SensorManager.java

blob: f05397669b34d196e87a2d36e77456685245a8b2 [file] [log] [blame]

```java
/*
 * Copyright (C) 2008 The Android Open Source Project
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package android.hardware;

import android.annotation.Nullable;
import android.annotation.SystemApi;
import android.annotation.SystemService;
import android.compat.annotation.UnsupportedAppUsage;
import android.content.Context;
import android.os.Build;
import android.os.Handler;
import android.os.MemoryFile;
import android.util.Log;
import android.util.SparseArray;
```

```java
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

/**
 * <p>
 * SensorManager lets you access the device's {@link android.hardware.Sensor
 * sensors}.
 * </p>
 * <p>
 * Always make sure to disable sensors you don't need, especially when your
 * activity is paused. Failing to do so can drain the battery in just a few
 * hours. Note that the system will <i>not</i> disable sensors automatically when
 * the screen turns off.
 * </p>
 * <p class="note">
 * Note: Don't use this mechanism with a Trigger Sensor, have a look
 * at {@link TriggerEventListener}. {@link Sensor#TYPE_SIGNIFICANT_MOTION}
 * is an example of a trigger sensor.
 * </p>
 * <p>
 * In order to access sensor data at high sampling rates (i.e. greater than 200 Hz
 * for {@link SensorEventListener} and greater than {@link SensorDirectChannel#RATE_NORMAL}
 * for {@link SensorDirectChannel}), apps must declare
 * the {@link android.Manifest.permission#HIGH_SAMPLING_RATE_SENSORS} permission
 * in their AndroidManifest.xml file.
 * </p>
 * <pre class="prettyprint">
 * public class SensorActivity extends Activity implements SensorEventListener {
 *     private final SensorManager mSensorManager;
 *     private final Sensor mAccelerometer;
 *
 *     public SensorActivity() {
 *         mSensorManager = (SensorManager)getSystemService(SENSOR_SERVICE);
 *         mAccelerometer = mSensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
 *     }
```

```
 *
 *     protected void onResume() {
 *         super.onResume();
 *         mSensorManager.registerListener(this, mAccelerometer, SensorManager.SENSOR_DELAY_NORMAL);
 *     }
 *
 *     protected void onPause() {
 *         super.onPause();
 *         mSensorManager.unregisterListener(this);
 *     }
 *
 *     public void onAccuracyChanged(Sensor sensor, int accuracy) {
 *     }
 *
 *     public void onSensorChanged(SensorEvent event) {
 *     }
 * }
 * </pre>
 *
 * @see SensorEventListener
 * @see SensorEvent
 * @see Sensor
 *
 */
@SystemService(Context.SENSOR_SERVICE)
public abstract class SensorManager {
    /** @hide */
    protected static final String TAG = "SensorManager";

    private static final float[] sTempMatrix = new float[16];

    // Cached lists of sensors by type.  Guarded by mSensorListByType.
    private final SparseArray<List<Sensor>> mSensorListByType =
            new SparseArray<List<Sensor>>();

    // Legacy sensor manager implementation.  Guarded by mSensorListByType during initialization.
    private LegacySensorManager mLegacySensorManager;
```

```java

    /* NOTE: sensor IDs must be a power of 2 */

    /**
     * A constant describing an orientation sensor. See
     * {@link android.hardware.SensorListener SensorListener} for more details.
     *
     * @deprecated use {@link android.hardware.Sensor Sensor} instead.
     */
    @Deprecated
    public static final int SENSOR_ORIENTATION = 1 << 0;

    /**
     * A constant describing an accelerometer. See
     * {@link android.hardware.SensorListener SensorListener} for more details.
     *
     * @deprecated use {@link android.hardware.Sensor Sensor} instead.
     */
    @Deprecated
    public static final int SENSOR_ACCELEROMETER = 1 << 1;

    /**
     * A constant describing a temperature sensor See
     * {@link android.hardware.SensorListener SensorListener} for more details.
     *
     * @deprecated use {@link android.hardware.Sensor Sensor} instead.
     */
    @Deprecated
    public static final int SENSOR_TEMPERATURE = 1 << 2;

    /**
     * A constant describing a magnetic sensor See
     * {@link android.hardware.SensorListener SensorListener} for more details.
     *
     * @deprecated use {@link android.hardware.Sensor Sensor} instead.
     */
    @Deprecated
```

```java
    public static final int SENSOR_MAGNETIC_FIELD = 1 << 3;

    /**
     * A constant describing an ambient light sensor See
     * {@link android.hardware.SensorListener SensorListener} for more details.
     *
     * @deprecated use {@link android.hardware.Sensor Sensor} instead.
     */
    @Deprecated
    public static final int SENSOR_LIGHT = 1 << 4;

    /**
     * A constant describing a proximity sensor See
     * {@link android.hardware.SensorListener SensorListener} for more details.
     *
     * @deprecated use {@link android.hardware.Sensor Sensor} instead.
     */
    @Deprecated
    public static final int SENSOR_PROXIMITY = 1 << 5;

    /**
     * A constant describing a Tricorder See
     * {@link android.hardware.SensorListener SensorListener} for more details.
     *
     * @deprecated use {@link android.hardware.Sensor Sensor} instead.
     */
    @Deprecated
    public static final int SENSOR_TRICORDER = 1 << 6;

    /**
     * A constant describing an orientation sensor. See
     * {@link android.hardware.SensorListener SensorListener} for more details.
     *
     * @deprecated use {@link android.hardware.Sensor Sensor} instead.
     */
    @Deprecated
    public static final int SENSOR_ORIENTATION_RAW = 1 << 7;
```

```java

    /**
     * A constant that includes all sensors
     *
     * @deprecated use {@link android.hardware.Sensor Sensor} instead.
     */
    @Deprecated
    public static final int SENSOR_ALL = 0x7F;

    /**
     * Smallest sensor ID
     *
     * @deprecated use {@link android.hardware.Sensor Sensor} instead.
     */
    @Deprecated
    public static final int SENSOR_MIN = SENSOR_ORIENTATION;

    /**
     * Largest sensor ID
     *
     * @deprecated use {@link android.hardware.Sensor Sensor} instead.
     */
    @Deprecated
    public static final int SENSOR_MAX = ((SENSOR_ALL + 1) >> 1);


    /**
     * Index of the X value in the array returned by
     * {@link android.hardware.SensorListener#onSensorChanged}
     *
     * @deprecated use {@link android.hardware.Sensor Sensor} instead.
     */
    @Deprecated
    public static final int DATA_X = 0;

    /**
     * Index of the Y value in the array returned by
```

```
     * {@link android.hardware.SensorListener#onSensorChanged}
     *
     * @deprecated use {@link android.hardware.Sensor Sensor} instead.
     */
    @Deprecated
    public static final int DATA_Y = 1;

    /**
     * Index of the Z value in the array returned by
     * {@link android.hardware.SensorListener#onSensorChanged}
     *
     * @deprecated use {@link android.hardware.Sensor Sensor} instead.
     */
    @Deprecated
    public static final int DATA_Z = 2;

    /**
     * Offset to the untransformed values in the array returned by
     * {@link android.hardware.SensorListener#onSensorChanged}
     *
     * @deprecated use {@link android.hardware.Sensor Sensor} instead.
     */
    @Deprecated
    public static final int RAW_DATA_INDEX = 3;

    /**
     * Index of the untransformed X value in the array returned by
     * {@link android.hardware.SensorListener#onSensorChanged}
     *
     * @deprecated use {@link android.hardware.Sensor Sensor} instead.
     */
    @Deprecated
    public static final int RAW_DATA_X = 3;

    /**
     * Index of the untransformed Y value in the array returned by
     * {@link android.hardware.SensorListener#onSensorChanged}
```

```java
     *
     * @deprecated use {@link android.hardware.Sensor Sensor} instead.
     */
    @Deprecated
    public static final int RAW_DATA_Y = 4;

    /**
     * Index of the untransformed Z value in the array returned by
     * {@link android.hardware.SensorListener#onSensorChanged}
     *
     * @deprecated use {@link android.hardware.Sensor Sensor} instead.
