Functional Key Handling in Vewd Core Based TV Browsers

> ⚠ 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.

This article offers advice for generic browsers based on the Vewd Core. For specific information about the Vewd App Store – which introduces further functionality, as well as additional requirements to authors – please refer to this separate article: Functional Key Handling in Vewd App Store Applications.

Spatial navigation and functional buttons

Browsers based on the Vewd Core are generally designed to use the standard four-way directional keys on a remote control for spatial navigation. 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, websites can simply rely on Vewd's spatial navigation to handle site navigation. There are simple mechanisms to further [tweak spatial navigation for TV browsing](/tv/tweaking-spatial-navigation-for-tv-browsing/) using CSS3.

However, for maximum control, web authors may also choose to handle the navigation of their site (or specific aspects of their site, such as individual image carousel elements for instance) 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).

Handling `keydown` events

In previous versions of the Vewd Core, authors were encouraged to handle `keypress` events. However, starting with the Vewd Core 3.4, Vewd 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.

In the past, websites often handled key events by simply adding an `onkeydown` attribute to an element in HTML and running some inline JavaScript.

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

However, we would recommend the much cleaner and flexible method of adding handler purely in 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);
```

Before the standardisation effort of DOM Level 3 Events, key handler functions would simply compare the `event.keyCode` value to hardcoded values corresponding to specific keys. For instance, to detect if the UP arrow was pressed, the handler script would have been something like:

```
function handler(event){
        …
        if (event.keyCode == 38){
                // UP (generally keyCode 38) was pressed … do something useful
        }
        …
}
```

The potential problem with this approach has always been that the specific numerical values of `event.keyCode` were never standardised – the same functional or navigation key, on different devices, may generate different key codes. For this reason, DOM Level 3 introduces a new, more abstracted method of identifying keys in the form of the DOM Level 3 Events Key values list and the new `event.key` (as well as `event.char`, which is however only applicable to keys which produce a character value).

Starting with the Vewd Core 3.4, the recommended method of identifying navigation and functional keys is therefore:

```
function handler(event){
        …
        if (event.key == 'Up'){
                // 'Up' 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.

To ensure backwards compatibility with previous Vewd Core versions, it may be necessary to combine the old and new way of comparing key events into a single expression:

```
function handler(event){
        …
        if ((event.key == 'Up') || (event.keyCode == 38)) {
                // 'Up'/key 38 was pressed … do something useful
        }
        …
}
```

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`

Starting with the Vewd Core 3.4, in accordance with the DOM Level 3 Events model, the browser 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 Vewd Core 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
        …
}
```

Preventing default spatial navigation

When handling key events directly, you will probably want to stop the Vewd Core 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();
        …
}
```