height for an expansion along with optionally specifying the use of a custom close indicator.

Before the ad calls `expand()`, the ad needs to identify the desired width and height for the expansion using `setExpandProperties`. The ad may also specify a custom close indicator graphic instead of using the hosts default indicator. The host ignores any expand properties set after `expand()` is called.

The following is an example of an `expandProperties` object:

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

| Property | Description |
|---|---|
| **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** | This is deprecated. In MRAID 3.0 host will ignore this request<br><br>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 (section 5.6), and is provided as a convenience for creators of expandable ads. |
| **isModal** | boolean – true, the container is modal for the expanded ad and other operations must be halted during expansion; false, the container is not modal for the expanded ad and other operations may continue during the expansion. This property is read-only for the ad and cannot be set. |

When the ad calls `getExpandProperties`, the host returns the whole `expandProperties` object.

| | |
|---|---|
| Syntax | `getExpandProperties()` |
| Parameters | None |
| Return Values | JavaScript object: contains the expand properties for the ad expansion described in the table above. |
| Related Event | None |

The ad uses `setExpandProperties` to set properties of the `expandProperties` object prior to initiating an ad expansion using `expand()`.

| | |
|---|---|
| Syntax | `setExpandProperties(properties)` |
| Parameters | JavaScript object: {width, height, useCustomClose} contains the width and height of the expanded ad. |
| Return Values | None |
| Related Event | None |

## 6.9 getMaxSize

The ad calls `getMaxSize` to query the host about the maximum size (in density-independent pixels) to which the ad may resize.

If the app runs full-screen on the device (covering the status bar), the host returns the full-screen dimensions. If the app runs at less than full screen on the device, usually to leave room for a status bar or other elements outside the app, then the host returns the size for the view that contains the app.

| Syntax | `getMaxSize()` |
|---|---|
| Parameters | None |
| Return Values | JavaScript object: {width, height} contains the maximum width and height of the webview (webview can resize to no larger than given width and height). |
| Related Event | None |

## 6.10 getScreenSize

The ad calls `getScreenSize` to query the host about the dimensions of the device screen size, especially before expanding an ad (for resizing use `getMaxSize`). The host returns current device screen width and height, in density-independent pixels.

This size changes when the device orientation changes. For example, a screen that is 640x960 in portrait mode changes to 960x640 when reoriented to landscape mode.

The entire screen size is returned, including any area reserved for status or system bars and other functional space that the app cannot override. To find out how much usable size is available on the app for ad display, the ad must call `getMaxSize()` described in section 6.9.

| Syntax | `getScreenSize()` |
|---|---|
| Parameters | None |
| Return Values | JavaScript object: {width, height} contains the width and height of the device screen, depending on its orientation |
| Related Event | None |

## 6.11 get/set resizeProperties

Before the ad executes a resize, it can call `getResizeProperties` to query the host about current settings for resizing the ad container. Then the ad can call `setResizeProperties` to update the `resizeProperties` object. The host uses the properties set in this object to adjust the ad container when the ad calls `resize()`.

The following is an example of the `resizeProperties` object:

