LEVERAGING OPEN SOURCE BROWSERS TO OPTIMIZE APPS AND UI PERFORMANCE ON SET-TOP BOXES

Albert Dahan, Co-Founder and CTO, Metrological
Wouter van Boesschoten, VP of Technology and Innovation, Metrological

Abstract

Embedded browsers present a host of performance challenges for operators. Today, the user experience of enhanced browser-based services is largely dependent on the performance of the browser itself. Embedded browsers face set-top box (STB) CPU power and memory limitations that affect performance and fidelity. As consumer demand for personalized OTT offerings increase (Netflix, Hulu, etc) operators need a browser experience that can support a greater number of apps and the mechanisms in place to search and discover the right apps for each person.

HTML5 creates a standard, but it doesn't solve the issue of having to create a uniform experience on every different device. Operators need an in-house browser supported by a framework that can run cloud-based services consistently across all devices. Using an open-source approach enables operators to leverage the rapid increase in innovations coming from contributors to open-source browsing.

The solution is to adopt a browser approach that leverages open-source "WebKit for Wayland" (WPE) components. This new software approach delivers high performance rendering of HTML5 apps and next-generation user interfaces, increasing browser performance with a smaller software footprint, and requires significantly less memory usage. WPE components enable robust rendering of cloud-based applications and next-generation user interfaces and provide better window management to control multiple applications. Operators can enable cloud-based apps to run on STBs with the speed and consistency of native or local apps while avoiding the costs normally associated with proprietary based approaches.

This paper will outline how to implement an open sourced based approach that enables high performance rendering of HTML5 apps and user interfaces. It will discuss the capabilities of the browser in detail along with its pros and cons when compared to other approaches and its ability to integrate into the STB ecosystem.

BROWSER ORIGINS

In the early 90's as the first building blocks of the Internet became available there was a need for a program to retrieve, traverse and present information from internet resources. This ultimately became what we now refer to as an Internet browser. The first widely adopted browser was the NCSA Mosiac browser, which was later renamed to Netscape in 1994. Microsoft responded with Internet Explorer in 1995 and Opera Software ASA responded with Opera version 2.0 in 1996. Apple's Safari was first released in 2003 however, the origins of WebKit go all the way back to 1998, as part of the KDE HTML Layout engine (KHTML) and KDE JavaScript Engine (KJS). The WebKit project was started within Apple in 2001 as a fork of KHTML and KJS.

EVOLUTION OF WEBPAGES

In 1992 CERN launched the first webpage under the HTML1.0 specification. At the time, a webpage was nothing more than text, static images and hyperlinks to other resources. During the first wave of browser launches the HTML specification evolved at a high pace. Within a few years HTML 2.0 (1995), HTML 3.2 with CSS1 (1996), and HTML 4.0 with CSS2 and ECMAScript 1 (1997) were released by the World Wide Web (W3) consortium.

With the general adoption of the Internet, webpages became less static with server side scripting executed through technologies such as Common Gateway Interface (CGI) and HyperText Preprocessor (PHP).

The Asychronous JavaScript and XML (AJAX) technology paved the road for webpages to start changing from simply offering static information to dynamically providing the latest (server side generated) information such as news and web portals. This transition from static webpages to web-applications meant that webpages could now load dynamic data by making client originated queries towards (other) Internet resources; more data could now be loaded from different resources without traversing towards a new webpage. With the rise of social media in the mid 2000's websites dynamically retrieved data and provided (through plugins such as Flash) the ability to playback various media. This in turn, allowed users to share their videos, pictures and other updates with one another through the browser. Webpages were no longer static single load and click to the next resource applications, but interactive applications that retrieved data and updated information as the user stayed within that Internet resource.

