**Sofia University “St. Kl. Ohridski”, Sofia**

Software Engineering

**Firefox OS Documentation**

*( Report for course of Operating Systems )*



**Prepared by: Supervisor:**

Yavor Mihaylov FN: 61528 Rumyana Antonova

Hristo Mohamed FN: 61498

Lachezar Tsonov FN: 61504

Georgi Antonov FN: 61499

Sofia, may 2013

1. **Firefox OS Introduction**

Firefox OS (also referred to by its codename "Boot to Gecko" or "B2G") is Mozilla's open source mobile operating system. It uses a Linux kernel and boots into a Gecko-based runtime engine, which lets users run applications developed entirely using HTML, JavaScript, and other open web application APIs. Firefox OS is a mobile operating system that's free from proprietary technology while still a powerful platform that provides application developers an opportunity to create excellent products. In addition, it's flexible and capable enough to make the end user happy.

For Web developers, the most important part to understand is that the entire user interface is a Web app, one that is capable of displaying and launching other Web apps. Any modifications you make to the user interface and any applications you create to run on Firefox OS are Web pages. It is designed to allow HTML5 applications to communicate directly with the device's hardware and services. This should make it easy for many to start working on Firefox OS apps or "port" existing web apps. This also means that the apps themselves don't have to reside on the device, they can "live" on the web itself.

For users, the advantage is that they don't have to install an app to use it and Mozilla is making the most of this with the search functionality built into Firefox OS, a core feature of the platform.

A search will return a mixture web results, direct links to buy listen to music, or even apps, depending on the query.

This makes it possible for the concept of "one time" apps to exist. If you need an app for a specific task at a certain point, you can search for it, use the app and then discard it, much like you would with a website.

This may sound like a downside for developers, but it should result in people trying out more apps than they would if they had to install them first, which should lead to more people eventually installing those apps, if they like them or use them often.

**Hardware Requirments**

It should be possible to port Firefox OS to most recent ARM-based mobile devices. This section covers the basic hardware requirements as well as the recommended hardware features.

| **Component** | **Minimum** | **Recommended** |
| --- | --- | --- |
| **CPU** | ARMv6 | Cortex A5 class or better ARMv7a with NEON |
| **GPU** | — | Adreno 200 class or better |
| **Connectivity** | — | WiFi 3G |
| **Sensors** | — | Accelerometer Proximity Ambient light A-GPS |

It's also suggested that the device offer a uniform color profile (which would be implemented by the graphics device driver) and headphone support for mute/unmute and stop/play. These are features common among modern smartphones.

Firefox OS is compatible with devices including: Otoro, PandaBoard, Emulator (ARM and x86), Desktop, Nexus S, Nexus S 4G, Samsung Galaxy S II, and Galaxy Nexus, Raspbery Pi

## Buttons and controls

A typical Firefox OS device has a small number of physical hardware buttons:

* *Home button -* This button is generally centered below the screen. Pressing it will return you to the app launcher. Holding it down opens the card switching view; swiping up on an app in that view will terminate it
* *Volume control rocker* - Along the left side is the volume rocker; pressing the top half of the rocker increases the audio volume and pressing the bottom half decreases the volume.
* *Power button* - The power button is at the top right corner of the device.

**History**

On July 25, 2011, Dr. Andreas Gal, Director of Research at Mozilla Corporation, announced the "Boot to Gecko" Project on the mozilla.dev.platform mailing list. The project proposal was to "pursue the goal of building a complete, standalone operating system for the open web" in order to "find the gaps that keep web developers from being able to build apps that are – in every way – the equals of native apps built for the iPhone (iOS), Android, and WP7 (Windows Phone 7)." The announcement identified these work areas: new web APIs to expose device and OS capabilities such as telephone and camera, a privilege model to safely expose these to web pages, applications to prove these capabilities, and low-level code to boot on an Android-compatible device.

This led to much blog coverage. According to Ars Technica, "Mozilla says that B2G is motivated by a desire to demonstrate that the standards-based open Web has the potential to be a competitive alternative to the existing single-vendor application development stacks offered by the dominant mobile operating systems."

In July 2012, Boot to Gecko was rebranded as 'Firefox OS', after Mozilla's well-known desktop browser, Firefox, and screenshots began appearing in August 2012.

In September 2012 analysts Strategy Analytics forecasted Firefox OS would account for 1% of the global smartphone market in 2013 – its first year of commercial availability.

In February 2013 Mozilla announced plans for global commercial roll-out of Firefox OS. Mozilla announced at a press conference before the start of Mobile World Congress in Barcelona that the first wave of Firefox OS devices will be available to consumers in Brazil, Colombia, Hungary, Mexico, Montenegro, Poland, Serbia, Spain and Venezuela. Firefox have also announced that LG.

Electronics, ZTE, Huawei and TCL Corporation have committed to making Firefox OS devices. At the beginning of 2013, Mozilla revealed a partnership with Spanish firm Geeksphone.

**Demonstrations**

At Mobile World Congress 2012, Mozilla and Telefónica announced that the Spanish telecommunications provider intended to deliver "open Web devices" in 2012 based on HTML5 and these APIs. Mozilla also announced support for the project from Adobe and Qualcomm, and that Deutsche Telekom’s Innovation Labs will join the project. Mozilla demonstrated a "sneak preview" of the software and apps running on Samsung Galaxy S II phones (replacing their usual Android operating system). In August 2012, a Nokia employee demonstrated the OS running on a Raspberry Pi.