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

}
```

Properties available in the `resizeProperties` object are as follows:

| Property | Description |
|---|---|
| **width*** | integer: width, in density independent pixels, to which the ad container must be resized |
| **height*** | integer: height, in density independent pixels, to which the ad container must be resized |
| **offsetX*** | integer: the horizontal delta from the current upper-left position to the desired resize upper-left position of the ad container. Positive integers move right; negative integers move left. |
| **offsetY*** | integer: the vertical delta from the current upper-left position to the desired resize upper-left position of the ad container. Positive integers move down; negative integers move up. |
| **customClosePosition** | this is deprecated in MRAID 3.0. The host will always add close indicator in top right corner. |
| **allowOffscreen** | boolean: indicates whether the resized ad container must be allowed to be drawn partially or fully offscreen.<br>● **true:** offscreen positioning is allowed; host must refrain from repositioning ad container despite resulting in offscreen placement.<br>● **false:** offscreen positioning must be avoid and the host must attempt to reposition the ad container |

| Property | Description |
| --- | --- |
| | so that it displays within the area specified in the `maxSize` property. |

*required; if no value is provided using `setResizeProperties` before the ad calls `resize()`, then the host retains the original settings and sends an error.

**When Resize Results in Repositioning Offscreen**

When the `allowOffscreen` property is set to 'false,' the host attempts to reposition the resized creative with in the dimensions define in the `maxSize` object. For example, if ad is positioned the top of the screen, and ad wants to resize upwards by 50 pixels, the host moves the ad 50 pixels down before executing the resize.

When `allowOffscreen` is set to 'true' in the same operation, the resized portion of the ad extends out of view 50 pixels beyond the top of the screen.

The `allowOffscreen` property cannot solve all positioning issues. For example, if an ad successfully resizes in landscape orientation followed by an orientation change that results in a larger ad, a 'false' `allowOffscreen` is ineffective.

Resize ads must be tested for quality before serving to the app for display. If the ad provides resize values that the host cannot successfully execute, the host sends an error.

**Checking and Setting Resize Properties**

The ad uses `getResizeProperties` to query the host for current resize properties. The host returns the `resizeProperties` object.

| Syntax | `getResizeProperties()` |
| --- | --- |
| Parameters | None |
| Return Values | JavaScript object: {width, height} contains the resize properties |
| Related Event | None |

The ad uses `setResizeProperties` to change the values in the `resizeProperties` object. The host replaces the values in the `resizeProperties` object with the values the ad provides.

| Syntax | `setResizeProperties(properties)` |
| --- | --- |
| Parameters | JavaScript 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 off screen. |
| Return Values | None |

| Related Event | None |
|---|---|

## 6.12 getLocation

Knowing the location of the device when an ad is active can enhance the creative possibilities for an ad, but not all apps support access to location details. The `supports` feature in MRAID 3.0 (see section 6.1) identifies whether "location" is supported.

If supported, location values are provided using the location object with the following properties:

| Property | Description |
|---|---|
| **lat** | the latitude coordinate for the device |
| **lon** | the longitudinal coordinate of the device |
| **type** | source of location data; recommended when passing lat/lon<br>1: GPS/Location service<br>2: IP Address<br>3: User Provided (e.g. registration data) |
| **accuracy** | estimated location accuracy in meters; recommended when lat/lon are specified and derived from a device’s location services (i.e., type = 1). Note that this is the accuracy as reported from the device. Consult OS specific documentation (e.g., Android, iOS) for exact interpretation |
| **lastfix** | number of seconds since this geolocation fix was established. Note that devices may cache location data across multiple fetches. Ideally, this value must be from the time the actual fix was taken |
| **ipservice** | service or provider used to determine geolocation from IP address if applicable (i.e., type = 2). |

If supported, the ad uses `getLocation` to query the host about the location of the device. The host returns the location object with details about device location data and where the data came from.

| Syntax | `getLocation()` |
|---|---|
| Parameters | None |
| Return Values | JavaScript object: the location object described in the table above. |
| Related Event | None |

HTML5 API for location must not be used to replace the above.

When lat and lon properties are not available or not allowed as per user permission, then error message of “-1” must be communicated for `getLocation()` method.

# 7 Events

Events are used to report that an action has occurred or to confirm a change in the state of the webview. The host sends events to communicate activity to the ad. The ad must create listeners for specific events to be informed on when they occur.

## 7.1 Error

When the host cannot execute a function that the ad calls, the host sends an error that describes the action attempted when the error occurred. The ad must register a listener in order to receive error events. Any number of listeners can monitor for errors of different types so that the ad can respond as needed.

An `error` event offers two parameters: one for an error message and one for the action being attempted at the time the error occurred. Both parameters are optional and the message option is typically used for debugging pre-flight creative.

Any of the following MRAID actions could potentially result in error:

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

While the ad may register for errors that result from any of the above actions, the three most common errors result from using `resize()`, `storePhoto()`, and `createCalendarEvent()`. The ad must register listeners for these errors and be prepared to react accordingly.

