Sensor APIs implementation in Chromium: Generic Sensor Framework

Mikhail Pozdnyakov <mikhail.pozdnyakov@intel.com>
Alexander Shalamov <alexander.shalamov@intel.com>
Maksim Sisov <maksim.sisov@intel.com>

Last updated: April 12 2017

Objective

This document explains how sensor APIs (such as Ambient Light Sensor, Accelerometer, Gyroscope, Magnetometer) based on Generic Sensor API are implemented in Chromium. This document describes the Generic Sensor API implementation on both renderer process and browser process sides, it also describes important implementation details, for example how data from a single platform sensor is distributed among multiple JS sensor instances and how sensor configurations are managed.

At the time of writing following specifications were implemented in Chromium under “Generic Sensor” feature flag.

ED Specs:
Generic Sensor API https://w3c.github.io/sensors/
Ambient Light Sensor API https://w3c.github.io/ambient-light/
Accelerometer Sensor API https://w3c.github.io/accelerometer/
Gyroscope Sensor API https://w3c.github.io/gyroscope/
Magnetometer Sensor API https://w3c.github.io/magnetometer/
Absolute Orientation Sensor API https://w3c.github.io/orientation-sensor/

Background

The Generic Sensor API defines base interfaces that should be implemented by concrete sensors. In most of the cases, concrete sensors should only define sensor specific data structures and if required, sensor configuration options. Same approach is applied on the implementation side: Generic Sensor API implementation (we call it Generic Sensor Framework or **GSF**) provides the common functionality that is reused by concrete sensor implementations, its goal is to decrease to a minimum the amount of code required for implementation of a new sensor type.

Generic Sensor Framework requirements

1. Share the crucial parts of functionality between the concrete sensor implementations. Avoid the code duplication and thus simplify maintenance and development of new features.
2. Support simultaneous existence and functioning of multiple JS Sensor instances of the same type that can have different configurations and life-time.
3. Support for both “slow” sensors that provide periodic updates (e.g. AmbientLight, Proximity), and “fast” streaming sensors that have low-latency requirements for sensor reading updates (motion sensors).

Overview/Scope

The GSF implementation consists of two main components: Sensor module in Blink (located at `third_party/WebKit/Source/modules/sensor`) which contains JS bindings for Generic Sensor API and concrete sensors APIs, and ‘generic_sensor’ component (located at `services/device/generic_sensor/`) - a set of classes living on Browser process side that eventually call system API to access the actual device sensors.

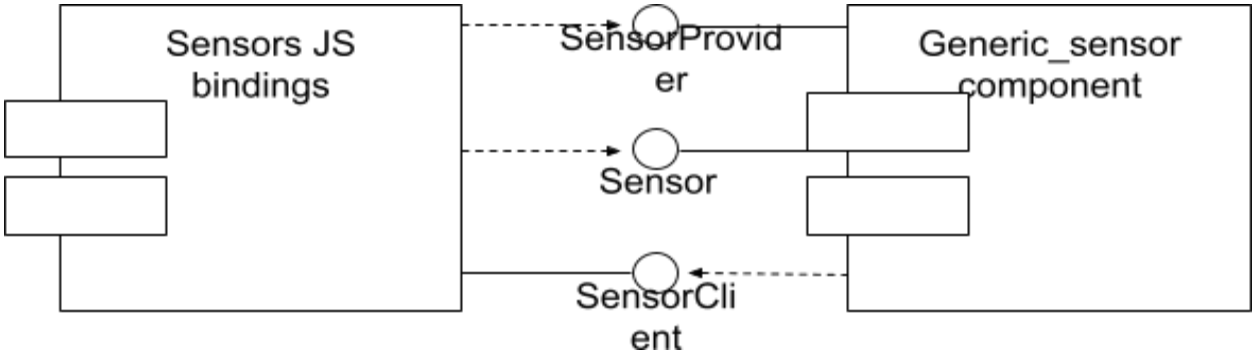
The ‘generic_sensor’ component exposes following mojo interfaces for communication with JS bindings:

- `SensorProvider` is a “factory like” interface that provides data about the sensors present on the device and their capabilities (reporting mode[1], maximum sampling frequency).
- `Sensor` is an interface wrapping a concrete device sensor. JS bindings code uses it to start polling the device sensor with the configurations obtained from JS.
- `SensorClient` is implemented by JS bindings code to be notified about errors occurred on platform side and about sensor reading updates for sensors with ‘onchange’ reporting mode.

Please note, that the fetched sensor data is not passed to JS bindings via mojo calls - shared memory buffer is used instead, thus we obviate bloating of mojo IPC channel with sensor data (for sensors with continuous reporting mode) when platform sensor has high sampling frequency and also avoid bringing of an extra latency.

[1] Sensors in GSF can have either ‘onchange’ or ‘continuous’ reporting mode, which correspond to the similar ones in Android SDK. Please refer to https://source.android.com/devices/sensors/report-modes.html for more details.

GSF component diagram is shown on figure below:



Detailed Design

Generic sensor component

The class diagram of Generic Sensor device component is shown below. All classes act on IO thread, except if documented otherwise.

`PlatformSensorProvider` is a singleton class, its main functionality is creating and tracking `PlatformSensor` instances. `PlatformSensorProvider` is also responsible for creating a shared buffer for sensor readings. Every platform has its own implementation of `PlatformSensorProvider` (`PlatformSensorProviderAndroid`, `PlatformSensorProviderWin`, ...), generic part of the functionality is encapsulated inside the inherited `PlatformSensorProviderBase` class.

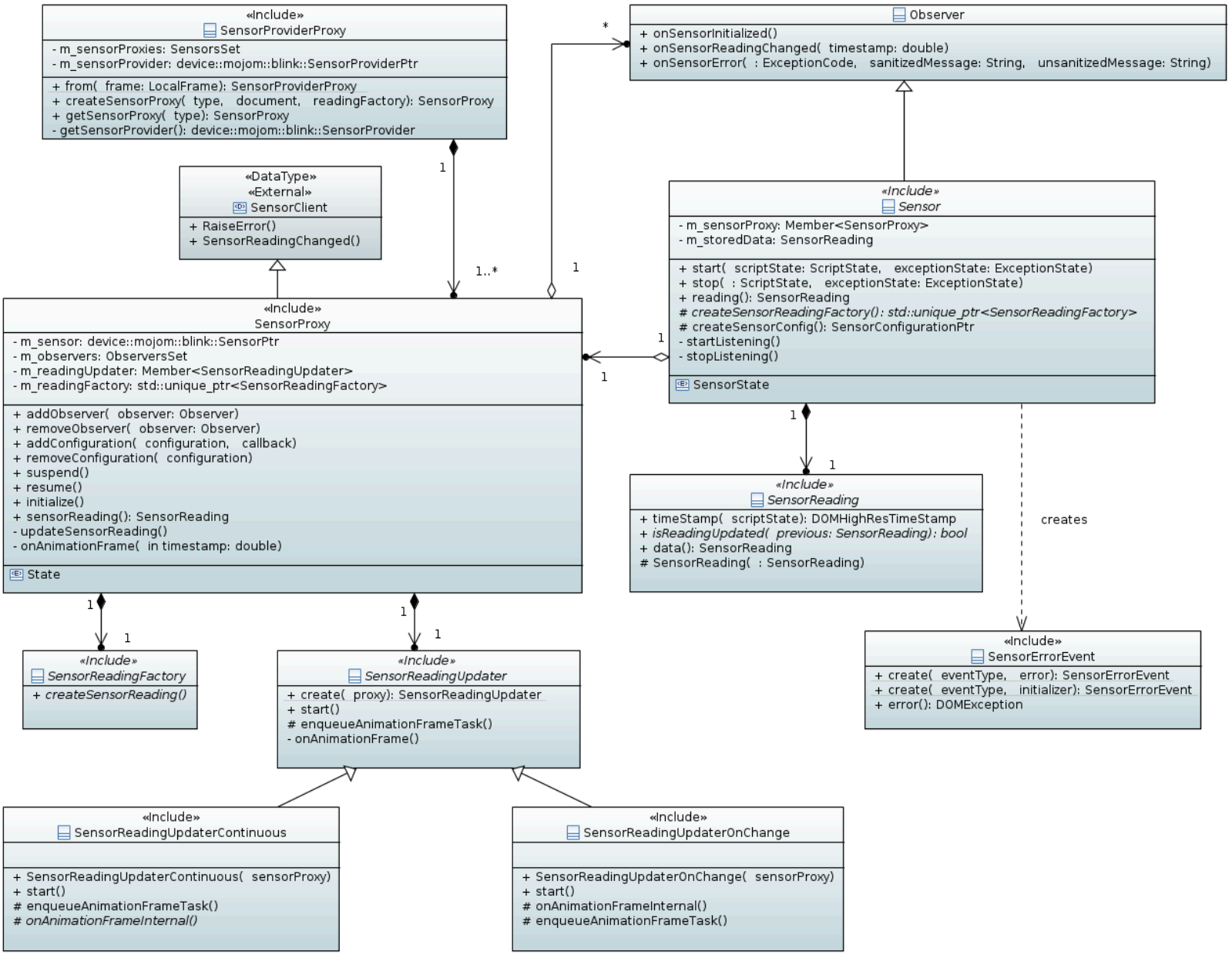
`PlatformSensor` represents device sensor of a given type, there can be only one `PlatformSensor` instance of the same type at the time, its ownership is shared between existing `SensorImpl` instances. `PlatformSensor` is an abstract class which encapsulates generic functionality and is inherited by the platform-specific implementations (`PlatformSensorAndroid`, `PlatformSensorWin`, ...).

`SensorImpl` class implements the exposed Sensor mojo interface and forwards IPC calls to the owned `PlatformSensor` instance. `SensorImpl` implements `PlatformSensor::Client` interface to receive notifications from `PlatformSensor`.

`SensorProviderImpl` class implements the exposed `SensorProvider` mojo interface and forwards IPC calls to the `PlatformSensorProvider` singleton instance.

Sensor module in Bink

Blink side class diagram is shown below.



Sensor - implements bindings for the `Sensor` interface. All classes that implement concrete sensor interfaces (such as AmbientLightSensor, Gyroscope, Accelerometer) must be inherited from it.