In December 2012, Mozilla rolled out another update and released Firefox OS Simulator 1.0 which can be downloaded as an add-on for Firefox.

**Goals**

When interviewed, Mozilla’s Director of Research Andreas Gal characterised the current set of mobile OS pages as "walled gardens" and presented Firefox OS as more accessible: "We use completely open standards and there’s no proprietary software or technology involved." Gal also said that because the software stack is entirely HTML5, there are already a large number of established developers. This assumption is employed in Mozilla's WebAPI. These are intended W3C standards that attempt to bridge the capability gap that currently exists between native frameworks and web applications. The goal of these efforts is to enable developers to build applications using WebAPI which would then run in any standards compliant browser without the need to rewrite their application for each platform.

"The primary aim of the project is to deliver a better smartphone experience to a higher proportion of the population, including at the low end of the device range portfolio."

Firefox phones are likely to be sold first in the developing world and Eastern Europe and will be at the cheaper end of the smartphone market, according to Jay Sullivan, vice president of products at Mozilla.

1. **Firefox OS Architecture**

**Terminology**

**Gaia:** The user interface of b2g. Everything drawn to screen after b2g starts up is some part of Gaia. Gaia implements a lock screen, home screen, telephone dialer, text-messaging application, camera app, ... and many more. Gaia is written entirely in HTML, CSS, and JavaScript. Its only interface to the underlying operating system is through Open Web APIs, which are implemented by Gecko. Gaia works *well* when run on top of b2g; however, since it only uses standard web APIs, it works on other OSes and in other web browsers (albeit with degraded functionality). Third-party applications can be installed alongside Gaia.

**Gecko:** The "application runtime" of b2g. At a high level, Gecko implements the open standards for HTML, CSS, and JS and makes those interfaces run well on all the OSes that Gecko supports. This means that Gecko consists of, among other things, a networking stack, graphics stack, layout engine, virtual machine (for JS), and porting layers.

**Gonk:** The lower-level "operating system" of b2g. Gonk consists of a linux kernel and userspace hardware abstraction layer (HAL). The kernel and several userspace libraries are common open-source projects: linux, libusb, bluez, etc. Some other parts of the HAL are shared with the android project: GPS, camera, among others. You could say that Gonk is an extremely simple linux distribution. Gonk is a *porting target* of Gecko; there is a port of Gecko to Gonk, just like there is a port of Gecko to OS X, and a port of Gecko to Android. Since the b2g project has full control over Gonk, we can expose interfaces to Gecko that aren't possible to expose on other OSes. For example, Gecko has direct access to the full telephony stack and display framebuffer on Gonk, but doesn't have this access on any other OS.

**Booting**

After turning on a b2g phone, execution starts in the primary bootloader. From there, the process of loading the main OS kernel happens in the usual way: a succession of higher-level bootloaders bootstrap the next loader in the chain. At the end of the process, execution is handed off to the linux kernel.

There's not a lot to say about the boot process, but there are a few things worth knowing

* The bootloaders usually show the first "splash screen" seen during device boot, which usually displays a vendor logo.
* The bootloaders implement flashing an image onto the device. Different devices use different protocols. Most phones use the fastboot protocol, but the Galaxy S II uses the "odin" protocol.
* By the end of the bootstrapping process, the modem image is usually loaded and running on the modem processor. How this happens is highly device-specific and possibly proprietary.

**Kernel (Linux)**

The linux kernel(s) in Gonk is reasonably close to upstream linux. There are a few modifications made by AOSP that are not in upstream yet. Vendors also modify the linux kernel and upstream those modifications on their own schedule. But in general, the linux kernel is close to stock.

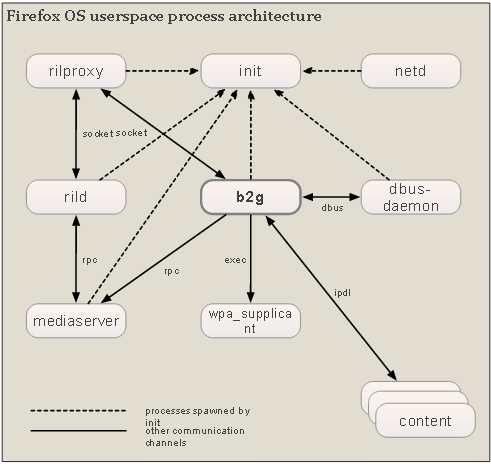
The startup process for linux is well documented elsewhere on the internet, so it's not covered here. At the end of kernel startup, a userspace "init" process is launched, like in most other UNIX-like OSes. At this point in execution, only a ramdisk is mounted. The ramdisk is built during the b2g build process, and consists of critical utilities (like init), other startup scripts, and loadable kernel modules.

After launching the init process, the linux kernel services system calls from userspace and interrupts etc. from hardware devices. Many devices are exposed to userspace through sysfs (documented elsewhere on the internet). For example, here's some code that reads the battery state in Gecko

**init**

The init process in Gonk handles mounting the required file systems and spawns system services. After that, it stays around to serve as a process manager. This is quite similar to init on other UNIX-like operating systems. It interprets scripts (that is, the init\*.rcfiles) that consist of commands describing what should be done to start various services. The Firefox OS init.rc is typically the stock Android init.rc for that device patched to include the things required to kick-start Firefox OS, and varies from device to device.