     */
    @Deprecated
    public static final int RAW_DATA_Z = 5;

    /** Standard gravity (g) on Earth. This value is equivalent to 1G */
    public static final float STANDARD_GRAVITY = 9.80665f;

    /** Sun's gravity in SI units (m/s^2) */
    public static final float GRAVITY_SUN             = 275.0f;
    /** Mercury's gravity in SI units (m/s^2) */
    public static final float GRAVITY_MERCURY         = 3.70f;
    /** Venus' gravity in SI units (m/s^2) */
    public static final float GRAVITY_VENUS           = 8.87f;
    /** Earth's gravity in SI units (m/s^2) */
    public static final float GRAVITY_EARTH           = 9.80665f;
    /** The Moon's gravity in SI units (m/s^2) */
    public static final float GRAVITY_MOON            = 1.6f;
    /** Mars' gravity in SI units (m/s^2) */
    public static final float GRAVITY_MARS            = 3.71f;
    /** Jupiter's gravity in SI units (m/s^2) */
    public static final float GRAVITY_JUPITER         = 23.12f;
    /** Saturn's gravity in SI units (m/s^2) */
    public static final float GRAVITY_SATURN          = 8.96f;
    /** Uranus' gravity in SI units (m/s^2) */
    public static final float GRAVITY_URANUS          = 8.69f;
    /** Neptune's gravity in SI units (m/s^2) */
```

```java
    public static final float GRAVITY_NEPTUNE         = 11.0f;
    /** Pluto's gravity in SI units (m/s^2) */
    public static final float GRAVITY_PLUTO           = 0.6f;
    /** Gravity (estimate) on the first Death Star in Empire units (m/s^2) */
    public static final float GRAVITY_DEATH_STAR_I    = 0.000000353036145f;
    /** Gravity on the island */
    public static final float GRAVITY_THE_ISLAND      = 4.815162342f;


    /** Maximum magnetic field on Earth's surface */
    public static final float MAGNETIC_FIELD_EARTH_MAX = 60.0f;
    /** Minimum magnetic field on Earth's surface */
    public static final float MAGNETIC_FIELD_EARTH_MIN = 30.0f;


    /** Standard atmosphere, or average sea-level pressure in hPa (millibar) */
    public static final float PRESSURE_STANDARD_ATMOSPHERE = 1013.25f;


    /** Maximum luminance of sunlight in lux */
    public static final float LIGHT_SUNLIGHT_MAX = 120000.0f;
    /** luminance of sunlight in lux */
    public static final float LIGHT_SUNLIGHT     = 110000.0f;
    /** luminance in shade in lux */
    public static final float LIGHT_SHADE        = 20000.0f;
    /** luminance under an overcast sky in lux */
    public static final float LIGHT_OVERCAST     = 10000.0f;
    /** luminance at sunrise in lux */
    public static final float LIGHT_SUNRISE      = 400.0f;
    /** luminance under a cloudy sky in lux */
    public static final float LIGHT_CLOUDY       = 100.0f;
    /** luminance at night with full moon in lux */
    public static final float LIGHT_FULLMOON     = 0.25f;
    /** luminance at night with no moon in lux*/
    public static final float LIGHT_NO_MOON      = 0.001f;
```

```java
    /** get sensor data as fast as possible */
    public static final int SENSOR_DELAY_FASTEST = 0;
    /** rate suitable for games */
    public static final int SENSOR_DELAY_GAME = 1;
    /** rate suitable for the user interface  */
    public static final int SENSOR_DELAY_UI = 2;
    /** rate (default) suitable for screen orientation changes */
    public static final int SENSOR_DELAY_NORMAL = 3;


    /**
      * The values returned by this sensor cannot be trusted because the sensor
      * had no contact with what it was measuring (for example, the heart rate
      * monitor is not in contact with the user).
      */
    public static final int SENSOR_STATUS_NO_CONTACT = -1;

    /**
     * The values returned by this sensor cannot be trusted, calibration is
     * needed or the environment doesn't allow readings
     */
    public static final int SENSOR_STATUS_UNRELIABLE = 0;

    /**
     * This sensor is reporting data with low accuracy, calibration with the
     * environment is needed
     */
    public static final int SENSOR_STATUS_ACCURACY_LOW = 1;

    /**
     * This sensor is reporting data with an average level of accuracy,
     * calibration with the environment may improve the readings
     */
    public static final int SENSOR_STATUS_ACCURACY_MEDIUM = 2;

    /** This sensor is reporting data with maximum accuracy */
    public static final int SENSOR_STATUS_ACCURACY_HIGH = 3;
```

```java

    /** see {@link #remapCoordinateSystem} */
    public static final int AXIS_X = 1;
    /** see {@link #remapCoordinateSystem} */
    public static final int AXIS_Y = 2;
    /** see {@link #remapCoordinateSystem} */
    public static final int AXIS_Z = 3;
    /** see {@link #remapCoordinateSystem} */
    public static final int AXIS_MINUS_X = AXIS_X | 0x80;
    /** see {@link #remapCoordinateSystem} */
    public static final int AXIS_MINUS_Y = AXIS_Y | 0x80;
    /** see {@link #remapCoordinateSystem} */
    public static final int AXIS_MINUS_Z = AXIS_Z | 0x80;


    /**
     * {@hide}
     */
    @UnsupportedAppUsage
    public SensorManager() {
    }

    /**
     * Gets the full list of sensors that are available.
     * @hide
     */
    protected abstract List<Sensor> getFullSensorList();

    /**
     * Gets the full list of dynamic sensors that are available.
     * @hide
     */
    protected abstract List<Sensor> getFullDynamicSensorList();

    /**
     * @return available sensors.
     * @deprecated This method is deprecated, use
```

```
     *             {@link SensorManager#getSensorList(int)} instead
     */
    @Deprecated
    public int getSensors() {
        return getLegacySensorManager().getSensors();
    }

    /**
     * Use this method to get the list of available sensors of a certain type.
     * Make multiple calls to get sensors of different types or use
     * {@link android.hardware.Sensor#TYPE_ALL Sensor.TYPE_ALL} to get all the
     * sensors. Note that the {@link android.hardware.Sensor#getName()} is
     * expected to yield a value that is unique across any sensors that return
     * the same value for {@link android.hardware.Sensor#getType()}.
     *
     * <p class="note">
     * NOTE: Both wake-up and non wake-up sensors matching the given type are
     * returned. Check {@link Sensor#isWakeUpSensor()} to know the wake-up properties
     * of the returned {@link Sensor}.
     * </p>
     *
     * @param type
     *        of sensors requested
     *
     * @return a list of sensors matching the asked type.
     *
     * @see #getDefaultSensor(int)
     * @see Sensor
     */
    public List<Sensor> getSensorList(int type) {
        // cache the returned lists the first time
        List<Sensor> list;
        final List<Sensor> fullList = getFullSensorList();
        synchronized (mSensorListByType) {
            list = mSensorListByType.get(type);
            if (list == null) {
                if (type == Sensor.TYPE_ALL) {
```

```java
                    list = fullList;
                } else {
                    list = new ArrayList<Sensor>();
                    for (Sensor i : fullList) {
                        if (i.getType() == type) {
                            list.add(i);
                        }
                    }
                }
                list = Collections.unmodifiableList(list);
                mSensorListByType.append(type, list);
            }
        }
        return list;
    }

    /**
     * Use this method to get a list of available dynamic sensors of a certain type.
     * Make multiple calls to get sensors of different types or use
     * {@link android.hardware.Sensor#TYPE_ALL Sensor.TYPE_ALL} to get all dynamic sensors.
     *
     * <p class="note">
     * NOTE: Both wake-up and non wake-up sensors matching the given type are
     * returned. Check {@link Sensor#isWakeUpSensor()} to know the wake-up properties
     * of the returned {@link Sensor}.
     * </p>
     *
     * @param type of sensors requested
     *
     * @return a list of dynamic sensors matching the requested type.
     *
     * @see Sensor
     */
    public List<Sensor> getDynamicSensorList(int type) {
        // cache the returned lists the first time
        final List<Sensor> fullList = getFullDynamicSensorList();
        if (type == Sensor.TYPE_ALL) {
```

```java
            return Collections.unmodifiableList(fullList);
        } else {
            List<Sensor> list = new ArrayList();
            for (Sensor i : fullList) {
                if (i.getType() == type) {
                    list.add(i);
                }
            }
            return Collections.unmodifiableList(list);
        }
    }

    /**
     * Use this method to get the default sensor for a given type. Note that the
     * returned sensor could be a composite sensor, and its data could be
     * averaged or filtered. If you need to access the raw sensors use
     * {@link SensorManager#getSensorList(int) getSensorList}.
     *
     * @param type
     *         of sensors requested
     *
     * @return the default sensor matching the requested type if one exists and the application
     *         has the necessary permissions, or null otherwise.
     *
     * @see #getSensorList(int)
     * @see Sensor
     */
    public @Nullable Sensor getDefaultSensor(int type) {
        // TODO: need to be smarter, for now, just return the 1st sensor
        List<Sensor> l = getSensorList(type);
        boolean wakeUpSensor = false;
        // For the following sensor types, return a wake-up sensor. These types are by default
        // defined as wake-up sensors. For the rest of the SDK defined sensor types return a
        // non_wake-up version.
        if (type == Sensor.TYPE_PROXIMITY || type == Sensor.TYPE_SIGNIFICANT_MOTION
                || type == Sensor.TYPE_TILT_DETECTOR || type == Sensor.TYPE_WAKE_GESTURE
                || type == Sensor.TYPE_GLANCE_GESTURE || type == Sensor.TYPE_PICK_UP_GESTURE
```

```java
                    || type == Sensor.TYPE_LOW_LATENCY_OFFBODY_DETECT
                    || type == Sensor.TYPE_WRIST_TILT_GESTURE
                    || type == Sensor.TYPE_DYNAMIC_SENSOR_META || type == Sensor.TYPE_HINGE_ANGLE) {
                wakeUpSensor = true;
            }

            for (Sensor sensor : l) {
                if (sensor.isWakeUpSensor() == wakeUpSensor) return sensor;
            }
            return null;
        }

        /**
         * Return a Sensor with the given type and wakeUp properties. If multiple sensors of this
         * type exist, any one of them may be returned.
         * <p>
         * For example,
         * <ul>
         *     <li>getDefaultSensor({@link Sensor#TYPE_ACCELEROMETER}, true) returns a wake-up
         *     accelerometer sensor if it exists. </li>
         *     <li>getDefaultSensor({@link Sensor#TYPE_PROXIMITY}, false) returns a non wake-up
         *     proximity sensor if it exists. </li>
         *     <li>getDefaultSensor({@link Sensor#TYPE_PROXIMITY}, true) returns a wake-up proximity
         *     sensor which is the same as the Sensor returned by {@link #getDefaultSensor(int)}. </li>
         * </ul>
         * </p>
         * <p class="note">
         * Note: Sensors like {@link Sensor#TYPE_PROXIMITY} and {@link Sensor#TYPE_SIGNIFICANT_MOTION}
         * are declared as wake-up sensors by default.