Since the host can handle errors either synchronously (in real-time) or asynchronously (time-shifted), the ad developer must consider how error details must be handled when received.

| Syntax | `"error" ➐ function(message, action)` |
|---|---|
| Parameters | message: String, description of the error that occurred<br>action: String, name of MRAID action attempted when the error occurred |
| Triggered by | anything that goes wrong |

## 7.2 ready

Before the ad is loaded, the host must make the MRAID library available to the ad so that the ad can make calls and register for listeners on host-sent events. At a minimum, the ad container must support `getState()`,and `addEventListener()` as soon as possible. Without at least these two functions, the ad cannot register a listener for the host ready event.

Ideally, the ready event is sent only after all MRAID functions are supported in the container and the host is ready to receive any MRAID call from the ad.

The ad must wait for the host to send the ready event check showing the container is ready before executing any rich media operations. In cases where the host has sent the ready event before the ad has registered to listen, the ad must use `getState()` in conjunction with the ready event as demonstrated in the following example:

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

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

The host sends the `ready` event when the ad container is loaded, initialized, and ready to receive calls from the ad. As a result, the MRAID JavaScript library is made available to the ad.

| Syntax | `"ready"` |
|---|---|
| Parameters | None |
| Triggered by | The container is fully loaded, initialized, and ready for any calls from the ad |

## 7.3 sizeChange

The host sends the `sizeChange` event whenever the ad container dimensions change in response to orientation, an ad resize request, or ad expand request. The event includes the new width and height in density-independent pixels.

| Syntax | `"sizeChange"` ❼ `function(width, height)` |
|---|---|
| Parameters | width: number, the width of the view<br>height: number, the height of the view |
| Triggered by | a change in the ad container width and height dimensions resulting from resize, expand, close, orientation, or the app registering a "size" event listener. |

## 7.4 stateChange

The host sends the `stateChange` event along with the updated state whenever the ad container state is changed in response to an ad call or a change in environment.

At any time the ad container may be in a state of:

- loading
- default
- expanded
- resized
- hidden

The ad container state may change as a result of: host initiation (loading), user-initiated close or other app interactions like window resize and orientation change, and ad calls for `expand()`, `resize()`, or `close()`. See section 4.2.1 for details on ad container states and how they're changed.

The ad can use `getState`, discussed in section 6.6, to query the host about the current state of the ad container.

| Syntax | `"stateChange"` ❼ `function(state)` |
|---|---|

| Parameters | state: String indicating either "loading", "default", "expanded", “resized”, or “hidden” |
|---|---|
| Triggered by | `expand()`, `close()`, or the app does something that changes the state of the webview |

## 7.5 exposureChange

The `exposureChange` event was introduced in MRAID 3.0 to more accurately reflect viewability.

The exposure change event is not supported in two-part ads.

When the ad registers a listener for the `exposureChange` event, the host sends the event asynchronously with the initial exposure state to all listeners for the ad. After the initial event, the host reports `exposureChange` events anytime the exposed area changes. An `exposedPercentage` value of 0.0 (zero) indicates the current ad container is not in view or has been moved to the background. Exposure changes can occur when the ad container or any of its parent views is scrolled, translated, or clipped. The host also reports `exposureChange` events when the visibility of the ad changes because of application or UI transitions.

The host must send `exposureChange` events when the exposed area of the ad view changes, even if the visible percentage does not change. Examples of such events include resizing, showing or hiding an interstitial, a banner expanding to fullscreen, and a banner ad being attached to the window. These exposure changes prompt the ad to recalculate viewability measurements based on the updated ad size.

The host may implement ‘`exposureChange`’ through native event handling or through polling in the native layer. See the section “Polling Rates and Event Thresholds” for requirements.

| Syntax | `"exposureChange" function(exposedPercentage, visibleRectangle, occlusionRectangles)` |
|---|---|
| Parameters | `exposedPercentage`: percentage of ad that is visible on screen, a floating-point number between 0.0 and 100.0, or 0.0 if not visible<br>`visibleRectangle`: the visible portion of the ad container, or null if not visible. It has the fields `{x, y, width, height}`, where `x` and `y` are the position of the upper-left corner of the visible area, relative of the upper-left corner of the ad container's current extent, and `width` and `height` are size of the visible area. If the visible |

| | |
|---|---|
| | area is non-rectangular, then this parameter is the bounding box of the visible portion, and the `occlusionRectangles` parameter describes the non-visible areas within the bounding box<br><br>`occlusionRectangles`: an array of rectangles describing the sections of the `visibleRectangle` that are not visible, or null if occlusion detection is not used or relevant. Each element of the array is has the fields `{x, y, width, height}`, where `x` and `y` are the position of the upper-left corner of the occluded area, relative of the upper-left corner of the ad container's current extent, and `width` and `height` are size of the occluded area. The rectangles must not overlap, and they must be sorted from largest area to smallest area. In common scenarios, the visible area is rectangular, and this parameter is null. If the implementation can detect non-rectangular exposures, then this parameter will be set |
| Triggered by | a change in the exposed area of the webview. See details below on what triggers an `exposureChange` event. |

Example function

