

[android](#) / [platform](#) / [frameworks](#) / [base](#) / [refs/heads/main](#) / [.](#) / [core](#) / [java](#) / [android](#) / [hardware](#) /

## SensorManager.java

blob: f05397669b34d196e87a2d36e77456685245a8b2 [[file](#)] [[log](#)] [[blame](#)]

```
1  /*
2   * Copyright (C) 2008 The Android Open Source Project
3   *
4   * Licensed under the Apache License, Version 2.0 (the "License");
5   * you may not use this file except in compliance with the License.
6   * You may obtain a copy of the License at
7   *
8   *     http://www.apache.org/licenses/LICENSE-2.0
9   *
10  * Unless required by applicable law or agreed to in writing, software
11  * distributed under the License is distributed on an "AS IS" BASIS,
12  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13  * See the License for the specific language governing permissions and
14  * limitations under the License.
15  */
16
17 package android.hardware;
18
19 import android.annotation.Nullable;
20 import android.annotation.SystemApi;
21 import android.annotation.SystemService;
22 import android.compat.annotation.UnsupportedAppUsage;
23 import android.content.Context;
24 import android.os.Build;
25 import android.os.Handler;
26 import android.os.MemoryFile;
27 import android.util.Log;
28 import android.util.SparseArray;
```

```
29
30 import java.util.ArrayList;
31 import java.util.Collections;
32 import java.util.List;
33
34 /**
35  * <p>
36  * SensorManager lets you access the device's {@link android.hardware.Sensor
37  * sensors}.
38  * </p>
39  * <p>
40  * Always make sure to disable sensors you don't need, especially when your
41  * activity is paused. Failing to do so can drain the battery in just a few
42  * hours. Note that the system will not disable sensors automatically when
43  * the screen turns off.
44  * </p>
45  * <p class="note">
46  * Note: Don't use this mechanism with a Trigger Sensor, have a look
47  * at {@link TriggerEventListener}. {@link Sensor#TYPE_SIGNIFICANT_MOTION}
48  * is an example of a trigger sensor.
49  * </p>
50  * <p>
51  * In order to access sensor data at high sampling rates (i.e. greater than 200 Hz
52  * for {@link SensorEventListener} and greater than {@link SensorDirectChannel#RATE_NORMAL}
53  * for {@link SensorDirectChannel}), apps must declare
54  * the {@link android.Manifest.permission#HIGH_SAMPLING_RATE_SENSORS} permission
55  * in their AndroidManifest.xml file.
56  * </p>
57  * <pre class="prettyprint">
58  * public class SensorActivity extends Activity implements SensorEventListener {
59  *     private final SensorManager mSensorManager;
60  *     private final Sensor mAccelerometer;
61  *
62  *     public SensorActivity() {
63  *         mSensorManager = (SensorManager) getSystemService(SENSOR_SERVICE);
64  *         mAccelerometer = mSensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
65  *     }

```

```

66  *
67  *     protected void onResume() {
68  *         super.onResume();
69  *         mSensorManager.registerListener(this, mAccelerometer, SensorManager.SENSOR_DELAY_NORMAL);
70  *     }
71  *
72  *     protected void onPause() {
73  *         super.onPause();
74  *         mSensorManager.unregisterListener(this);
75  *     }
76  *
77  *     public void onAccuracyChanged(Sensor sensor, int accuracy) {
78  *     }
79  *
80  *     public void onSensorChanged(SensorEvent event) {
81  *     }
82  * }
83  * </pre>
84  *
85  * @see SensorEventListener
86  * @see SensorEvent
87  * @see Sensor
88  *
89  */
90 @SystemService(Context.SENSOR_SERVICE)
91 public abstract class SensorManager {
92     /** @hide */
93     protected static final String TAG = "SensorManager";
94
95     private static final float[] sTempMatrix = new float[16];
96
97     // Cached lists of sensors by type. Guarded by mSensorListByType.
98     private final SparseArray<List<Sensor>> mSensorListByType =
99         new SparseArray<List<Sensor>>();
100
101     // Legacy sensor manager implementation. Guarded by mSensorListByType during initialization.
102     private LegacySensorManager mLegacySensorManager;

```

```
103
104     /* NOTE: sensor IDs must be a power of 2 */
105
106     /**
107      * A constant describing an orientation sensor. See
108      * {@link android.hardware.SensorListener SensorListener} for more details.
109      *
110      * @deprecated use {@link android.hardware.Sensor Sensor} instead.
111      */
112     @Deprecated
113     public static final int SENSOR_ORIENTATION = 1 << 0;
114
115     /**
116      * A constant describing an accelerometer. See
117      * {@link android.hardware.SensorListener SensorListener} for more details.
118      *
119      * @deprecated use {@link android.hardware.Sensor Sensor} instead.
120      */
121     @Deprecated
122     public static final int SENSOR_ACCELEROMETER = 1 << 1;
123
124     /**
125      * A constant describing a temperature sensor See
126      * {@link android.hardware.SensorListener SensorListener} for more details.
127      *
128      * @deprecated use {@link android.hardware.Sensor Sensor} instead.
129      */
130     @Deprecated
131     public static final int SENSOR_TEMPERATURE = 1 << 2;
132
133     /**
134      * A constant describing a magnetic sensor See
135      * {@link android.hardware.SensorListener SensorListener} for more details.
136      *
137      * @deprecated use {@link android.hardware.Sensor Sensor} instead.
138      */
139     @Deprecated
```

```
140 public static final int SENSOR_MAGNETIC_FIELD = 1 << 3;
141
142 /**
143  * A constant describing an ambient light sensor See
144  * {@link android.hardware.SensorListener SensorListener} for more details.
145  *
146  * @deprecated use {@link android.hardware.Sensor Sensor} instead.
147  */
148 @Deprecated
149 public static final int SENSOR_LIGHT = 1 << 4;
150
151 /**
152  * A constant describing a proximity sensor See
153  * {@link android.hardware.SensorListener SensorListener} for more details.
154  *
155  * @deprecated use {@link android.hardware.Sensor Sensor} instead.
156  */
157 @Deprecated
158 public static final int SENSOR_PROXIMITY = 1 << 5;
159
160 /**
161  * A constant describing a Tricorder See
162  * {@link android.hardware.SensorListener SensorListener} for more details.
163  *
164  * @deprecated use {@link android.hardware.Sensor Sensor} instead.
165  */
166 @Deprecated
167 public static final int SENSOR_TRICORDER = 1 << 6;
168
169 /**
170  * A constant describing an orientation sensor. See
171  * {@link android.hardware.SensorListener SensorListener} for more details.
172  *
173  * @deprecated use {@link android.hardware.Sensor Sensor} instead.
174  */
175 @Deprecated
176 public static final int SENSOR_ORIENTATION_RAW = 1 << 7;
```

```
177
178 /**
179  * A constant that includes all sensors
180  *
181  * @deprecated use {@link android.hardware.Sensor Sensor} instead.
182  */
183 @Deprecated
184 public static final int SENSOR_ALL = 0x7F;
185
186 /**
187  * Smallest sensor ID
188  *
189  * @deprecated use {@link android.hardware.Sensor Sensor} instead.
190  */
191 @Deprecated
192 public static final int SENSOR_MIN = SENSOR_ORIENTATION;
193
194 /**
195  * Largest sensor ID
196  *
197  * @deprecated use {@link android.hardware.Sensor Sensor} instead.
198  */
199 @Deprecated
200 public static final int SENSOR_MAX = ((SENSOR_ALL + 1) >> 1);
201
202
203 /**
204  * Index of the X value in the array returned by
205  * {@link android.hardware.SensorListener#onSensorChanged}
206  *
207  * @deprecated use {@link android.hardware.Sensor Sensor} instead.
208  */
209 @Deprecated
210 public static final int DATA_X = 0;
211
212 /**
213  * Index of the Y value in the array returned by
```

```
214     * {@link android.hardware.SensorListener#onSensorChanged}
215     *
216     * @deprecated use {@link android.hardware.Sensor Sensor} instead.
217     */
218     @Deprecated
219     public static final int DATA_Y = 1;
220
221     /**
222     * Index of the Z value in the array returned by
223     * {@link android.hardware.SensorListener#onSensorChanged}
224     *
225     * @deprecated use {@link android.hardware.Sensor Sensor} instead.
226     */
227     @Deprecated
228     public static final int DATA_Z = 2;
229
230     /**
231     * Offset to the untransformed values in the array returned by
232     * {@link android.hardware.SensorListener#onSensorChanged}
233     *
234     * @deprecated use {@link android.hardware.Sensor Sensor} instead.
235     */
236     @Deprecated
237     public static final int RAW_DATA_INDEX = 3;
238
239     /**
240     * Index of the untransformed X value in the array returned by
241     * {@link android.hardware.SensorListener#onSensorChanged}
242     *
243     * @deprecated use {@link android.hardware.Sensor Sensor} instead.
244     */
245     @Deprecated
246     public static final int RAW_DATA_X = 3;
247
248     /**
249     * Index of the untransformed Y value in the array returned by
250     * {@link android.hardware.SensorListener#onSensorChanged}
```

```

251     *
252     * @deprecated use {@link android.hardware.Sensor Sensor} instead.
253     */
254     @Deprecated
255     public static final int RAW_DATA_Y = 4;
256
257     /**
258     * Index of the untransformed Z value in the array returned by
259     * {@link android.hardware.SensorListener#onSensorChanged}
260     *
261     * @deprecated use {@link android.hardware.Sensor Sensor} instead.
262     */
263     @Deprecated
264     public static final int RAW_DATA_Z = 5;
265
266     /** Standard gravity (g) on Earth. This value is equivalent to 1G */
267     public static final float STANDARD_GRAVITY = 9.80665f;
268
269     /** Sun's gravity in SI units (m/s^2) */
270     public static final float GRAVITY_SUN = 275.0f;
271     /** Mercury's gravity in SI units (m/s^2) */
272     public static final float GRAVITY_MERCURY = 3.70f;
273     /** Venus' gravity in SI units (m/s^2) */
274     public static final float GRAVITY_VENUS = 8.87f;
275     /** Earth's gravity in SI units (m/s^2) */
276     public static final float GRAVITY_EARTH = 9.80665f;
277     /** The Moon's gravity in SI units (m/s^2) */
278     public static final float GRAVITY_MOON = 1.6f;
279     /** Mars' gravity in SI units (m/s^2) */
280     public static final float GRAVITY_MARS = 3.71f;
281     /** Jupiter's gravity in SI units (m/s^2) */
282     public static final float GRAVITY_JUPITER = 23.12f;
283     /** Saturn's gravity in SI units (m/s^2) */
284     public static final float GRAVITY_SATURN = 8.96f;
285     /** Uranus' gravity in SI units (m/s^2) */
286     public static final float GRAVITY_URANUS = 8.69f;
287     /** Neptune's gravity in SI units (m/s^2) */