         * </p>
         * @param type
         *        type of sensor requested
         * @param wakeUp
         *        flag to indicate whether the Sensor is a wake-up or non wake-up sensor.
         * @return the default sensor matching the requested type and wakeUp properties if one exists
         *         and the application has the necessary permissions, or null otherwise.
         * @see Sensor#isWakeUpSensor()
```

```java
     */
    public @Nullable Sensor getDefaultSensor(int type, boolean wakeUp) {
        List<Sensor> l = getSensorList(type);
        for (Sensor sensor : l) {
            if (sensor.isWakeUpSensor() == wakeUp) {
                return sensor;
            }
        }
        return null;
    }

    /**
     * Registers a listener for given sensors.
     *
     * @deprecated This method is deprecated, use
     *             {@link SensorManager#registerListener(SensorEventListener, Sensor, int)}
     *             instead.
     *
     * @param listener
     *        sensor listener object
     *
     * @param sensors
     *        a bit masks of the sensors to register to
     *
     * @return <code>true</code> if the sensor is supported and successfully
     *         enabled
     */
    @Deprecated
    public boolean registerListener(SensorListener listener, int sensors) {
        return registerListener(listener, sensors, SENSOR_DELAY_NORMAL);
    }

    /**
     * Registers a SensorListener for given sensors.
     *
     * @deprecated This method is deprecated, use
     *             {@link SensorManager#registerListener(SensorEventListener, Sensor, int)}
```

```java
     *             instead.
     *
     * @param listener
     *        sensor listener object
     *
     * @param sensors
     *        a bit masks of the sensors to register to
     *
     * @param rate
     *        rate of events. This is only a hint to the system. events may be
     *        received faster or slower than the specified rate. Usually events
     *        are received faster. The value must be one of
     *        {@link #SENSOR_DELAY_NORMAL}, {@link #SENSOR_DELAY_UI},
     *        {@link #SENSOR_DELAY_GAME}, or {@link #SENSOR_DELAY_FASTEST}.
     *
     * @return <code>true</code> if the sensor is supported and successfully
     *         enabled
     */
    @Deprecated
    public boolean registerListener(SensorListener listener, int sensors, int rate) {
        return getLegacySensorManager().registerListener(listener, sensors, rate);
    }

    /**
     * Unregisters a listener for all sensors.
     *
     * @deprecated This method is deprecated, use
     *             {@link SensorManager#unregisterListener(SensorEventListener)}
     *             instead.
     *
     * @param listener
     *        a SensorListener object
     */
    @Deprecated
    public void unregisterListener(SensorListener listener) {
        unregisterListener(listener, SENSOR_ALL | SENSOR_ORIENTATION_RAW);
    }
```

```java

    /**
     * Unregisters a listener for the sensors with which it is registered.
     *
     * @deprecated This method is deprecated, use
     *             {@link SensorManager#unregisterListener(SensorEventListener, Sensor)}
     *             instead.
     *
     * @param listener
     *        a SensorListener object
     *
     * @param sensors
     *        a bit masks of the sensors to unregister from
     */
    @Deprecated
    public void unregisterListener(SensorListener listener, int sensors) {
        getLegacySensorManager().unregisterListener(listener, sensors);
    }

    /**
     * Unregisters a listener for the sensors with which it is registered.
     *
     * <p class="note">
     * Note: Don't use this method with a one shot trigger sensor such as
     * {@link Sensor#TYPE_SIGNIFICANT_MOTION}.
     * Use {@link #cancelTriggerSensor(TriggerEventListener, Sensor)} instead.
     * </p>
     *
     * @param listener
     *        a SensorEventListener object
     *
     * @param sensor
     *        the sensor to unregister from
     *
     * @see #unregisterListener(SensorEventListener)
     * @see #registerListener(SensorEventListener, Sensor, int)
     */
```

```java
    public void unregisterListener(SensorEventListener listener, Sensor sensor) {
        if (listener == null || sensor == null) {
            return;
        }

        unregisterListenerImpl(listener, sensor);
    }

    /**
     * Unregisters a listener for all sensors.
     *
     * @param listener
     *        a SensorListener object
     *
     * @see #unregisterListener(SensorEventListener, Sensor)
     * @see #registerListener(SensorEventListener, Sensor, int)
     *
     */
    public void unregisterListener(SensorEventListener listener) {
        if (listener == null) {
            return;
        }

        unregisterListenerImpl(listener, null);
    }

    /** @hide */
    protected abstract void unregisterListenerImpl(SensorEventListener listener, Sensor sensor);

    /**
     * Registers a {@link android.hardware.SensorEventListener SensorEventListener} for the given
     * sensor at the given sampling frequency.
     * <p>
     * The events will be delivered to the provided {@code SensorEventListener} as soon as they are
     * available. To reduce the power consumption, applications can use
     * {@link #registerListener(SensorEventListener, Sensor, int, int)} instead and specify a
     * positive non-zero maximum reporting latency.
```

```
     * </p>
     * <p>
     * In the case of non-wake-up sensors, the events are only delivered while the Application
     * Processor (AP) is not in suspend mode. See {@link Sensor#isWakeUpSensor()} for more details.
     * To ensure delivery of events from non-wake-up sensors even when the screen is OFF, the
     * application registering to the sensor must hold a partial wake-lock to keep the AP awake,
     * otherwise some events might be lost while the AP is asleep. Note that although events might
     * be lost while the AP is asleep, the sensor will still consume power if it is not explicitly
     * deactivated by the application. Applications must unregister their {@code
     * SensorEventListener}s in their activity's {@code onPause()} method to avoid consuming power
     * while the device is inactive.  See {@link #registerListener(SensorEventListener, Sensor, int,
     * int)} for more details on hardware FIFO (queueing) capabilities and when some sensor events
     * might be lost.
     * </p>
     * <p>
     * In the case of wake-up sensors, each event generated by the sensor will cause the AP to
     * wake-up, ensuring that each event can be delivered. Because of this, registering to a wake-up
     * sensor has very significant power implications. Call {@link Sensor#isWakeUpSensor()} to check
     * whether a sensor is a wake-up sensor. See
     * {@link #registerListener(SensorEventListener, Sensor, int, int)} for information on how to
     * reduce the power impact of registering to wake-up sensors.
     * </p>
     * <p class="note">
     * Note: Don't use this method with one-shot trigger sensors such as
     * {@link Sensor#TYPE_SIGNIFICANT_MOTION}. Use
     * {@link #requestTriggerSensor(TriggerEventListener, Sensor)} instead. Use
     * {@link Sensor#getReportingMode()} to obtain the reporting mode of a given sensor.
     * </p>
     *
     * @param listener A {@link android.hardware.SensorEventListener SensorEventListener} object.
     * @param sensor The {@link android.hardware.Sensor Sensor} to register to.
     * @param samplingPeriodUs The rate {@link android.hardware.SensorEvent sensor events} are
     *            delivered at. This is only a hint to the system. Events may be received faster or
     *            slower than the specified rate. Usually events are received faster. The value must
     *            be one of {@link #SENSOR_DELAY_NORMAL}, {@link #SENSOR_DELAY_UI},
     *            {@link #SENSOR_DELAY_GAME}, or {@link #SENSOR_DELAY_FASTEST} or, the desired delay
     *            between events in microseconds. Specifying the delay in microseconds only works
```

```java
     *             from Android 2.3 (API level 9) onwards. For earlier releases, you must use one of
     *             the {@code SENSOR_DELAY_*} constants.
     * @return <code>true</code> if the sensor is supported and successfully enabled.
     * @see #registerListener(SensorEventListener, Sensor, int, Handler)
     * @see #unregisterListener(SensorEventListener)
     * @see #unregisterListener(SensorEventListener, Sensor)
     */
    public boolean registerListener(SensorEventListener listener, Sensor sensor,
            int samplingPeriodUs) {
        return registerListener(listener, sensor, samplingPeriodUs, null);
    }

    /**
     * Registers a {@link android.hardware.SensorEventListener SensorEventListener} for the given
     * sensor at the given sampling frequency and the given maximum reporting latency.
     * <p>
     * This function is similar to {@link #registerListener(SensorEventListener, Sensor, int)} but
     * it allows events to stay temporarily in the hardware FIFO (queue) before being delivered. The
     * events can be stored in the hardware FIFO up to {@code maxReportLatencyUs} microseconds. Once
     * one of the events in the FIFO needs to be reported, all of the events in the FIFO are
     * reported sequentially. This means that some events will be reported before the maximum
     * reporting latency has elapsed.
     * </p><p>
     * When {@code maxReportLatencyUs} is 0, the call is equivalent to a call to
     * {@link #registerListener(SensorEventListener, Sensor, int)}, as it requires the events to be
     * delivered as soon as possible.
     * </p><p>
     * When {@code sensor.maxFifoEventCount()} is 0, the sensor does not use a FIFO, so the call
     * will also be equivalent to {@link #registerListener(SensorEventListener, Sensor, int)}.
     * </p><p>
     * Setting {@code maxReportLatencyUs} to a positive value allows to reduce the number of
     * interrupts the AP (Application Processor) receives, hence reducing power consumption, as the
     * AP can switch to a lower power state while the sensor is capturing the data. This is
     * especially important when registering to wake-up sensors, for which each interrupt causes the
     * AP to wake up if it was in suspend mode. See {@link Sensor#isWakeUpSensor()} for more
     * information on wake-up sensors.