`SensorReading` - implements bindings for the `SensorReading` IDL interface. All classes that implement concrete sensor reading interfaces (e.g. `GyroscopeReading`) must be inherited from it.

`SensorProxy` wraps the `mojom::Sensor` mojo interface proxy and itself implements `mojom::SensorClient` interface. It provides nested `SensorProxy::Observer` interface which is inherited by Sensor class in order to receive notification from platform side.
`SensorProxy` contains the `SensorReading` instance which is shared between all Sensor instances inside the frame.

Inside a frame there can be only one `SensorProxy` instance for a concrete sensor type (i.e. ambient light, accelerometer) at the time and its ownership is shared between Sensor instances. `SensorProxy` instance is created at first `Sensor.start()` method call and destroyed when there are no more active Sensor instances left.

`SensorProviderProxy` wraps 'SensorProvider' mojo interface proxy and manages 'SensorProxy' instances. Sensor implementation obtains `SensorProviderProxy` instance as a `LocalFrame` supplement and uses it to the get `SensorProxy` instance for the needed type.

`SensorReadingUpdater` abstract class (and its implementations `SensorReadingUpdaterOnChange` and `SensorReadingUpdaterContinuous`) encapsulates the logic for sending 'onchange' event which depends on sensor's reporting mode.

Sensor shared buffer

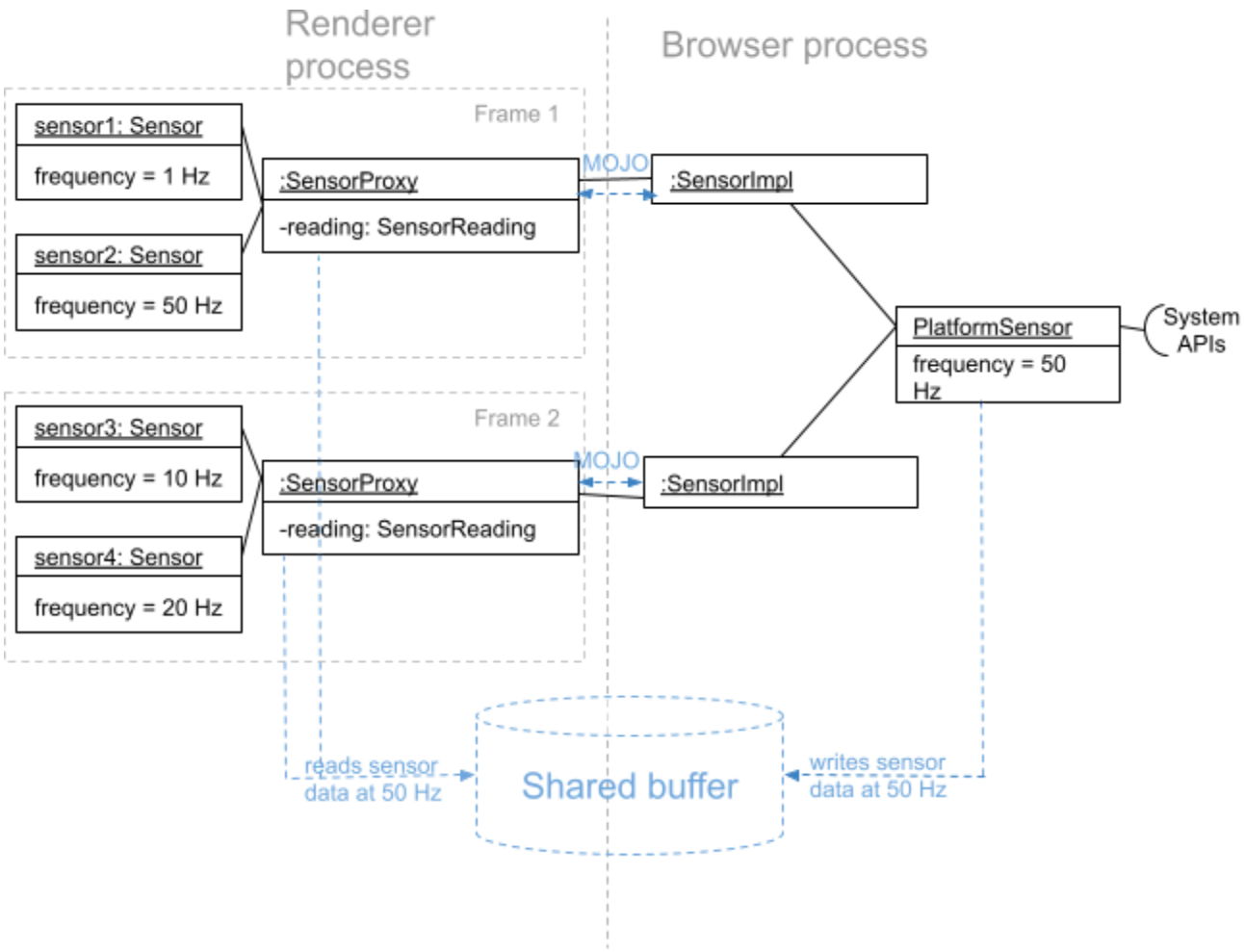
Sensor shared buffer is used to transfer sensor readings from browser process to renderer process. Read-write operations are synchronized via seqlock mechanism.

GSF uses a single shared memory buffer which is divided into chunks - sensor reading buffers, one chunk per sensor type. Every sensor reading buffer contains 6 tightly packed 64-bit floating fields: { seqlock, timestamp, sensor reading 1, sensor reading 2, sensor reading 3, sensor reading 4 }, so it has fixed size 6 * 8 = 48 bytes.

Please see `sensor_reading.h` for more details.

Sensor configurations management

This paragraph describes the implementation of logic behind `Sensor.reading` update and sending of `Sensor.onchange` event. The issue here is that relationship between device sensor and corresponding JS sensor instances is one-to-many, and each JS sensor instance may have different configuration.
In GSF the resulting configuration which is applied to the device sensor is the highest from the currently existing JS sensors configurations. The following object diagram illustrates these logic with example of four sensor instances of the same type and they have different sampling frequencies: 1Hz, 50Hz, 10Hz, 20Hz.



The maximum sampling frequency used is 50 Hz and `PlatformSensor` updates shared buffer using this frequency.

Note: the given sampling frequency value is capped to 60 Hz for security reasons, that are explained in “Security and Privacy” section of this document.

On platform side (in `generic_sensor` component) sensor configurations are managed inside `PlatformSensor` class.

On Blink side, for ‘continuous’ reporting mode, `SensorProxy` continuously updates the stored reading instance from shared buffer using periodic timer and then notifies all dependent Sensor instances that sensor reading has changed. Further, Sensor instance may send ‘onchange’ event considering its own frequency and based on the timestamp delta between the newly arrived reading and the one that had been previously send.

For ‘onchange’ reporting mode the behavior is a bit different: `SensorProxy` updates reading from shared buffer, however, unlike the ‘continuous’ reporting mode case, it does not do it all the

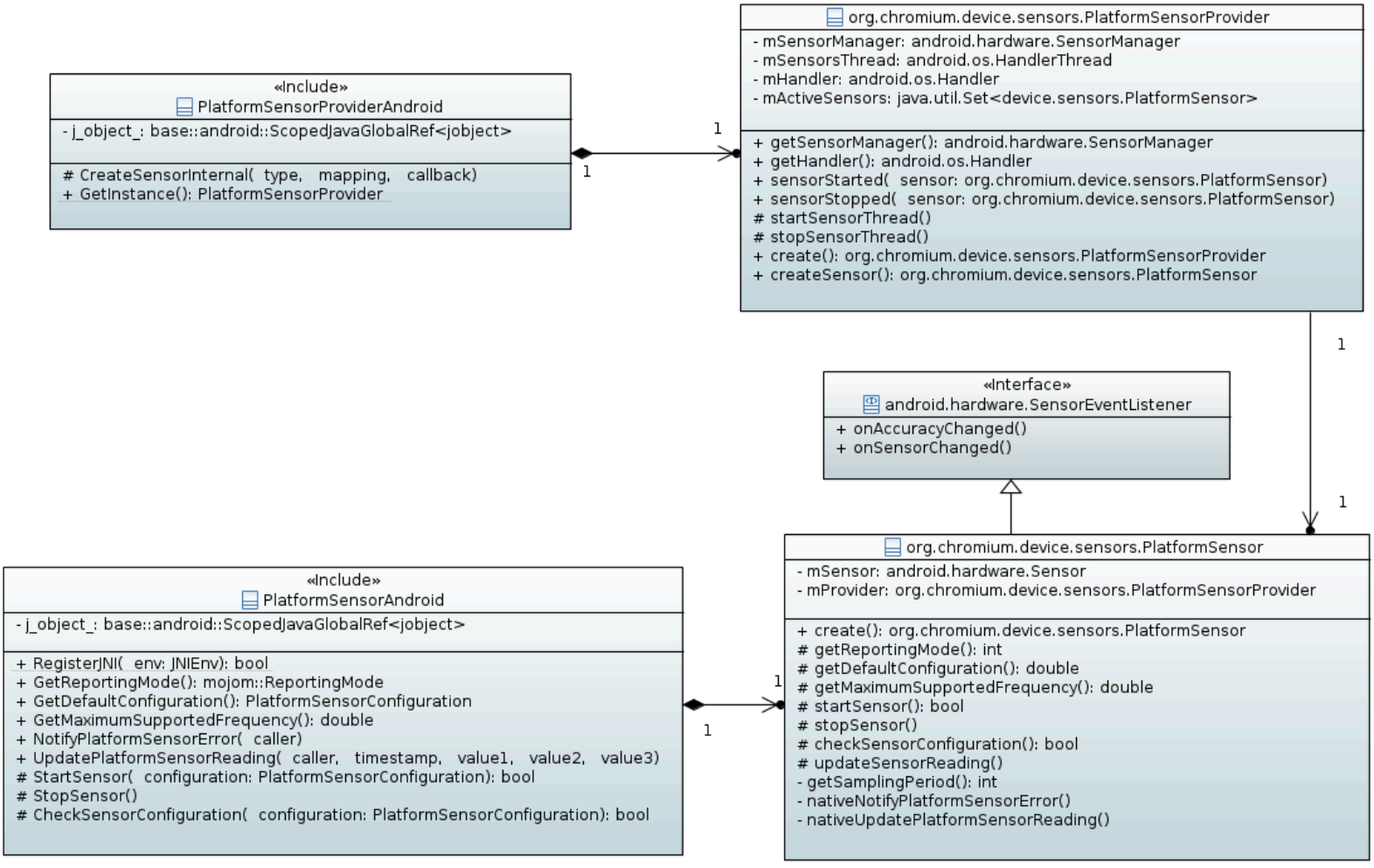
time. Reading updates start after '`SensorReadingChanged()`' mojo call and continue for the period of time equal to

$$T = 1 \div Fmin$$

where Fmin is the minimal sampling frequency from the currently present Sensor instances, in the example above it would be $1 \div 1Hz = 1s$.

