University of Victoria, Computer Science Department

# Self-Adaptive Systems – Assignment 1 Part III

Tory Borsboom, Sean Debroni, Stephan Heinemann

May 29, 2015

# 1 Part III – Sensor APIs

## 1.1 Android Sensor API

Android devices have a wide variety of sensors available to them, of both the hardware and software variety. For Android, hardware sensors are physical components build into the Android device, while software sensors are built on top of one or more hardware sensors, and interpret their data in a more useful fashion. Sensors for Android devices usually fall into one of these three categories: motion sensors, environmental sensors, and position sensors.

The Sensor Application Programming Interface (API) for Android consists of four main classes and interfaces: `SensorManager`, `Sensor`, `SensorEventListener`, and `SensorEvent`.

The `SensorManager` class is used to access a device's sensors. The following code demonstrates the instantiation of this class:

```
SensorManager sm = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
```

The `Sensor` class represents one of the device's sensors. Each sensor has a unique type, which is used to get an instance of that sensor. For example, to get a gyroscope sensor instance, the following code snippet is used:

```
Sensor s = sm.getDefaultSensor(Sensor.TYPE_GYROSCOPE);
```

`SensorEventListener` is an interface that contains two listener methods to be implemented by the user:

```
onSensorChanged(SensorEvent event)
onAccuracyChanged(Sensor sensor, int accuracy)
```

Assuming an implementation of `SensorEventListener` in a class called `MySensorEventListener`, for the gyroscope sensor, its registration code would look like this:

```
MySensorEventListener msel = new MySensorEventListener();
sm.registerListener(msel, s, SensorManager.SENSOR_DELAY_UI);
```

`SENSOR_DELAY_UI` is one of four different delay settings, measured in microseconds, that controls how often `onSensorChanged` listener method is called. These are, in order of fastest to slowest, `SENSOR_DELAY_FASTEST`, `SENSOR_DELAY_GAME`, `SENSOR_DELAY_NORMAL`, and `SENSOR_DELAY_UI`. This timing is just a hint to the system, events may be received faster or slower than the specified rate. From Android 2.3 onwards, any delay in microseconds can be specified, not just these four constants.

A `SensorEvent` consists of four fields – the accuracy of the event, which sensor generated the event, a timestamp indicating the time in nanoseconds that the event occurred at, and a float array of values, whose length and contents depends upon the type of sensor being monitored. For the gyroscope sensor, this float array contains three values, the angular speed around the $x$, $y$, and $z$ axis' in radians/second.

The Android Sensor API consists of these four classes and interfaces; it does not matter what sensor is being monitored, the only thing that changes is the implementation of `onSensorChanged` and `onAccuracyChanged`. This is a very nice generic interface for Sensors.[1]

## 1.2 iOS Magnetic Compass API

The *iOS*[2] platform used on *Apple*[3] phones and tablets supports a variety of motion services via an object called `CMMotionManager`. This object is the gateway to raw *accelerometer*, *gyroscope* and *magnetometer* data, as well as some processed data such as *rotation rate*, *attitude*, *calibrated magnetic fields*, *direction of gravity*, and the *direction of acceleration* of the device. Of particular interest here, however, is the magnetometer and how the device

[1]All information is sourced from the Android documentation:
developer.android.com/guide/topics/sensors/sensors_overview.html
[2]https://www.apple.com/ca/ios/what-is/
[3]https://www.apple.com

handles the data collected by this sensor.

The magnetometer in the device collects magnetic field data, such as strength and polarity, which is accessible via the `CMMotionManager` object discussed above by either setting an interval at which updates can be pushed, or by polling the sensor whenever an update is required. If requesting updates at intervals the `accelerometerUpdate-Interval` property must be specified and the `startMagnetometerUpdatesToQueue:withHandler:` method is called which requires a `CMMagnetometerHandler` type object to be passed into it. This method will add data to a queue which can then be processed as needed. The alternative to requesting repeated information at intervals is periodic sampling of the data which is performed by calling `startMagnetometerUpdates` and then simply reading the `magnetomerData` property. For both of these methods it is important that the `stopMagnetometerUpdates` method is called when the system no longer needs to process magnetometer data.