     * </p>
```

```
     * <p class="note">
     * </p>
     * Note: Don't use this method with one-shot trigger sensors such as
     * {@link Sensor#TYPE_SIGNIFICANT_MOTION}. Use
     * {@link #requestTriggerSensor(TriggerEventListener, Sensor)} instead. </p>
     *
     * @param listener A {@link android.hardware.SensorEventListener SensorEventListener} object
     *            that will receive the sensor events. If the application is interested in receiving
     *            flush complete notifications, it should register with
     *            {@link android.hardware.SensorEventListener SensorEventListener2} instead.
     * @param sensor The {@link android.hardware.Sensor Sensor} to register to.
     * @param samplingPeriodUs The desired delay between two consecutive events in microseconds.
     *            This is only a hint to the system. Events may be received faster or slower than
     *            the specified rate. Usually events are received faster. Can be one of
     *            {@link #SENSOR_DELAY_NORMAL}, {@link #SENSOR_DELAY_UI},
     *            {@link #SENSOR_DELAY_GAME}, {@link #SENSOR_DELAY_FASTEST} or the delay in
     *            microseconds.
     * @param maxReportLatencyUs Maximum time in microseconds that events can be delayed before
     *            being reported to the application. A large value allows reducing the power
     *            consumption associated with the sensor. If maxReportLatencyUs is set to zero,
     *            events are delivered as soon as they are available, which is equivalent to calling
     *            {@link #registerListener(SensorEventListener, Sensor, int)}.
     * @return <code>true</code> if the sensor is supported and successfully enabled.
     * @see #registerListener(SensorEventListener, Sensor, int)
     * @see #unregisterListener(SensorEventListener)
     * @see #flush(SensorEventListener)
     */
    public boolean registerListener(SensorEventListener listener, Sensor sensor,
            int samplingPeriodUs, int maxReportLatencyUs) {
        int delay = getDelay(samplingPeriodUs);
        return registerListenerImpl(listener, sensor, delay, null, maxReportLatencyUs, 0);
    }

    /**
     * Registers a {@link android.hardware.SensorEventListener SensorEventListener} for the given
     * sensor. Events are delivered in continuous mode as soon as they are available. To reduce the
     * power consumption, applications can use
```

```java
     * {@link #registerListener(SensorEventListener, Sensor, int, int)} instead and specify a
     * positive non-zero maximum reporting latency.
     * <p class="note">
     * </p>
     * Note: Don't use this method with a one shot trigger sensor such as
     * {@link Sensor#TYPE_SIGNIFICANT_MOTION}. Use
     * {@link #requestTriggerSensor(TriggerEventListener, Sensor)} instead. </p>
     *
     * @param listener A {@link android.hardware.SensorEventListener SensorEventListener} object.
     * @param sensor The {@link android.hardware.Sensor Sensor} to register to.
     * @param samplingPeriodUs The rate {@link android.hardware.SensorEvent sensor events} are
     *            delivered at. This is only a hint to the system. Events may be received faster or
     *            slower than the specified rate. Usually events are received faster. The value must
     *            be one of {@link #SENSOR_DELAY_NORMAL}, {@link #SENSOR_DELAY_UI},
     *            {@link #SENSOR_DELAY_GAME}, or {@link #SENSOR_DELAY_FASTEST} or, the desired
     *            delay between events in microseconds. Specifying the delay in microseconds only
     *            works from Android 2.3 (API level 9) onwards. For earlier releases, you must use
     *            one of the {@code SENSOR_DELAY_*} constants.
     * @param handler The {@link android.os.Handler Handler} the {@link android.hardware.SensorEvent
     *            sensor events} will be delivered to.
     * @return <code>true</code> if the sensor is supported and successfully enabled.
     * @see #registerListener(SensorEventListener, Sensor, int)
     * @see #unregisterListener(SensorEventListener)
     * @see #unregisterListener(SensorEventListener, Sensor)
     */
    public boolean registerListener(SensorEventListener listener, Sensor sensor,
            int samplingPeriodUs, Handler handler) {
        int delay = getDelay(samplingPeriodUs);
        return registerListenerImpl(listener, sensor, delay, handler, 0, 0);
    }

    /**
     * Registers a {@link android.hardware.SensorEventListener SensorEventListener} for the given
     * sensor at the given sampling frequency and the given maximum reporting latency.
     *
     * @param listener A {@link android.hardware.SensorEventListener SensorEventListener} object
     *            that will receive the sensor events. If the application is interested in receiving
```

```java
     *             flush complete notifications, it should register with
     *             {@link android.hardware.SensorEventListener SensorEventListener2} instead.
     * @param sensor The {@link android.hardware.Sensor Sensor} to register to.
     * @param samplingPeriodUs The desired delay between two consecutive events in microseconds.
     *             This is only a hint to the system. Events may be received faster or slower than
     *             the specified rate. Usually events are received faster. Can be one of
     *             {@link #SENSOR_DELAY_NORMAL}, {@link #SENSOR_DELAY_UI},
     *             {@link #SENSOR_DELAY_GAME}, {@link #SENSOR_DELAY_FASTEST} or the delay in
     *             microseconds.
     * @param maxReportLatencyUs Maximum time in microseconds that events can be delayed before
     *             being reported to the application. A large value allows reducing the power
     *             consumption associated with the sensor. If maxReportLatencyUs is set to zero,
     *             events are delivered as soon as they are available, which is equivalent to calling
     *             {@link #registerListener(SensorEventListener, Sensor, int)}.
     * @param handler The {@link android.os.Handler Handler} the {@link android.hardware.SensorEvent
     *             sensor events} will be delivered to.
     * @return <code>true</code> if the sensor is supported and successfully enabled.
     * @see #registerListener(SensorEventListener, Sensor, int, int)
     */
    public boolean registerListener(SensorEventListener listener, Sensor sensor,
            int samplingPeriodUs, int maxReportLatencyUs, Handler handler) {
        int delayUs = getDelay(samplingPeriodUs);
        return registerListenerImpl(listener, sensor, delayUs, handler, maxReportLatencyUs, 0);
    }

    /** @hide */
    protected abstract boolean registerListenerImpl(SensorEventListener listener, Sensor sensor,
            int delayUs, Handler handler, int maxReportLatencyUs, int reservedFlags);


    /**
     * Flushes the FIFO of all the sensors registered for this listener. If there are events
     * in the FIFO of the sensor, they are returned as if the maxReportLatency of the FIFO has
     * expired. Events are returned in the usual way through the SensorEventListener.
     * This call doesn't affect the maxReportLatency for this sensor. This call is asynchronous and
     * returns immediately.
     * {@link android.hardware.SensorEventListener2#onFlushCompleted onFlushCompleted} is called
```

```
     * after all the events in the batch at the time of calling this method have been delivered
     * successfully. If the hardware doesn't support flush, it still returns true and a trivial
     * flush complete event is sent after the current event for all the clients registered for this
     * sensor.
     *
     * @param listener A {@link android.hardware.SensorEventListener SensorEventListener} object
     *        which was previously used in a registerListener call.
     * @return <code>true</code> if the flush is initiated successfully on all the sensors
     *         registered for this listener, false if no sensor is previously registered for this
     *         listener or flush on one of the sensors fails.
     * @see #registerListener(SensorEventListener, Sensor, int, int)
     * @throws IllegalArgumentException when listener is null.
     */
    public boolean flush(SensorEventListener listener) {
        return flushImpl(listener);
    }

    /** @hide */
    protected abstract boolean flushImpl(SensorEventListener listener);


    /**
     * Create a sensor direct channel backed by shared memory wrapped in MemoryFile object.
     *
     * The resulting channel can be used for delivering sensor events to native code, other
     * processes, GPU/DSP or other co-processors without CPU intervention. This is the recommended
     * for high performance sensor applications that use high sensor rates (e.g. greater than 200Hz)
     * and cares about sensor event latency.
     *
     * Use the returned {@link android.hardware.SensorDirectChannel} object to configure direct
     * report of sensor events. After use, call {@link android.hardware.SensorDirectChannel#close()}
     * to free up resource in sensor system associated with the direct channel.
     *
     * @param mem A {@link android.os.MemoryFile} shared memory object.
     * @return A {@link android.hardware.SensorDirectChannel} object.
     * @throws NullPointerException when mem is null.
     * @throws UncheckedIOException if not able to create channel.
```

```java
     * @see SensorDirectChannel#close()
     */
    public SensorDirectChannel createDirectChannel(MemoryFile mem) {
        return createDirectChannelImpl(mem, null);
    }

    /**
     * Create a sensor direct channel backed by shared memory wrapped in HardwareBuffer object.
     *
     * The resulting channel can be used for delivering sensor events to native code, other
     * processes, GPU/DSP or other co-processors without CPU intervention. This is the recommended
     * for high performance sensor applications that use high sensor rates (e.g. greater than 200Hz)
     * and cares about sensor event latency.
     *
     * Use the returned {@link android.hardware.SensorDirectChannel} object to configure direct
     * report of sensor events. After use, call {@link android.hardware.SensorDirectChannel#close()}
     * to free up resource in sensor system associated with the direct channel.
     *
     * @param mem A {@link android.hardware.HardwareBuffer} shared memory object.
     * @return A {@link android.hardware.SensorDirectChannel} object.
     * @throws NullPointerException when mem is null.
     * @throws UncheckedIOException if not able to create channel.
     * @see SensorDirectChannel#close()
     */
    public SensorDirectChannel createDirectChannel(HardwareBuffer mem) {
        return createDirectChannelImpl(null, mem);
    }

    /** @hide */
    protected abstract SensorDirectChannel createDirectChannelImpl(
            MemoryFile memoryFile, HardwareBuffer hardwareBuffer);

    /** @hide */
    void destroyDirectChannel(SensorDirectChannel channel) {
        destroyDirectChannelImpl(channel);
    }

```

```java
    /** @hide */
    protected abstract void destroyDirectChannelImpl(SensorDirectChannel channel);

    /** @hide */
    protected abstract int configureDirectChannelImpl(
            SensorDirectChannel channel, Sensor s, int rate);

    /**
     * Used for receiving notifications from the SensorManager when dynamic sensors are connected or
     * disconnected.