Platform Implementation details

Implementation on Android



The adaptation for Android platform consists of two parts, native (C++) and java. Native adaptation includes `PlatformSensorProviderAndroid` and `PlatformSensorAndroid` C++ classes, while java counterpart consists of `PlatformSensorProvider` and `PlatformSensor`

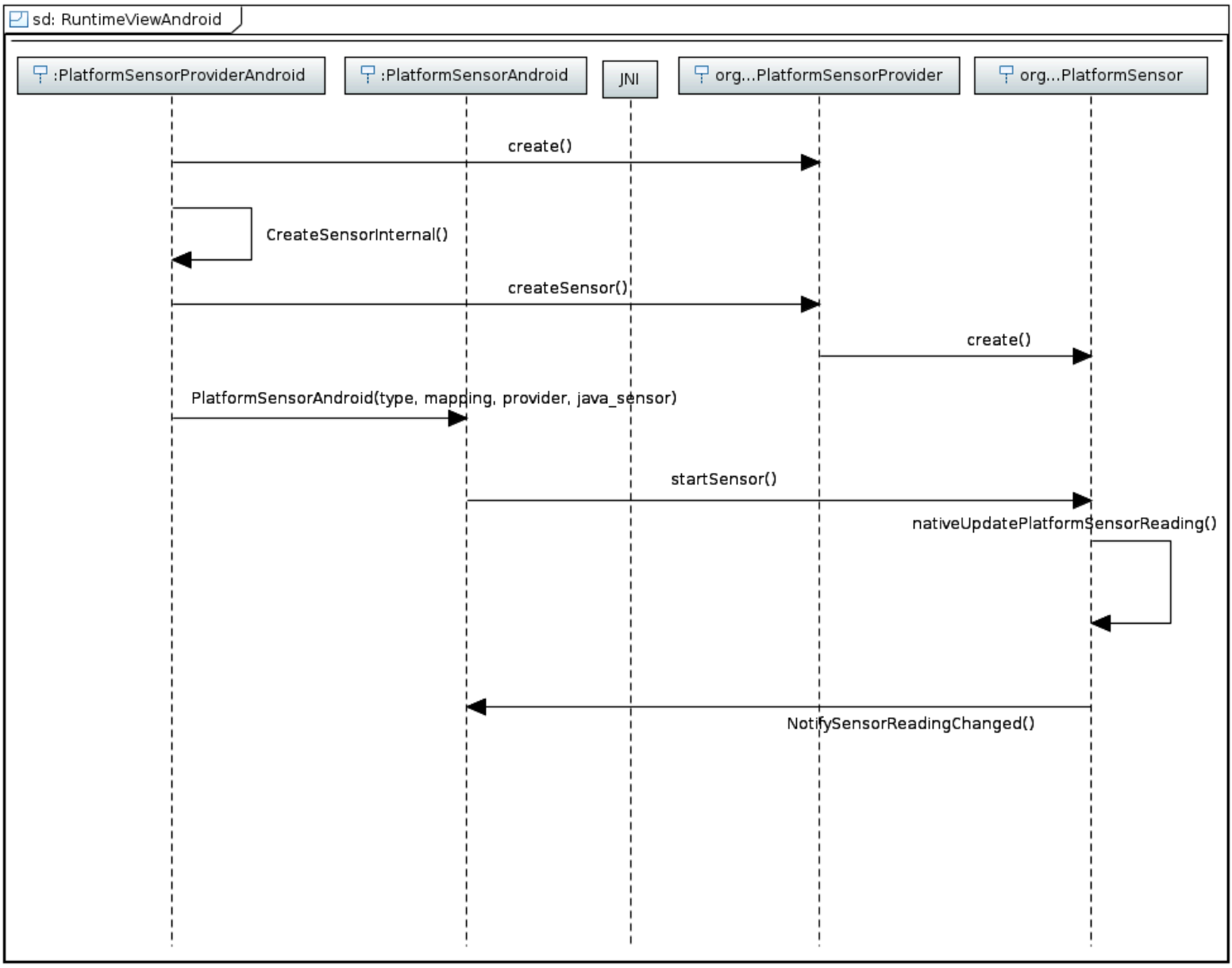
java classes that are included in `org.chromium.device.sensors` package. Java classes interface with Android Sensor API to fetch reading from device sensors. Native side and java classes communicate with each other over JNI interface.

The `PlatformSensorProviderAndroid` class implements `PlatformSensorProvider` interface and responsible for creation of `PlatformSensorProvider` instance over JNI, when java object is created, all sensor creation requests are forwarded to java object.

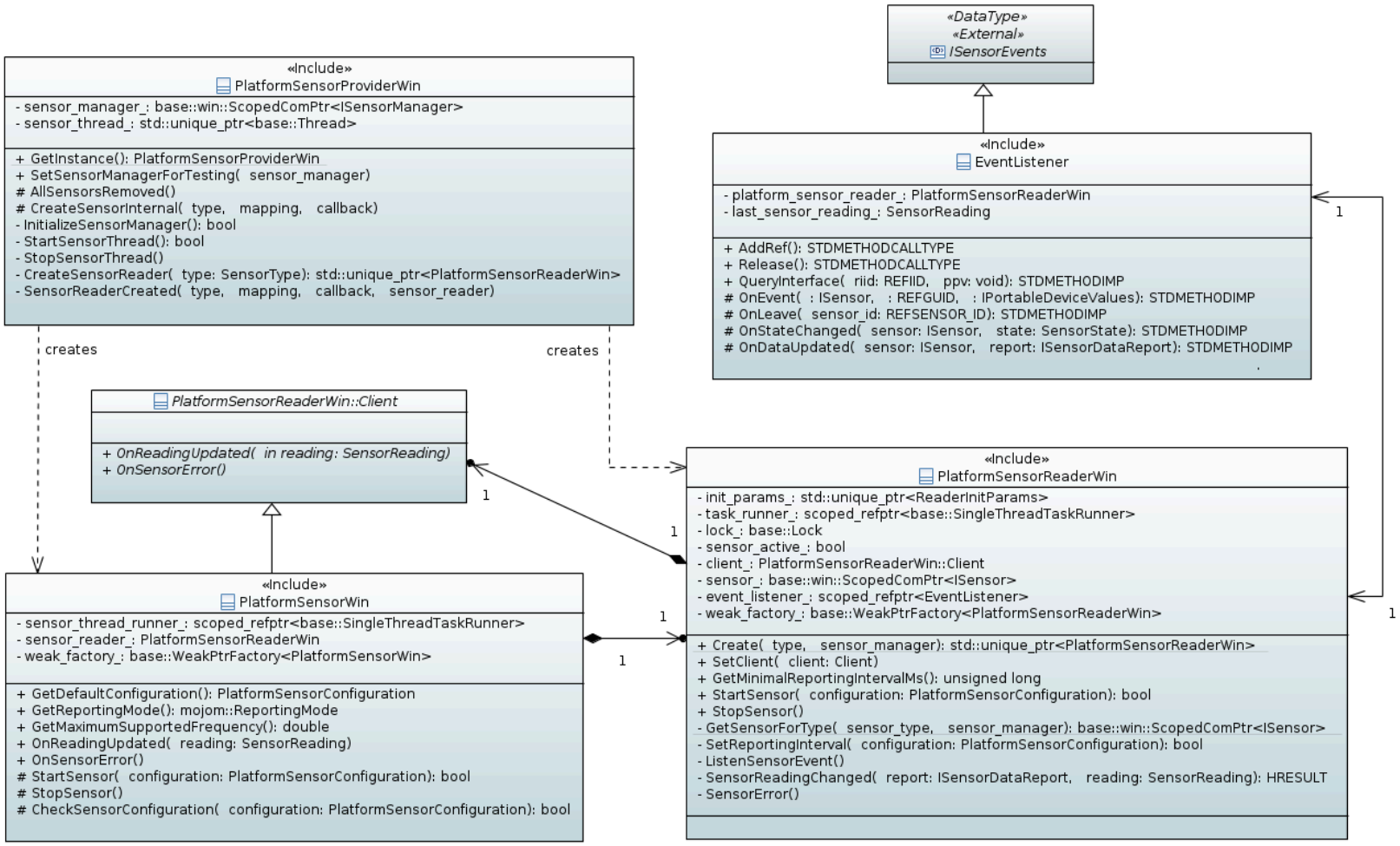
The `PlatformSensorAndroid` class implements `PlatformSensor` interface, owns `PlatformSensor` java object and forwards start, stop and other requests to it.
The `PlatformSensorProvider` java object is responsible for thread management, and `PlatformSensor` creation, it also owns Android `SensorManager` object that is accessed by `PlatformSensor` java objects.

The `PlatformSensor` java object implements `SensorEventListener` interface and owns Android Sensor object. The `PlatformSensor` adds itself as a event listener for a Sensor to receive sensor reading updates and forwards them to native side using native* methods.

Simplified runtime view.


sd: RuntimeViewAndroid
:PlatformSensorProviderAndroid
:PlatformSensorAndroid
JNI
org...PlatformSensorProvider
org...PlatformSensor
create()
CreateSensorInternal()
createSensor()
create()
PlatformSensorAndroid(type, mapping, provider, java_sensor)
startSensor()
nativeUpdatePlatformSensorReading()
NotifySensorReadingChanged()

Implementation on Windows



The adaptation layer for Generic Sensors on Windows platform uses Windows Sensor API that provides COM interfaces to interact with platform sensors. The adaptation consists of three main classes: `PlatformSensorProviderWin`, `PlatformSensorWin` and `PlatformSensorReaderWin`.

