Functional Key Handling in Vewd App Store Applications

Please note that parts of this article refers to older Presto-based versions of the Vewd products Vewd Core 3.x). These versions are still live in a lot of devices that are active in all markets, but most devices shipped since 2015 are based on the newer Blink-based Vewd product.

Update history:

- 8 June 2012: added requirements for Back/Return button handling
- 24 August 2012: changes to key handling in line with DOM Level 3 Events, additional information on repeating key events.
- 4 September 2012: alternative and refined approach for handling repeating key events, clarification of extra `<ELEMENT onkeydown="…">` complications to get event object, inclusion of HTML5 history API trick to circumvent automatic app closure.

Spatial navigation and functional buttons

The Vewd App Store is designed to use the standard four-way directional keys on a remote control for spatial navigation. Authors should test that their applications work correctly using the default spatial navigation built into the Vewd App Store browser.

Vewd's spatial navigation works in a similar way to traditional TAB based keyboard access in most browsers, allowing users to move between focusable elements (links, form controls, image map areas). In addition, spatial navigation also employs heuristics that make arbitrary elements with attached `click` and `mouseover` JavaScript events focusable as well. Lastly, as the name implies, spatial navigation in Vewd allows the user to move between those elements based on their spatial relationship on screen, rather than in source order (as with `TAB` navigation).

In most cases, authors can simply rely on Vewd's spatial navigation to handle their application's controls. There are simple mechanisms to further tweak spatial navigation for TV browsing using CSS3.

For maximum control, authors may also choose to handle the navigation of their application themselves by intercepting key presses from the remote control. This makes it possible to not only react to the basic directional buttons (`UP`, `RIGHT`, `DOWN`, `LEFT`), but to further bind functionality to the various shortcut and functional keys (such as `BACK`, `INFO`, `OPTIONS` or the RED button). As the exact key codes for remote control keys vary between different devices, the Vewd App Store browser provides built-in **global constants** mapped to the hardware-specific codes used by the current device.

List of available functional buttons

Hardware key	Key code constant	Availability in devices
	VK_UP	Always present in remote controllers*
	VK_RIGHT	Always present in remote controllers*
	VK_DOWN	Always present in remote controllers*
	VK_LEFT	Always present in remote controllers*
Confirm/Select/OK	VK_ENTER	Always present in remote controllers*
Exit	N/A	Usually present in remote controllers
Back/Return	VK_BACK_SPACE	Always present in remote controllers
BLUE	VK_BLUE	Usually present in remote controllers
RED	VK_RED	Usually present in remote controllers
GREEN	VK_GREEN	Usually present in remote controllers
YELLOW	VK_YELLOW	Usually present in remote controllers
Menu	VK_MENU	Not available in some remote controllers
0	VK_0	Not available in some remote controllers
1	VK_1	Not available in some remote controllers

2	VK_2	Not available in some remote controllers
3	VK_3	Not available in some remote controllers
4	VK_4	Not available in some remote controllers
5	VK_5	Not available in some remote controllers
6	VK_6	Not available in some remote controllers
7	VK_7	Not available in some remote controllers
8	VK_8	Not available in some remote controllers
9	VK_9	Not available in some remote controllers
PLAY	VK_PLAY	Not available in some remote controllers
PAUSE	VK_PAUSE	Not available in some remote controllers
STOP	VK_STOP	Not available in some remote controllers
NEXT	VK_TRACK_NEXT	Not available in some remote controllers
PREV	VK_TRACK_PREV	Not available in some remote controllers
FF (Fast-Forward)	VK_FAST_FWD	Not available in some remote controllers
REWIND	VK_REWIND	Not available in some remote controllers
SUBTITLE	VK_SUBTITLE	Not available in some remote controllers
INFORMATION	VK_INFO	Not available in some remote controllers

Note: `CONFIRM`, `EXIT` and directional buttons are mandatory for device manufacturers to implement, so they are always available for the end user via the remote control of any device that the Vewd App Store is integrated with. The `EXIT` key is handled by the Vewd App Store browser itself, to ensure that each application can be closed. For this reason `VK_EXIT` will not be sent to the application. Not all keys from the table above are always present in various remote controllers (see Availability column). It is important to take this into account when designing an app. For example, it should be possible to play and pause video, even if the PLAY or PAUSE buttons are not available.