**The userspace process architecture**

****Now it's useful to take a high-level look at how the various components of Firefox OS fit together and interact with one another. This diagram shows the primary userspace processes in Firefox OS.

**B2G**

The b2g process may, in turn, spawn a number of low-rights **content processes**. These processes are where web applications and other web content are loaded. These processes communicate with the main Gecko server process through IPLD, a message-passing system.

**rild**

The rild process is the interface to the modem processor. rild is the daemon that implements the **Radio Interface Layer** (RIL). It's a proprietary piece of code that's implemented by the hardware vendor to talk to their modem hardware. rild makes it possible for client code to connect to a UNIX-domain socket to which it binds.

**rildproxy**

In Firefox OS, the rild client is the rilproxy process. This acts as a dumb forwarding proxy between rild and b2g. This proxy is needed as an implementation detail; suffice it to say, it is indeed necessary.

**mediaserver**

The  mediaserver process controls audio and video playback. Gecko talks to it through an Android Remote Procedure Call (RPC) mechanism. Some of the media that Gecko can play (OGG Vorbis audio, OGG Theora video, and WebM video) are decoded by Gecko and sent directly to the mediaserver process. Other media files are decoded by libstagefright, which is capable of accessing proprietary codecs and hardware encoders.

**netid**

The netd process is used to configure network interfaces.

**wpa\_supplicant**

The wpa\_supplicant process is the standard UNIX-style daemon that handles connectivity with WiFi access points.

**dbus-daemon**

The dbus-daemon implements D-Bus, a message bus system that Firefox OS uses for Bluetooth communication.

1. **Gecko**

Gecko, as previously mentioned, is the implementation of web standards (HTML, CSS, and JavaScript) that is used to implement everything the user sees on Firefox OS.

Most action inside Gecko is triggered by user actions. These actions are represented by input events (such as button presses, touches to a touch screen device, and so forth). These events enter Gecko through the Gonk implemention of nsIAppShell, a Gecko interface that is used to represent the primary entrance points for a Gecko application; that is, the input device driver calls methods on the nsAppShell object that represents the Gecko subsystem in order to send events to the user interface.

For example:

void GeckoInputDispatcher::notifyKey(nsecs\_t eventTime,

                                     int32\_t deviceId,

                                     int32\_t source,

                                     uint32\_t policyFlags,

                                     int32\_t action,

                                     int32\_t flags,

                                     int32\_t keyCode,

                                     int32\_t scanCode,

                                     int32\_t metaState,

                                     nsecs\_t downTime) {

  UserInputData data;

  data.timeMs = nanosecsToMillisecs(eventTime);

  data.type = UserInputData::KEY\_DATA;

  data.action = action;

  data.flags = flags;

  data.metaState = metaState;

  data.key.keyCode = keyCode;

  data.key.scanCode = scanCode;

  {

    MutexAutoLock lock(mQueueLock);

    mEventQueue.push(data);

  }

  gAppShell->NotifyNativeEvent();

}

These events come from the standard Linux input\_event system. Firefox OS uses a light abstraction layer over that; this provides some nice features like event filtering. You can see the code that creates input events in the EventHub::getEvents() method in widget/gonk/libui/EventHub.cpp.

Once the events are received by Gecko, they're dispatched into the DOM by nsAppShell:

static nsEventStatus sendKeyEventWithMsg(uint32\_t keyCode,

                                         uint32\_t msg,

                                         uint64\_t timeMs,

                                         uint32\_t flags) {

    nsKeyEvent event(true, msg, NULL);

    event.keyCode = keyCode;

    event.location = nsIDOMKeyEvent::DOM\_KEY\_LOCATION\_MOBILE;

    event.time = timeMs;

    event.flags |= flags;

    return nsWindow::DispatchInputEvent(event);

}

After that, the events are either consumed by Gecko itself or are dispatched to Web applications as DOM events for further processing.

**Graphics**

At the very lowest level, Gecko uses OpenGL ES 2.0 to draw to an GL context that wraps the hardware frame buffers. This is done in the Gonk implementation of nsWindow by code similar to this:

gNativeWindow = new android::FramebufferNativeWindow();

sGLContext = GLContextProvider::CreateForWindow(this);

The FramebufferNativeWindow class is brought in directly from Android; see *FramebufferNativeWindow.cpp*. This uses the **gralloc** API to access the graphics driver in order to map buffers from the framebuffer device into memory.

Gecko uses its Layers system to composite drawn content to the screen. In summary, what happens is this:

1. Gecko draws separate regions of pages into memory buffers. Sometimes these buffers are in system memory; other times, they're textures mapped into Gecko's address space, which means that Gecko is drawing directly into video memory. This istypically done in the method BasicThebesLayer::PaintThebes().
2. Gecko then composites all of these textures to the screen using OpenGL commands. This composition occurs in ThebesLayerOGL::RenderTo().

The details of how Gecko handles the rendering of web content is outside the scope of this document.