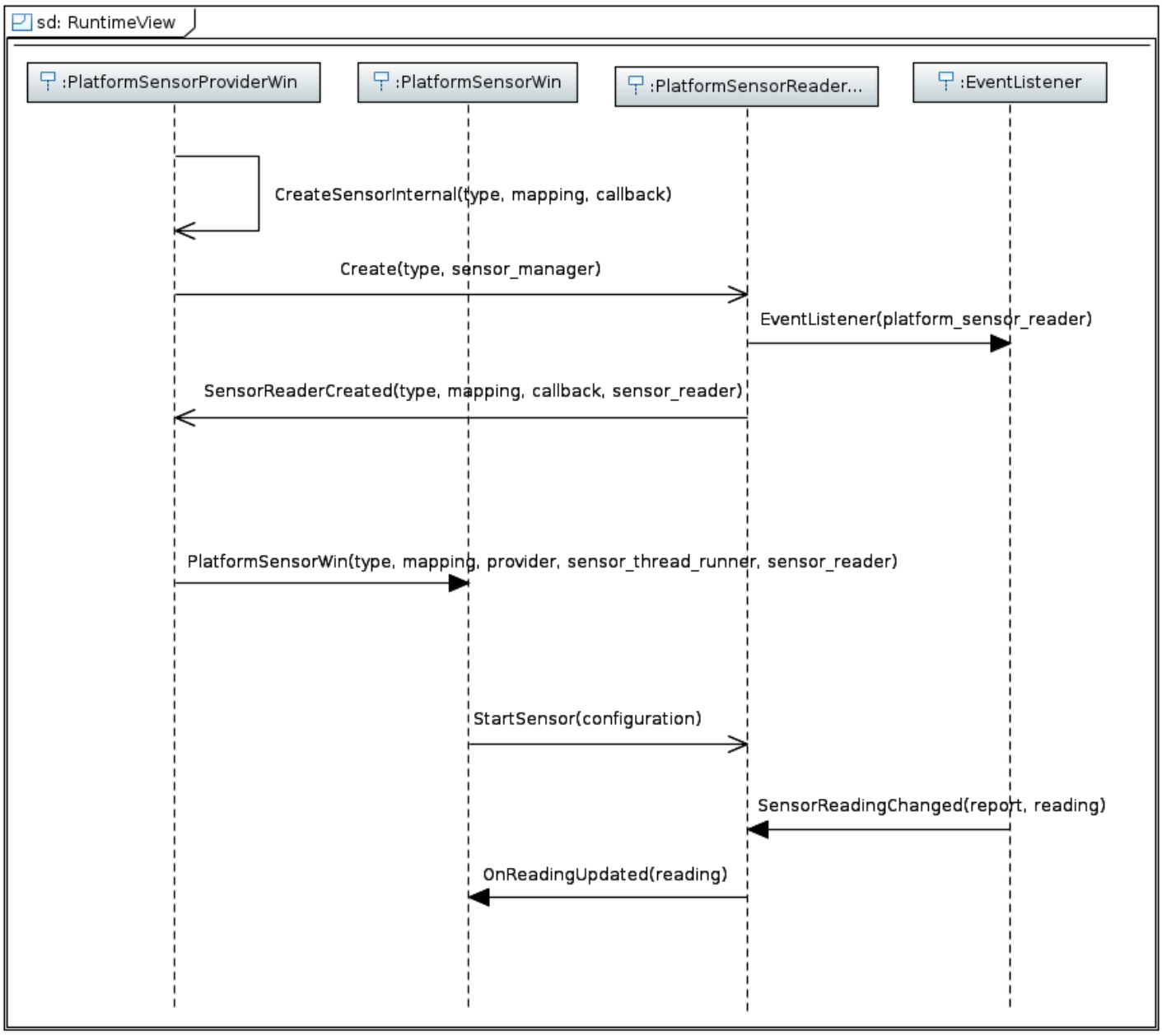
All Windows Sensor API COM objects and `PlatformSensorReaderWin` are created on sensors thread, while `PlatformSensorProviderWin` and `PlatformSensorWin` live on IPC thread and communicate with Generic Sensor API mojo interfaces.

The `PlatformSensorProviderWin` implements `PlatformSensorProvider` interface, it is responsible for creation of `PlatformSensorWin` and `PlatformSensorReaderWin` instances. It also manages COM object `ISensorManager` and sensor thread where all COM objects are created.

The `PlatformSensorReaderWin` communicates with `ISensor` interface to configure it and get readings from `ISensor` COM object. The `EventListener` class implements `ISensorEvents` interface to get notifications about sensor state changes and delivers sensor readings to parent class `PlatformSensorReaderWin` that in turn, forwards sensor readings to `PlatformSensorWin` through `PlatformSensorReaderWin::Client` interface that is implemented by `PlatformSensorWin`.

The `PlatformSensorWin` implements `PlatformSensor` and controls `PlatformSensorReaderWin` state using `StartSensor()` and `StopSensor()` methods. Implements `PlatformSensorReaderWin::Client` interface to receive notifications about new readings or error conditions.

Simplified runtime view.



Implementation on Chrome OS and Linux

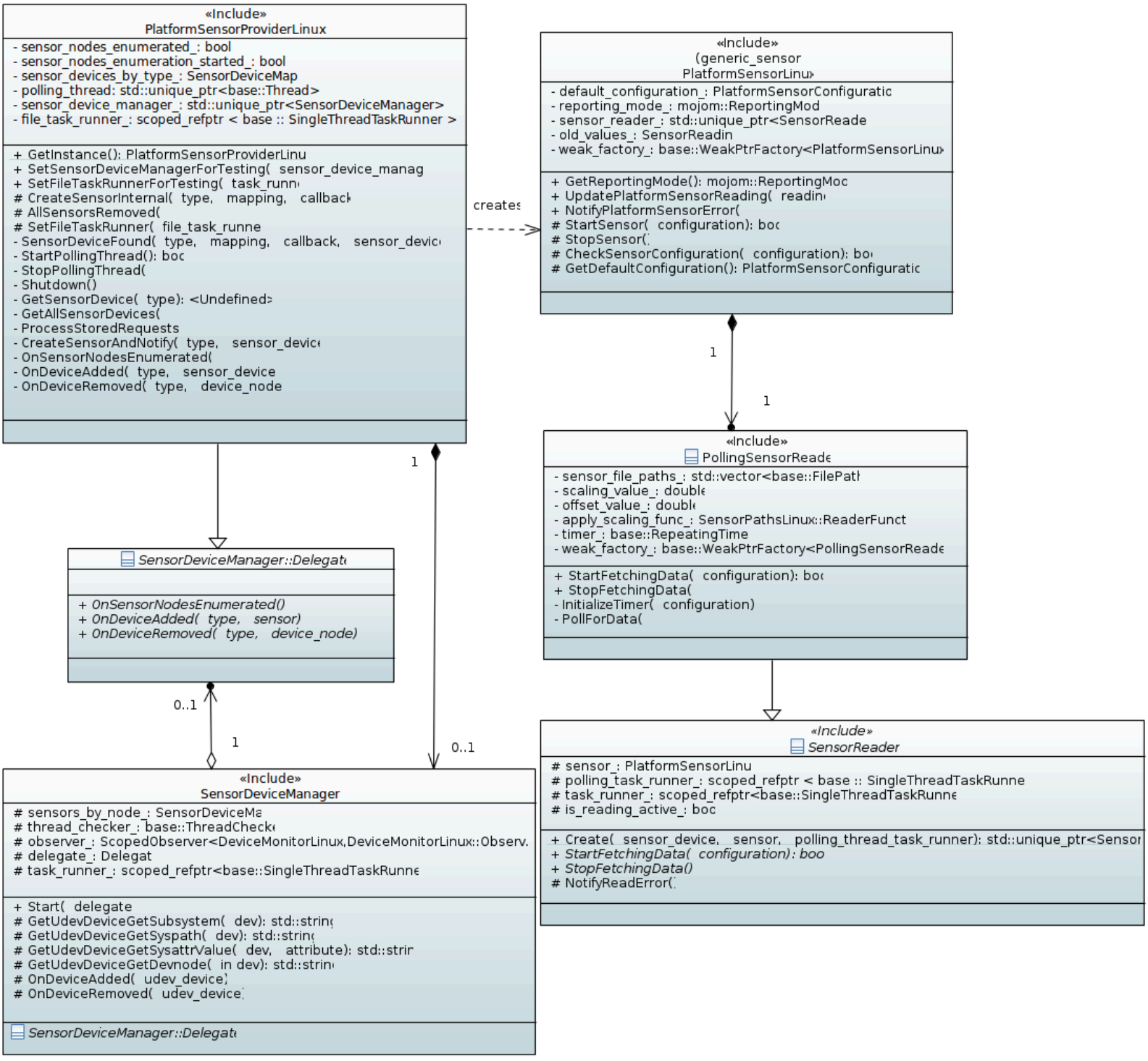
ChromeOS (CrOS) and Linux Operating Systems (OS) share the same code base except for some auxiliary data structures, which are used to read sensor values in a right order and in accordance to Generic Sensor API specifications, and functions, which are used to apply scaling value, offset value and other values to keep readings in single units of measurement.

Sensor data is read using Industrial Input/Output (IIO) APIs. There are two ways to read data:

1. Using sysfs paths.
2. Using device node interface.

Our implementation is using `sysfs` for both CrOS and Linux platforms. The problem with the device node interface is that it requires device nodes to be accessible not only by root but by a user without root rights, who runs Chromium or Chrome browser. While the mentioned problem concerns only Linux distributions, CrOS involves another problem - data cannot be read by multiple clients simultaneously, which is true for Linux platform as well. As far as we know, there is an `AccelerometerReader` that uses this interface to read accelerometer data for CrOS specific components in the browser. To be more precise, the interface uses a ring buffer, whose values are erased after reading happens or buffer is full. If there are two simultaneous clients trying to read from the same buffer, both of them will miss data which will cause performance and reliability issues. There is a new feature in the latest kernel - two and more simultaneous reads can be done for the same device node interface, but this feature is still in staging.