In order to access the data acquired from the most recent polling of the sensor, `magnetometerData` must be accessed, this property is a structure of type `CMMagnetometerData`. It contains three `int` values $x$, $y$, and $z$. These `int` values represent the directional magnetic field relative to the body of the device in microteslas. In the coordinate system used, the $z$ axis is positive up through the front of the screen, the $y$ axis is positive up through the top of the phone, and the $x$ axis is positive to the right of the device if the screen is facing the user. There are also two methods made for simply polling the sensor to see if it is working or available: `magnetometerActive` and `magnetometerAvailable`. The method `magnetometerActive` checks to see if the sensor is currently taking readings and `magnetometerAvailable` checks to see if the sensor is available for use.

The purpose of having the two separate collection methods is one of resource management. If the purpose of the app being written is something like a game or some sort of mapping software, it is unnecessary to acquire data rapidly and keep it in a queue. Most programs intended for light users will simply need to query the sensor once per frame rendered on the screen at the very most. The only time it would be necessary to use the queuing method is if the data needs to be incredibly precise and with little latency between updates, such as if the device were being used as a magnetometer in an experiment measuring magnetic flux.

## 1.3 NuttX / ArduPilot PX4 Optical Flow and Sonar API

The *ArduPilot*[4] autopilot stack supports a wide range of sensors to be used with any of the supported vehicle types including *Plane*, *Copter* and *Rover*. It builds on top of the *NuttX*[5] operating system and requires the applicable firmware for the underlying hardware such as the *Pixhawk*[6] autopilot module.

The supported sensor types include those to capture data about the vehicle's *environment* as well as those related to its own *capabilities* in order to establish an adequate *situational awareness.* Supported environmental sensors gather *inertial* (attitude and acceleration), *barometric* (altitude and airspeed), *magnetic* (direction), *range* (distance), *optical* (orientation and recognition) and *satellite* (position) data. Sensors that capture the vehicle's capability state include a *battery monitor* (power consumption and endurance) and a *performance monitor* (scheduler latency). Environmental sensors themselves may provide self-reflective *health* and *quality* monitoring functions. Furthermore, all information that is provided by the *NuttX* operating system about its *Pixhawk* platform including the available *memory* and *filesystem* space can be instrumented to enhance the vehicle's capability state.

The ArduPilot API separates sensor *frontends* (interfaces) from their actual *backends* (implementations) in order to easily support different sensors of the same class. These sensors and other Hardware Abstraction Layer (HAL) modules can be found in the `libraries` directory[7] and are, hence, separate from any concrete vehicle-related software. The *Copter* vehicle employs the sensor API through its `sensors.pde` arduino sketch file which represents an even higher level API[8] to the actual vehicle functions such as different autoflight modes.

[4]http://ardupilot.com/
[5]http://nuttx.org/
[6]https://pixhawk.org
[7]https://github.com/diydrones/ardupilot
[8]http://www.arduino.cc/en/pmwiki.php?n=Reference/HomePage

As a representative example, the structure of the *PX4 Optical Flow / Sonar*[9] sensor interface shall be explained. It is connected to the *Pixhawk* autopilot module as shown in Figure 2.1.
This sensor allows capturing the optical flow, that is, the rate of translational and rotational changes in a captured image, in order to establish positional information. The optical flow camera is mounted on the bottom of the *Iris*[10] quadcopter body pointing downwards and sampling the floor pattern that is being overflown. Since the aircraft can be tilted (pitch and bank) and flown at different altitudes, a range finder (sonar sensor) on the same sensor board is employed to relate the optical flow information to the measured ground distances accordingly. Assuming the actual attitude of the aircraft based on inertial sensors is known, an actual ground distance can be derived from the sonar slant range readings.

Figure 2.2 shows a small extract of the optical flow sensor frontend API in the `libraries/ AP_OpticalFlow` directory possessing the sensor state data structure as well as methods that determine flow and body rates in a two-dimensional image. The interface itself furthermore consists of methods that determine the health of the sensor and the quality of the data being captured by it.