     */
    public abstract static class DynamicSensorCallback {
        /**
         * Called when there is a dynamic sensor being connected to the system.
         *
         * @param sensor the newly connected sensor. See {@link android.hardware.Sensor Sensor}.
         */
        public void onDynamicSensorConnected(Sensor sensor) {}

        /**
         * Called when there is a dynamic sensor being disconnected from the system.
         *
         * @param sensor the disconnected sensor. See {@link android.hardware.Sensor Sensor}.
         */
        public void onDynamicSensorDisconnected(Sensor sensor) {}
    }


    /**
     * Add a {@link android.hardware.SensorManager.DynamicSensorCallback
     * DynamicSensorCallback} to receive dynamic sensor connection callbacks. Repeat
     * registration with the already registered callback object will have no additional effect.
     *
     * @param callback An object that implements the
     *        {@link android.hardware.SensorManager.DynamicSensorCallback
     *        DynamicSensorCallback}
     *        interface for receiving callbacks.
```

```java
     * @see #registerDynamicSensorCallback(DynamicSensorCallback, Handler)
     *
     * @throws IllegalArgumentException when callback is null.
     */
    public void registerDynamicSensorCallback(DynamicSensorCallback callback) {
        registerDynamicSensorCallback(callback, null);
    }

    /**
     * Add a {@link android.hardware.SensorManager.DynamicSensorCallback
     * DynamicSensorCallback} to receive dynamic sensor connection callbacks. Repeat
     * registration with the already registered callback object will have no additional effect.
     *
     * @param callback An object that implements the
     *        {@link android.hardware.SensorManager.DynamicSensorCallback
     *        DynamicSensorCallback} interface for receiving callbacks.
     * @param handler The {@link android.os.Handler Handler} the {@link
     *        android.hardware.SensorManager.DynamicSensorCallback
     *        sensor connection events} will be delivered to.
     *
     * @throws IllegalArgumentException when callback is null.
     */
    public void registerDynamicSensorCallback(
            DynamicSensorCallback callback, Handler handler) {
        registerDynamicSensorCallbackImpl(callback, handler);
    }

    /**
     * Remove a {@link android.hardware.SensorManager.DynamicSensorCallback
     * DynamicSensorCallback} to stop sending dynamic sensor connection events to that
     * callback.
     *
     * @param callback An object that implements the
     *        {@link android.hardware.SensorManager.DynamicSensorCallback
     *        DynamicSensorCallback}
     *        interface for receiving callbacks.
     */
```

```java
    public void unregisterDynamicSensorCallback(DynamicSensorCallback callback) {
        unregisterDynamicSensorCallbackImpl(callback);
    }

    /**
     * Tell if dynamic sensor discovery feature is supported by system.
     *
     * @return <code>true</code> if dynamic sensor discovery is supported, <code>false</code>
     * otherwise.
     */
    public boolean isDynamicSensorDiscoverySupported() {
        List<Sensor> sensors = getSensorList(Sensor.TYPE_DYNAMIC_SENSOR_META);
        return sensors.size() > 0;
    }

    /** @hide */
    protected abstract void registerDynamicSensorCallbackImpl(
            DynamicSensorCallback callback, Handler handler);

    /** @hide */
    protected abstract void unregisterDynamicSensorCallbackImpl(
            DynamicSensorCallback callback);

    /**
     * <p>
     * Computes the inclination matrix <b>I</b> as well as the rotation matrix
     * <b>R</b> transforming a vector from the device coordinate system to the
     * world's coordinate system which is defined as a direct orthonormal basis,
     * where:
     * </p>
     *
     * <ul>
     * <li>X is defined as the vector product <b>Y.Z</b> (It is tangential to
     * the ground at the device's current location and roughly points East).</li>
     * <li>Y is tangential to the ground at the device's current location and
     * points towards the magnetic North Pole.</li>
     * <li>Z points towards the sky and is perpendicular to the ground.</li>
```

```
     * </ul>
     *
     * <p>
     * <center><img src="../../../images/axis_globe.png"
     * alt="World coordinate-system diagram." border="0" /></center>
     * </p>
     *
     * <p>
     * <hr>
     * <p>
     * By definition:
     * <p>
     * [0 0 g] = <b>R</b> * <b>gravity</b> (g = magnitude of gravity)
     * <p>
     * [0 m 0] = <b>I</b> * <b>R</b> * <b>geomagnetic</b> (m = magnitude of
     * geomagnetic field)
     * <p>
     * <b>R</b> is the identity matrix when the device is aligned with the
     * world's coordinate system, that is, when the device's X axis points
     * toward East, the Y axis points to the North Pole and the device is facing
     * the sky.
     *
     * <p>
     * <b>I</b> is a rotation matrix transforming the geomagnetic vector into
     * the same coordinate space as gravity (the world's coordinate space).
     * <b>I</b> is a simple rotation around the X axis. The inclination angle in
     * radians can be computed with {@link #getInclination}.
     * <hr>
     *
     * <p>
     * Each matrix is returned either as a 3x3 or 4x4 row-major matrix depending
     * on the length of the passed array:
     * <p>
     * <u>If the array length is 16:</u>
     *
     * <pre>
     *   /  M[ 0]   M[ 1]   M[ 2]   M[ 3]  \
```

```
     *   |  M[ 4]   M[ 5]   M[ 6]   M[ 7]  |
     *   |  M[ 8]   M[ 9]   M[10]   M[11]  |
     *   \  M[12]   M[13]   M[14]   M[15]  /
     *</pre>
     *
     * This matrix is ready to be used by OpenGL ES's
     * {@link javax.microedition.khronos.opengles.GL10#glLoadMatrixf(float[], int)
     * glLoadMatrixf(float[], int)}.
     * <p>
     * Note that because OpenGL matrices are column-major matrices you must
     * transpose the matrix before using it. However, since the matrix is a
     * rotation matrix, its transpose is also its inverse, conveniently, it is
     * often the inverse of the rotation that is needed for rendering; it can
     * therefore be used with OpenGL ES directly.
     * <p>
     * Also note that the returned matrices always have this form:
     *
     * <pre>
     *   /  M[ 0]   M[ 1]   M[ 2]   0  \
     *   |  M[ 4]   M[ 5]   M[ 6]   0  |
     *   |  M[ 8]   M[ 9]   M[10]   0  |
     *   \      0       0       0   1  /
     *</pre>
     *
     * <p>
     * <u>If the array length is 9:</u>
     *
     * <pre>
     *   /  M[ 0]   M[ 1]   M[ 2]  \
     *   |  M[ 3]   M[ 4]   M[ 5]  |
     *   \  M[ 6]   M[ 7]   M[ 8]  /
     *</pre>
     *
     * <hr>
     * <p>
     * The inverse of each matrix can be computed easily by taking its
     * transpose.
```

```
     *
     * <p>
     * The matrices returned by this function are meaningful only when the
     * device is not free-falling and it is not close to the magnetic north. If
     * the device is accelerating, or placed into a strong magnetic field, the
     * returned matrices may be inaccurate.
     *
     * @param R
     *        is an array of 9 floats holding the rotation matrix <b>R</b> when
     *        this function returns. R can be null.
     *        <p>
     *
     * @param I
     *        is an array of 9 floats holding the rotation matrix <b>I</b> when
     *        this function returns. I can be null.
     *        <p>
     *
     * @param gravity
     *        is an array of 3 floats containing the gravity vector expressed in
     *        the device's coordinate. You can simply use the
     *        {@link android.hardware.SensorEvent#values values} returned by a
     *        {@link android.hardware.SensorEvent SensorEvent} of a
     *        {@link android.hardware.Sensor Sensor} of type
     *        {@link android.hardware.Sensor#TYPE_ACCELEROMETER
     *        TYPE_ACCELEROMETER}.
     *        <p>
     *
     * @param geomagnetic
     *        is an array of 3 floats containing the geomagnetic vector
     *        expressed in the device's coordinate. You can simply use the
     *        {@link android.hardware.SensorEvent#values values} returned by a
     *        {@link android.hardware.SensorEvent SensorEvent} of a
     *        {@link android.hardware.Sensor Sensor} of type
     *        {@link android.hardware.Sensor#TYPE_MAGNETIC_FIELD
     *        TYPE_MAGNETIC_FIELD}.
     *
     * @return <code>true</code> on success, <code>false</code> on failure (for
```

```java
     *         instance, if the device is in free fall). Free fall is defined as
     *         condition when the magnitude of the gravity is less than 1/10 of
     *         the nominal value. On failure the output matrices are not modified.