The implementation on CrOS and Linux involves several classes - a `PlatformSensorProviderLinux`, derived from a `PlatformSensorProvider`, a PlatformSensorLinux derived from a `PlatformSensor`, a `SensorDeviceManager` and a `SensorReader`, which is a base class for a `PollingSensorReader`.



CrOS/Linux UML class diagram

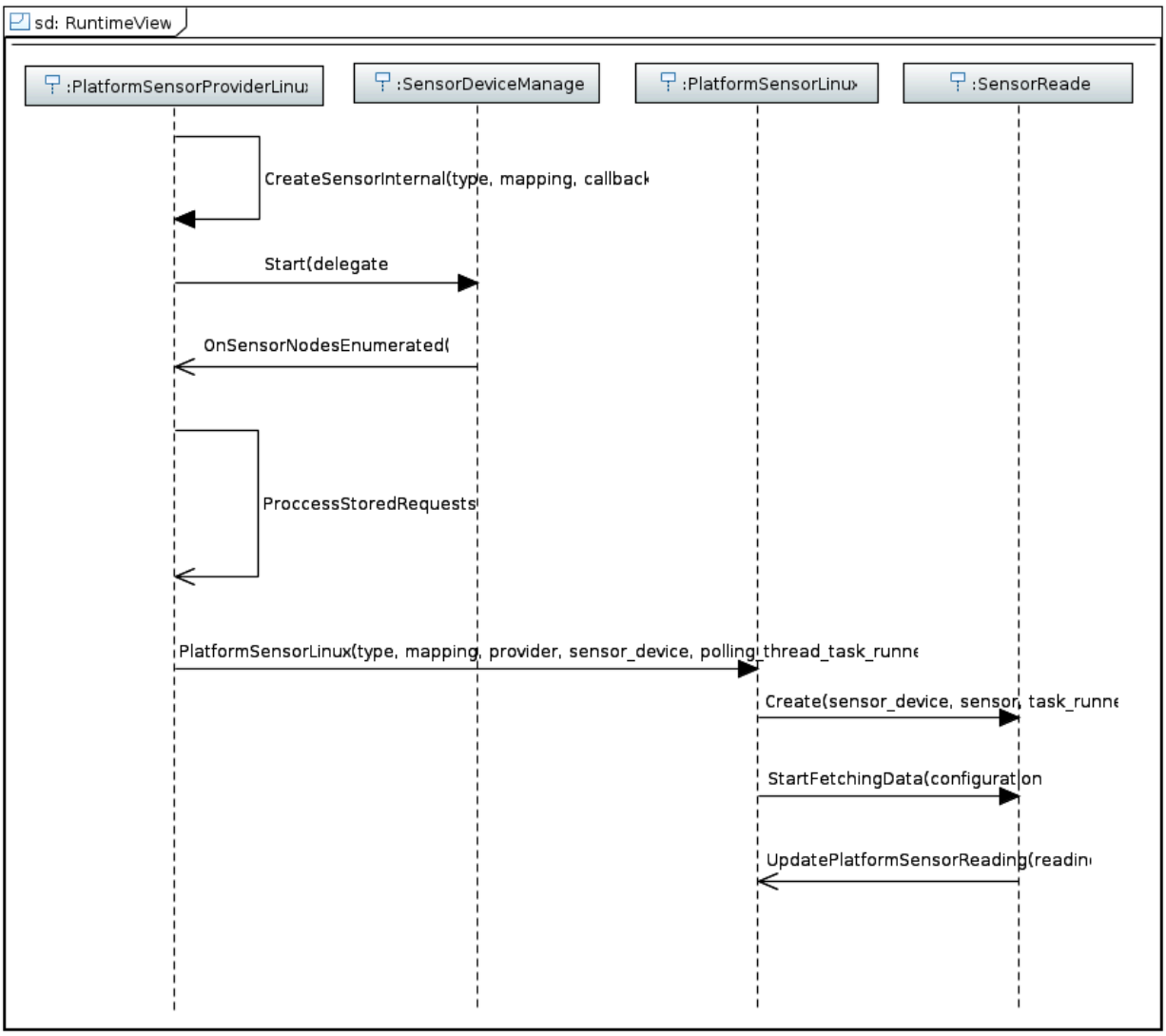
PlatformSensorProviderLinux is a singleton class that processes requests for new sensors. It uses composition and holds a unique pointer to a SensorDeviceManager object, which sends notifications about added and removed iio sensors back to it using a Delegate pattern. The PlatformSensorProviderLinux has a SensorDeviceMap cache, which is an unordered map that stores a pair of a mojom::SensorType key and a std::unique_ptr<SensorInfoLinux> structure, which represents a structure of an existing iio device. The structure's members will be discussed further along with the implementation of the SensorDeviceManager.

The SensorDeviceMap is a cache, which is used to store type to structure pairs of all known IIO sensors provided by an OS and is used to create sensors of requested types. When a request for a specific type of sensor comes (PlatformSensorProviderLinux::CreateSensorInternal is called), the provider checks if the SensorDeviceManager has been started and enumeration has been done. If not, the provider starts the manager and waits until it enumerates all iio devices available to create PlatformSensorLinux sensors asynchronously, otherwise the provider checks if it has a SensorInfoLinux for a specific sensor type in its cache. Then it creates a platform sensor passing SensorInfoLinux to it, which will use the structure to set own frequency, reporting mode and then passes the structure to a SensorReader created by that sensor.

Once enumeration is done, the provider starts to process stored in std::vector<mojom::SensorType> requests according their types by enumerating own cache and looking for a SensorInfoLinux structure that represents the requested type of a sensor.

As previously said, the PlatformSensorLinux creates a SensorReader and passes a SensorInfoLinux structure to it. The reader has a static factory method SensorReader::Create, which creates a PollingSensorReader (we will implement triggered sensor reader, which will use the device node interface in the future once all problems regarding this reading strategy are resolved). The reader uses the SensorInfoLinux structure and stores sysfs paths, which are used to read iio sensor values, offset value, scaling value and a functor - SensorPathsLinux::ReaderFunctor, which is used to apply scaling and offset values and invert signs if needed. A base::RepeatingTimer is used to instantiate readings using a frequency provided by a PlatformSensorLinux.

SensorDeviceManager is a class that uses LinuxDeviceMonitor to enumerate devices, listen to "add/removed" events and then notify the PlatformSensorProviderLinux about added or removed IIO devices. It has own cache to speed up an identification process of removed devices. The LinuxDeviceMonitor is a class that listens for notifications from libudev about connected/disconnected devices. When the manager is started, it adds itself as an observer to the LinuxDeviceMonitor and asks to enumerate devices. During enumeration, the provider gets notifications about found sensors and updates its cache.



Simplified runtime diagram on CrOS/Linux platform

Chrome OS and Linux: threading model

The threading model differs from other Generic Sensor API's platform implementations. Three threads are used to preserve Chromium or Chrome fast and responsive. Those are an I/O browser thread, a browser file thread and a custom polling thread.

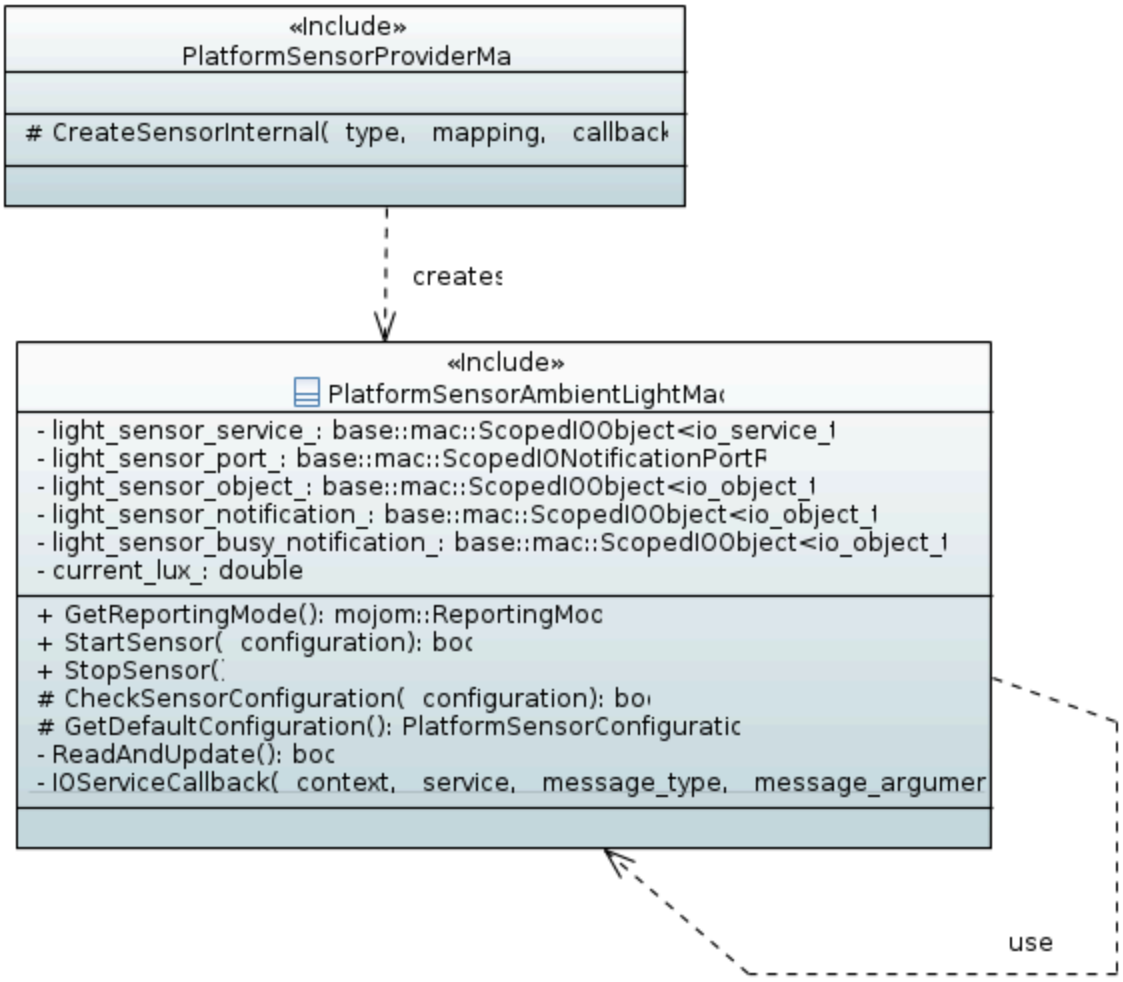
The `PlatformSensorProviderLinux` and `PlatformSensorLinux` use the I/O browser thread and all the communications with another Generic Sensor framework code happens there. The `SensorDeviceManager` uses the browser file thread and communicates with the `PlatformSensorProviderLinux` using I/O thread's task runners. The polling thread is created and owned by the `PlatformSensorProviderLinux` and stopped once there are no sensors left. The provider passes the polling thread's task runner to the `PlatformSensorLinux`, which uses that to communicate with the `SensorReader`. The `SensorReader` is created on a I/O thread, but detached in it's constructor and attached to a custom polling thread once it's methods are called by the sensor.