**Hardware Abstraction Layer (HAL)**

The Gecko hardware abstraction layer is one of the porting layers of Gecko. It handles low-level access to system interfaces across multiple platforms using a C++ API that's accessible to the higher levels of Gecko. These APIs are implemented on a per-platform basis inside the Gecko HAL itself. This hardware abstraction layer is not exposed directly to JavaScript code in Gecko.

**How the HAL works**

Let's consider the Vibration API as an example. The Gecko HAL for this API is defined in hal/Hal.h. In essence (simplifying the method signature for clarity's sake), you have this function:

void Vibrate(const nsTArray<uint32> &pattern);

This is the function called by Gecko code to turn on vibration of the device according to the specified pattern; a corresponding function exists to cancel any ongoing vibration. The Gonk implementation of this method is in *hal/conk/GonkHal.cpp*:

void Vibrate(const nsTArray<uint32\_t> &pattern) {

  EnsureVibratorThreadInitialized();

  sVibratorRunnable->Vibrate(pattern);

}

This code sends the request to start vibrating the device to another thread, which is implemented in VibratorRunnable::Run(). This thread's main loop looks like this:

while (!mShuttingDown) {

  if (mIndex < mPattern.Length()) {

    uint32\_t duration = mPattern[mIndex];

    if (mIndex % 2 == 0) {

      vibrator\_on(duration);

    }

    mIndex++;

    mMonitor.Wait(PR\_MillisecondsToInterval(duration));

  }

  else {

    mMonitor.Wait();

  }

}

*vibrator\_on()* is the Gonk HAL API that turns on the vibrator motor. Internally, this method sends a message to the kernel driver by writing a value to a kernel object using sysfs.

#### Fallback HAL API implementations

The Gecko HAL APIs are supported across all platforms. When Gecko is built for a platform that doesn't expose an interface to vibration motors (such as a desktop computer), then a fallback implementation of the HAL API is used. For vibration, this is implemented in *hal/fallback/FallbackVibration.cpp*.

void Vibrate(const nsTArray<uint32\_t> &pattern) { .. }

#### Sandbox implementations

Because most web content runs in content processes with low privileges, we can't assume those processes have the privileges needed to be able to (for example), turn on and off the vibration motor. In addition, we want to have a central location for handling potential race conditions. In the Gecko HAL, this is done through a "sandbox" implementation of the HAL. This sandbox implementation simply proxies requests made by content processes and forwards them to the "Gecko server" process. The proxy requests are sent using IPDL.

For vibration, this is handled by the *Vibrate()* function implemented in *hal/sandbox/SandboxHal.cpp*:

void Vibrate(const nsTArray<uint32\_t>& pattern, const WindowIdentifier &id) {

  AutoInfallibleTArray<uint32\_t, 8> p(pattern);

  WindowIdentifier newID(id);

  newID.AppendProcessID();

  Hal()->SendVibrate(p, newID.AsArray(), GetTabChildFrom(newID.GetWindow()));

}

This sends a message defined by the PHal interface, described by IPDL in *hal/sandbox/PHal.ipdl*. This method is described more or less as follows:

Vibrate(uint32\_t[] pattern);

The receiver of this message is *the HalParent::RecvVibrate()* method in *hal/sandbox/SandboxHal.cpp*, which looks like this:

virtual bool RecvVibrate(const InfallibleTArray<unsigned int>& pattern,

            const InfallibleTArray<uint64\_t> &id,

            PBrowserParent \*browserParent) MOZ\_OVERRIDE {

  hal::Vibrate(pattern, newID);

  return true;

}

This omits some details that aren't relevant to this discussion; however, it shows how the message progresses from a content process through Gecko to Gonk, then to the Gonk HAL implementation of Vibrate(), and eventually to the graphics driver.

**DOM APIs**

**DOM interfaces** are, in essence, how web content communicates with Gecko. There's more involved than that, and if you're interested in added details, you can read about the DOM. DOM interfaces are defined using IDL, which comprises both a foreign function interface (FFI) and object model (OM) between JavaScript and C++.

The vibration API is exposed to web content through an IDL interface, which is provided in *nsIDOMNavigator.idl*:

[implicit\_jscontext] void mozVibrate(in jsval aPattern);

The *jsval* argument indicates that *mozVibrate()* (which is our vendor-prefixed implementation of this non-finalized vibration specification) accepts as input any JavaScript value. The IDL compiler, *xpidl*, generates a C++ interface that's then implemented by the Navigator class in *Navigator.cpp.*

NS\_IMETHODIMP Navigator::MozVibrate(const jsval& aPattern, JSContext\* cx) {

  // ...

  hal::Vibrate(pattern);

  return NS\_OK;

}

There's a lot more code in this method than what you see here, but it's not important for the purposes of this discussion. The point is that the call *to hal::Vibrate()* transfers control from the DOM to the Gecko HAL. From there, we enter the HAL implementation discussed in the previous section and work our way down toward the graphics driver. On top of that, the DOM implementation doesn't care at all what platform it's running on (Gonk, Windows, Mac OS X, or anything else). It also doesn't care whether the code is running in a content process or in the Gecko server process. Those details are all left to lower levels of the system to deal with.

1. **Radio Interface Layer (RIL)**

The RIL was mentioned in the section The userspace process architecture. This section will examine how the various pieces of this layer interact in a bit more detail.