```
{
 "exposedPercentage": 78,
 "viewport": {
    "width": 375,
    "height": 667
 },
 "visibleRectangle": {
    "x": 27.5,
    "y": 65,
    "width": 300,
    "height": 50
    "occlusionRectangle":
      {
        "x": 27.5,
        "y": 65,
        "width": 50,
        "height": 50
      }
```

**Triggers for exposureChange events**

An ad is considered exposed when all of these properties are true:

- The device is not sleeping or locked and the screen is on.
- The application is in the foreground.

A foreground application is the active application on device screen and available for user input. On Android, an activity is the foreground between calls to the `Activity.onResume()` and `Activity.onPause()` methods. On iOS, an app is in the foreground between the `UIApplicationDidBecomeActiveNotification` and `UIApplicationWillResignActiveNotification` events.

- The ad view has at least one pixel on-screen, taking into account clipping and scrolling within the app's view hierarchy. (This calculation does not need to account for Z-order occlusion within the view hierarchy.) Note that this is a change from the MRAID 2.0 specification, which did not have any pixel thresholds for a visible ad.
- There are no modal interfaces blocking the ad. Modal interfaces can include MRAID features like an in-app browser or app store displayed via `mraid.open()`, video playback via `mraid.playVideo()`, photo album access via `mraid.storePicture()`, and calendar access via `mraid.createCalendarEvent()`, as well as other in-app modal screens.

If any of these conditions don't hold, the ad is out of view and not exposed.

The host reports an exposure change as follows:

| Condition | Exposure change reported |
|---|---|
| Ad container is exposed | exposureChange events have non-zero percentage argument |
| Ad container is out of view or behind other active views | exposureChange events have zero exposedPercentage argument |
| Ad container changes from exposed to out of view | host sends exposureChange with zero (0) exposedPercentage argument |
| Ad container changes from out of view to exposed | host sends exposureChange with non-zero exposedPercent argument |
| Ad container state or size changes, but exposure percentage remains the same | host sends an exposureChange event with same exposedPercent argument as previously reported |

## 7.6 audioVolumeChange

The host uses `audioVolumeChange` to report changes in audio volume percentage. The audio volume may be calculated by multiplying the device volume by any gating factors, which may include the application audio focus, application-specific volume level, device muting, and other factors. If the

application is not able to play audio, or if it cannot determine the audio volume, this event reports null instead of a percentage.

| Syntax | `"audioVolumeChange"` ➐ `function(volumePercentage)` |
|---|---|
| Parameters | volume_Percentage: percentage of maximum audio playback volume, a floating-point number between 0.0 and 100.0, or 0.0 if playback is not allowed, or null if volume can't be determined |
| Triggered by | a change in the audio playback volume of the ad |

When an ad registers for an `audioVolumeChange` event, the host asynchronously sends the event with the initial audio volume level to all listeners for that ad. After that initial event, the host reports `audioVolumeChange` events if the audio volume changes.

Typical causes of volume changes are when the user changes the device volume or mutes the device. The host also reports volume change when the audio focus changes because of application or UI transitions. Multiple audio change events may be reported with the same percentage value.

The host may determine the audio level through native event handling or through polling in the native layer. If the native layer uses polling, it must send an `audioVolumeChange` event no later than 1.0 second after an audio change.

The volume change event is the effective volume outside the control of the ad. The ad may also have some control over volume, such as mute control. Volume control that occurs directly in the ad is not reflected in the `audioVolumeChange` event.

In some cases, the host may not have any access to audio levels when the ad is not playing audio. Additionally, the host may not be able to tell that the audio is muted or not playing, such as when the mute switch on, but headphones still work. In such situations, the host reports a `volume_Percentage` of null before the ads plays, and then another `audioVolumeChange` event with a numeric volume level just after playback starts and the host can detect volume levels.

**Sample Usage**

When host offers the `audioVolumeChange` event in addition to the other features in MRAID, the ad will be able to determine audibility. The ad might use the following pseudocode:

1. Register a listener for the `audioVolumeChange` event. For example,
   `mraid.addEventListener('audioVolumeChange', handleVolumeChange);`

2. The `audioVolumeChange` handler may compare the volume with an audibility threshold and log time when the threshold is reached:

   ```
   function handleVolumeChange(volume_percentage) {
      if (volumePercentage && volumePercentage >= 10.0) {
          // log audible time ...
      }
   }
   ```

3. When the ad has been sampled long enough for the desired audibility metric, remove the `audioVolumeChange` listener. For example,

   ```
   mraid.removeEventListener('audioVolumeChange',
   handleVolumeChange);
   ```

**Implementation Notes**

This JavaScript event-driven API was chosen over a status API (e.g. "`getAudioVolume()`") for the following reasons:

- Fetching values immediately from the native layer for a "`getAudioVolume()`" status call in JavaScript would involve blocking the webview execution thread, which negatively impacts performance.
- In order to prevent a "`getAudioVolume()`" status call from blocking the JavaScript thread, the native layer would have to check audio levels (and cache the result) whether or not creative ever used the value. By using an event interface, only ads interested in managing audio volume incur the cost of determining the value.
- An event interface avoids polling loops in the JavaScript layer.
- Adding a "`getAudioVolume()`" status API in addition to the "`audioVolumeChange`" event interface is superfluous. The JavaScript for a creative can easily implement such a function based on its handler for the event.

***Android***

On Android, the class `android.media.AudioManager` provides access to the device audio level. That value must be gated by the audio focus status of the webview displaying the ad. The following pseudocode will return the device audio volume.