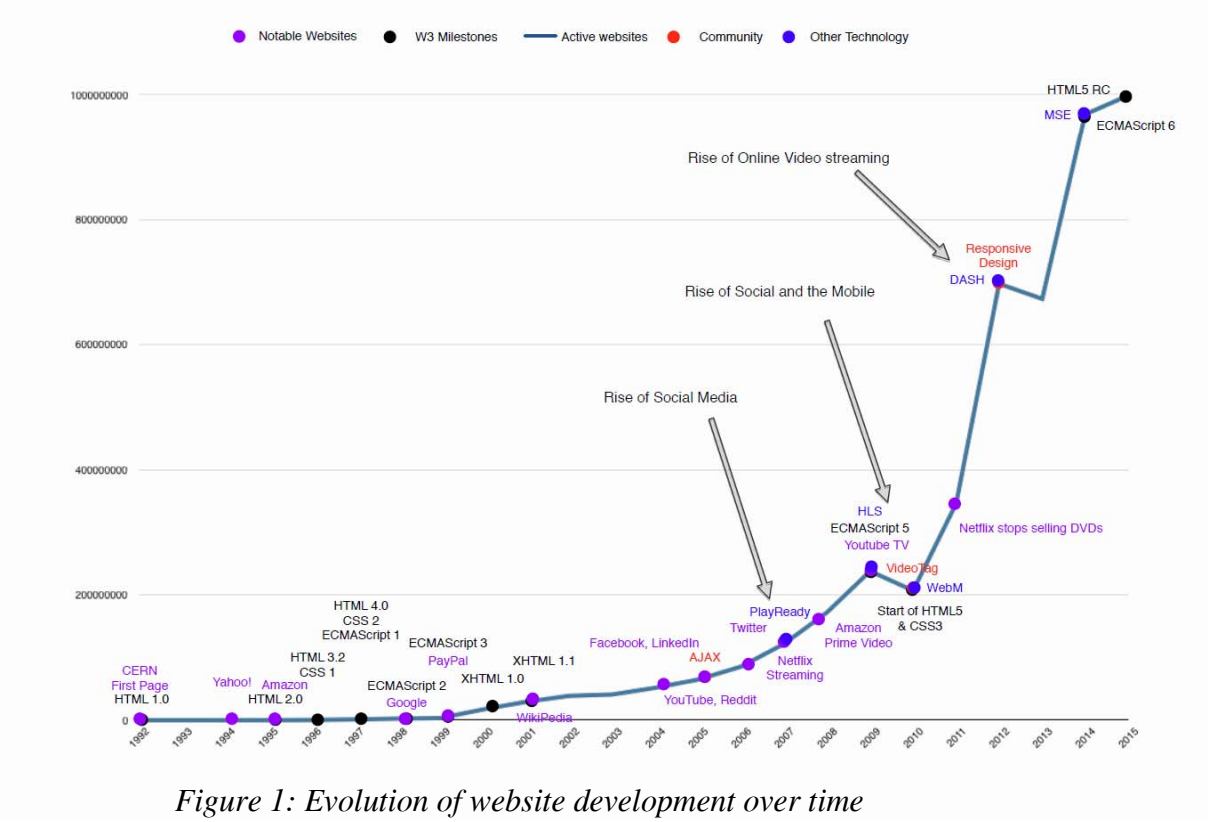

Figure 1: Evolution of website development over time

As mobile devices became popular in the late 2000's (ex: iOS, Android) the concepts of responsive design and single page Model-View-Controller web applications became popular. Various JavaScript frameworks provided the developer with tooling to build full-blown stateful graphical applications on the browser platform.

As mobile devices began to struggle with browser plugins such as Flash, Video Tag was proposed and early drafts of HTML5 and CSS3 started to emerge. Interestingly, the drafts of HTML5 came almost along 10 years after its predecessor (HTML4). While browsers started to adopt portions of the HTML5 specifications, social media platforms were dominating the Internet. Following the rise of social media, along came the rise of online video streaming, which provided further focus on the browser's capability to retrieve and render video. With websites such as YouTube serving millions of videos every day, companies like Netflix, Amazon Prime Video, Hulu started to get immensely popular. This in turn drove the need for adaptive streaming and common content encryption capabilities in the browser.