The main components involved in the RIL are:

**rild:** The daemon that talks to the proprietary modem firmware.

**rilproxy:** The daemon that proxies messages between rild and Gecko (which is implemented in the b2g process). This overcomes the permission problem that arises when trying to talk to rild directly, since rild can only be communicated with from within the radio group.

**b2g:** This process, also known as the **chrome process**, implements Gecko. The portions of it that relate to the Radio Interface Layer are *dom/system/gonk/ril\_worker.js* (which implements a worker thread that talks to *rild* through *rilproxy* and implements the radio state machine; and the *nsIRadioInterfaceLayer* interface, which is the main thread's XPCOM service that acts primarily as a message exchange between the *ril\_worker.js* thread and various other Gecko components, including the Gecko content process.

**Gecko's content process:** Within Gecko's content process, the *nsIRILContentHelper* interface provides an XPCOM service that lets code implementing parts of the DOM, such as the *Telephony* and *SMS* APIs talk to the radio interface, which is in the chrome process.

**Example: Communicating from rild to the DOM**

Let's take a look at an example of how the lower level parts of the system communicate with DOM code. When the modem receives an incoming call, it notifies *rild* using a proprietary mechanism. *rild* then prepares a message for its client according to the "open" protocol, which is described in *ril.h*. In the case of an incoming call, a *RIL\_UNSOL\_RESPONSE\_CALL\_STATE\_CHANGED* message is generated and sent by rild to rilproxy.

*rilproxy*, implemented in *rilproxy.c*, receives this message in its main loop, which polls its connection to *rild* using code like this:

ret = read(rilproxy\_rw, data, 1024);

if(ret > 0) {

  writeToSocket(rild\_rw, data, ret);

}

Once the message is received from *rild*, it's then forwarded along to Gecko on the socket that connects *rilproxy* to Gecko. Gecko receives the forwarded message on its IPC thread:

int ret = read(fd, mIncoming->Data, 1024);

// ... handle errors ...

mIncoming->mSize = ret;

sConsumer->MessageReceived(mIncoming.forget());

The consumer of these messages is *SystemWorkerManager*, which repackages the messages and dispatches them to the *ril\_worker.js* thread that implements the RIL state machine; this is done in *the RILReceiver::MessageReceived()* method:

virtual void MessageReceived(RilRawData \*aMessage) {

  nsRefPtr<DispatchRILEvent> dre(new DispatchRILEvent(aMessage));

  mDispatcher->PostTask(dre);

}

The task posted to that thread in turn calls the *onRILMessage()* function, which is implemented in JavaScript. This is done using the JavaScript API function *JS\_CallFunctionName():*

return JS\_CallFunctionName(aCx, obj, "onRILMessage", NS\_ARRAY\_LENGTH(argv),argv, argv);

*onRILMessage()* is implemented in *dom/system/gonk/ril\_worker.js*, which processes the message bytes and chops them into parcels. Each complete parcel is then dispatched to individual handler methods as appropriate:

handleParcel: function handleParcel(request\_type, length) {

  let method = this[request\_type];

  if (typeof method == "function") {

    if (DEBUG) debug("Handling parcel as " + method.name);

    method.call(this, length);

  }

}

This code works by getting the request type from the object, making sure it's defined as a function in the JavaScript code, then calling the method. Since ril\_worker.js implements each request type in a method given the same name as the request type, this is very simple.

In our example*, RIL\_UNSOL\_RESPONSE\_CALL\_STATE\_CHANGED*, the following handler is called:

RIL[UNSOLICITED\_RESPONSE\_CALL\_STATE\_CHANGED] = function UNSOLICITED\_RESPONSE\_CALL\_STATE\_CHANGED() {

  this.getCurrentCalls();

};

As you see in the code above, when notification is received that the call state has changed, the state machine simply fetches the current call state by calling the *getCurrentCall()* method:

getCurrentCalls: function getCurrentCalls() {

  Buf.simpleRequest(REQUEST\_GET\_CURRENT\_CALLS);

}

This sends a request back to rild to request the state of all currently active calls. The request flows back along a similar path theRIL\_UNSOL\_RESPONSE\_CALL\_STATE\_CHANGED message followed, but in the opposite direction (that is, from ril\_worker.js toSystemWorkerManager to Ril.cpp, then rilproxy and finally to the rild socket). rild then responds in kind, back along the same path, eventually arriving in ril\_worker.js's handler for the REQUEST\_GET\_CURRENT\_CALLS message. And thus bidirectional communication occurs.

The call state is then processed and compared to the previous state; if there's a change of state, ril\_worker.js notifies the *nsIRadioInterfaceLayer* service on the main thread:

\_handleChangedCallState: function \_handleChangedCallState(changedCall) {

  let message = {type: "callStateChange",

                 call: changedCall};

  this.sendDOMMessage(message);

}

*nsIRadioInterfaceLayer* is implemented in *dom/system/gonk/*RadioInterfaceLayer.js; the message is received by its *onmessage()* method:

 onmessage: function onmessage(event) {

   let message = event.data;

   debug("Received message from worker: " + JSON.stringify(message));

   switch (message.type) {

     case "callStateChange":

       // This one will handle its own notifications.

       this.handleCallStateChange(message.call);

       break;

   ...