```
Double getAudioVolumePercentage() {

   if (!adHasAudioFocus()) return null;
   AudioManager audioManager = (AudioManager)
        getContext().getSystemService(Context.AUDIO_SERVICE)
   int currentVolume =
        audioManager.getStreamVolume(AudioManager.STREAM_MUSIC);
   int maxVolume =
        audioManager.getStreamMaxVolume(AudioManager.STREAM_MUSIC)
;
   return new Double((100.0 * currentVolume) / maxVolume);
}
```

The method `adHasAudioFocus()` in the pseudocode would call the MRAID implementation's handling of audio focus and activity background checks.

***iOS***
On iOS, the class `AVAudioSession` provides access to the device audio volume level. However, the volume level is valid only when the app has audio focus (the active audio session). The following pseudocode will return the device audio volume.

```
-(NSNumber*)getAudioVolumePercentage {
   if (![self adHasAudioFocus]) return nil;
   return @(100.0 * [AVAudioSession
sharedInstance].outputVolume);
}
```

The method `adHasAudioFocus` in the pseudocode would call the MRAID implementation's handling of audio session and app background checks.

## 7.7 viewableChange (deprecated)

The `viewableChange` event is deprecated in MRAID 3.0 and may be removed in future versions of MRAID. However, the host must still support `viewableChange` in MRAID 3.0 to maintain backward compatibility. The host sends `viewableChange` when the ad container moves from on-screen to off-screen or off-screen to on-screen. In any situation where the container may be loaded off screen, the ad should check on its viewable state and/or register for `viewableChange` before taking any action.

| | |
|---|---|
| Syntax | `"viewableChange" ➐ function(boolean)` |
| Parameters | 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 |

# 8 Working with Device Features

MRAID is designed to provide interaction with mobile device features such as geolocation, orientation, image storage, the calendar, and the native video player. Guidelines and examples for developing an ad that uses these features are provided in the following sections.

## 8.1 Device Orientation

The ad should not use window.orientation to determine orientation or orientationChange events to determine a change in orientation. On newer OS

verisons, the host cannot control orientation updates that the webview reports and sometimes the webview reports orientation inaccurately.

Instead of `window.orientation` to determine orientation, the ad must use `mraid.getCurrentPosition`() and compare width and height values to identify a portrait or landscape layout. Instead of using `orientationChange` when the layout changes, the ad must use it as a cue to check the width and height values reported in `sizeChange` to verify ad orientation.

The `getOrientationProperties` feature offered in MRAID 3.0 should be used to determine orientation when the ad is not managing ad size. See section 6.3 for details on using the `orientationProperties` object.

## 8.2 Store 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 must 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

## 8.3 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 must treat that device as if it does not support adding calendar events.

Calendar event data must 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'})
```

## 8.4 Video

Video on mobile devices can be played either inline (within the current webview, app, or mobile web page) or by opening a native player on the device. For many advertising applications, inline playback is preferred: it is less disruptive to the viewer's experience, and playback within a webview 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 must support inline playback where possible, and permit ad designers to specify if video creative must 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 must consistently insert any necessary enabling tags into the webview 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