Today's websites, which are loaded by browsers, are no longer web pages. Even the term 'web page' seems to indicate something static. These words depict something that resembles a page in book or text in a newspaper. Today's web pages do not fit that description at all. These web pages provide user interaction, dynamically loaded information (sometimes even real-time), advertisements, two-way communication, audio/video streaming and 3D graphics on a platform that is uniform across multiple devices. Interestingly, this application environment isn't controlled by a single technology company. It is truly an open platform where the community drives the need for changes. Open source is leading the way and the browser is by far the most widely used application platform on the Internet. The browser is no longer 'a program to retrieve, traverse and present information from Internet resources'. It is a full application environment.

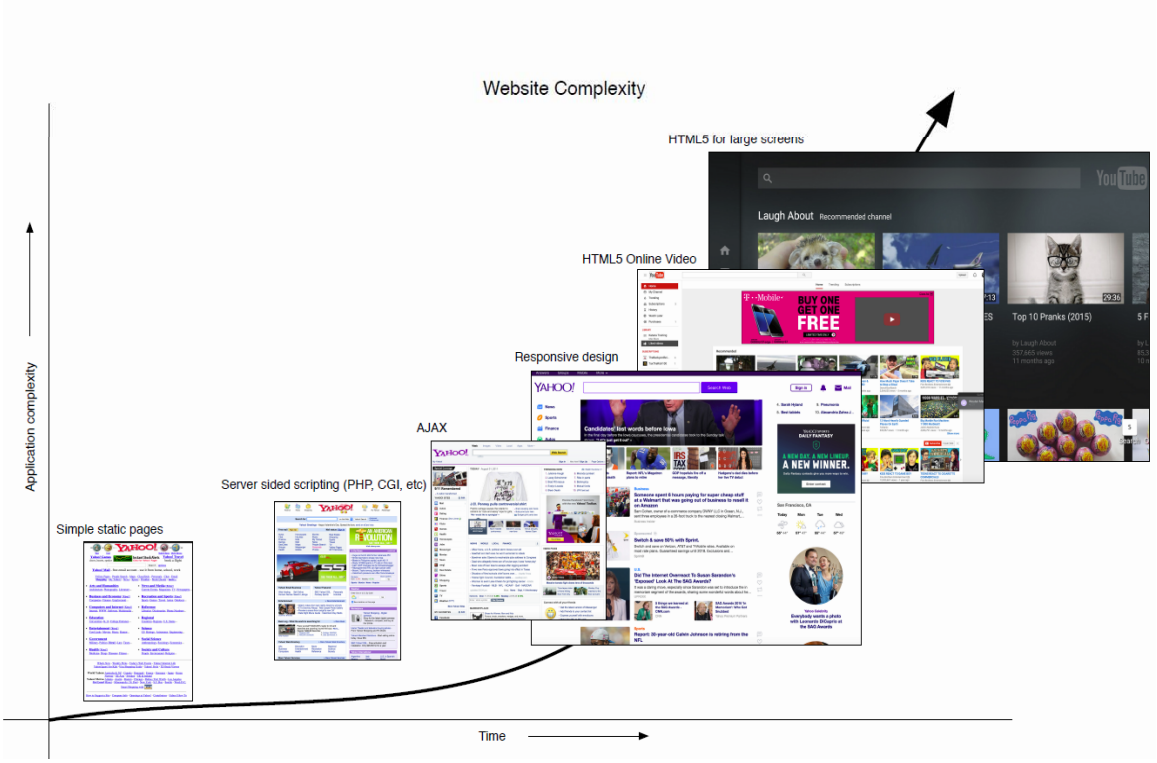

Figure 2: Website complexity over time

EMBEDDED BROWSER IMPLEMENTATIONS

Embedded devices have always been a bit of a niche market for browsers. With the rise of mobile devices there was no unification on the browser and no massive adoption of one or two browsers. Unlike the desktop market, mobile users do not have a 'preferred browser'. Historically, PC users install their own browser on their desktop. This is because at least initially operating systems did not come with a latest and greatest browser (i.e. Windows and Internet explorer). Mobile devices, on the other hand, are different. They are already bundled with a browser from its respective company (e.g. iOS comes with Safari, Android with Chrome, etc.) and use the app store mechanism to keep it up to date (an approach later adopted by desktop environments).