All this really does is dispatch the message to the content process using the Parent Process Message Manager (PPMM):

handleCallStateChange: function handleCallStateChange(call) {

  [some internal state updating]

  ppmm.sendAsyncMessage("RIL:CallStateChanged", call);

}

In the content process, the message is received by *receiveMessage()* method in the *nsIRILContentHelper* service, from the Child Process Message Manager (CPMM):

receiveMessage: function receiveMessage(msg) {

  let request;

  debug("Received message '" + msg.name + "': " + JSON.stringify(msg.json));

  switch (msg.name) {

    case "RIL:CallStateChanged":

      this.\_deliverTelephonyCallback("callStateChanged",

                                     [msg.json.callIndex, msg.json.state,

                                     msg.json.number, msg.json.isActive]);

      break;

This, in turn, calls the *nsIRILTelephonyCallback.callStateChanged()* methods on every registered telephony callback object. Every web application that accesses the *window.navigator.mozTelephony* API has registered one such callback object that dispatches events to the JavaScript code in the web application, either as a state change of an existing call object or a new incoming call event.

NS\_IMETHODIMP Telephony::CallStateChanged(PRUint32 aCallIndex, PRUint16 aCallState,

                                          const nsAString& aNumber, bool aIsActive) {

  [...]

  if (modifiedCall) {

    // Change state.

    modifiedCall->ChangeState(aCallState);

    // See if this should replace our current active call.

    if (aIsActive) {

      mActiveCall = modifiedCall;

    }

    return NS\_OK;

  }

  nsRefPtr<TelephonyCall> call =

          TelephonyCall::Create(this, aNumber, aCallState, aCallIndex);

  nsRefPtr<CallEvent> event = CallEvent::Create(call);

  nsresult rv = event->Dispatch(ToIDOMEventTarget(), NS\_LITERAL\_STRING("incoming"));

  NS\_ENSURE\_SUCCESS(rv, rv);

  return NS\_OK;

}

Applications can receive these events and update their user interface and so forth:

handleEvent: function fm\_handleEvent(evt) {

  switch (evt.call.state) {

    case 'connected':

      this.connected();

      break;

    case 'disconnected':

      this.disconnected();

      break;

    default:

      break;

  }

}

**3G Data**

There is a RIL message that places a "data call" to the cellular tower, which enables data-transfer mode in the modem. This data call ends up creating/activating a PPP interface device in the linux kernel that can be configured through usual interfaces.

**Wi-Fi**

The wifi backend for B2G simply uses wpa\_supplicant to do all of the heavy lifting. That means that its main purpose is to simply manage supplicant (and do some auxiliary tasks like loading the wifi driver and enabling or disabling the network interface). Effectively, this means that the backend is a state machine, with the states following the state of the supplicant. Bugs in the backend tend to stem from the supplicant following a state change that the code wasn't prepared to deal with.

**WifiWorker.js**

This file implements the main logic behind the wifi. That means that it runs in the chrome process (in e10s builds) and is instantiated by the SystemWorkerManager. The file is generally broken into two sections: a giant anonymous function and WifiWorker (and its prototype). The giant anonymous function, ends up being the WifiManager. It provides a local API including notifications for events like connection to the supplicant and scan results being available. In general, it contains relatively little logic, letting its one consumer "drive" while it notifies it of events and controls the details of the connection with the supplicant.

The second part of WifiWorker.js sits between the WifiManager and the DOM. It reacts to events and forwards them to the DOM and it receives requests from the DOM and performs the appropriate actions on the supplicant. It also maintains state about the wpa\_supplicant and what it needs to do next.

**DOMWiFiManager.js**

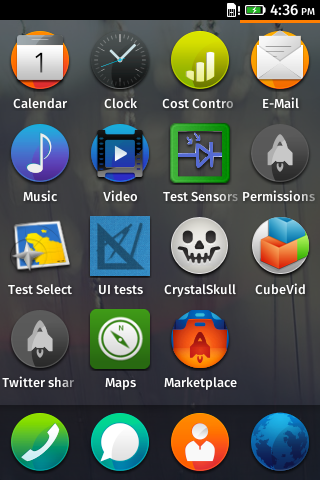
This file implements the DOM API, ferrying messages back and forth to the actual worker. There is very little logic in this file. That being said: one note: in order to avoid synchronous messages to the chrome process, we do need to cache the state based on the event that came in. There is a single synchronous message, sent at the time that the DOM API is instantiated in order to get the current state of the supplicant.

1. **Gaia**

Gaia is the user interface level of Firefox OS. Everything that appears on the screen after Firefox OS starts up is drawn by Gaia. It includes by default implementations of a lock screen, home screen, telephone dialer and contacts application, text-messaging application, camera application and a gallery support, plus the classic phone apps: mail, calendar, calculator and marketplace. Gaia is written entirely in HTML, CSS, and JavaScript. It interfaces with the operating system through Open Web APIs, which are implemented by Gecko.

Because of this design, Gaia can not only be run on Firefox OS devices, but on other operating systems and in other web browsers (albeit with potentially degraded functionality depending on the capabilities of the browser).

Third party applications onto the device installed alongside Gaia can be launched by Gaia.

**Default Interface**

Тhe default interface in Gaia is similar to what you see on most typical smartphones, as seen here.