     *
     * @see #getInclination(float[])
     * @see #getOrientation(float[], float[])
     * @see #remapCoordinateSystem(float[], int, int, float[])
     */

    public static boolean getRotationMatrix(float[] R, float[] I,
            float[] gravity, float[] geomagnetic) {
        // TODO: move this to native code for efficiency
        float Ax = gravity[0];
        float Ay = gravity[1];
        float Az = gravity[2];

        final float normsqA = (Ax * Ax + Ay * Ay + Az * Az);
        final float g = 9.81f;
        final float freeFallGravitySquared = 0.01f * g * g;
        if (normsqA < freeFallGravitySquared) {
            // gravity less than 10% of normal value
            return false;
        }

        final float Ex = geomagnetic[0];
        final float Ey = geomagnetic[1];
        final float Ez = geomagnetic[2];
        float Hx = Ey * Az - Ez * Ay;
        float Hy = Ez * Ax - Ex * Az;
        float Hz = Ex * Ay - Ey * Ax;
        final float normH = (float) Math.sqrt(Hx * Hx + Hy * Hy + Hz * Hz);

        if (normH < 0.1f) {
            // device is close to free fall (or in space?), or close to
            // magnetic north pole. Typical values are  > 100.
            return false;
        }
```

```
        final float invH = 1.0f / normH;
        Hx *= invH;
        Hy *= invH;
        Hz *= invH;
        final float invA = 1.0f / (float) Math.sqrt(Ax * Ax + Ay * Ay + Az * Az);
        Ax *= invA;
        Ay *= invA;
        Az *= invA;
        final float Mx = Ay * Hz - Az * Hy;
        final float My = Az * Hx - Ax * Hz;
        final float Mz = Ax * Hy - Ay * Hx;
        if (R != null) {
            if (R.length == 9) {
                R[0] = Hx;     R[1] = Hy;     R[2] = Hz;
                R[3] = Mx;     R[4] = My;     R[5] = Mz;
                R[6] = Ax;     R[7] = Ay;     R[8] = Az;
            } else if (R.length == 16) {
                R[0]  = Hx;    R[1]  = Hy;    R[2]  = Hz;   R[3]  = 0;
                R[4]  = Mx;    R[5]  = My;    R[6]  = Mz;   R[7]  = 0;
                R[8]  = Ax;    R[9]  = Ay;    R[10] = Az;   R[11] = 0;
                R[12] = 0;     R[13] = 0;     R[14] = 0;    R[15] = 1;
            }
        }
        if (I != null) {
            // compute the inclination matrix by projecting the geomagnetic
            // vector onto the Z (gravity) and X (horizontal component
            // of geomagnetic vector) axes.
            final float invE = 1.0f / (float) Math.sqrt(Ex * Ex + Ey * Ey + Ez * Ez);
            final float c = (Ex * Mx + Ey * My + Ez * Mz) * invE;
            final float s = (Ex * Ax + Ey * Ay + Ez * Az) * invE;
            if (I.length == 9) {
                I[0] = 1;     I[1] = 0;     I[2] = 0;
                I[3] = 0;     I[4] = c;     I[5] = s;
                I[6] = 0;     I[7] = -s;     I[8] = c;
            } else if (I.length == 16) {
                I[0] = 1;     I[1] = 0;     I[2] = 0;
                I[4] = 0;     I[5] = c;     I[6] = s;
```

```java
                I[8] = 0;     I[9] = -s;     I[10] = c;
                I[3] = I[7] = I[11] = I[12] = I[13] = I[14] = 0;
                I[15] = 1;
            }
        }
        return true;
    }

    /**
     * Computes the geomagnetic inclination angle in radians from the
     * inclination matrix <b>I</b> returned by {@link #getRotationMatrix}.
     *
     * @param I
     *        inclination matrix see {@link #getRotationMatrix}.
     *
     * @return The geomagnetic inclination angle in radians.
     *
     * @see #getRotationMatrix(float[], float[], float[], float[])
     * @see #getOrientation(float[], float[])
     * @see GeomagneticField
     *
     */
    public static float getInclination(float[] I) {
        if (I.length == 9) {
            return (float) Math.atan2(I[5], I[4]);
        } else {
            return (float) Math.atan2(I[6], I[5]);
        }
    }

    /**
     * <p>
     * Rotates the supplied rotation matrix so it is expressed in a different
     * coordinate system. This is typically used when an application needs to
     * compute the three orientation angles of the device (see
     * {@link #getOrientation}) in a different coordinate system.
     * </p>
```

```
     *
     * <p>
     * When the rotation matrix is used for drawing (for instance with OpenGL
     * ES), it usually <b>doesn't need</b> to be transformed by this function,
     * unless the screen is physically rotated, in which case you can use
     * {@link android.view.Display#getRotation() Display.getRotation()} to
     * retrieve the current rotation of the screen. Note that because the user
     * is generally free to rotate their screen, you often should consider the
     * rotation in deciding the parameters to use here.
     * </p>
     *
     * <p>
     * <u>Examples:</u>
     * <p>
     *
     * <ul>
     * <li>Using the camera (Y axis along the camera's axis) for an augmented
     * reality application where the rotation angles are needed:</li>
     *
     * <p>
     * <ul>
     * <code>remapCoordinateSystem(inR, AXIS_X, AXIS_Z, outR);</code>
     * </ul>
     * </p>
     *
     * <li>Using the device as a mechanical compass when rotation is
     * {@link android.view.Surface#ROTATION_90 Surface.ROTATION_90}:</li>
     *
     * <p>
     * <ul>
     * <code>remapCoordinateSystem(inR, AXIS_Y, AXIS_MINUS_X, outR);</code>
     * </ul>
     * </p>
     *
     * Beware of the above example. This call is needed only to account for a
     * rotation from its natural orientation when calculating the rotation
     * angles (see {@link #getOrientation}). If the rotation matrix is also used
```

```java
     * for rendering, it may not need to be transformed, for instance if your
     * {@link android.app.Activity Activity} is running in landscape mode.
     * </ul>
     *
     * <p>
     * Since the resulting coordinate system is orthonormal, only two axes need
     * to be specified.
     *
     * @param inR
     *        the rotation matrix to be transformed. Usually it is the matrix
     *        returned by {@link #getRotationMatrix}.
     *
     * @param X
     *        defines the axis of the new coordinate system that coincide with the X axis of the
     *        original coordinate system.
     *
     * @param Y
     *        defines the axis of the new coordinate system that coincide with the Y axis of the
     *        original coordinate system.
     *
     * @param outR
     *        the transformed rotation matrix. inR and outR should not be the same
     *        array.
     *
     * @return <code>true</code> on success. <code>false</code> if the input
     *         parameters are incorrect, for instance if X and Y define the same
     *         axis. Or if inR and outR don't have the same length.
     *
     * @see #getRotationMatrix(float[], float[], float[], float[])
     */

    public static boolean remapCoordinateSystem(float[] inR, int X, int Y, float[] outR) {
        if (inR == outR) {
            final float[] temp = sTempMatrix;
            synchronized (temp) {
                // we don't expect to have a lot of contention
                if (remapCoordinateSystemImpl(inR, X, Y, temp)) {
```

```java
                    final int size = outR.length;
                    for (int i = 0; i < size; i++) {
                        outR[i] = temp[i];
                    }
                    return true;
                }
            }
        }
        return remapCoordinateSystemImpl(inR, X, Y, outR);
    }

    private static boolean remapCoordinateSystemImpl(float[] inR, int X, int Y, float[] outR) {
        /*
         * X and Y define a rotation matrix 'r':
         *
         *  (X==1)?((X&0x80)?-1:1):0    (X==2)?((X&0x80)?-1:1):0    (X==3)?((X&0x80)?-1:1):0
         *  (Y==1)?((Y&0x80)?-1:1):0    (Y==2)?((Y&0x80)?-1:1):0    (Y==3)?((X&0x80)?-1:1):0
         *                              r[0] ^ r[1]
         *
         * where the 3rd line is the vector product of the first 2 lines
         *
         */

        final int length = outR.length;
        if (inR.length != length) {
            return false;   // invalid parameter
        }
        if ((X & 0x7C) != 0 || (Y & 0x7C) != 0) {
            return false;   // invalid parameter
        }
        if (((X & 0x3) == 0) || ((Y & 0x3) == 0)) {
            return false;   // no axis specified
        }
        if ((X & 0x3) == (Y & 0x3)) {
            return false;   // same axis specified
        }

```

```java
        // Z is "the other" axis, its sign is either +/- sign(X)*sign(Y)
        // this can be calculated by exclusive-or'ing X and Y; except for
        // the sign inversion (+/-) which is calculated below.
        int Z = X ^ Y;

        // extract the axis (remove the sign), offset in the range 0 to 2.
        final int x = (X & 0x3) - 1;
        final int y = (Y & 0x3) - 1;
        final int z = (Z & 0x3) - 1;

        // compute the sign of Z (whether it needs to be inverted)
        final int axis_y = (z + 1) % 3;
        final int axis_z = (z + 2) % 3;
        if (((x ^ axis_y) | (y ^ axis_z)) != 0) {
            Z ^= 0x80;
        }

        final boolean sx = (X >= 0x80);
        final boolean sy = (Y >= 0x80);
        final boolean sz = (Z >= 0x80);

        // Perform R * r, in avoiding actual muls and adds.
        final int rowLength = ((length == 16) ? 4 : 3);
        for (int j = 0; j < 3; j++) {
            final int offset = j * rowLength;
            for (int i = 0; i < 3; i++) {
                if (x == i)   outR[offset + i] = sx ? -inR[offset + 0] : inR[offset + 0];
                if (y == i)   outR[offset + i] = sy ? -inR[offset + 1] : inR[offset + 1];
                if (z == i)   outR[offset + i] = sz ? -inR[offset + 2] : inR[offset + 2];
            }
        }
        if (length == 16) {
            outR[3] = outR[7] = outR[11] = outR[12] = outR[13] = outR[14] = 0;
            outR[15] = 1;
        }
        return true;
    }
```

```
    /**
     * Computes the device's orientation based on the rotation matrix.
     * <p>
     * When it returns, the array values are as follows:
     * <ul>
     * <li>values[0]: <i>Azimuth</i>, angle of rotation about the -z axis.
     *                This value represents the angle between the device's y
     *                axis and the magnetic north pole. When facing north, this
     *                angle is 0, when facing south, this angle is &pi;.