```



```
288 public static final float GRAVITY_NEPTUNE = 11.0f;
289 /** Pluto's gravity in SI units (m/s^2) */
290 public static final float GRAVITY_PLUTO = 0.6f;
291 /** Gravity (estimate) on the first Death Star in Empire units (m/s^2) */
292 public static final float GRAVITY_DEATH_STAR_I = 0.000000353036145f;
293 /** Gravity on the island */
294 public static final float GRAVITY_THE_ISLAND = 4.815162342f;
295
296
297 /** Maximum magnetic field on Earth's surface */
298 public static final float MAGNETIC_FIELD_EARTH_MAX = 60.0f;
299 /** Minimum magnetic field on Earth's surface */
300 public static final float MAGNETIC_FIELD_EARTH_MIN = 30.0f;
301
302
303 /** Standard atmosphere, or average sea-level pressure in hPa (millibar) */
304 public static final float PRESSURE_STANDARD_ATMOSPHERE = 1013.25f;
305
306
307 /** Maximum luminance of sunlight in lux */
308 public static final float LIGHT_SUNLIGHT_MAX = 120000.0f;
309 /** luminance of sunlight in lux */
310 public static final float LIGHT_SUNLIGHT = 110000.0f;
311 /** luminance in shade in lux */
312 public static final float LIGHT_SHADE = 20000.0f;
313 /** luminance under an overcast sky in lux */
314 public static final float LIGHT_OVERCAST = 10000.0f;
315 /** luminance at sunrise in lux */
316 public static final float LIGHT_SUNRISE = 400.0f;
317 /** luminance under a cloudy sky in lux */
318 public static final float LIGHT_CLOUDY = 100.0f;
319 /** luminance at night with full moon in lux */
320 public static final float LIGHT_FULLMOON = 0.25f;
321 /** luminance at night with no moon in lux */
322 public static final float LIGHT_NO_MOON = 0.001f;
323
324
```

```
325  /** get sensor data as fast as possible */
326  public static final int SENSOR_DELAY_FASTEST = 0;
327  /** rate suitable for games */
328  public static final int SENSOR_DELAY_GAME = 1;
329  /** rate suitable for the user interface */
330  public static final int SENSOR_DELAY_UI = 2;
331  /** rate (default) suitable for screen orientation changes */
332  public static final int SENSOR_DELAY_NORMAL = 3;
333
334
335  /**
336   * The values returned by this sensor cannot be trusted because the sensor
337   * had no contact with what it was measuring (for example, the heart rate
338   * monitor is not in contact with the user).
339   */
340  public static final int SENSOR_STATUS_NO_CONTACT = -1;
341
342  /**
343   * The values returned by this sensor cannot be trusted, calibration is
344   * needed or the environment doesn't allow readings
345   */
346  public static final int SENSOR_STATUS_UNRELIABLE = 0;
347
348  /**
349   * This sensor is reporting data with low accuracy, calibration with the
350   * environment is needed
351   */
352  public static final int SENSOR_STATUS_ACCURACY_LOW = 1;
353
354  /**
355   * This sensor is reporting data with an average level of accuracy,
356   * calibration with the environment may improve the readings
357   */
358  public static final int SENSOR_STATUS_ACCURACY_MEDIUM = 2;
359
360  /** This sensor is reporting data with maximum accuracy */
361  public static final int SENSOR_STATUS_ACCURACY_HIGH = 3;
```

```
362
363     /** see {@link #remapCoordinateSystem} */
364     public static final int AXIS_X = 1;
365     /** see {@link #remapCoordinateSystem} */
366     public static final int AXIS_Y = 2;
367     /** see {@link #remapCoordinateSystem} */
368     public static final int AXIS_Z = 3;
369     /** see {@link #remapCoordinateSystem} */
370     public static final int AXIS_MINUS_X = AXIS_X | 0x80;
371     /** see {@link #remapCoordinateSystem} */
372     public static final int AXIS_MINUS_Y = AXIS_Y | 0x80;
373     /** see {@link #remapCoordinateSystem} */
374     public static final int AXIS_MINUS_Z = AXIS_Z | 0x80;
375
376
377     /**
378      * {@hide}
379      */
380     @UnsupportedAppUsage
381     public SensorManager() {
382     }
383
384     /**
385      * Gets the full list of sensors that are available.
386      * @hide
387      */
388     protected abstract List<Sensor> getFullSensorList();
389
390     /**
391      * Gets the full list of dynamic sensors that are available.
392      * @hide
393      */
394     protected abstract List<Sensor> getFullDynamicSensorList();
395
396     /**
397      * @return available sensors.
398      * @deprecated This method is deprecated, use
```

```

399         *      {@link SensorManager#getSensorList(int)} instead
400     */
401     @Deprecated
402     public int getSensors() {
403         return getLegacySensorManager().getSensors();
404     }
405
406     /**
407      * Use this method to get the list of available sensors of a certain type.
408      * Make multiple calls to get sensors of different types or use
409      * {@link android.hardware.Sensor#TYPE_ALL Sensor.TYPE_ALL} to get all the
410      * sensors. Note that the {@link android.hardware.Sensor#getName()} is
411      * expected to yield a value that is unique across any sensors that return
412      * the same value for {@link android.hardware.Sensor#getType()}.
413      *
414      * <p class="note">
415      * NOTE: Both wake-up and non wake-up sensors matching the given type are
416      * returned. Check {@link Sensor#isWakeUpSensor()} to know the wake-up properties
417      * of the returned {@link Sensor}.
418      * </p>
419      *
420      * @param type
421      *         of sensors requested
422      *
423      * @return a list of sensors matching the asked type.
424      *
425      * @see #getDefaultSensor(int)
426      * @see Sensor
427      */
428     public List<Sensor> getSensorList(int type) {
429         // cache the returned lists the first time
430         List<Sensor> list;
431         final List<Sensor> fullList = getFullSensorList();
432         synchronized (mSensorListByType) {
433             list = mSensorListByType.get(type);
434             if (list == null) {
435                 if (type == Sensor.TYPE_ALL) {