For embedded devices that are not running iOS, Android or Windows Mobile there is not a lot of choice. There are a few proprietary solutions available which require a license, but often these proprietary solutions struggle to keep up with the high pace of new HTML5 specifications.

Through the Chromium project, the Blink-based source code was available for porting towards embedded devices. This provides a Chrome-based browser for mobile devices. However, Blink is built for a desktop and requires desktop resources in terms of available memory, Central Processing Unit (CPU) and graphics power. It runs great on expensive embedded hardware, but not all embedded projects can afford 600 dollars' worth of hardware.

For quite some time the only license free solution was a WebKit port on top of the QT application framework. With the introduction of QT 5.4 this became increasingly harder due to licensing changes by QT. Since QT port isn't maintained upstream, WebKit is clearly the best choice due to its lightweight nature and BSD v2/LGPL v2 licensing. Being maintained by Apple, Adobe, KDE (graphical desktop environment for UNIX workstations) and others, the WebKit project is sure to quickly adopt the latest W3 specifications for HTML5.1 and beyond. All that was missing was a free and lightweight graphics framework.

FUTURE TRENDS OF W3C AND THE IMPACT ON EMBEDDED DEVICES

The shift from static web pages to an application environment isn't accidental. The W3 consortium started its catch up with other application environments with the start of the HTML5 specification. The major highlights in past iterations included video tag, drag and drop, offline apps and Canvas. The new HTML 5.1 specification enables media source extensions, encrypted media extensions, full screen Application Program Interfaces (APIs), geolocation, Indexed Database API (IndexedDB) and Web Audio support.

However W3 isn't stopping there. A closer look at the task forces that are present within the W3 reveals that they are planning to add functionality for automotive, mobile devices and TV and broadcasting. There are already W3 drafts/proposals for functionality such as screen orientation, lock screen for mobile, reading metrics for cars and the TVAPI for tuning and recording functionality on broadcasting devices.

W3 is clearly going after these other application environments and with the focus on mobile, cars and TV, the need for an

embedded browser is much more important than ever.

PERFORMANCE CHALLENGES OF EMBEDDED BROWSERS

Anyone who has developed on embedded devices will acknowledge that performance is the Achilles' heel of embedded solutions. Embedded solutions continuously balance hardware costs with performance. Like all computer hardware, the more expensive the hardware components the more resources a software program will likely have. The more resources a software program can utilize the better the performance, especially for graphical applications.

Those who expect high-end smartphone and gaming console performance out of a 60 dollar device should think again. The actual cost of a smartphone or gaming console that can support cutting edge graphics and high performance capabilities would run almost tenfold of that in terms of costs. The right balance can be hard to find. If the manufacturing costs are too high, the end consumer product will likely be too expensive. A product with functionality that is too limited, even with a small price tag, will have too many hardware compromises. This results in an end product that underperforms, yielding a negative end user experience.

Software matters when considering the capability of hardware. It won't make a 60 dollar device outperform a 600 dollar one. However, software should be at a level where it meets the hardware capabilities. In order to achieve the right performance there are several factors, such as complexity and optimizations that are crucial. However, complexity and optimizations are almost orthogonal, as optimizations can quickly get complex. The key is to combine both at the right place. Use simple solutions where possible and leverage complex optimizations that are already out there and vetted by the community.

Need for a new approach

WebKit for Wayland (WPE) does exactly that. Leveraging state-of-the-art browser optimizations capabilities provided by WebKit, such as the JavaScript Core Fourth-tier optimizing (FTL) JIT compiler, and combining that with Wayland. Wayland provides simple, elegant, graphics compositing integration between different layers using EGL (interface between Khronos rendering APIs). Wayland started as a replacement for the X Windowing System (X11). Wayland is not an implementation but a protocol specification between a display server and its clients. Its implementations are lightweight with a small footprint. Wayland is primarily focused on performance, code maintainability and security.