Figure 2.3 shows an API extract of the range finder component of the optical flow sensor board. On the *PX4 Optical Flow* board, a sonar sensor is employed to obtain range in order to relate optical flow to ground distance. This concrete implementation is abstracted by the more generic range finder API in the `libraries/AP_RangeFinder` directory.

The `sensors.pde` arduino sketch file uses the above API in order to provide the *Copter* vehicle with necessary sensor data to perform its functions such as automatic flight modes. The overall static structure is shown in Figure 2.4.

## 1.4 Application Example

A small application example makes use of the *PX4 Optical Flow* board connected to the *Iris Pixhawk* autopilot module. This application realizes a flight envelope warning alert. The alert is triggered whenever translational flow or rotational body rates exceed a certain limit. It is also triggered when the Attitude and Heading Reference System (AHRS)[11] which fuses different sensor inputs for the current height of the aircraft including the sonar distance reports a height above ground that exceeds a certain threshold.

The alert consist a series of high pitch tones whenever the normal envelope is violated and a long low pitch tone as soon as the normal flight envelope is re-established. Furthermore, the aircraft status Light-Emitting Diode (LED) continues to flash red as long as the normal envelope is violated. Snippets of the documented application code can be found in the Appendix 2.

## 1.5 Application in the Cloud

The optical flow sensor can be used to establish position and, hence, to support higher-level autopilot modes such as loitering and following a sequence of legs between waypoints. It can be fused with other sensor data and provide a higher degree of redundancy. The tracks and sensor data of small Unmanned Aerial Vehicles (UAVs) can be shared in the cloud. Such an existing project is called *Droneshare*[12]. Having access to and being able to mine shared mission data enables more efficient planning and execution of coordinated missions that involve multiple UAVs. Examples of coordinated missions include *aerial surveillance*, *crop spraying* and *search and rescue* among many other applications. Sharing position and altitude of UAVs can also be used for enforcing compliance with airspace restrictions.

[9] https://pixhawk.org/modules/px4flow
[10] http://3drobotics.com/iris/
[11] http://en.wikipedia.org/wiki/Attitude_and_heading_reference_system
[12] http://www.droneshare.com/

## 2 FIGURES

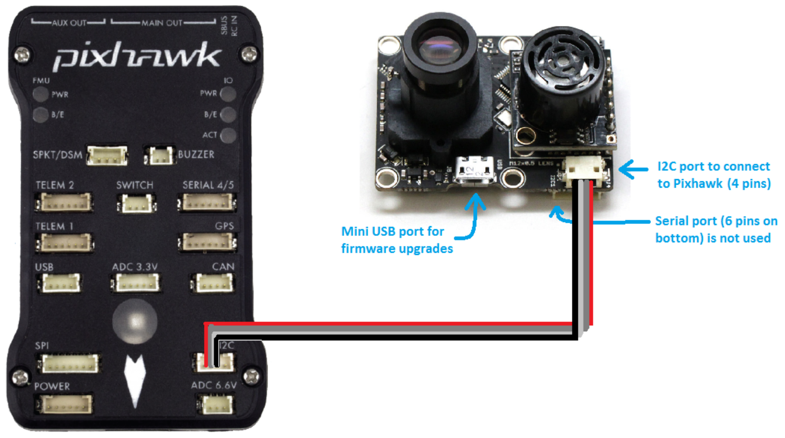
I2C port to connect to Pixhawk (4 pins)
Serial port (6 pins on bottom) is not used
Mini USB port for firmware upgrades

Figure 2.1: Pixhawk PX4 Flow Connection

```
class OpticalFlow { //...
public: //...
        bool enabled() const { return _enabled; }
        bool healthy() const { return backend != NULL && _flags.healthy; }
        uint8_t quality() const { return _state.surface_quality; }
        const Vector2f& flowRate() const { return _state.flowRate; }
        const Vector2f& bodyRate() const { return _state.bodyRate; }
        uint8_t device_id() const { return _state.device_id; }
    //...
    struct OpticalFlow_state {
        uint8_t device_id;
        uint8_t  surface_quality;
        Vector2f flowRate;
        Vector2f bodyRate;
    };
private: //...
};
```