     *                Likewise, when facing east, this angle is &pi;/2, and
     *                when facing west, this angle is -&pi;/2. The range of
     *                values is -&pi; to &pi;.</li>
     * <li>values[1]: <i>Pitch</i>, angle of rotation about the x axis.
     *                This value represents the angle between a plane parallel
     *                to the device's screen and a plane parallel to the ground.
     *                Assuming that the bottom edge of the device faces the
     *                user and that the screen is face-up, tilting the top edge
     *                of the device toward the ground creates a positive pitch
     *                angle. The range of values is -&pi;/2 to &pi;/2.</li>
     * <li>values[2]: <i>Roll</i>, angle of rotation about the y axis. This
     *                value represents the angle between a plane perpendicular
     *                to the device's screen and a plane perpendicular to the
     *                ground. Assuming that the bottom edge of the device faces
     *                the user and that the screen is face-up, tilting the left
     *                edge of the device toward the ground creates a positive
     *                roll angle. The range of values is -&pi; to &pi;.</li>
     * </ul>
     * <p>
     * Applying these three rotations in the azimuth, pitch, roll order
     * transforms an identity matrix to the rotation matrix passed into this
     * method. Also, note that all three orientation angles are expressed in
     * <b>radians</b>.
     *
     * @param R
     *        rotation matrix see {@link #getRotationMatrix}.
     *
```

```java
     * @param values
     *        an array of 3 floats to hold the result.
     *
     * @return The array values passed as argument.
     *
     * @see #getRotationMatrix(float[], float[], float[], float[])
     * @see GeomagneticField
     */
    public static float[] getOrientation(float[] R, float[] values) {
        /*
         * 4x4 (length=16) case:
         *   /  R[ 0]   R[ 1]   R[ 2]   0  \
         *   |  R[ 4]   R[ 5]   R[ 6]   0  |
         *   |  R[ 8]   R[ 9]   R[10]   0  |
         *   \      0       0       0   1  /
         *
         * 3x3 (length=9) case:
         *   /  R[ 0]   R[ 1]   R[ 2]  \
         *   |  R[ 3]   R[ 4]   R[ 5]  |
         *   \  R[ 6]   R[ 7]   R[ 8]  /
         *
         */
        if (R.length == 9) {
            values[0] = (float) Math.atan2(R[1], R[4]);
            values[1] = (float) Math.asin(-R[7]);
            values[2] = (float) Math.atan2(-R[6], R[8]);
        } else {
            values[0] = (float) Math.atan2(R[1], R[5]);
            values[1] = (float) Math.asin(-R[9]);
            values[2] = (float) Math.atan2(-R[8], R[10]);
        }

        return values;
    }

    /**
     * Computes the Altitude in meters from the atmospheric pressure and the
```

```java
     * pressure at sea level.
     * <p>
     * Typically the atmospheric pressure is read from a
     * {@link Sensor#TYPE_PRESSURE} sensor. The pressure at sea level must be
     * known, usually it can be retrieved from airport databases in the
     * vicinity. If unknown, you can use {@link #PRESSURE_STANDARD_ATMOSPHERE}
     * as an approximation, but absolute altitudes won't be accurate.
     * </p>
     * <p>
     * To calculate altitude differences, you must calculate the difference
     * between the altitudes at both points. If you don't know the altitude
     * as sea level, you can use {@link #PRESSURE_STANDARD_ATMOSPHERE} instead,
     * which will give good results considering the range of pressure typically
     * involved.
     * </p>
     * <p>
     * <code><ul>
     *  float altitude_difference =
     *      getAltitude(SensorManager.PRESSURE_STANDARD_ATMOSPHERE, pressure_at_point2)
     *      - getAltitude(SensorManager.PRESSURE_STANDARD_ATMOSPHERE, pressure_at_point1);
     * </ul></code>
     * </p>
     *
     * @param p0 pressure at sea level
     * @param p atmospheric pressure
     * @return Altitude in meters
     */
    public static float getAltitude(float p0, float p) {
        final float coef = 1.0f / 5.255f;
        return 44330.0f * (1.0f - (float) Math.pow(p / p0, coef));
    }

    /** Helper function to compute the angle change between two rotation matrices.
     *  Given a current rotation matrix (R) and a previous rotation matrix
     *  (prevR) computes the intrinsic rotation around the z, x, and y axes which
     *  transforms prevR to R.
     *  outputs a 3 element vector containing the z, x, and y angle
```

```java
     *  change at indexes 0, 1, and 2 respectively.
     * <p> Each input matrix is either as a 3x3 or 4x4 row-major matrix
     * depending on the length of the passed array:
     * <p>If the array length is 9, then the array elements represent this matrix
     * <pre>
     *   /  R[ 0]   R[ 1]   R[ 2]   \
     *   |  R[ 3]   R[ 4]   R[ 5]   |
     *   \  R[ 6]   R[ 7]   R[ 8]   /
     *</pre>
     * <p>If the array length is 16, then the array elements represent this matrix
     * <pre>
     *   /  R[ 0]   R[ 1]   R[ 2]   R[ 3]  \
     *   |  R[ 4]   R[ 5]   R[ 6]   R[ 7]  |
     *   |  R[ 8]   R[ 9]   R[10]   R[11]  |
     *   \  R[12]   R[13]   R[14]   R[15]  /
     *</pre>
     *
     * See {@link #getOrientation} for more detailed definition of the output.
     *
     * @param R current rotation matrix
     * @param prevR previous rotation matrix
     * @param angleChange an an array of floats (z, x, and y) in which the angle change
     *        (in radians) is stored
     */

    public static void getAngleChange(float[] angleChange, float[] R, float[] prevR) {
        float rd1 = 0, rd4 = 0, rd6 = 0, rd7 = 0, rd8 = 0;
        float ri0 = 0, ri1 = 0, ri2 = 0, ri3 = 0, ri4 = 0, ri5 = 0, ri6 = 0, ri7 = 0, ri8 = 0;
        float pri0 = 0, pri1 = 0, pri2 = 0, pri3 = 0, pri4 = 0;
        float pri5 = 0, pri6 = 0, pri7 = 0, pri8 = 0;

        if (R.length == 9) {
            ri0 = R[0];
            ri1 = R[1];
            ri2 = R[2];
            ri3 = R[3];
            ri4 = R[4];
```

```
                ri5 = R[5];
                ri6 = R[6];
                ri7 = R[7];
                ri8 = R[8];
            } else if (R.length == 16) {
                ri0 = R[0];
                ri1 = R[1];
                ri2 = R[2];
                ri3 = R[4];
                ri4 = R[5];
                ri5 = R[6];
                ri6 = R[8];
                ri7 = R[9];
                ri8 = R[10];
            }

            if (prevR.length == 9) {
                pri0 = prevR[0];
                pri1 = prevR[1];
                pri2 = prevR[2];
                pri3 = prevR[3];
                pri4 = prevR[4];
                pri5 = prevR[5];
                pri6 = prevR[6];
                pri7 = prevR[7];
                pri8 = prevR[8];
            } else if (prevR.length == 16) {
                pri0 = prevR[0];
                pri1 = prevR[1];
                pri2 = prevR[2];
                pri3 = prevR[4];
                pri4 = prevR[5];
                pri5 = prevR[6];
                pri6 = prevR[8];
                pri7 = prevR[9];
                pri8 = prevR[10];
            }
```

```java

        // calculate the parts of the rotation difference matrix we need
        // rd[i][j] = pri[0][i] * ri[0][j] + pri[1][i] * ri[1][j] + pri[2][i] * ri[2][j];

        rd1 = pri0 * ri1 + pri3 * ri4 + pri6 * ri7; //rd[0][1]
        rd4 = pri1 * ri1 + pri4 * ri4 + pri7 * ri7; //rd[1][1]
        rd6 = pri2 * ri0 + pri5 * ri3 + pri8 * ri6; //rd[2][0]
        rd7 = pri2 * ri1 + pri5 * ri4 + pri8 * ri7; //rd[2][1]
        rd8 = pri2 * ri2 + pri5 * ri5 + pri8 * ri8; //rd[2][2]

        angleChange[0] = (float) Math.atan2(rd1, rd4);
        angleChange[1] = (float) Math.asin(-rd7);
        angleChange[2] = (float) Math.atan2(-rd6, rd8);

    }

    /** Helper function to convert a rotation vector to a rotation matrix.
     *  Given a rotation vector (presumably from a ROTATION_VECTOR sensor), returns a
     *  9  or 16 element rotation matrix in the array R.  R must have length 9 or 16.