```

```

436         list = fullList;
437     } else {
438         list = new ArrayList<Sensor>();
439         for (Sensor i : fullList) {
440             if (i.getType() == type) {
441                 list.add(i);
442             }
443         }
444     }
445     list = Collections.unmodifiableList(list);
446     mSensorListByType.append(type, list);
447 }
448 }
449 return list;
450 }
451
452 /**
453  * Use this method to get a list of available dynamic sensors of a certain type.
454  * Make multiple calls to get sensors of different types or use
455  * {@link android.hardware.Sensor#TYPE_ALL Sensor.TYPE_ALL} to get all dynamic sensors.
456  *
457  * <p class="note">
458  * NOTE: Both wake-up and non wake-up sensors matching the given type are
459  * returned. Check {@link Sensor#isWakeUpSensor()} to know the wake-up properties
460  * of the returned {@link Sensor}.
461  * </p>
462  *
463  * @param type of sensors requested
464  *
465  * @return a list of dynamic sensors matching the requested type.
466  *
467  * @see Sensor
468  */
469 public List<Sensor> getDynamicSensorList(int type) {
470     // cache the returned lists the first time
471     final List<Sensor> fullList = getFullDynamicSensorList();
472     if (type == Sensor.TYPE_ALL) {

```

```

473         return Collections.unmodifiableList(fullList);
474     } else {
475         List<Sensor> list = new ArrayList();
476         for (Sensor i : fullList) {
477             if (i.getType() == type) {
478                 list.add(i);
479             }
480         }
481         return Collections.unmodifiableList(list);
482     }
483 }
484
485 /**
486  * Use this method to get the default sensor for a given type. Note that the
487  * returned sensor could be a composite sensor, and its data could be
488  * averaged or filtered. If you need to access the raw sensors use
489  * {@link SensorManager#getSensorList(int) getSensorList}.
490  *
491  * @param type
492  *         of sensors requested
493  *
494  * @return the default sensor matching the requested type if one exists and the application
495  *         has the necessary permissions, or null otherwise.
496  *
497  * @see #getSensorList(int)
498  * @see Sensor
499  */
500 public @Nullable Sensor getDefaultSensor(int type) {
501     // TODO: need to be smarter, for now, just return the 1st sensor
502     List<Sensor> l = getSensorList(type);
503     boolean wakeUpSensor = false;
504     // For the following sensor types, return a wake-up sensor. These types are by default
505     // defined as wake-up sensors. For the rest of the SDK defined sensor types return a
506     // non_wake-up version.
507     if (type == Sensor.TYPE_PROXIMITY || type == Sensor.TYPE_SIGNIFICANT_MOTION
508         || type == Sensor.TYPE_TILT_DETECTOR || type == Sensor.TYPE_WAKE_GESTURE
509         || type == Sensor.TYPE_GLANCE_GESTURE || type == Sensor.TYPE_PICK_UP_GESTURE

```

```

510         || type == Sensor.TYPE_LOW_LATENCY_OFFBODY_DETECT
511         || type == Sensor.TYPE_WRIST_TILT_GESTURE
512         || type == Sensor.TYPE_DYNAMIC_SENSOR_META || type == Sensor.TYPE_HINGE_ANGLE) {
513     wakeUpSensor = true;
514 }
515
516     for (Sensor sensor : l) {
517         if (sensor.isWakeUpSensor() == wakeUpSensor) return sensor;
518     }
519     return null;
520 }
521
522 /**
523  * Return a Sensor with the given type and wakeUp properties. If multiple sensors of this
524  * type exist, any one of them may be returned.
525  * <p>
526  * For example,
527  * <ul>
528  *     <li>getDefaultSensor({@link Sensor#TYPE_ACCELEROMETER}, true) returns a wake-up
529  *     accelerometer sensor if it exists. </li>
530  *     <li>getDefaultSensor({@link Sensor#TYPE_PROXIMITY}, false) returns a non wake-up
531  *     proximity sensor if it exists. </li>
532  *     <li>getDefaultSensor({@link Sensor#TYPE_PROXIMITY}, true) returns a wake-up proximity
533  *     sensor which is the same as the Sensor returned by {@link #getDefaultSensor(int)}. </li>
534  * </ul>
535  * </p>
536  * <p class="note">
537  * Note: Sensors like {@link Sensor#TYPE_PROXIMITY} and {@link Sensor#TYPE_SIGNIFICANT_MOTION}
538  * are declared as wake-up sensors by default.
539  * </p>
540  * @param type
541  *         type of sensor requested
542  * @param wakeUp
543  *         flag to indicate whether the Sensor is a wake-up or non wake-up sensor.
544  * @return the default sensor matching the requested type and wakeUp properties if one exists
545  *         and the application has the necessary permissions, or null otherwise.
546  * @see Sensor#isWakeUpSensor()

```

```

547     */
548     public @Nullable Sensor getDefaultSensor(int type, boolean wakeUp) {
549         List<Sensor> l = getSensorList(type);
550         for (Sensor sensor : l) {
551             if (sensor.isWakeUpSensor() == wakeUp) {
552                 return sensor;
553             }
554         }
555         return null;
556     }
557
558     /**
559     * Registers a listener for given sensors.
560     *
561     * @deprecated This method is deprecated, use
562     *             {@link SensorManager#registerListener(SensorEventListener, Sensor, int)}
563     *             instead.
564     *
565     * @param listener
566     *         sensor listener object
567     *
568     * @param sensors
569     *         a bit masks of the sensors to register to
570     *
571     * @return <code>true</code> if the sensor is supported and successfully
572     *         enabled
573     */
574     @Deprecated
575     public boolean registerListener(SensorListener listener, int sensors) {
576         return registerListener(listener, sensors, SENSOR_DELAY_NORMAL);
577     }
578
579     /**
580     * Registers a SensorListener for given sensors.
581     *
582     * @deprecated This method is deprecated, use
583     *             {@link SensorManager#registerListener(SensorEventListener, Sensor, int)}

```



```

584         *         instead.
585     *
586     * @param listener
587     *         sensor listener object
588     *
589     * @param sensors
590     *         a bit masks of the sensors to register to
591     *
592     * @param rate
593     *         rate of events. This is only a hint to the system. events may be
594     *         received faster or slower than the specified rate. Usually events
595     *         are received faster. The value must be one of
596     *         {@link #SENSOR_DELAY_NORMAL}, {@link #SENSOR_DELAY_UI},
597     *         {@link #SENSOR_DELAY_GAME}, or {@link #SENSOR_DELAY_FASTEST}.
598     *
599     * @return <code>true</code> if the sensor is supported and successfully
600     *         enabled
601     */
602     @Deprecated
603     public boolean registerListener(SensorListener listener, int sensors, int rate) {
604         return getLegacySensorManager().registerListener(listener, sensors, rate);
605     }
606
607     /**
608     * Unregisters a listener for all sensors.
609     *
610     * @deprecated This method is deprecated, use
611     *         {@link SensorManager#unregisterListener(SensorEventListener)}
612     *         instead.
613     *
614     * @param listener
615     *         a SensorListener object
616     */
617     @Deprecated
618     public void unregisterListener(SensorListener listener) {
619         unregisterListener(listener, SENSOR_ALL | SENSOR_ORIENTATION_RAW);
620     }

```

```

621
622 /**
623  * Unregisters a listener for the sensors with which it is registered.
624  *
625  * @deprecated This method is deprecated, use
626  *             {@link SensorManager#unregisterListener(SensorEventListener, Sensor)}
627  *             instead.
628  *
629  * @param listener
630  *       a SensorListener object
631  *
632  * @param sensors
633  *       a bit masks of the sensors to unregister from
634  */
635 @Deprecated
636 public void unregisterListener(SensorListener listener, int sensors) {
637     getLegacySensorManager().unregisterListener(listener, sensors);
638 }
639
640 /**
641  * Unregisters a listener for the sensors with which it is registered.
642  *
643  * <p class="note">
644  * Note: Don't use this method with a one shot trigger sensor such as
645  * {@link Sensor#TYPE_SIGNIFICANT_MOTION}.
646  * Use {@link #cancelTriggerSensor(TriggerEventListener, Sensor)} instead.
647  * </p>
648  *
649  * @param listener
650  *       a SensorEventListener object
651  *
652  * @param sensor
653  *       the sensor to unregister from
654  *
655  * @see #unregisterListener(SensorEventListener)
656  * @see #registerListener(SensorEventListener, Sensor, int)
657  */