Handling `keydown` events

Previously, authors were encouraged to handle `keypress` events. However, starting with the Vewd Core 3.4, the Vewd App Store is aligned with the DOM Level 3 Events model.

The most notable change here is that the `keypress` event is now only fired for *keys which produce a character value*. From the list of functional buttons above, this means that only the number keys `0-9` and the `ENTER` (Confirm/Select/Ok) buttons can be detected via `keypress`.

Additionally, this specification deprecates the `keypress` event, meaning that future versions of the specification — and, as a result, future versions of conformant browsers — should not fire this event anymore. For compatibility with existing content, it is unlikely that browsers will drop legacy support for this event, but we would still recommend using `keydown` instead of `keypress` going forward.

The simplest, but least elegant, way to add key events is to directly add an `onkeydown` attribute to an element. When that element has focus, the key event code will be fired. Note, though, that this old-school method requires extra work to determine the event (and the related properties, like `keyCode`) that caused the handler to be called, and doesn't necessarily work cross-platform.

```
<ELEMENT onkeydown="handler()">

function handler( ) {
        // extra hoop to jump through to get event
        if (!event) { event=window.event; }
        …
}
```

A much cleaner and flexible way would be to do this directly via JavaScript, either by attaching the handler function directly to the `onkeydown` property of the element or using `addEventListener`. This automatically passes on the event object associated with the call, avoiding any ugly `window.event` hacks:

```
object.onkeydown = handler;
object.addEventListener('keydown', handler, useCapture);
```

In the `handler` function, you can then compare the `event.keyCode` to the set of global constants for functional keys provided in the Vewd App Store.

```
function handler(event){
        …
        if (VK_RED == event.keyCode){
                /* VK_RED was pressed … do something useful */
        }
        …
}
```

Although the DOM Level 3 Events model normatively uses `event.key` and `event.char`, it still retains information on legacy key attributes such as `event.keyCode`. For compatibility with existing content, it is likely that `event.keyCode` will continue to be available for the time being. As the new event properties are not backwards-compatible, we recommend still using the current `event.keyCode` property.

Depending on the application, it is advisable not to include a large number of separate event handlers to various elements in the page, but to instead take advantage of event capture / bubbling and use an event delegation mechanism, hooking the `keydown` handler on a top-level element (for instance, the `body`) or object (`window` or similar):

```
window.addEventListener('keydown', handler, useCapture);
```

In your `handler` function, you may need to determine the element where the event originated. A reference to this can be easily obtained from the `event.target`:

```
function handler(event){
        …
        var target = event.target;
        …
}
```

Repeating key events

What happens when a user keeps a functional button on their remote control pressed is dependant on their specific device. Some devices will only send a single `keydown` event until the button is released. Others may send a series of `keydown` (and `keypress`, if it's a key that produces a *character value*) and `keyup` events (as if the button was manually being pressed and released multiple times). Lastly, platforms that do support proper key repeats will send a continuous stream of `keydown` (and `keypress`, if it's a key that produces a *character value*) events, and only fire `keyup` once the user releases the button.

In general, since it cannot be guaranteed that a device has full key repeat support, we'd recommend not making an application reliant on this behaviour.

If your applications does need to handle repeating / long-press button events, the switch to the DOM Level 3 Events model in the Vewd Core 3.4 may require some additional work in order to ensure backwards- and forwards-compatibility.

Previously, repeating keys (on supporting platforms) used to fire:

- `keydown` > [multiple `keypress`] > `keyup`

New versions of the Vewd App Store, in accordance with the DOM Level 3 Events model, will instead fire:

- [multiple `keydown` and `keypress`] > `keyup` (for keys that produce a *character value*)
- [multiple `keydown`] > `keyup` (for all other keys)

If for previous versions of the TV Store your code listened to repeating `keypress` events, the best way to remain compatible is to register your handlers for **both** `keydown` and `keypress`. To avoid having functionality being triggered twice (for the first button press in the old SDK, and for repeating *character value* keys in the new SDK), you can take advantage of the `event.repeat` property introduced in DOM Level 3 Events to filter out unwanted duplicate events:

```
// example using event delegation
window.addEventListener('keydown', handler, useCapture);
window.addEventListener('keypress', handler, useCapture);

function handler(event){
        if ((event.type=='keydown' && !('repeat' in event)) ||
                (event.type=='keypress' && ('repeat' in event))) return;
        …
}
```