Integration effort, time to market

Creating proprietary solutions for a single device with limited requirements will always be viable. Performance can be safeguarded as the complexity of the solution is within controllable limits. However these kind of proprietary solutions do not scale very well over multiple devices. Even using open source does not solve the scaling issue. Using community driven components does not necessarily mean it will lower integration efforts for every device. In other open source browser solutions substantial time is lost on the integration of low-level primitives for the browser such as graphics and input. This means a full stack integration and effort on both the Software Development Kit (SDK)/driver level as well as how the browser uses those primitives.

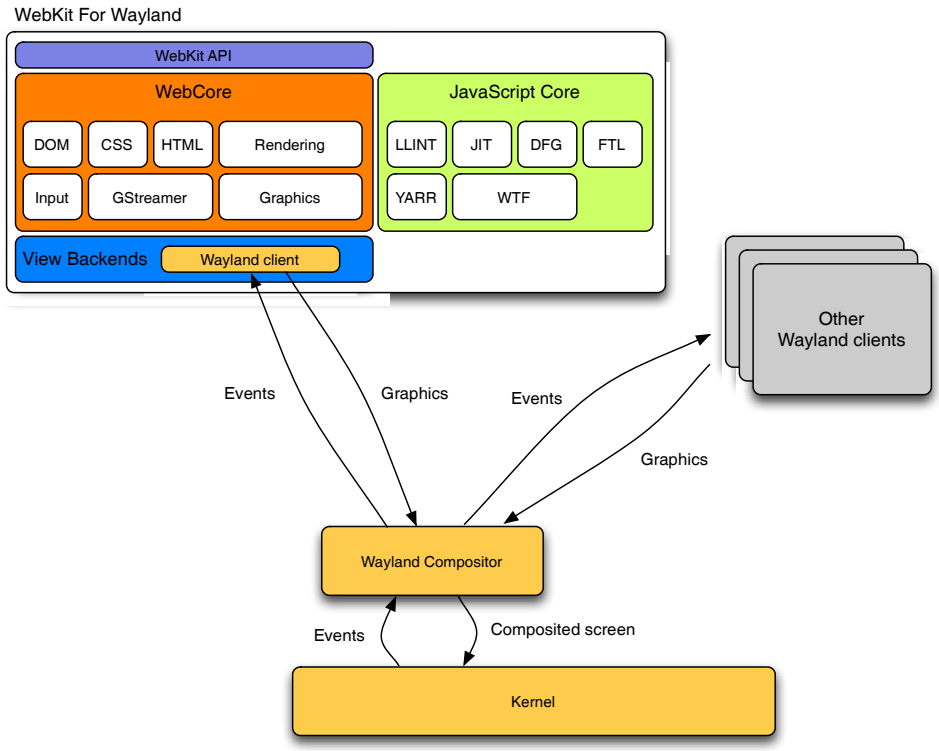

Figure 3: WebKit for Wayland architecture

To solve the issue, Wayland separates these responsibilities. It provides a unified protocol specification for a multi-program graphical compositing environment and user input. Simply put, the hardware/SDK supplier can ensure that their drivers work with Wayland by using a reference implementation (e.g. Weston) without even touching a browser. These hardware/SDK suppliers can ensure the drivers are operational with a minimal stack. And, the browser can also be validated and tested without doing full-stack integration on the target device. The browser will still need to be validated when the two are put together, but the SDK/drivers and the browser components can be independently validated. This independent validation lowers the risk of integration issues and integration effort, thus speeding up time to market.

Code maintainability

The bigger the footprint, the harder it is to maintain something. Simply put, more lines of code equal more maintenance. Open source is great, but it's important to use the right solution for the problem. The QT application framework served a broader, bigger scope than just rendering webpages. This broader scope meant a maintaining a lot of layers, abstractions and lines of code for a browser.