     *  If R.length == 9, the following matrix is returned:
     * <pre>
     *   /  R[ 0]   R[ 1]   R[ 2]   \
     *   |  R[ 3]   R[ 4]   R[ 5]   |
     *   \  R[ 6]   R[ 7]   R[ 8]   /
     *</pre>
     * If R.length == 16, the following matrix is returned:
     * <pre>
     *   /  R[ 0]   R[ 1]   R[ 2]   0  \
     *   |  R[ 4]   R[ 5]   R[ 6]   0  |
     *   |  R[ 8]   R[ 9]   R[10]   0  |
     *   \  0       0       0       1  /
     *</pre>
     *  @param rotationVector the rotation vector to convert
     *  @param R an array of floats in which to store the rotation matrix
     */
    public static void getRotationMatrixFromVector(float[] R, float[] rotationVector) {

```

```
        float q0;
        float q1 = rotationVector[0];
        float q2 = rotationVector[1];
        float q3 = rotationVector[2];

        if (rotationVector.length >= 4) {
            q0 = rotationVector[3];
        } else {
            q0 = 1 - q1 * q1 - q2 * q2 - q3 * q3;
            q0 = (q0 > 0) ? (float) Math.sqrt(q0) : 0;
        }

        float sq_q1 = 2 * q1 * q1;
        float sq_q2 = 2 * q2 * q2;
        float sq_q3 = 2 * q3 * q3;
        float q1_q2 = 2 * q1 * q2;
        float q3_q0 = 2 * q3 * q0;
        float q1_q3 = 2 * q1 * q3;
        float q2_q0 = 2 * q2 * q0;
        float q2_q3 = 2 * q2 * q3;
        float q1_q0 = 2 * q1 * q0;

        if (R.length == 9) {
            R[0] = 1 - sq_q2 - sq_q3;
            R[1] = q1_q2 - q3_q0;
            R[2] = q1_q3 + q2_q0;

            R[3] = q1_q2 + q3_q0;
            R[4] = 1 - sq_q1 - sq_q3;
            R[5] = q2_q3 - q1_q0;

            R[6] = q1_q3 - q2_q0;
            R[7] = q2_q3 + q1_q0;
            R[8] = 1 - sq_q1 - sq_q2;
        } else if (R.length == 16) {
            R[0] = 1 - sq_q2 - sq_q3;
            R[1] = q1_q2 - q3_q0;
```

```
            R[2] = q1_q3 + q2_q0;
            R[3] = 0.0f;

            R[4] = q1_q2 + q3_q0;
            R[5] = 1 - sq_q1 - sq_q3;
            R[6] = q2_q3 - q1_q0;
            R[7] = 0.0f;

            R[8] = q1_q3 - q2_q0;
            R[9] = q2_q3 + q1_q0;
            R[10] = 1 - sq_q1 - sq_q2;
            R[11] = 0.0f;

            R[12] = R[13] = R[14] = 0.0f;
            R[15] = 1.0f;
        }
    }

    /** Helper function to convert a rotation vector to a normalized quaternion.
     *  Given a rotation vector (presumably from a ROTATION_VECTOR sensor), returns a normalized
     *  quaternion in the array Q.  The quaternion is stored as [w, x, y, z]
     *  @param rv the rotation vector to convert
     *  @param Q an array of floats in which to store the computed quaternion
     */
    public static void getQuaternionFromVector(float[] Q, float[] rv) {
        if (rv.length >= 4) {
            Q[0] = rv[3];
        } else {
            Q[0] = 1 - rv[0] * rv[0] - rv[1] * rv[1] - rv[2] * rv[2];
            Q[0] = (Q[0] > 0) ? (float) Math.sqrt(Q[0]) : 0;
        }
        Q[1] = rv[0];
        Q[2] = rv[1];
        Q[3] = rv[2];
    }

    /**
```

```java
     * Requests receiving trigger events for a trigger sensor.
     *
     * <p>
     * When the sensor detects a trigger event condition, such as significant motion in
     * the case of the {@link Sensor#TYPE_SIGNIFICANT_MOTION}, the provided trigger listener
     * will be invoked once and then its request to receive trigger events will be canceled.
     * To continue receiving trigger events, the application must request to receive trigger
     * events again.
     * </p>
     *
     * @param listener The listener on which the
     *        {@link TriggerEventListener#onTrigger(TriggerEvent)} will be delivered.
     * @param sensor The sensor to be enabled.
     *
     * @return true if the sensor was successfully enabled.
     *
     * @throws IllegalArgumentException when sensor is null or not a trigger sensor.
     */
    public boolean requestTriggerSensor(TriggerEventListener listener, Sensor sensor) {
        return requestTriggerSensorImpl(listener, sensor);
    }

    /**
     * @hide
     */
    protected abstract boolean requestTriggerSensorImpl(TriggerEventListener listener,
            Sensor sensor);

    /**
     * Cancels receiving trigger events for a trigger sensor.
     *
     * <p>
     * Note that a Trigger sensor will be auto disabled if
     * {@link TriggerEventListener#onTrigger(TriggerEvent)} has triggered.
     * This method is provided in case the user wants to explicitly cancel the request
     * to receive trigger events.
     * </p>
```

```java
     *
     * @param listener The listener on which the
     *        {@link TriggerEventListener#onTrigger(TriggerEvent)}
     *        is delivered.It should be the same as the one used
     *        in {@link #requestTriggerSensor(TriggerEventListener, Sensor)}
     * @param sensor The sensor for which the trigger request should be canceled.
     *        If null, it cancels receiving trigger for all sensors associated
     *        with the listener.
     *
     * @return true if successfully canceled.
     *
     * @throws IllegalArgumentException when sensor is a trigger sensor.
     */
    public boolean cancelTriggerSensor(TriggerEventListener listener, Sensor sensor) {
        return cancelTriggerSensorImpl(listener, sensor, true);
    }

    /**
     * @hide
     */
    protected abstract boolean cancelTriggerSensorImpl(TriggerEventListener listener,
            Sensor sensor, boolean disable);


    /**
     * For testing purposes only. Not for third party applications.
     *
     * Initialize data injection mode and create a client for data injection. SensorService should
     * already be operating in DATA_INJECTION mode for this call succeed. To set SensorService into
     * DATA_INJECTION mode "adb shell dumpsys sensorservice data_injection" needs to be called
     * through adb. Typically this is done using a host side test.  This mode is expected to be used
     * only for testing purposes. If the HAL is set to data injection mode, it will ignore the input
     * from physical sensors and read sensor data that is injected from the test application. This
     * mode is used for testing vendor implementations for various algorithms like Rotation Vector,
     * Significant Motion, Step Counter etc. Not all HALs support DATA_INJECTION. This method will
     * fail in those cases. Once this method succeeds, the test can call
     * {@link injectSensorData(Sensor, float[], int, long)} to inject sensor data into the HAL.
```

```
     *
     * @param enable True to initialize a client in DATA_INJECTION mode.
     *               False to clean up the native resources.
     *
     * @return true if the HAL supports data injection and false
     *         otherwise.
     * @hide
     */
    @SystemApi
    public boolean initDataInjection(boolean enable) {
        return initDataInjectionImpl(enable);
    }

    /**
     * @hide
     */
    protected abstract boolean initDataInjectionImpl(boolean enable);

    /**
     * For testing purposes only. Not for third party applications.
     *
     * This method is used to inject raw sensor data into the HAL.  Call {@link
     * initDataInjection(boolean)} before this method to set the HAL in data injection mode. This
     * method should be called only if a previous call to initDataInjection has been successful and
     * the HAL and SensorService are already operating in data injection mode.
     *
     * @param sensor The sensor to inject.
     * @param values Sensor values to inject. The length of this
     *               array must be exactly equal to the number of
     *               values reported by the sensor type.
     * @param accuracy Accuracy of the sensor.
     * @param timestamp Sensor timestamp associated with the event.
     *
     * @return boolean True if the data injection succeeds, false
     *         otherwise.
     * @throws IllegalArgumentException when the sensor is null,
     *         data injection is not supported by the sensor, values
```

```java
     *         are null, incorrect number of values for the sensor,
     *         sensor accuracy is incorrect or timestamps are
     *         invalid.
     * @hide
     */
    @SystemApi
    public boolean injectSensorData(Sensor sensor, float[] values, int accuracy,
                long timestamp) {
        if (sensor == null) {
            throw new IllegalArgumentException("sensor cannot be null");
        }
        if (!sensor.isDataInjectionSupported()) {
            throw new IllegalArgumentException("sensor does not support data injection");
        }
        if (values == null) {
            throw new IllegalArgumentException("sensor data cannot be null");
        }
        int expectedNumValues = Sensor.getMaxLengthValuesArray(sensor, Build.VERSION_CODES.M);
        if (values.length != expectedNumValues) {
            throw new  IllegalArgumentException("Wrong number of values for sensor "
                    + sensor.getName() + " actual=" + values.length + " expected="
                    + expectedNumValues);
        }
        if (accuracy < SENSOR_STATUS_NO_CONTACT || accuracy > SENSOR_STATUS_ACCURACY_HIGH) {
            throw new IllegalArgumentException("Invalid sensor accuracy");
        }
        if (timestamp <= 0) {
            throw new IllegalArgumentException("Negative or zero sensor timestamp");
        }
        return injectSensorDataImpl(sensor, values, accuracy, timestamp);
    }

    /**
     * @hide
     */
    protected abstract boolean injectSensorDataImpl(Sensor sensor, float[] values, int accuracy,
                long timestamp);
```

```java

    private LegacySensorManager getLegacySensorManager() {
        synchronized (mSensorListByType) {
            if (mLegacySensorManager == null) {
                Log.i(TAG, "This application is using deprecated SensorManager API which will "
                        + "be removed someday.  Please consider switching to the new API.");
                mLegacySensorManager = new LegacySensorManager(this);
            }
            return mLegacySensorManager;
        }
    }

    private static int getDelay(int rate) {
        int delay = -1;
        switch (rate) {
            case SENSOR_DELAY_FASTEST:
                delay = 0;
                break;
            case SENSOR_DELAY_GAME:
                delay = 20000;
                break;
            case SENSOR_DELAY_UI:
                delay = 66667;
                break;
            case SENSOR_DELAY_NORMAL:
                delay = 200000;
                break;
            default:
                delay = rate;
                break;
        }
        return delay;
    }

    /** @hide */
    public boolean setOperationParameter(SensorAdditionalInfo parameter) {
        return setOperationParameterImpl(parameter);
```

```java
    }

    /** @hide */
    protected abstract boolean setOperationParameterImpl(SensorAdditionalInfo parameter);
}
```