Implementation on macOS

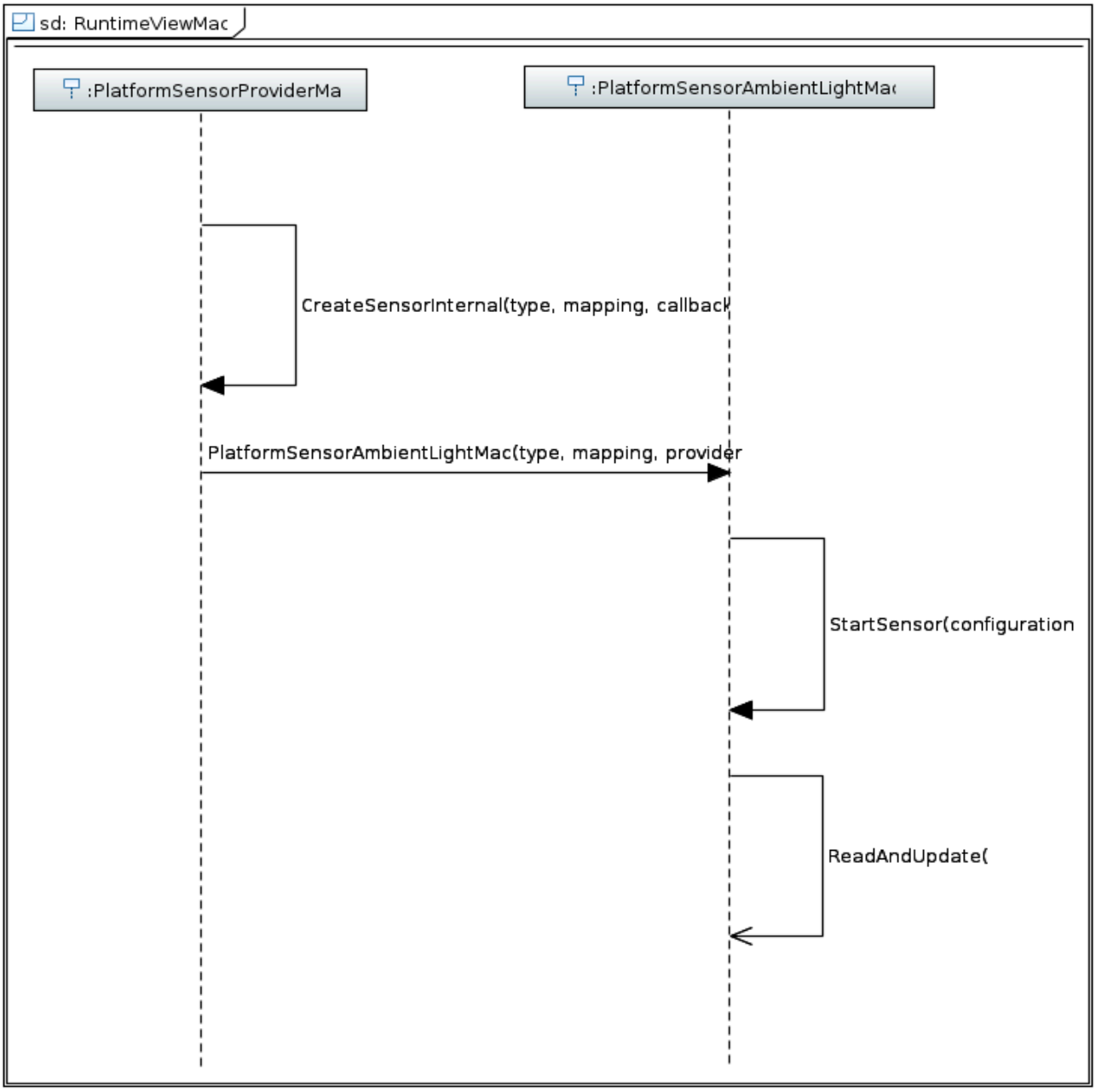
The implementation for Mac platform consists of two classes: `PlatformSensorProviderMac` (singleton) and `PlatformSensorAmbientLightMac`. The reason of a precise naming of the sensor class is that the platform has only ambient light sensors embedded into hardware.

`PlatformSensorProviderMac` implements `PlatformSensorProvider's` interface and is responsible for creation of `PlatformSensorAmbientLightMac` object. Both of them live on I/O browser thread and communicate with the rest Generic Sensor API code using mojo interfaces.

`PlatformSensorAmbientLightMac` utilizes `IOKit` to get information from the platform and callback
when the value of the sensor is changed. In order to get a right callback notification, `IOServiceAddInterestNotification` is used.



Mac UML class diagram



Simplified runtime diagram for Mac platform

Security and Privacy

Generic Sensor APIs can be only accessed by top-level secure browsing contexts. Only focused browsing context is able to access sensor data. When browsing context (tab) is unfocused, sensors that are associated with are stopped to reduce power consumption and avoid exposing sensor data. Generic Sensors API specification addresses security and privacy in chapter 5. Security and Privacy considerations.

Generic Sensor implementation in Chromium is using Permission mojo service to obtain permission from the user.

In order to avoid privacy information leakage, sensors that might expose privacy sensitive data must be protected by permission system and maximum allowed polling frequency should capped to complicate ‘gyrophone’ [1] or keylogger [2,3,4,5] type of attacks.

The ambient light sensor is prone to keylogging attacks [5], therefore, must have separate permission token, so that UA / web page is able to control access to data provided by the sensor. The data could be rounded to its integer part and only illuminance values returned to the users of the API without exposing RGB data. Also, ambient light sensor might be used to track what end-user is watching at the moment on the TV or tell whether user has moved from one room to another.

The accelerometer and gyroscope sensors might be used for keylogger type of attacks [2,3,4] or, for example, identify users by walking patterns, therefore, must be protected by separate permission token.

The magnetometer sensor provides information about magnetic field and in theory, can expose location of a user. For example, attack vector could be pre-magnetized surface in a particular location, or mapping between location and constant magnetic field disturbances caused by the building. Due to non-uniform strength of the Earth’s magnetic field, another attack vector could be exposure / validation of user location. For example, if end-user is connected through VPN, magnetic field associated with GEO IP information can be compared with magnetometer readings at real location, therefore, tell whether user is using VPN or not.

Orientation sensor that provides quaternion or rotation matrix data is a fusion sensor that uses accelerometer, gyroscope and optionally magnetometer. It fuses data from different sources, therefore, it is difficult to reconstruct original data provided by low-level sensors. Research paper [6] indicates that orientation sensors can be used for keylogger type of attacks.

To avoid out-of-band communication between different origins, actual polling frequency is not exposed to JS objects.

Generic Sensor APIs functions the same in incognito and regular windows.

Discussion of past issues discovered in Blink's Device Orientation and Motion APIs is here, we believe they are all addressed by the above mitigations.

Sensor permissions considerations

Given the privacy and security risks described above the most sensitive data is fetched from the low-level sensors (accelerometer and gyroscope in particular), therefore access to these sensor interfaces is better to be protected with permission mechanism.

Proposed security policies

Sensor	Fusion sensor	Access to low level data	Security impact	Proposed security policy
Accelerometer	No	Easy	High	auto-grant + opt-out
LinearAccelerationSensor	Yes	Easy	High	auto-grant + opt-out
GravitySensor	Yes	Difficult	High	auto-grant + opt-out
Gyroscope	No	Easy	High	auto-grant + opt-out
Magnetometer	No	Easy	Medium ?	auto-grant + upt-out UI
AmbientLight, rounded lux	No	Easy	Medium ?	auto-grant + upt-out UI
AbsoluteOrientation	Yes	Difficult	High	auto-grant + opt-out
RelativeOrientation	Yes	Difficult	High	auto-grant + opt-out
GeomagneticOrientation	Yes	Difficult	High	auto-grant + opt-out