This image is obviously of a prerelease version of the operating system, with placeholder icons (and some test applications). The status bar at the top indicates the network on which the phone is operating (or "No SIM card" for a device without a network), the network strength, WiFi signal strength, battery level, and current time.

The middle area of the display shows icons for the applications; swiping left and right pages through screens of icons.

At the bottom of the screen is a dock with room for up to seven of your most commonly used applications. You can drag and drop apps to the dock from the middle area.

**Default Applications**

By default Gaia includes implementations of: Browser, Calendar, Camera, Clock, Contacts, Dialer, Email, FM Radio, Gallery, Home, Lock, Marketplace, Music, PDF Viewer, Settings, SMS/MMS, Video.

**Calendar**

* View
* Mounth
* Week
* Day
* Agenda
* Event
* View summary
* View details
* Edit
  + Time
  + Attendees
  + Location
* Create
* Delete
* Import Calendars
* Sync between Calendars

**Camera**

***Gaia v1***

* User launches the Camera app and has the ability to see a preview of what the subject and can snap a photo
* Photos that are taken are saved to the Gallery in standard, default resolution (A X B resolution)
* Gallery access is available directly from the Camera app's primary screen
* User has the ability to apply 3 basic filters after taking a photo (filters, P2)
* User has the ability to turn the flash on/off directly from the Camera preview screen (HW dependent, P2)
* User has the ability to toggle between the front-facing and rear cameras (HW dependent, P2)
* User has the ability to toggle between mode: Video [or] Photo
* Videos that are taken are saved to the Video app for to be played/managed/deleted
* The Camera app has the ability to auto-focus on the subject
* User has the ability to crop photos after it's been captured
* Storage of photos will default to on-board flash memory (storage size TBD) and user will have the option to select the SD card slot as alternative storage
  + Settings will incorporate the option to manage the memory usage of apps and cached data
* Thumbnail creation of all photos taken for the Gallery
* Record Metadata/exif data for photos taken
* Encoding formats:
  + - Video: H.264
    - Audio: AAC
* Default resolution (camera sensor hardware dependent)

***Gaia v2***

* User has the ability to digitally zoom into a subject in the Camera mode.
* User has the ability to select focus area by tapping on preview.
* User has the ability to zoom into a subject in Video mode and then begin filming
* The Camera app has the ability to auto-detect the amount of light and adjust the exposure sensitivity
* The Camera app has the ability to auto-enhance an image right after it's been taken
* The Camera app offers the option to geo-tag all photos taken based on the user's GPS location
* The Camera app offers facial recognition in the preview mode before taking a photo
* User has the option to select between multiple resolutions

**Clock**

***Gaia v1***

* Clock home page
  + Show current time
  + Show all active alarms
* Alarm clock:
  + Set an alarm
    - Set time using the time picker
    - Set repeat option:
      * Select individual days (Monday, Tuesday, ... Saturday, Sunday)
      * All days are unselected by default
    - Set snooze time:
      * Select 5 (default), 10, 20, or 30 minutes
    - Set alarm label option:
      * Default string is "Alarm", or overwrite with your own label
    - Set alarm sound:
      * Select from our predefined list of sounds
    - Set alarm color:
      * Select from our predefined list of colors
    - Delete alarm
  + Set multiple alarms
    - Add new alarm
    - Turn alarms on and off from the clock home page (alarms are on by default)
    - Each alarm should show:
      * Time (e.g.6:50pm or 18:50)
      * Label string
      * Repeat state (e.g. Weekdays, Weekends, Mon, Tue, Wed)
      * Alarm color
      * Alarm on/off state
      * Press and hold to delete
    - After setting an alarm, show a countdown indicator: "This alarm is set for 10 hour and 20 minutes from now."
* Lock screen
  + Indicator that there is an upcoming alarm set
    - Alarm icon and color
    - Alarm time
  + When the alarm goes off
    - Show alarm icon and color
    - Show alarm label
    - Show alarm time
    - Show big snooze button
      * Tap to close alarm dialog
      * Show "Snooze for n minutes" toast
    - Show big stop button
* Time Zones
  + Set time zone automatically, but allow user to override
* Time Format
  + Obey system 24/12 hr setting
* Alarm Sounds
  + Alarm sound should gradually fade up (10-30 seconds) so it's not jarring when it goes off (wake up scenario)

***Gaia v2***

* Clock
  + Select clock themes (styles, analog, large size, etc.)
  + Display date
  + Display weather
  + World clocks
* Stopwatch
* Timer
  + Sound for the timer should not gradually fade up like the alarm clock

**Contacts**

***Gaia v1***

* The Contacts app offers several features:
  + Ability to add a new contact - available fields:
    - First Name
    - Last Name
    - Profile photo
    - Phone number - mobile (primary) and offers the ability add work and home numbers
    - Email address - home (primary) and offers the ability to add work and other email addresses
  + Scrollable contacts list
    - 'Quick search' allows user to pan through the alphabet to find a specific contact by name
  + Detailed contact view
    - In view mode, user has the ability to:
      * Select a phone number to dial it
      * Select an email address to compose an email
      * Select 'send a text message' to compose an SMS
      * Add to Favorites in the Dialer app
    - In edit mode, user has the ability to:
      * Add/edit name
      * Add/edit profile photo
      * Add/edit phone numbers
      * Add/edit email address
      * Delete contact
  + Ability to import Contacts from:
    - Previous feature/smart phone (via SIM card)
    - Online service contacts (confirming with partners)
  + Contacts in the cloud
    - Tied to a Persona account, users will have the ability to take existing imported (or entered) data in the Contacts app and 'back it up' to Mozilla's cloud servers
    - Users who need to reset their B2G phone will have the ability to log into their phone with their Persona account and their contacts would populate the Contacts app
    - Users who upgrade/switch to a new B2G phone will also be able to log into their phone and it would populate the Contacts app