# 9 VPAID Events and Methods

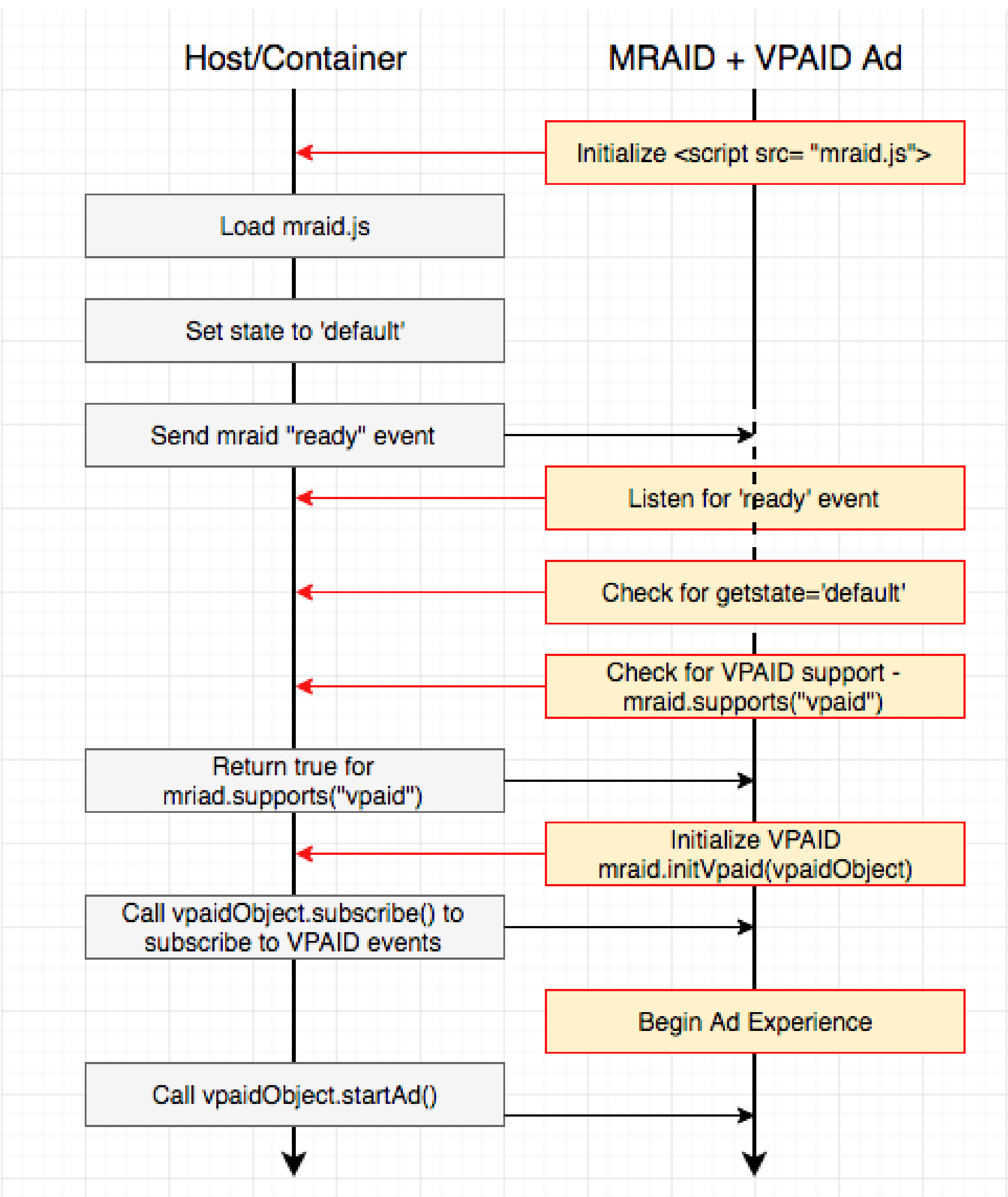
MRAID handles the interaction between rich media ads and the app or the webview. When the ad includes a video component, the host can indicate to the ad that it is interested in video playback events as well as video timing for tracking and user experience.

IAB's Digital In-Stream Video Player-Ad Interface Definition (VPAID) is an interface that handles communication between the ad and the video player. An addendum to MRAID 2.0 introduced support for initiating VPAID and reporting certain events. MRAID 3.0 integrates this addendum and offers optional compliance for the host to support ads developed using both MRAID and VPAID.

The VPAID integration with MRAID does NOT handle ad interactions with the player. The integration enables the MRAID host to listen for VPAID events.

## 9.1 VPAID Interaction in MRAID Ads

The following diagram illustrates the interactive process of a VPAID video in an MRAID creative.




### 9.1.1 Initializing VPAID in the MRAID Context

An MRAID/VPAID ad initializes using the initialization mechanism described in Section 3. Once the container returns the "ready" event or sets the container's state to "default," the ad may check whether the container supports VPAID.

Support for VPAID can be checked using the `mraid.supports()` method using the value of "vpaid" in the following format:

```
mraid.supports("vpaid")
```

Support for VPAID in MRAID 3.0 is optional, but an app that supports MRAID 3.0 is required to return a Boolean value: true if VPAID is supported or false if not supported.