Proposed security tokens

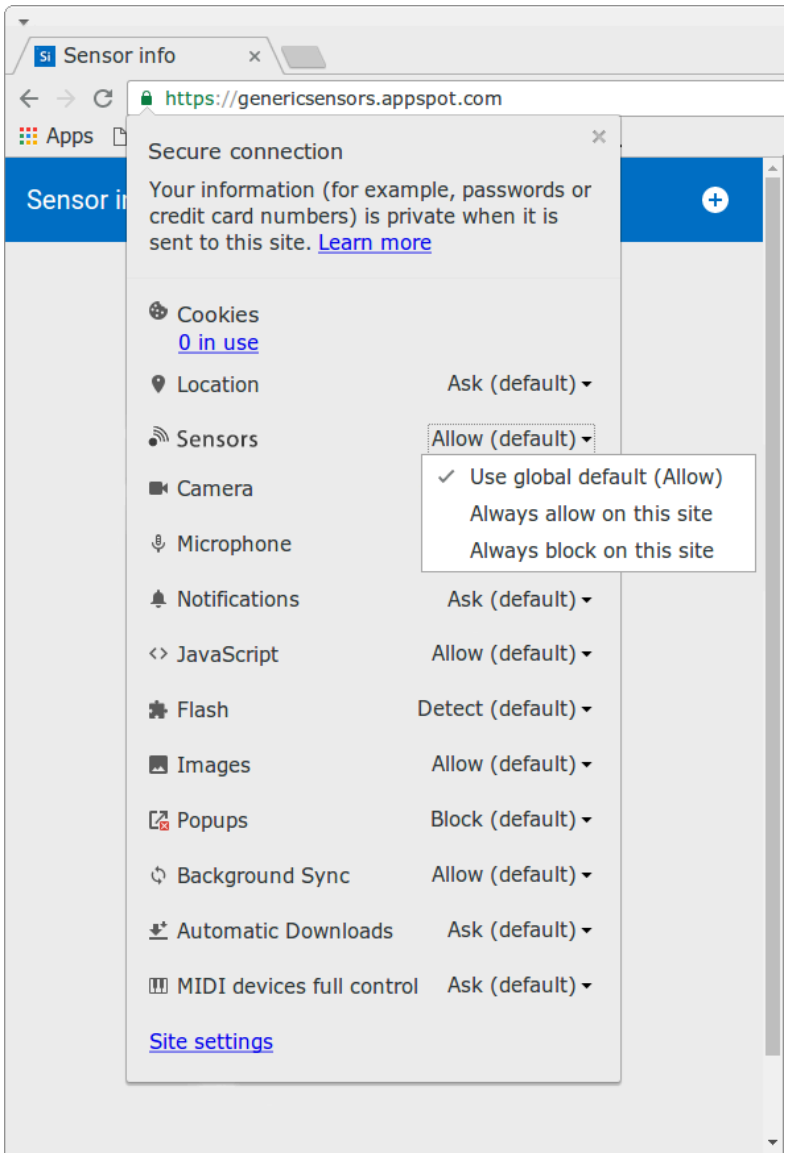
Sensor	Security tokens
Accelerometer	“accelerometer”
LinearAccelerationSensor	“accelerometer”
GravitySensor	“accelerometer”
Gyroscope	“gyroscope”
Magnetometer	“magnetometer”
AmbientLight, rounded lux	“ambient-light-sensor”
AbsoluteOrientationSensor	[“accelerometer”, “gyroscope”, “magnetometer”]
RelativeOrientationSensor	[“accelerometer”, “gyroscope”]
GeomagneticOrientationSensor	[“accelerometer”, “magnetometer”]

Chrome UI

The site settings UI could contain single option entry dedicated for sensors. The user might forbid access, thus disabling sensors that are under ‘opt-out’ group. If the default permission policy for particular sensor is ‘ASK’ web page could request permissions using Permission API, thus, triggering permission dialog. Therefore, localized strings and possibly icons would be required for the UI.

Mock (opt-out) UI

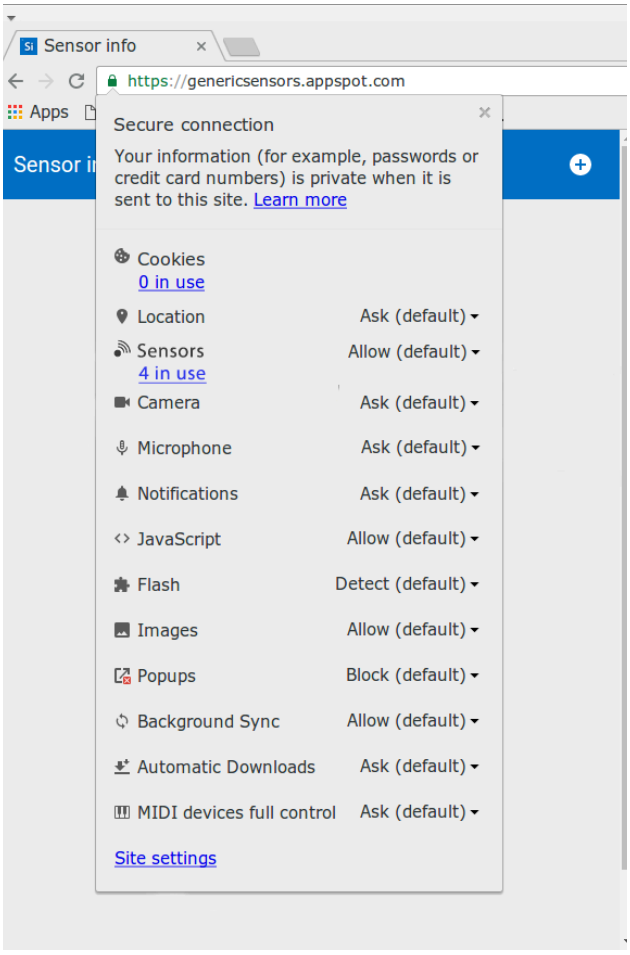
Simple site settings UI

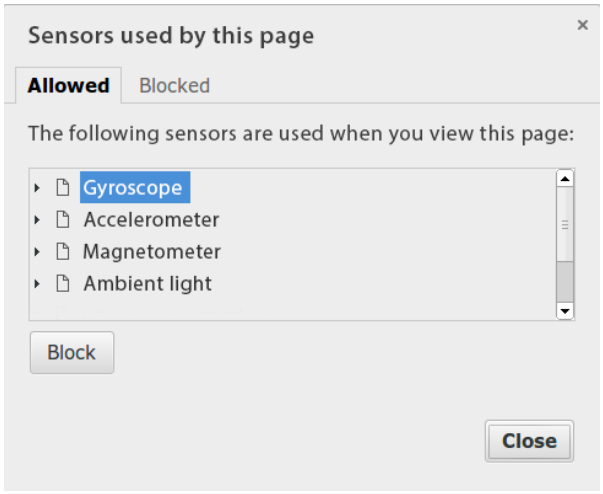


If “always block” is selected by the user, Chrome should (options):

- Block all sensors?
- Block particular set of sensors?

Site settings UI with granular permission settings





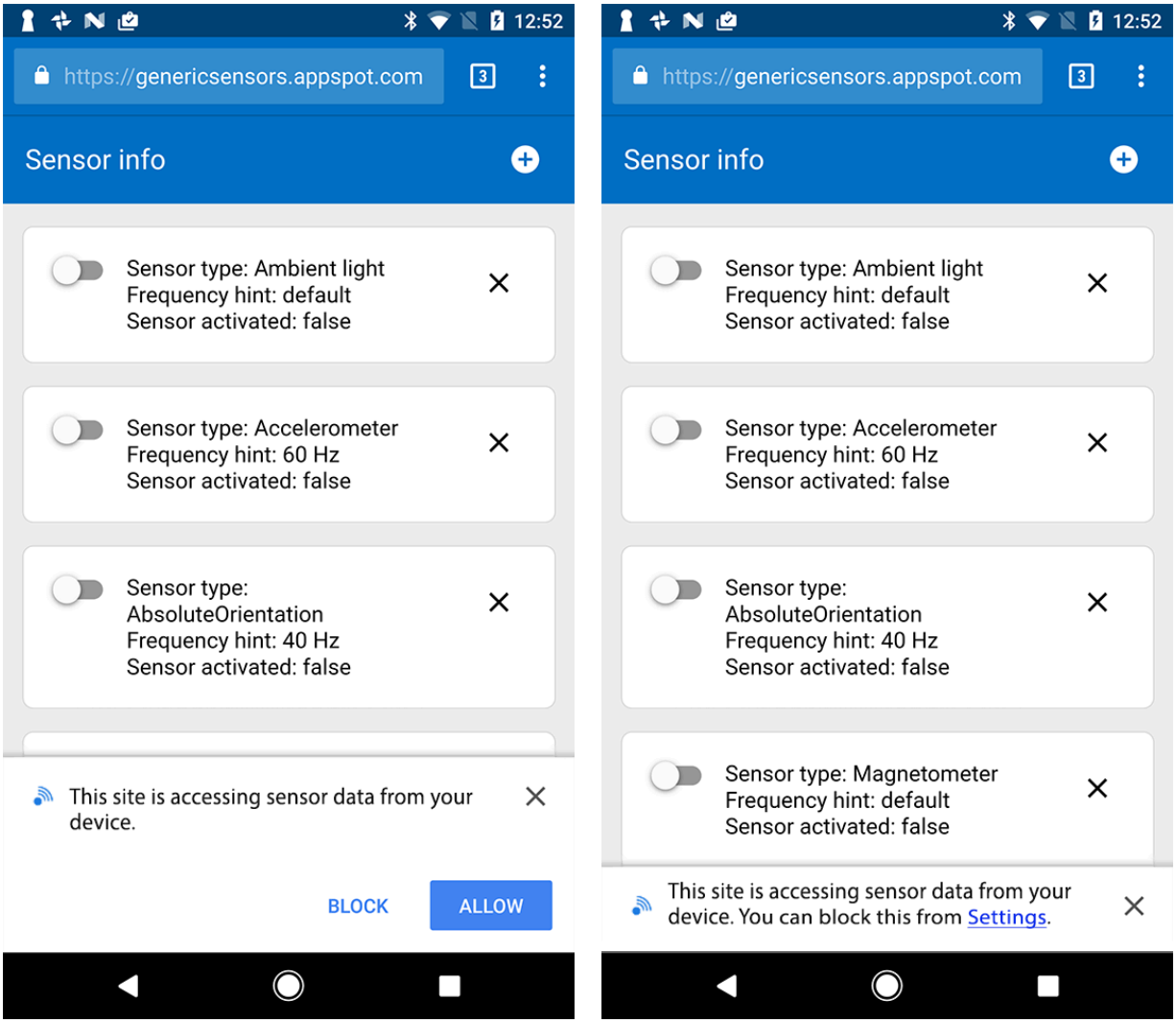
Pros:

- User is able to see what sensors are being used by the web page.
- Block all sensors or particular sensor in “granular” UI, similar to cookies.