```

```

658     public void unregisterListener(SensorEventListener listener, Sensor sensor) {
659         if (listener == null || sensor == null) {
660             return;
661         }
662
663         unregisterListenerImpl(listener, sensor);
664     }
665
666     /**
667      * Unregisters a listener for all sensors.
668      *
669      * @param listener
670      *      a SensorListener object
671      *
672      * @see #unregisterListener(SensorEventListener, Sensor)
673      * @see #registerListener(SensorEventListener, Sensor, int)
674      *
675      */
676     public void unregisterListener(SensorEventListener listener) {
677         if (listener == null) {
678             return;
679         }
680
681         unregisterListenerImpl(listener, null);
682     }
683
684     /** @hide */
685     protected abstract void unregisterListenerImpl(SensorEventListener listener, Sensor sensor);
686
687     /**
688      * Registers a {@link android.hardware.SensorEventListener SensorEventListener} for the given
689      * sensor at the given sampling frequency.
690      * <p>
691      * The events will be delivered to the provided {@code SensorEventListener} as soon as they are
692      * available. To reduce the power consumption, applications can use
693      * {@link #registerListener(SensorEventListener, Sensor, int, int)} instead and specify a
694      * 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

```

732     *      from Android 2.3 (API level 9) onwards. For earlier releases, you must use one of
733     *      the {@code SENSOR_DELAY_*} constants.
734     * @return <code>true</code> if the sensor is supported and successfully enabled.
735     * @see #registerListener(SensorEventListener, Sensor, int, Handler)
736     * @see #unregisterListener(SensorEventListener)
737     * @see #unregisterListener(SensorEventListener, Sensor)
738     */
739     public boolean registerListener(SensorEventListener listener, Sensor sensor,
740         int samplingPeriodUs) {
741         return registerListener(listener, sensor, samplingPeriodUs, null);
742     }
743
744     /**
745     * Registers a {@link android.hardware.SensorEventListener SensorEventListener} for the given
746     * sensor at the given sampling frequency and the given maximum reporting latency.
747     * <p>
748     * This function is similar to {@link #registerListener(SensorEventListener, Sensor, int)} but
749     * it allows events to stay temporarily in the hardware FIFO (queue) before being delivered. The
750     * events can be stored in the hardware FIFO up to {@code maxReportLatencyUs} microseconds. Once
751     * one of the events in the FIFO needs to be reported, all of the events in the FIFO are
752     * reported sequentially. This means that some events will be reported before the maximum
753     * reporting latency has elapsed.
754     * </p><p>
755     * When {@code maxReportLatencyUs} is 0, the call is equivalent to a call to
756     * {@link #registerListener(SensorEventListener, Sensor, int)}, as it requires the events to be
757     * delivered as soon as possible.
758     * </p><p>
759     * When {@code sensor.maxFifoEventCount()} is 0, the sensor does not use a FIFO, so the call
760     * will also be equivalent to {@link #registerListener(SensorEventListener, Sensor, int)}.
761     * </p><p>
762     * Setting {@code maxReportLatencyUs} to a positive value allows to reduce the number of
763     * interrupts the AP (Application Processor) receives, hence reducing power consumption, as the
764     * AP can switch to a lower power state while the sensor is capturing the data. This is
765     * especially important when registering to wake-up sensors, for which each interrupt causes the
766     * AP to wake up if it was in suspend mode. See {@link Sensor#isWakeUpSensor()} for more
767     * information on wake-up sensors.
768     * </p>

```

```

769 * <p class="note">
770 * </p>
771 * Note: Don't use this method with one-shot trigger sensors such as
772 * {@link Sensor#TYPE_SIGNIFICANT_MOTION}. Use
773 * {@link #requestTriggerSensor(TriggerEventListener, Sensor)} instead. </p>
774 *
775 * @param listener A {@link android.hardware.SensorEventListener SensorEventListener} object
776 *         that will receive the sensor events. If the application is interested in receiving
777 *         flush complete notifications, it should register with
778 *         {@link android.hardware.SensorEventListener SensorEventListener2} instead.
779 * @param sensor The {@link android.hardware.Sensor Sensor} to register to.
780 * @param samplingPeriodUs The desired delay between two consecutive events in microseconds.
781 *         This is only a hint to the system. Events may be received faster or slower than
782 *         the specified rate. Usually events are received faster. Can be one of
783 *         {@link #SENSOR_DELAY_NORMAL}, {@link #SENSOR_DELAY_UI},
784 *         {@link #SENSOR_DELAY_GAME}, {@link #SENSOR_DELAY_FASTEST} or the delay in
785 *         microseconds.
786 * @param maxReportLatencyUs Maximum time in microseconds that events can be delayed before
787 *         being reported to the application. A large value allows reducing the power
788 *         consumption associated with the sensor. If maxReportLatencyUs is set to zero,
789 *         events are delivered as soon as they are available, which is equivalent to calling
790 *         {@link #registerListener(SensorEventListener, Sensor, int)}.
791 * @return <code>true</code> if the sensor is supported and successfully enabled.
792 * @see #registerListener(SensorEventListener, Sensor, int)
793 * @see #unregisterListener(SensorEventListener)
794 * @see #flush(SensorEventListener)
795 */
796 public boolean registerListener(SensorEventListener listener, Sensor sensor,
797         int samplingPeriodUs, int maxReportLatencyUs) {
798     int delay = getDelay(samplingPeriodUs);
799     return registerListenerImpl(listener, sensor, delay, null, maxReportLatencyUs, 0);
800 }
801
802 /**
803 * Registers a {@link android.hardware.SensorEventListener SensorEventListener} for the given
804 * sensor. Events are delivered in continuous mode as soon as they are available. To reduce the
805 * power consumption, applications can use

```

```

806 * {@link #registerListener(SensorEventListener, Sensor, int, int)} instead and specify a
807 * positive non-zero maximum reporting latency.
808 * <p class="note">
809 * </p>
810 * Note: Don't use this method with a one shot trigger sensor such as
811 * {@link Sensor#TYPE_SIGNIFICANT_MOTION}. Use
812 * {@link #requestTriggerSensor(TriggerEventListener, Sensor)} instead. </p>
813 *
814 * @param listener A {@link android.hardware.SensorEventListener SensorEventListener} object.
815 * @param sensor The {@link android.hardware.Sensor Sensor} to register to.
816 * @param samplingPeriodUs The rate {@link android.hardware.SensorEvent sensor events} are
817 *         delivered at. This is only a hint to the system. Events may be received faster or
818 *         slower than the specified rate. Usually events are received faster. The value must
819 *         be one of {@link #SENSOR_DELAY_NORMAL}, {@link #SENSOR_DELAY_UI},
820 *         {@link #SENSOR_DELAY_GAME}, or {@link #SENSOR_DELAY_FASTEST} or, the desired
821 *         delay between events in microseconds. Specifying the delay in microseconds only
822 *         works from Android 2.3 (API level 9) onwards. For earlier releases, you must use
823 *         one of the {@code SENSOR_DELAY_*} constants.
824 * @param handler The {@link android.os.Handler Handler} the {@link android.hardware.SensorEvent
825 *         sensor events} will be delivered to.
826 * @return <code>true</code> if the sensor is supported and successfully enabled.
827 * @see #registerListener(SensorEventListener, Sensor, int)
828 * @see #unregisterListener(SensorEventListener)
829 * @see #unregisterListener(SensorEventListener, Sensor)
830 */
831 public boolean registerListener(SensorEventListener listener, Sensor sensor,
832     int samplingPeriodUs, Handler handler) {
833     int delay = getDelay(samplingPeriodUs);
834     return registerListenerImpl(listener, sensor, delay, handler, 0, 0);
835 }
836
837 /**
838 * Registers a {@link android.hardware.SensorEventListener SensorEventListener} for the given
839 * sensor at the given sampling frequency and the given maximum reporting latency.
840 *
841 * @param listener A {@link android.hardware.SensorEventListener SensorEventListener} object
842 *         that will receive the sensor events. If the application is interested in receiving

```