Figure 2.2: Optical Flow Sensor API

```
class RangeFinder {
public: //...
        enum RangeFinder_Status {
                RangeFinder_NotConnected = 0,
                RangeFinder_NoData,
                RangeFinder_OutOfRangeLow,
                RangeFinder_OutOfRangeHigh,
                RangeFinder_Good
        };

        struct RangeFinder_State {
                uint8_t instance;
                uint16_t distance_cm;
                uint16_t voltage_mv;
                enum RangeFinder_Status status;
                uint8_t range_valid_count;
                bool pre_arm_check;
                uint16_t pre_arm_distance_min;
                uint16_t pre_arm_distance_max;
        };
        //...
        uint16_t distance_cm() const { return distance_cm(primary_instance); }
        uint16_t voltage_mv() const { return voltage_mv(primary_instance); }
        int16_t ground_clearance_cm() const { return _ground_clearance_cm[primary_instance]; }
        RangeFinder_Status status(void) const { return status(primary_instance);
        //...
private: //...
};
```

Figure 2.3: Range Finder API

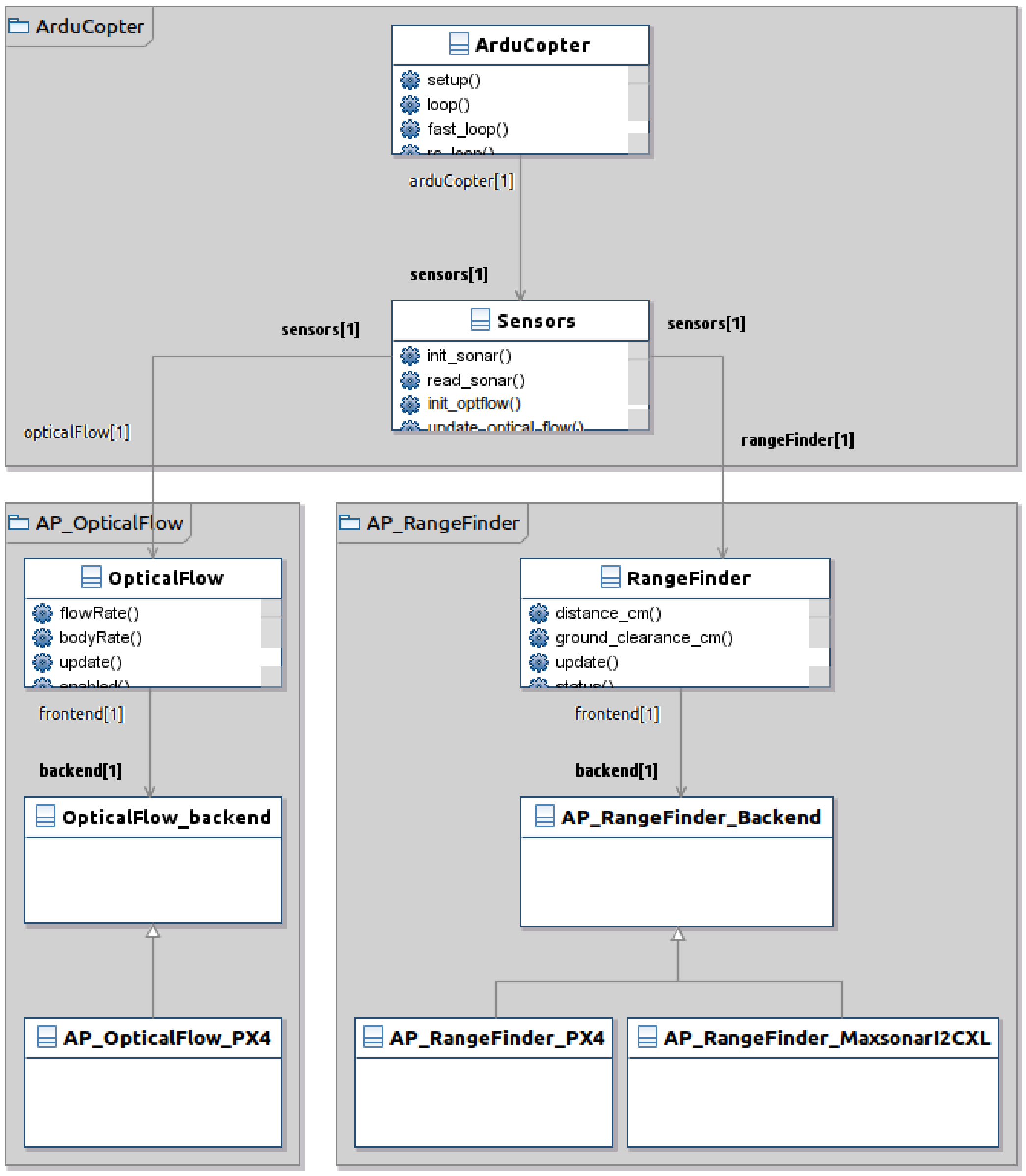

Figure 2.4: PX4 Optical Flow Sensor Board API

```
static void stabilize_run()
{
    //...
    // in the stabilizing flight mode check the current flight envelope
    // measured by the optical flow sensor
    checkEnvelope();
    //...
}

#define BODY_RATE_ENVELOPE 4.0f
#define FLOW_RATE_ENVELOPE 1.0f
#define MAX_ALT 1.0f

static bool checkBodyRateEnvelope() {
        Vector2f bodyRate = optflow.bodyRate();
        return ((bodyRate.x > -BODY_RATE_ENVELOPE) && (bodyRate.y < BODY_RATE_ENVELOPE));
}

static bool checkFlowRateEnvelope() {
        Vector2f flowRate = optflow.flowRate();
        return ((flowRate.x > -FLOW_RATE_ENVELOPE) && (flowRate.y < FLOW_RATE_ENVELOPE));
}

static bool checkAltitudeEnvelope() {
        float hagl = 0.0f;
        ahrs.get_NavEKF().getHAGL(hagl);
        return (hagl <= MAX_ALT);
}

static void checkEnvelope() {
        // if the flight envelope is violated, then notify an envelope alert
        // notification subscribers include the tone alert and LED
        if (!checkBodyRateEnvelope() || !checkFlowRateEnvelope() || !checkAltitudeEnvelope()) {
                hal.console->printf("envelope␣violated\n");