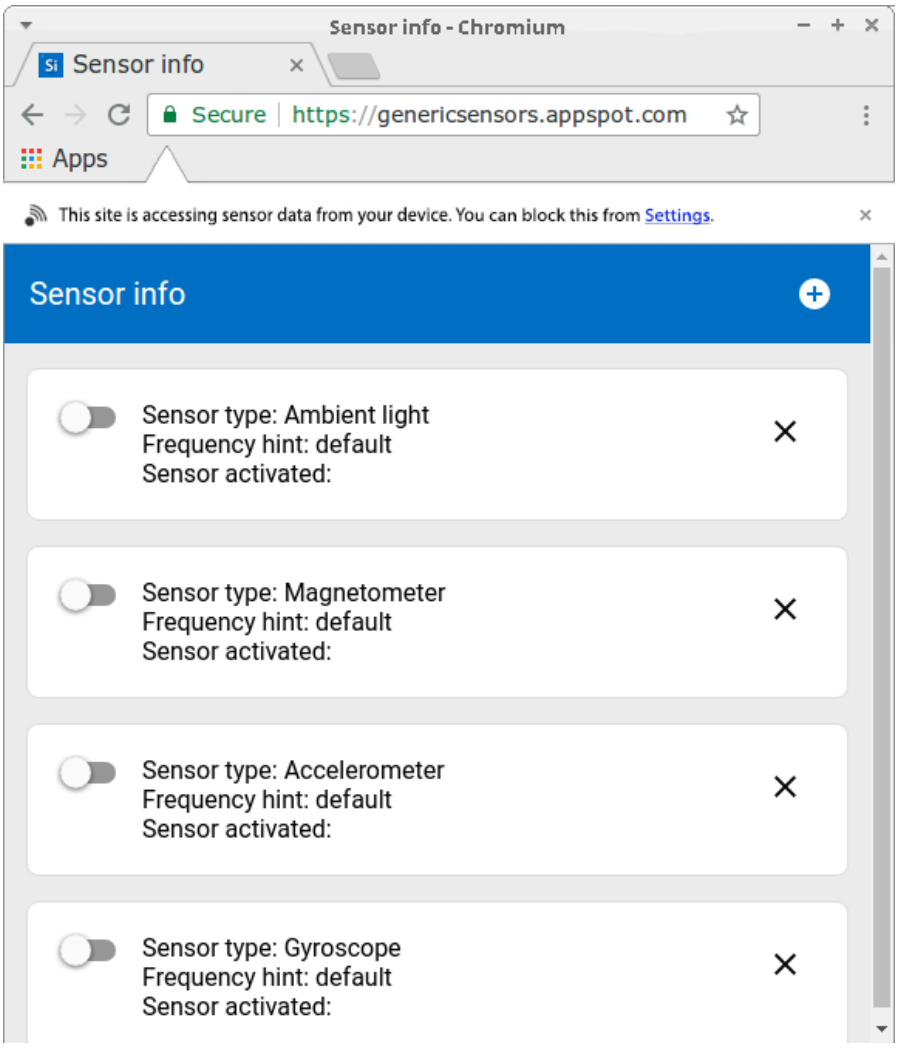
Cons:

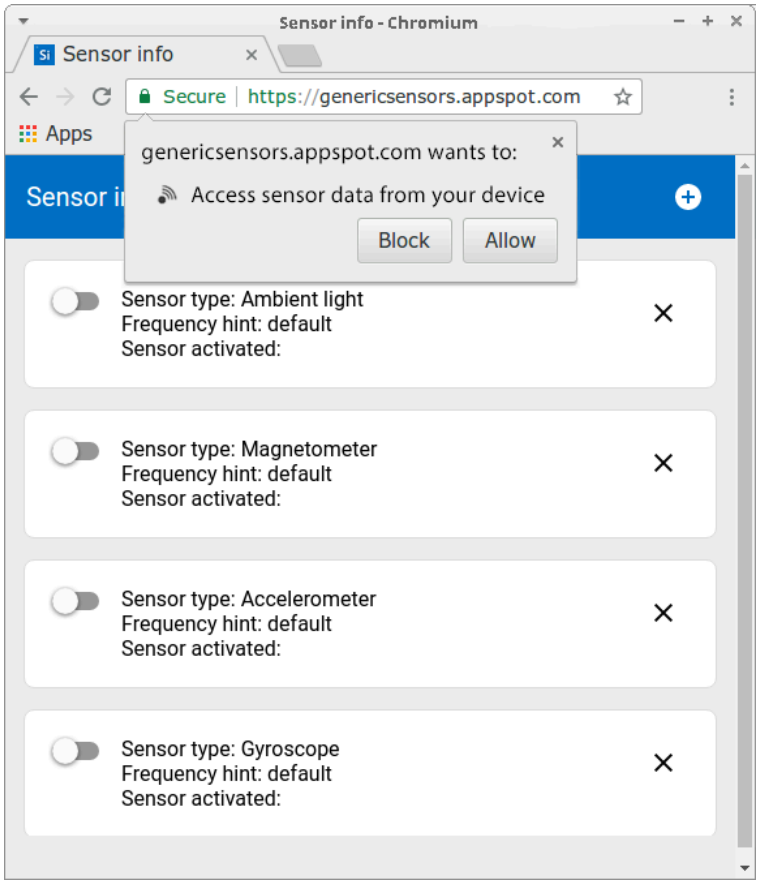
- Unclear whether this can be implemented on mobile devices, e.g., Android.

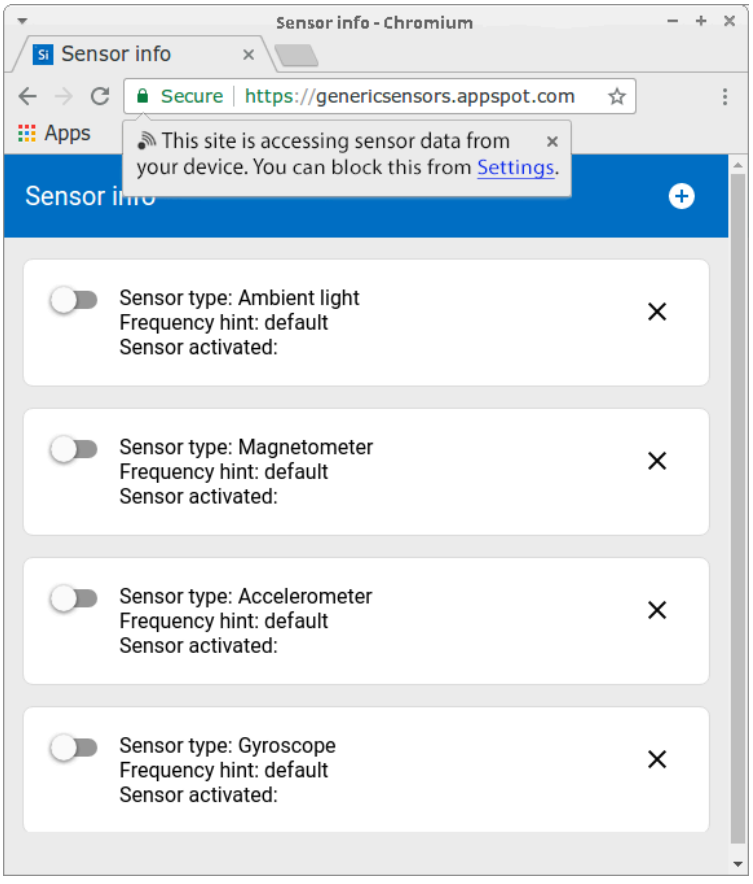
Android infobar (popup)



Desktop infobar (popup)




Sensor info - Chromium
Sensor info
Secure | https://genericsensors.appspot.com
Apps
genericsensors.appspot.com wants to:
Access sensor data from your device
Block
Allow
Sensor type: Ambient light
Frequency hint: default
Sensor activated:
Sensor type: Magnetometer
Frequency hint: default
Sensor activated:
Sensor type: Accelerometer
Frequency hint: default
Sensor activated:
Sensor type: Gyroscope
Frequency hint: default
Sensor activated:


Sensor info - Chromium
Sensor info
Secure | https://genericsensors.appspot.com
Apps
This site is accessing sensor data from your device. You can block this from Settings.
Sensor type: Ambient light
Frequency hint: default
Sensor activated:
Sensor type: Magnetometer
Frequency hint: default
Sensor activated:
Sensor type: Accelerometer
Frequency hint: default
Sensor activated:
Sensor type: Gyroscope
Frequency hint: default
Sensor activated:

References

[1] - Michalevsky, Y., Boneh, D. and Nakibly, G., 2014, August. Gyrophone: Recognizing Speech from Gyroscope Signals. In USENIX Security (pp. 1053-1067).
URL: https://crypto.stanford.edu/gyrophone/files/gyromic.pdf

[2] - Owusu, E., Han, J., Das, S., Perrig, A. and Zhang, J., 2012, February. ACCessory: password inference using accelerometers on smartphones. In Proceedings of the Twelfth Workshop on Mobile Computing Systems & Applications (p. 9). ACM.
URL: https://pdfs.semanticscholar.org/3673/2ae9fbf61f84eab43e60bc2bcb0a48d05b67.pdf

[3] - Mehrnezhad, Maryam, et al. "Touchsignatures: identification of user touch actions and pins based on mobile sensor data via javascript." Journal of Information Security and Applications 26 (2016): 23-38.
URL: https://arxiv.org/pdf/1602.04115.pdf

[4] - Spreitzer, R., Moonsamy, V., Korak, T. and Mangard, S., 2016. SoK: Systematic Classification of Side-Channel Attacks on Mobile Devices. arXiv preprint arXiv:1611.03748.
URL: https://arxiv.org/pdf/1611.03748.pdf

[5] - Spreitzer, Raphael. "Pin skimming: Exploiting the ambient-light sensor in mobile devices." Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices. ACM, 2014.
URL: https://arxiv.org/pdf/1405.3760.pdf

[6] - Xu, Z., Bai, K. and Zhu, S., 2012. TapLogger: Inferring User Inputs On Smartphone Touchscreens Using On-board Motion Sensors.
URL: https://pdfs.semanticscholar.org/c860/4311321f1b8f8fdc8acff8871a5bad2ad4ac.pdf