A browser consists of a piece that processes, parses and runs the application (DOM, JS and CSS) and a piece that renders graphics. WebKit handles the first series of functions. Wayland handles the rendering with a much leaner fit. Not only does Wayland more concisely solve the rendering problem, but it also provides an easy path forward for multiple instances of WebKit and

sharing graphical/input resources between other applications through the Wayland protocol. This functionality is often sought after in any browser solution and is more important than ever in the embedded market where multiple video streaming applications are emerging fast.

When selecting an open source project another requirement is to stay open. Quite often in the past, proprietary solutions were based off of something that was open source, but then diverged from open source with proprietary or closed modifications. Breaking too far away from the open source architecture means that the two are so far apart they can't be merged anymore. Staying code compatible and even bug-to-bug compatible with the upstream project is paramount. This way, if someone finds an issue they can solve and share it upstream and upstream fixes can be easily pulled in. Keeping the code upstream compatible is essential in order to leverage the open source community. By contributing solutions back into the community everybody benefits.

WPE contributes back to the community and is on average seven days behind the main WebKit trunk. The WebKit for Wayland browser is upstream compatible and will continue to follow the tip of the trunk from WebKit.

Native or local implementations

Quite often skeptics say, "A native user interface (UI) implementation is better then a browser". Let's focus on what this implies.

First, what is a native UI? Technically, native means something that exists or belongs to one by nature, but in this context that meaning doesn't apply. Software is (cross) compiled on a machine that uses human readable source code to turn it into something a machine can understand. That doesn't fit the meaning of native here. So let's paraphrase native and assume skeptics mean something that is closer to the hardware primitives. Often with embedded development, as mentioned above, fewer layers mean less complexity and could actually benefit performance. And that's correct. Writing code in assembly (if you know what you are doing) will always outperform code that is written in a high level programming language. Writing code in assembly would take years, if not decades. Which is why brilliant engineers in the early 1970's came up with generic purpose programming languages, such as C/C++ and beyond.

In essence it's not about writing the entire application in assembly or low-level hardware primitives. Let's focus one level up from those primitives like C and OpenGL and look at the second part of the sentence: "A native UI implementation is **better then a browser**". One can't compare a browser to a specific UI application written in C. The browser by itself doesn't do anything. It's necessary to write an application that runs within the browser. Would an application written straight on top of OpenGL outperform an HTML5 application that runs in the browser? That really depends on the implementation details of the HTML5 application. Would it outperform a set of animations in CSS? Not likely, as a browser is very efficient in determining its strategy to render CSS and it has years of experience to back that up. Technologies such as the threaded compositing provide huge benefits in terms of performance that will need to be replicated in the C application. This in turn, takes a lot of time (and money) and adds to the complexity.

Next, let's look at complex 3D graphics. Will an application that renders complex 3D graphics written in C directly on top of OpenGL outperform a HTML5 application using CSS animations? Yes, without a doubt. CSS is not meant to compete against complex 3D graphics. Cascading style sheets are created to style the text and blocks within the Document Object Model (DOM) tree. In lieu of that, the browser supports doing Canvas and Web Graphics Library (WebGL) straight from the browser for complex graphics. The browser exposes methods to access those OpenGL primitives straight from the HTML5 application. You get access to the same low-level interface. In that sense it is as 'native', to use the words of the skeptics, as writing your application straight on top of OpenGL for 3D graphics in C. So if both components are using the same APIs there isn't much difference in the one or the other for graphics. This makes the argument of 'native' no longer about access to graphical primitives, as both environments have the same accessibility.

That leaves the difference of writing an application in C versus writing an application in the browser in terms of CPU and memory performance. What is the added value of a browser over doing an application directly in C? An application in C, if written well, will be faster than anything one can create in a browser. However the key thing to note here is "if written well". C and C++ are general-purpose low-level programming languages, meaning a software developer can write code in a generic syntax and access low-level APIs. This has benefits over writing the code in assembly. C and C++ still provide access to those low-level primitives such as access to memory and other hardware APIs. However, writing against low-levels takes a great deal of time and it's important to note that C and C++ can be very unforgiving. Since it has access to low-level primitives, making a mistake can have disastrous effects to the runtime of the application. Due to its low-level nature it takes a lot of time to do basic things that are not exposed the same way as in high-level languages such as JavaScript. This includes type casting, threading (service workers/web workers), non-blocking code and all the utility the browser gives (network stack, player interfaces, etc.).