### 9.1.2 Sending and Receiving VPAID events

Once ad verifies that the host supports VPAID, the ad must invoke `mraid.initVpaid(vpaidObject)` to pass the VPAID object to the container. The host then subscribes to VPAID events and calls `vpaidObject.startAd()`. The ad can then start the video creative and send VPAID events as appropriate.

### 9.1.3 If VPAID Is Not Supported

If VPAID is not supported, the call for VPAID support returns false, results in an error, or returns an undefined result. When this happens, the creative must begin the ad experience without waiting for the `vpaidObject.startAd()`. Supported VPAID Methods and Events

Not all VPAID events are supported in an MRAID context. In order to avoid overlap with MRAID or conflicting commands, only the following VPAID methods and events are supported. When the host supports VPAID integration, VPAID methods are called using vpaidObject (for example, `vpaidObject.subscribe()` to subscribe to VPAID events). See section 5.14 for details on using VPAID methods in MRAID.

| **Supported VPAID Methods** | **Unsupported VPAID Methods** |
|---|---|
| • subscribe()<br>• unsubscribe()<br>• getAdDuration()<br>• getAdRemainingTime()<br>• startAd() | • handshakeVersion<br>• initAd()<br>• resizeAd()<br>• stopAd()<br>• pauseAd()<br>• resumeAd()<br>• expandAd()<br>• collapseAd()<br>• skipAd |
| **Supported VPAID Events** | **Unsupported VPAID Events** |
| • AdClickThru<br>• AdError<br>• AdImpression<br>• AdPaused<br>• AdPlaying | • AdDurationChange<br>• AdExpandedChange<br>• AdInteraction<br>• AdLinearChange<br>• AdLoaded |

| | |
|---|---|
| • AdVideoComplete<br>• AdVideoFirstQuartile<br>• AdVideoMidpoint<br>• AdVideoStart<br>• AdVideoThirdQuartile | • AdLog<br>• AdRemainingTimeChange (Deprecated in VPAID 2.0)<br>• AdSizeChange<br>• AdSkippableStateChange<br>• AdSkipped<br>• AdStarted<br>• AdStopped<br>• AdUserAccept Invitation<br>• AdUserClose<br>• AdUserMinimize<br>• AdVolumeChange |

Any VPAID method or event not listed here is not supported in MRAID. The methods and events listed should help track interactions on any video portion of an MRAID creative.

### 9.1.4 VPAID AdClickThru Event

In VPAID, the ad creative uses the `AdClickThru` event to communicate how a clickthrough URL must be handled. VPAID offers three parameters to define the URL (url), an ID for tracking purposes (id), and a Boolean for identifying whether the player or the ad creative must open the URL (playerHandles). The MRAID container must ignore these parameters and instead the ad must use `mraid.open()` to offer the host instructions on how to handle the video clickthrough.

### 9.1.5 VPAID AdPaused, AdPlaying Events

In VPAID, the `AdPaused` and `AdPlaying` events are used in response to the player commands, `pauseAd()` and `resumeAd()`. However, in an MRAID context, the commands are not issued so the `AdPaused` and `AdPlaying` events are instead used to notify the host when these events have occurred.

### 9.1.6 Support for Auto-Start Video

To support the optimal interactive video experience, the container webview must support inline video and autostart video. The video element in the ad creative must be able to start without user interaction and play inline.

Reference to the autostart support parameter for iOS is already included in the MRAID spec under the additional information provided for inline video. Containers supporting this addendum must add autostart support to a webview.

If autostart is not supported on a given device, operating system or app, then the container must return `mraid.supports("vpaid")` as "false", so that the ad creative can either play a non-interactive version of the video content, allow the user to initiate video playback manually, or take some other action.

Note, however, that video ads requiring manual playback should be an edge case - advertisers must make sure that publisher/network ad servers target around OS versions that do not support autostart to make sure this is the case.

## 9.2 Clickthrough Behavior and Viewability

When the user clicks an ad, the container opens a new browser window in the app e.g. SFSafariViewController or WKWebView in iOS or custom tabs in Android or sends user to the device's default browser using the mraid `open()` method. In this new window, the container lacks any method to communicate the shift from the webview of the ad to a new browser window. Likewise, the container cannot indicate when a user closes the new browser window and returns to the first. Because of the blind spot in the new browser window, the creative may not know when to pause any animation or video element.
MRAID 3.0 uses the new `exposureChange` event to report a shift in exposure. This event will report the change by reporting the `exposedPercentage` and `occlusionRectangle` parameters.