Alternatively, if you're binding event handlers via JavaScript already, you can use the new `window.KeyboardEvent` interface as an indicator for DOM 3 support, and only bind your event handler to either `keydown` or `keypress`.

```
// example using event delegation
if (window.KeyboardEvent){
        window.addEventListener('keydown', handler, useCapture);
} else {
        window.addEventListener('keypress', handler, useCapture);
}

function handler(event){
        // no need to de-dupe events
        …
}
```

Requirements for the `Back/Return` button

Most remote controllers have a `Back` or `Return` button. In the Vewd TV Emulator, this is equivalent to pressing the BACKSPACE key. The Vewd App Store requires that the `Back/Return` button works consistently in each application as follows:

- Pressing `Back/Return` must return the user to the previous page or screen.
- If the user is at the first page of the application, the application should close.
- If the user is at an "Exit app" confirmation dialog, the application should close.

In other words, if the user presses `Back/Return` repeatedly, they will eventually exit the application and return to the TV Store menu.

If your application consists of regular pages loaded one after another, the `Back/Return` button should work without any extra effort — the correct behaviour is handled automatically by Vewd. If your application is using AJAX, overlays or history modifications, however, then Back/Return must be handled by your application. Here are examples of how such behaviour can be coded:

```
// To close overlays with Back/Return
function handler(event) {
        // assuming there's a global boolean overlay_opened
        if (overlay_opened && event.keyCode == VK_BACK_SPACE) {
                event.preventDefault();
                overlay_opened = false;
                closeOverlay();
        }
}
```

```
// To close the application with Back/Return
function handler(event) {
        // assuming there's a global boolean main_page
        if (main_page && event.keyCode == VK_BACK_SPACE) {
                event.preventDefault();
                main_page = false;
                window.close();
        }
}
```

Currently, the Vewd App Store has implemented some additional hardcoded behavior which automatically closes an application if the user presses Back /Return and the browser history is empty, without the possibility to `preventDefault()` the event. The idea is to avoid faulty applications from inadvertently trapping users.

To circumvent this behavior, a slightly hacky workaround is to user the HTML5 history management API to inject entries into the browser history.

```
window.history.pushState({}, document.title, '#dummy_url');
```

After this, the preceding code snippets to manually handle the Back/Return key using `preventDefault()` will work as expected.

Preventing default spatial navigation

When handling key events directly, you will probably want to stop the Vewd App Store browser from carrying out its normal spatial navigation and element activation behaviours. This can simply be suppressed in the `handler` function:

```
function handler(event){
        …
        event.preventDefault();
        …
}
```

Determining support for a specific functional key

Authors can check if a specific functional key has been defined on the current device using a simple JavaScript check. If a button is supported, the constant will contain the device-specific key code of the button; otherwise, the constant will return a `null` value. For example, to test for the VK_RED key:

```
if (VK_RED !== null) {
        /* VK_RED is supported */
        …
}
```

Although all devices running the Vewd App Store should have all the global constants listed above defined (though their value may be `null`, if the device's default remote control doesn't have a particular button), it is still advisable to also check for the existence of the constant before using it, to avoid any `Unhandled Error: Undefined variable` JavaScript errors.

```
if (('VK_RED' in window)&&(VK_RED !== null)) {
        /* VK_RED is supported */
        …
}
```

This precaution should also be taken when checking `event.keyCode` values against those constants:

```
function handler(event){
        …
        if (('VK_RED' in window)&&(VK_RED == event.keyCode)){
                /* VK_RED was pressed … do something useful */
        }
        …
}
```

The list of `VK_*` global constants is currently set in a `user.js` file, which OEMs include as part of their Vewd App Store installation at integration time. Device manufacturers will include the buttons and their respective key codes based on the default remote controls that ship with their devices. With this approach, however, any third-party or alternative remote controls may not match the standard remote's set of functional buttons — the constants may be defined and present, but the actual remote doesn't have those physical keys. For this reason, it's still advisable to use caution and to make applications work with the minimal set of *Always available* keys.