```

843     * flush complete notifications, it should register with
844     * {@link android.hardware.SensorEventListener SensorEventListener2} instead.
845     * @param sensor The {@link android.hardware.Sensor Sensor} to register to.
846     * @param samplingPeriodUs The desired delay between two consecutive events in microseconds.
847     * This is only a hint to the system. Events may be received faster or slower than
848     * the specified rate. Usually events are received faster. Can be one of
849     * {@link #SENSOR_DELAY_NORMAL}, {@link #SENSOR_DELAY_UI},
850     * {@link #SENSOR_DELAY_GAME}, {@link #SENSOR_DELAY_FASTEST} or the delay in
851     * microseconds.
852     * @param maxReportLatencyUs Maximum time in microseconds that events can be delayed before
853     * being reported to the application. A large value allows reducing the power
854     * consumption associated with the sensor. If maxReportLatencyUs is set to zero,
855     * events are delivered as soon as they are available, which is equivalent to calling
856     * {@link #registerListener(SensorEventListener, Sensor, int)}.
857     * @param handler The {@link android.os.Handler Handler} the {@link android.hardware.SensorEvent
858     * sensor events} will be delivered to.
859     * @return <code>true</code> if the sensor is supported and successfully enabled.
860     * @see #registerListener(SensorEventListener, Sensor, int, int)
861     */
862     public boolean registerListener(SensorEventListener listener, Sensor sensor,
863         int samplingPeriodUs, int maxReportLatencyUs, Handler handler) {
864         int delayUs = getDelay(samplingPeriodUs);
865         return registerListenerImpl(listener, sensor, delayUs, handler, maxReportLatencyUs, 0);
866     }
867
868     /** @hide */
869     protected abstract boolean registerListenerImpl(SensorEventListener listener, Sensor sensor,
870         int delayUs, Handler handler, int maxReportLatencyUs, int reservedFlags);
871
872
873     /**
874     * Flushes the FIFO of all the sensors registered for this listener. If there are events
875     * in the FIFO of the sensor, they are returned as if the maxReportLatency of the FIFO has
876     * expired. Events are returned in the usual way through the SensorEventListener.
877     * This call doesn't affect the maxReportLatency for this sensor. This call is asynchronous and
878     * returns immediately.
879     * {@link android.hardware.SensorEventListener2#onFlushCompleted onFlushCompleted} is called

```



```

880     * after all the events in the batch at the time of calling this method have been delivered
881     * successfully. If the hardware doesn't support flush, it still returns true and a trivial
882     * flush complete event is sent after the current event for all the clients registered for this
883     * sensor.
884     *
885     * @param listener A {@link android.hardware.SensorEventListener SensorEventListener} object
886     *       which was previously used in a registerListener call.
887     * @return <code>true</code> if the flush is initiated successfully on all the sensors
888     *       registered for this listener, false if no sensor is previously registered for this
889     *       listener or flush on one of the sensors fails.
890     * @see #registerListener(SensorEventListener, Sensor, int, int)
891     * @throws IllegalArgumentException when listener is null.
892     */
893     public boolean flush(SensorEventListener listener) {
894         return flushImpl(listener);
895     }
896
897     /** @hide */
898     protected abstract boolean flushImpl(SensorEventListener listener);
899
900
901     /**
902     * Create a sensor direct channel backed by shared memory wrapped in MemoryFile object.
903     *
904     * The resulting channel can be used for delivering sensor events to native code, other
905     * processes, GPU/DSP or other co-processors without CPU intervention. This is the recommended
906     * for high performance sensor applications that use high sensor rates (e.g. greater than 200Hz)
907     * and cares about sensor event latency.
908     *
909     * Use the returned {@link android.hardware.SensorDirectChannel} object to configure direct
910     * report of sensor events. After use, call {@link android.hardware.SensorDirectChannel#close()}
911     * to free up resource in sensor system associated with the direct channel.
912     *
913     * @param mem A {@link android.os.MemoryFile} shared memory object.
914     * @return A {@link android.hardware.SensorDirectChannel} object.
915     * @throws NullPointerException when mem is null.
916     * @throws UncheckedIOException if not able to create channel.

```

```

917     * @see SensorDirectChannel#close()
918     */
919     public SensorDirectChannel createDirectChannel(MemoryFile mem) {
920         return createDirectChannelImpl(mem, null);
921     }
922
923     /**
924     * Create a sensor direct channel backed by shared memory wrapped in HardwareBuffer object.
925     *
926     * The resulting channel can be used for delivering sensor events to native code, other
927     * processes, GPU/DSP or other co-processors without CPU intervention. This is the recommended
928     * for high performance sensor applications that use high sensor rates (e.g. greater than 200Hz)
929     * and cares about sensor event latency.
930     *
931     * Use the returned {@link android.hardware.SensorDirectChannel} object to configure direct
932     * report of sensor events. After use, call {@link android.hardware.SensorDirectChannel#close()}
933     * to free up resource in sensor system associated with the direct channel.
934     *
935     * @param mem A {@link android.hardware.HardwareBuffer} shared memory object.
936     * @return A {@link android.hardware.SensorDirectChannel} object.
937     * @throws NullPointerException when mem is null.
938     * @throws UncheckedIOException if not able to create channel.
939     * @see SensorDirectChannel#close()
940     */
941     public SensorDirectChannel createDirectChannel(HardwareBuffer mem) {
942         return createDirectChannelImpl(null, mem);
943     }
944
945     /** @hide */
946     protected abstract SensorDirectChannel createDirectChannelImpl(
947         MemoryFile memoryFile, HardwareBuffer hardwareBuffer);
948
949     /** @hide */
950     void destroyDirectChannel(SensorDirectChannel channel) {
951         destroyDirectChannelImpl(channel);
952     }
953

```

```

954     /** @hide */
955     protected abstract void destroyDirectChannelImpl(SensorDirectChannel channel);
956
957     /** @hide */
958     protected abstract int configureDirectChannelImpl(
959         SensorDirectChannel channel, Sensor s, int rate);
960
961     /**
962      * Used for receiving notifications from the SensorManager when dynamic sensors are connected or
963      * disconnected.
964      */
965     public abstract static class DynamicSensorCallback {
966         /**
967          * Called when there is a dynamic sensor being connected to the system.
968          *
969          * @param sensor the newly connected sensor. See {@link android.hardware.Sensor Sensor}.
970          */
971         public void onDynamicSensorConnected(Sensor sensor) {}
972
973         /**
974          * Called when there is a dynamic sensor being disconnected from the system.
975          *
976          * @param sensor the disconnected sensor. See {@link android.hardware.Sensor Sensor}.
977          */
978         public void onDynamicSensorDisconnected(Sensor sensor) {}
979     }
980
981
982     /**
983      * Add a {@link android.hardware.SensorManager.DynamicSensorCallback
984      * DynamicSensorCallback} to receive dynamic sensor connection callbacks. Repeat
985      * registration with the already registered callback object will have no additional effect.
986      *
987      * @param callback An object that implements the
988      *      {@link android.hardware.SensorManager.DynamicSensorCallback
989      *      DynamicSensorCallback}
990      *      interface for receiving callbacks.

```

```

991     * @see #registerDynamicSensorCallback(DynamicSensorCallback, Handler)
992     *
993     * @throws IllegalArgumentException when callback is null.
994     */
995     public void registerDynamicSensorCallback(DynamicSensorCallback callback) {
996         registerDynamicSensorCallback(callback, null);
997     }
998
999     /**
1000     * Add a {@link android.hardware.SensorManager.DynamicSensorCallback
1001     * DynamicSensorCallback} to receive dynamic sensor connection callbacks. Repeat
1002     * registration with the already registered callback object will have no additional effect.
1003     *
1004     * @param callback An object that implements the
1005     *     {@link android.hardware.SensorManager.DynamicSensorCallback
1006     *     DynamicSensorCallback} interface for receiving callbacks.
1007     * @param handler The {@link android.os.Handler Handler} the {@link
1008     *     android.hardware.SensorManager.DynamicSensorCallback
1009     *     sensor connection events} will be delivered to.
1010     *
1011     * @throws IllegalArgumentException when callback is null.
1012     */
1013     public void registerDynamicSensorCallback(
1014         DynamicSensorCallback callback, Handler handler) {
1015         registerDynamicSensorCallbackImpl(callback, handler);
1016     }
1017
1018     /**
1019     * Remove a {@link android.hardware.SensorManager.DynamicSensorCallback
1020     * DynamicSensorCallback} to stop sending dynamic sensor connection events to that
1021     * callback.
1022     *
1023     * @param callback An object that implements the
1024     *     {@link android.hardware.SensorManager.DynamicSensorCallback
1025     *     DynamicSensorCallback}
1026     *     interface for receiving callbacks.
1027     */

```