                AP_Notify::flags.envelope_alert = true;
        } else {
                hal.console->printf("envelope␣satisfied\n");
                AP_Notify::flags.envelope_alert = false;
        }
}
```

Figure 2.5: Application Snippets

## 3 Acronyms

**AHRS** Attitude and Heading Reference System
An attitude and heading reference system consists of sensors on three axes that provide attitude information for aircraft, including heading, pitch and yaw.They are designed to replace traditional mechanical gyroscopic flight instruments and provide superior reliability and accuracy. [13]

**API** Application Programming Interface
In computer programming, an application programming interface is a set of routines, protocols, and tools for building software applications. An API expresses a software component in terms of its operations, inputs, outputs, and underlying types. An API defines functionalities that are independent of their respective implementations, which allows definitions and implementations to vary without compromising each other. [14]

**HAL** Hardware Abstraction Layer
Hardware abstractions are sets of routines in software that emulate some platform-specific details, giving programs direct access to the hardware resources. [15]

**LED** Light-Emitting Diode A light-emitting diode is a two-lead semiconductor light source. It is a pn-junction diode, which emits light when activated. When a suitable voltage is applied to the leads, electrons are able to recombine with electron holes within the device, releasing energy in the form of photons. [16]

**UAV** Unmanned Aerial Vehicle
An unmanned aerial vehicle, known in the mainstream as a drone and also referred to as an unpiloted aerial vehicle and a remotely piloted aircraft by the International Civil Aviation Organization, is an aircraft without a human pilot aboard. [17]

## 4 Group 12 Members

seng480a-tory-borsboom-v00807557
csc485a-sean-debroni-v00754822
csc586a-stephan-heinemann-v00814821

[13]http://en.wikipedia.org/wiki/Attitude_and_heading_reference_system
[14]http://en.wikipedia.org/wiki/Application_programming_interface
[15]http://en.wikipedia.org/wiki/Hardware_abstraction
[16]http://en.wikipedia.org/wiki/Light-emitting_diode
[17]http://en.wikipedia.org/wiki/Unmanned_aerial_vehicle