The clickthrough process should be managed as follows:

1. On requesting the URL to display (this can either be from the main ad unit, or from within an interactive engagement state), the ad unit calls `mraid.open()`.
2. The container opens a new browser window within the app or in the default device browser and fires `exposureChange()`. This new browser window should contain a close button and may also contain other controls, such as a back button or return to app indicator.
3. Upon calling `mraid.open()`, the creative must also fire the VPAID `AdClickThru` event if a VPAID object has been registered with the host so that the integration layer may track the click using VPAID.
4. If the ad pauses the video, the ad must fire the VPAID `AdPaused` event.
5. When the new browser window is closed or user returns from default device browser to the app, the container must send the `exposureChange()` to report that the new `exposedPercentage` and new `occlusionRectangle` values. The creative can then resume execution as designed. If video play resumes, the VPAID `AdPlaying` event must be sent.

## 9.3 Counting Impressions

MRAID and VPAID are technical specifications for managing interaction between the ad and a mobile app or mobile web app. Neither specification was intended as measurement guidelines. However, both specifications are designed to support measurement guidelines.

When the ad is a video using VPAID events to count impressions, reports must reflect compliance with the latest IAB guidelines on counting digital video ad impressions.

The following is an excerpt from the [Digital In-Stream Video Impression Measurement Guidelines]:

> *A valid digital video ad impression may only be counted when an ad counter (logging server) receives and responds to an HTTP request for a tracking asset from a client. The count must happen after the initiation of the stream, post-buffering, as opposed to the linked digital video content itself.*
>
> *Specifically, measurement should not occur when the buffer is initiated, rather measurement should occur when the ad itself begins to appear on the user's browser, closest to the opportunity to see.*

In-stream video ad impressions must be counted using the VPAID `AdImpression` event. If the ad is treated as a rich media ad, the impression can be counted on the `adload` response using MRAID. When counted in MRAID, the VPAID `AdImpression` event is not needed and the ad container must not wait for it.

MRAID video ads that can be counted using VPAID events are only those that interrupt the flow of content and fill most of the ad container, such as interstitial video ads. MRAID ads with a video component that run inline must use MRAID events to count ads.

The following table identifies six common scenarios that include video component in an MRAID ad and specifies when VPAID may be used to count the video impression.

| Format | Description or Example | MRAID placementType | Report VPAID events? |
|---|---|---|---|
| **Video interstitial, non-dismissible** | E.g. a pre-app ad or a video interstitial between two levels of a game | interstitial | Yes |

| | | | |
|---|---|---|---|
| **skippable pre-roll** | E.g. a 640x480 placement on an iPad that takes over 3/4 of a screen and plays before the video content | interstitial | Yes |
| **Post-roll with an end-card** | E.g., a 15 second post-roll video with an interactive end-card that is displayed and stays on screen until the user closes the ad | interstitial | Yes |
| **Simple banner that plays a video on click** | E.g., a 300x250 banner in a newspaper app | inline | No |
| **A banner that on click expands and plays a video** | E.g. a 320x50 static banner that on tap calls mraid.expand() and plays a video in a <video> element. After the video finishes, the ad collapses. For the user, experience is almost the same as using mraid.playVideo() from the banner; a basic tap-to-video | inline | No |
| **A banner that on click expands and offers a grid of videos** | Similar to the previous, but when expanding, instead of a video playing immediately, the creative contains a grid of poster images for different videos. The user may choose and play multiple times before closing | inline | No |

# 10 Glossary of Terminology

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 roughly 1/160 of an inch (1 device pixel on a device with roughly 160 DPI).

On iOS, these must 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 must use the following formula:

css_pixels * viewport_scale = density_independent_pixels

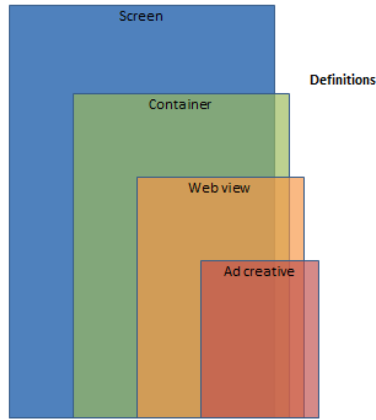
**Inline Ad:** An ad that appears onscreen accompanied by other kinds of content, e.g., a banner on a web page or in an app.

**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:** Software 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.

**Webview:** The HTML-based viewer that displays the ad creative. The webview is used to perform rendering of HTML- and Javascript-enabled ads.




# 11 Appendix: W3C CalendarEvent 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)

Copyright © 2011 W3C® (MIT, ERCIM, Keio), All Rights Reserved. W3C liability, trademark and document use rules apply.

---

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.*