```

1028     public void unregisterDynamicSensorCallback(DynamicSensorCallback callback) {
1029         unregisterDynamicSensorCallbackImpl(callback);
1030     }
1031
1032     /**
1033      * Tell if dynamic sensor discovery feature is supported by system.
1034      *
1035      * @return <code>true</code> if dynamic sensor discovery is supported, <code>false</code>
1036      * otherwise.
1037      */
1038     public boolean isDynamicSensorDiscoverySupported() {
1039         List<Sensor> sensors = getSensorList(Sensor.TYPE_DYNAMIC_SENSOR_META);
1040         return sensors.size() > 0;
1041     }
1042
1043     /** @hide */
1044     protected abstract void registerDynamicSensorCallbackImpl(
1045         DynamicSensorCallback callback, Handler handler);
1046
1047     /** @hide */
1048     protected abstract void unregisterDynamicSensorCallbackImpl(
1049         DynamicSensorCallback callback);
1050
1051     /**
1052      * <p>
1053      * Computes the inclination matrix <b>I</b> as well as the rotation matrix
1054      * <b>R</b> transforming a vector from the device coordinate system to the
1055      * world's coordinate system which is defined as a direct orthonormal basis,
1056      * where:
1057      * </p>
1058      *
1059      * <ul>
1060      * <li>X is defined as the vector product <b>Y.Z</b> (It is tangential to
1061      * the ground at the device's current location and roughly points East).</li>
1062      * <li>Y is tangential to the ground at the device's current location and
1063      * points towards the magnetic North Pole.</li>
1064      * <li>Z points towards the sky and is perpendicular to the ground.</li>

```

```

1065      * </ul>
1066      *
1067      * <p>
1068      * <center></center>
1070      * </p>
1071      *
1072      * <p>
1073      * <hr>
1074      * <p>
1075      * By definition:
1076      * <p>
1077      * [0 0 g] = <b>R</b> * <b>gravity</b> (g = magnitude of gravity)
1078      * <p>
1079      * [0 m 0] = <b>I</b> * <b>R</b> * <b>geomagnetic</b> (m = magnitude of
1080      * geomagnetic field)
1081      * <p>
1082      * <b>R</b> is the identity matrix when the device is aligned with the
1083      * world's coordinate system, that is, when the device's X axis points
1084      * toward East, the Y axis points to the North Pole and the device is facing
1085      * the sky.
1086      *
1087      * <p>
1088      * <b>I</b> is a rotation matrix transforming the geomagnetic vector into
1089      * the same coordinate space as gravity (the world's coordinate space).
1090      * <b>I</b> is a simple rotation around the X axis. The inclination angle in
1091      * radians can be computed with {@link #getInclination}.
1092      * <hr>
1093      *
1094      * <p>
1095      * Each matrix is returned either as a 3x3 or 4x4 row-major matrix depending
1096      * on the length of the passed array:
1097      * <p>
1098      * <u>If the array length is 16:</u>
1099      *
1100      * <pre>
1101      * / M[ 0]  M[ 1]  M[ 2]  M[ 3] \

```

```

1102      *   |   M[ 4]   M[ 5]   M[ 6]   M[ 7]   |
1103      *   |   M[ 8]   M[ 9]   M[10]   M[11]   |
1104      *   \   M[12]   M[13]   M[14]   M[15]   /
1105      *</pre>
1106      *
1107      * This matrix is ready to be used by OpenGL ES's
1108      * {@link javax.microedition.khronos.opengles.GL10#glLoadMatrixf(float[], int)
1109      * glLoadMatrixf(float[], int)}.
1110      * <p>
1111      * Note that because OpenGL matrices are column-major matrices you must
1112      * transpose the matrix before using it. However, since the matrix is a
1113      * rotation matrix, its transpose is also its inverse, conveniently, it is
1114      * often the inverse of the rotation that is needed for rendering; it can
1115      * therefore be used with OpenGL ES directly.
1116      * <p>
1117      * Also note that the returned matrices always have this form:
1118      *
1119      * <pre>
1120      *   /   M[ 0]   M[ 1]   M[ 2]   0   \
1121      *   |   M[ 4]   M[ 5]   M[ 6]   0   |
1122      *   |   M[ 8]   M[ 9]   M[10]   0   |
1123      *   \           0           0           0   1   /
1124      *</pre>
1125      *
1126      * <p>
1127      * <u>If the array length is 9:</u>
1128      *
1129      * <pre>
1130      *   /   M[ 0]   M[ 1]   M[ 2]   \
1131      *   |   M[ 3]   M[ 4]   M[ 5]   |
1132      *   \   M[ 6]   M[ 7]   M[ 8]   /
1133      *</pre>
1134      *
1135      * <hr>
1136      * <p>
1137      * The inverse of each matrix can be computed easily by taking its
1138      * transpose.

```

```
1139 *
1140 * <p>
1141 * The matrices returned by this function are meaningful only when the
1142 * device is not free-falling and it is not close to the magnetic north. If
1143 * the device is accelerating, or placed into a strong magnetic field, the
1144 * returned matrices may be inaccurate.
1145 *
1146 * @param R
1147 *     is an array of 9 floats holding the rotation matrix <b>R</b> when
1148 *     this function returns. R can be null.
1149 *     <p>
1150 *
1151 * @param I
1152 *     is an array of 9 floats holding the rotation matrix <b>I</b> when
1153 *     this function returns. I can be null.
1154 *     <p>
1155 *
1156 * @param gravity
1157 *     is an array of 3 floats containing the gravity vector expressed in
1158 *     the device's coordinate. You can simply use the
1159 *     {@link android.hardware.SensorEvent#values values} returned by a
1160 *     {@link android.hardware.SensorEvent SensorEvent} of a
1161 *     {@link android.hardware.Sensor Sensor} of type
1162 *     {@link android.hardware.Sensor#TYPE_ACCELEROMETER
1163 *     TYPE_ACCELEROMETER}.
1164 *     <p>
1165 *
1166 * @param geomagnetic
1167 *     is an array of 3 floats containing the geomagnetic vector
1168 *     expressed in the device's coordinate. You can simply use the
1169 *     {@link android.hardware.SensorEvent#values values} returned by a
1170 *     {@link android.hardware.SensorEvent SensorEvent} of a
1171 *     {@link android.hardware.Sensor Sensor} of type
1172 *     {@link android.hardware.Sensor#TYPE_MAGNETIC_FIELD
1173 *     TYPE_MAGNETIC_FIELD}.
1174 *
1175 * @return <code>true</code> on success, <code>false</code> on failure (for
```



```

1176     *         instance, if the device is in free fall). Free fall is defined as
1177     *         condition when the magnitude of the gravity is less than 1/10 of
1178     *         the nominal value. On failure the output matrices are not modified.
1179     *
1180     * @see #getInclination(float[])
1181     * @see #getOrientation(float[], float[])
1182     * @see #remapCoordinateSystem(float[], int, int, float[])
1183     */
1184
1185     public static boolean getRotationMatrix(float[] R, float[] I,
1186         float[] gravity, float[] geomagnetic) {
1187         // TODO: move this to native code for efficiency
1188         float Ax = gravity[0];
1189         float Ay = gravity[1];
1190         float Az = gravity[2];
1191
1192         final float normsqa = (Ax * Ax + Ay * Ay + Az * Az);
1193         final float g = 9.81f;
1194         final float freeFallGravitySquared = 0.01f * g * g;
1195         if (normsqa < freeFallGravitySquared) {
1196             // gravity less than 10% of normal value
1197             return false;
1198         }
1199
1200         final float Ex = geomagnetic[0];
1201         final float Ey = geomagnetic[1];
1202         final float Ez = geomagnetic[2];
1203         float Hx = Ey * Az - Ez * Ay;
1204         float Hy = Ez * Ax - Ex * Az;
1205         float Hz = Ex * Ay - Ey * Ax;
1206         final float normH = (float) Math.sqrt(Hx * Hx + Hy * Hy + Hz * Hz);
1207
1208         if (normH < 0.1f) {
1209             // device is close to free fall (or in space?), or close to
1210             // magnetic north pole. Typical values are > 100.
1211             return false;
1212         }

```