***Gaia v2***

* User has the ability to easily scroll through their list sorted by:
  + First Name
  + Last Name
  + User is able to set this preference in Gaia/Settings
* User has the ability to add more data fields:
  + Contact address
  + Contact URL
  + Contact specific ringtone/text tone
  + Contact job info
  + Contact birthday
* Ability to import Contacts from:
  + Gmail
  + Yahoo! Mail
  + Hotmail
  + LinkedIn

**Dialer**

***Gaia v1***

* Dialer app offers several key areas:
  + Standard numeric dialer
    - "+" support for intl numbers
    - Holding down the delete key should repeat the delete action
  + Recents List should support:
    - Incoming calls
    - Outgoing calls
    - Missed incoming calls
    - User needs to be able to dial the number directly from this list
    - Phone numbers that have been stored in Contacts should appear as contact names - all other phone numbers should appear formatted in their native format (i.e. (XXX) XXX-XXXX)
  + Favorites (aka Speed dial)
    - User has the ability to add any existing contact as a favorite
    - User has the ability to re-arrange the order of this list
    - User has the ability to delete any contact from this list
    - User has the ability to create a new contact and add it as a favorite to this list
  + Contacts
  + Voicemail
    - Working with the carrier on accessing a user's voicemail, the user has the ability to access/manage their voicemail via a phone number in their Contacts list titled "Voicemail".
  + When receiving calls, users should be able to see who is calling them (caller ID)
    - Phone number for non-Contact specific numbers
    - Contact Name for numbers that are stored in Contacts
* Phone calling functionality:
  + During a call, the user has the ability to:
    - Mute
    - Enable/disable speakerphone
    - Show a keypad for input purposes
    - End call
  + While on call A and a user receives call B, they have the option to:
    - End their call A and answer call B
    - Ignore call B and continue with call A
    - Put call A on Hold and answer call B with the option to return to call A after call B is complete

***Gaia v2***

* Visual Voicemail
* During a phone call, the user has the ability to place a call on Hold and make a call with another contact
  + The user can have two separate calls at the same time - contact A on hold while user talks to contact B and vice versa
  + The user has the ability to conference in both contact A with contact B and carry a 3-way conversaion

**Gallery**

***Gaia v1***

* User has the ability to swipe through the thumbnail view of photos
* Full screen view of photos
* User has the ability to pan between photos in full screen view
* User has the ability to delete one or multiple photos at the same time
* Pull photos from both on-board storage as well as the SD card (if applicable)
* User has the current sharing capabilities (ability up to select 5 photos if the service supports multi-photo sharing):
  + Create an extensible menu for adding app services
  + Email
  + Social Networks/Storage
* User has the ability to set the wallpaper with any photo
* User has the ability to set a specific contact with any photo
* Respect exif orientation data when viewing photo in full and thumbnail view
* User has the ability to export/copy photos off the device when plugged into a desktop machine
  + This should appear on the desktop like a USB thumbnail drive and the user can access the 'Photos' directory
* Photo Editing
  + Exposure Compensation
  + Basic Cropping (freeform aspect ratio)
  + Basic Effects (B&W, Sepia)
  + Basic Borders (B&W thin and thick)
* Bluetooth sharing
  + Ability to share photos via Bluetooth file transfer.

***Gaia v2***

* User has the ability to move photos from SD card to on-board storage directory
* User has the ability to create collection of photos
* User has the ability to click 'play' to view a slideshow of photos
* Photo Editing
  + Cropping with fixed aspect ration options
  + Fancy Effects
  + Fancy Borders
  + Photo Captions

**Lock screen**

The Lock Screen’s primary function is to prevent accidental or insecure device access by forcing manual input(s) to access full functionality.

* User can view date and time
* User can make emergency calls (with or without SIM)
* User can unlock their device (eg: slide gesture upwards)
  + User can require a 4-digit PIN entry in order to unlock device.
    - Repeated failed attempts to unlock phone can result in timed lock-downs (eg: device locks for 5 minutes after 3 failed attempts, and for 1 day after 6 failed attempts, etc)
* User can set background image
* User can review single notifications as they arrive. Each displaying (as per Notifications spec):
  + Icon
  + String 1 (eg: sender app)
  + String 2 (eg: notification content)
  + Time stamp
* User can dismiss notifications.
* User can directly access certain functions from lock screen
  + Camera
  + Phone
  + Messaging
  + Music
    - Transport controls: Play, Pause, Stop, Next track, Prev track
    - Device controls
* User can review notifications in more sophisticated ways:
  + Multiples simultaneously
    - Grouped: chronologically, or by sender app (eg: multiple missed SMS messages grouped together)
  + User configurable stances: Alert or Passive
  + User actionable: with inputs for quick access to the issuing app