If someone makes a mistake in JavaScript the compiler will deal with it. Mistakes are unavoidable. The JavaScript Core in the browser combined with four tiers of JIT compilers will kick in as functions are used more frequently within an application. The JavaScript Core JIT compilers optimize the code to a level that is hard to accomplish with one's own C/C++ implementation. A lot of research and development went into these different JIT tiers. Because of this, it's hard to beat with your own application. At least not without spending millions of hours on optimization alone. Hours that someone has already spent and made available within the community as part of the browser.

On top of that an HTML5 application is a lot easier to maintain with equal amount of functionality in a C/C++ application. Since the code does not need to deal with a lot of primitives the code becomes smaller, simpler and less complex. This makes future maintenance easier. Finding developers is easy and often cheaper as it does not require very specific expertise that would otherwise be required in a C/C++ application. It provides equal or better performance due to the level of optimizations the browser can apply without involving the application developer.

Back to the question: “What added value does a browser have over doing your application directly in C?” Well, quite a few: ease of large-scale development, code maintainability, time to market, threaded compositing and built-in performance through its JITs to name just a few.

WEBKIT FOR WAYLAND INTEGRATION

With the removal of the QT application framework and providing a lean and mean integration towards Wayland the WebKit for Wayland browser can be easily integrated in low cost devices. If the device supports a Wayland compositor the integration is extremely straightforward. However in cases where the System on a Chip (SoC) vendor does not provide the required support for the Wayland integration the WPE ViewBackend can be extended to support direct graphics integration with the required SoC drivers. This can be used as a fallback mechanism in cases where the SoC is no longer actively developed on and where it is unlikely the hardware drivers will be extended or modified to meet the requirements of Wayland.

WPE comes with reference build environment recipes for Buildroot and OpenEmbedded for easy adoption within (existing) build systems. At the writing of this article WPE is mostly supported on all major hardware platforms, including various Broadcom chipsets, Intel CE chipsets, Nvidia gaming platforms and the Raspberry Pi family. The latter is mainly used for validation of WPE and because the Raspberry Pi is widely adopted within the open source community. This enables the open source community to quickly develop and validate using a widely available and cheap device.

Much of the development done for WPE is fed back to the community. Lots of changes originally developed for WPE have already been provided back to Apple WebKit and G-Streamer. WPE will continue to pull in changes from upstream WebKit. WPE is dedicated to continued support and contributions to the open source community.

Because of its open source nature anyone can access the source code. The source code is available on a GitHub repository: https://github.com/Metrological/WebKitForWayland.

Obtaining the source code is just a matter of cloning the repository and building for the right target machine (don’t forget to checkout the Buildroot and OpenEmbedded repositories). Contributing to WebKit for Wayland is as simple as forking the repository, make your changes, run the tests and create a pull request. The changes will be reviewed by the open source community, and if approved, merged into the trunk. That’s it. No license fees, no hours of meetings, no masses of documents and no other overhead. Just code.

CONCLUSION

Embedded browsers have always been the underdog of browsers. Historically, they were available solely as proprietary or license restrictive solutions, which didn’t offer a lot of choice. With HTML5 features growing at a rapid pace and entering new territories such as the mobile, TV and automotive industries, the need for a fast, open, bleeding edge embedded browser is greater then ever.

Embedded browsers should be widely available to anyone. Similar to most desktop browsers the embedded browser should be

free and open source. By combining the latest WebKit and Wayland integration is simple and straightforward.

The browser should follow the bleeding edge of the W3 standards. By following the WebKit main trunk closely, all new features can be pulled in with ease.

WPE provides a simple, high performance, low footprint, well maintained open source browser. By truly being part of the open source community, everybody is invited to contribute to the better future of embedded browsers.