```

1213     final float invH = 1.0f / normH;
1214     Hx *= invH;
1215     Hy *= invH;
1216     Hz *= invH;
1217     final float invA = 1.0f / (float) Math.sqrt(Ax * Ax + Ay * Ay + Az * Az);
1218     Ax *= invA;
1219     Ay *= invA;
1220     Az *= invA;
1221     final float Mx = Ay * Hz - Az * Hy;
1222     final float My = Az * Hx - Ax * Hz;
1223     final float Mz = Ax * Hy - Ay * Hx;
1224     if (R != null) {
1225         if (R.length == 9) {
1226             R[0] = Hx;    R[1] = Hy;    R[2] = Hz;
1227             R[3] = Mx;    R[4] = My;    R[5] = Mz;
1228             R[6] = Ax;    R[7] = Ay;    R[8] = Az;
1229         } else if (R.length == 16) {
1230             R[0] = Hx;    R[1] = Hy;    R[2] = Hz;    R[3] = 0;
1231             R[4] = Mx;    R[5] = My;    R[6] = Mz;    R[7] = 0;
1232             R[8] = Ax;    R[9] = Ay;    R[10] = Az;    R[11] = 0;
1233             R[12] = 0;    R[13] = 0;    R[14] = 0;    R[15] = 1;
1234         }
1235     }
1236     if (I != null) {
1237         // compute the inclination matrix by projecting the geomagnetic
1238         // vector onto the Z (gravity) and X (horizontal component
1239         // of geomagnetic vector) axes.
1240         final float invE = 1.0f / (float) Math.sqrt(Ex * Ex + Ey * Ey + Ez * Ez);
1241         final float c = (Ex * Mx + Ey * My + Ez * Mz) * invE;
1242         final float s = (Ex * Ax + Ey * Ay + Ez * Az) * invE;
1243         if (I.length == 9) {
1244             I[0] = 1;    I[1] = 0;    I[2] = 0;
1245             I[3] = 0;    I[4] = c;    I[5] = s;
1246             I[6] = 0;    I[7] = -s;    I[8] = c;
1247         } else if (I.length == 16) {
1248             I[0] = 1;    I[1] = 0;    I[2] = 0;
1249             I[4] = 0;    I[5] = c;    I[6] = s;

```

```

1250         I[8] = 0;      I[9] = -s;      I[10] = c;
1251         I[3] = I[7] = I[11] = I[12] = I[13] = I[14] = 0;
1252         I[15] = 1;
1253     }
1254 }
1255     return true;
1256 }
1257
1258 /**
1259  * Computes the geomagnetic inclination angle in radians from the
1260  * inclination matrix <b>I</b> returned by {@link #getRotationMatrix}.
1261  *
1262  * @param I
1263  *         inclination matrix see {@link #getRotationMatrix}.
1264  *
1265  * @return The geomagnetic inclination angle in radians.
1266  *
1267  * @see #getRotationMatrix(float[], float[], float[], float[])
1268  * @see #getOrientation(float[], float[])
1269  * @see GeomagneticField
1270  *
1271  */
1272 public static float getInclination(float[] I) {
1273     if (I.length == 9) {
1274         return (float) Math.atan2(I[5], I[4]);
1275     } else {
1276         return (float) Math.atan2(I[6], I[5]);
1277     }
1278 }
1279
1280 /**
1281  * <p>
1282  * Rotates the supplied rotation matrix so it is expressed in a different
1283  * coordinate system. This is typically used when an application needs to
1284  * compute the three orientation angles of the device (see
1285  * {@link #getOrientation}) in a different coordinate system.
1286  * </p>

```

```
1287 *
1288 * <p>
1289 * When the rotation matrix is used for drawing (for instance with OpenGL
1290 * ES), it usually <b>doesn't need</b> to be transformed by this function,
1291 * unless the screen is physically rotated, in which case you can use
1292 * {@link android.view.Display#getRotation() Display.getRotation()} to
1293 * retrieve the current rotation of the screen. Note that because the user
1294 * is generally free to rotate their screen, you often should consider the
1295 * rotation in deciding the parameters to use here.
1296 * </p>
1297 *
1298 * <p>
1299 * <u>Examples:</u>
1300 * <p>
1301 *
1302 * <ul>
1303 * <li>Using the camera (Y axis along the camera's axis) for an augmented
1304 * reality application where the rotation angles are needed:</li>
1305 *
1306 * <p>
1307 * <ul>
1308 * <code>remapCoordinateSystem(inR, AXIS_X, AXIS_Z, outR);</code>
1309 * </ul>
1310 * </p>
1311 *
1312 * <li>Using the device as a mechanical compass when rotation is
1313 * {@link android.view.Surface#ROTATION_90 Surface.ROTATION_90}:</li>
1314 *
1315 * <p>
1316 * <ul>
1317 * <code>remapCoordinateSystem(inR, AXIS_Y, AXIS_MINUS_X, outR);</code>
1318 * </ul>
1319 * </p>
1320 *
1321 * Beware of the above example. This call is needed only to account for a
1322 * rotation from its natural orientation when calculating the rotation
1323 * angles (see {@link #getOrientation}). If the rotation matrix is also used
```

```

1324 * for rendering, it may not need to be transformed, for instance if your
1325 * {@link android.app.Activity Activity} is running in landscape mode.
1326 * </ul>
1327 *
1328 * <p>
1329 * Since the resulting coordinate system is orthonormal, only two axes need
1330 * to be specified.
1331 *
1332 * @param inR
1333 *         the rotation matrix to be transformed. Usually it is the matrix
1334 *         returned by {@link #getRotationMatrix}.
1335 *
1336 * @param X
1337 *         defines the axis of the new coordinate system that coincide with the X axis of the
1338 *         original coordinate system.
1339 *
1340 * @param Y
1341 *         defines the axis of the new coordinate system that coincide with the Y axis of the
1342 *         original coordinate system.
1343 *
1344 * @param outR
1345 *         the transformed rotation matrix. inR and outR should not be the same
1346 *         array.
1347 *
1348 * @return <code>true</code> on success. <code>false</code> if the input
1349 *         parameters are incorrect, for instance if X and Y define the same
1350 *         axis. Or if inR and outR don't have the same length.
1351 *
1352 * @see #getRotationMatrix(float[], float[], float[], float[])
1353 */
1354
1355 public static boolean remapCoordinateSystem(float[] inR, int X, int Y, float[] outR) {
1356     if (inR == outR) {
1357         final float[] temp = sTempMatrix;
1358         synchronized (temp) {
1359             // we don't expect to have a lot of contention
1360             if (remapCoordinateSystemImpl(inR, X, Y, temp)) {

```

```

1361         final int size = outR.length;
1362         for (int i = 0; i < size; i++) {
1363             outR[i] = temp[i];
1364         }
1365         return true;
1366     }
1367 }
1368 }
1369 return remapCoordinateSystemImpl(inR, X, Y, outR);
1370 }
1371
1372 private static boolean remapCoordinateSystemImpl(float[] inR, int X, int Y, float[] outR) {
1373     /*
1374      * X and Y define a rotation matrix 'r':
1375      *
1376      * (X==1)?((X&0x80)?-1:1):0    (X==2)?((X&0x80)?-1:1):0    (X==3)?((X&0x80)?-1:1):0
1377      * (Y==1)?((Y&0x80)?-1:1):0    (Y==2)?((Y&0x80)?-1:1):0    (Y==3)?((X&0x80)?-1:1):0
1378      *
1379      * r[0] ^ r[1]
1380
1381      * where the 3rd line is the vector product of the first 2 lines
1382      */
1383
1384     final int length = outR.length;
1385     if (inR.length != length) {
1386         return false;    // invalid parameter
1387     }
1388     if ((X & 0x7C) != 0 || (Y & 0x7C) != 0) {
1389         return false;    // invalid parameter
1390     }
1391     if (((X & 0x3) == 0) || ((Y & 0x3) == 0)) {
1392         return false;    // no axis specified
1393     }
1394     if ((X & 0x3) == (Y & 0x3)) {
1395         return false;    // same axis specified
1396     }
1397

```

```

1398 // Z is "the other" axis, its sign is either +/- sign(X)*sign(Y)
1399 // this can be calculated by exclusive-or'ing X and Y; except for
1400 // the sign inversion (+/-) which is calculated below.
1401 int Z = X ^ Y;
1402
1403 // extract the axis (remove the sign), offset in the range 0 to 2.
1404 final int x = (X & 0x3) - 1;
1405 final int y = (Y & 0x3) - 1;
1406 final int z = (Z & 0x3) - 1;
1407
1408 // compute the sign of Z (whether it needs to be inverted)
1409 final int axis_y = (z + 1) % 3;
1410 final int axis_z = (z + 2) % 3;
1411 if (((x ^ axis_y) | (y ^ axis_z)) != 0) {
1412     Z ^= 0x80;
1413 }
1414
1415 final boolean sx = (X >= 0x80);
1416 final boolean sy = (Y >= 0x80);
1417 final boolean sz = (Z >= 0x80);
1418
1419 // Perform R * r, in avoiding actual mul's and adds.
1420 final int rowLength = ((length == 16) ? 4 : 3);
1421 for (int j = 0; j < 3; j++) {
1422     final int offset = j * rowLength;
1423     for (int i = 0; i < 3; i++) {
1424         if (x == i) outR[offset + i] = sx ? -inR[offset + 0] : inR[offset + 0];
1425         if (y == i) outR[offset + i] = sy ? -inR[offset + 1] : inR[offset + 1];
1426         if (z == i) outR[offset + i] = sz ? -inR[offset + 2] : inR[offset + 2];
1427     }
1428 }
1429 if (length == 16) {
1430     outR[3] = outR[7] = outR[11] = outR[12] = outR[13] = outR[14] = 0;
1431     outR[15] = 1;
1432 }
1433 return true;
1434 }

```

```

1435
1436 /**
1437  * Computes the device's orientation based on the rotation matrix.
1438  * <p>
1439  * When it returns, the array values are as follows:
1440  * <ul>
1441  * <li>values[0]: <i>Azimuth</i>, angle of rotation about the -z axis.
1442  *         This value represents the angle between the device's y
1443  *         axis and the magnetic north pole. When facing north, this
1444  *         angle is 0, when facing south, this angle is  $\pi$ .
1445  *         Likewise, when facing east, this angle is  $\pi/2$ , and
1446  *         when facing west, this angle is  $-\pi/2$ . The range of
1447  *         values is  $-\pi$  to  $\pi$ .</li>
1448  * <li>values[1]: <i>Pitch</i>, angle of rotation about the x axis.
1449  *         This value represents the angle between a plane parallel
1450  *         to the device's screen and a plane parallel to the ground.
1451  *         Assuming that the bottom edge of the device faces the
1452  *         user and that the screen is face-up, tilting the top edge
1453  *         of the device toward the ground creates a positive pitch
1454  *         angle. The range of values is  $-\pi/2$  to  $\pi/2$ .</li>
1455  * <li>values[2]: <i>Roll</i>, angle of rotation about the y axis. This
1456  *         value represents the angle between a plane perpendicular
1457  *         to the device's screen and a plane perpendicular to the
1458  *         ground. Assuming that the bottom edge of the device faces
1459  *         the user and that the screen is face-up, tilting the left
1460  *         edge of the device toward the ground creates a positive
1461  *         roll angle. The range of values is  $-\pi$  to  $\pi$ .</li>
1462  * </ul>
1463  * <p>
1464  * Applying these three rotations in the azimuth, pitch, roll order
1465  * transforms an identity matrix to the rotation matrix passed into this
1466  * method. Also, note that all three orientation angles are expressed in
1467  * <b>radians</b>.
1468  *
1469  * @param R
1470  *         rotation matrix see {@link #getRotationMatrix}.
1471  *

```



```

1472     * @param values
1473     *         an array of 3 floats to hold the result.
1474     *
1475     * @return The array values passed as argument.
1476     *
1477     * @see #getRotationMatrix(float[], float[], float[], float[])
1478     * @see GeomagneticField
1479     */
1480     public static float[] getOrientation(float[] R, float[] values) {
1481         /*
1482         * 4x4 (length=16) case:
1483         *   /  R[ 0]  R[ 1]  R[ 2]  0  \
1484         *   |  R[ 4]  R[ 5]  R[ 6]  0  |
1485         *   |  R[ 8]  R[ 9]  R[10]  0  |
1486         *   \      0      0      0  1  /
1487         *
1488         * 3x3 (length=9) case:
1489         *   /  R[ 0]  R[ 1]  R[ 2]  \
1490         *   |  R[ 3]  R[ 4]  R[ 5]  |
1491         *   \  R[ 6]  R[ 7]  R[ 8]  /
1492         *
1493         */
1494         if (R.length == 9) {
1495             values[0] = (float) Math.atan2(R[1], R[4]);
1496             values[1] = (float) Math.asin(-R[7]);
1497             values[2] = (float) Math.atan2(-R[6], R[8]);
1498         } else {
1499             values[0] = (float) Math.atan2(R[1], R[5]);
1500             values[1] = (float) Math.asin(-R[9]);
1501             values[2] = (float) Math.atan2(-R[8], R[10]);
1502         }
1503
1504         return values;
1505     }
1506
1507     /**
1508     * Computes the Altitude in meters from the atmospheric pressure and the

```

```

1509     * pressure at sea level.
1510     * <p>
1511     * Typically the atmospheric pressure is read from a
1512     * {@link Sensor#TYPE_PRESSURE} sensor. The pressure at sea level must be
1513     * known, usually it can be retrieved from airport databases in the
1514     * vicinity. If unknown, you can use {@link #PRESSURE_STANDARD_ATMOSPHERE}
1515     * as an approximation, but absolute altitudes won't be accurate.
1516     * </p>
1517     * <p>
1518     * To calculate altitude differences, you must calculate the difference
1519     * between the altitudes at both points. If you don't know the altitude
1520     * as sea level, you can use {@link #PRESSURE_STANDARD_ATMOSPHERE} instead,
1521     * which will give good results considering the range of pressure typically
1522     * involved.
1523     * </p>
1524     * <p>
1525     * <code><ul>
1526     *     float altitude_difference =
1527     *         getAltitude(SensorManager.PRESSURE_STANDARD_ATMOSPHERE, pressure_at_point2)
1528     *         - getAltitude(SensorManager.PRESSURE_STANDARD_ATMOSPHERE, pressure_at_point1);
1529     * </ul></code>
1530     * </p>
1531     *
1532     * @param p0 pressure at sea level
1533     * @param p atmospheric pressure
1534     * @return Altitude in meters
1535     */
1536     public static float getAltitude(float p0, float p) {
1537         final float coef = 1.0f / 5.255f;
1538         return 44330.0f * (1.0f - (float) Math.pow(p / p0, coef));
1539     }
1540
1541     /** Helper function to compute the angle change between two rotation matrices.
1542     * Given a current rotation matrix (R) and a previous rotation matrix
1543     * (prevR) computes the intrinsic rotation around the z, x, and y axes which
1544     * transforms prevR to R.
1545     * outputs a 3 element vector containing the z, x, and y angle

```

```

1546 * change at indexes 0, 1, and 2 respectively.
1547 * <p> Each input matrix is either as a 3x3 or 4x4 row-major matrix
1548 * depending on the length of the passed array:
1549 * <p>If the array length is 9, then the array elements represent this matrix
1550 * <pre>
1551 *   /  R[ 0]   R[ 1]   R[ 2]   \
1552 *   |  R[ 3]   R[ 4]   R[ 5]   |
1553 *   \  R[ 6]   R[ 7]   R[ 8]   /
1554 *</pre>
1555 * <p>If the array length is 16, then the array elements represent this matrix
1556 * <pre>
1557 *   /  R[ 0]   R[ 1]   R[ 2]   R[ 3]   \
1558 *   |  R[ 4]   R[ 5]   R[ 6]   R[ 7]   |
1559 *   |  R[ 8]   R[ 9]   R[10]   R[11]   |
1560 *   \  R[12]   R[13]   R[14]   R[15]   /
1561 *</pre>
1562 *
1563 * See {@link #getOrientation} for more detailed definition of the output.
1564 *
1565 * @param R current rotation matrix
1566 * @param prevR previous rotation matrix
1567 * @param angleChange an array of floats (z, x, and y) in which the angle change
1568 *                    (in radians) is stored
1569 */
1570
1571 public static void getAngleChange(float[] angleChange, float[] R, float[] prevR) {
1572     float rd1 = 0, rd4 = 0, rd6 = 0, rd7 = 0, rd8 = 0;
1573     float ri0 = 0, ri1 = 0, ri2 = 0, ri3 = 0, ri4 = 0, ri5 = 0, ri6 = 0, ri7 = 0, ri8 = 0;
1574     float pri0 = 0, pri1 = 0, pri2 = 0, pri3 = 0, pri4 = 0;
1575     float pri5 = 0, pri6 = 0, pri7 = 0, pri8 = 0;
1576
1577     if (R.length == 9) {
1578         ri0 = R[0];
1579         ri1 = R[1];
1580         ri2 = R[2];
1581         ri3 = R[3];
1582         ri4 = R[4];

```

```
1583         ri5 = R[5];
1584         ri6 = R[6];
1585         ri7 = R[7];
1586         ri8 = R[8];
1587     } else if (R.length == 16) {
1588         ri0 = R[0];
1589         ri1 = R[1];
1590         ri2 = R[2];
1591         ri3 = R[4];
1592         ri4 = R[5];
1593         ri5 = R[6];
1594         ri6 = R[8];
1595         ri7 = R[9];
1596         ri8 = R[10];
1597     }
1598
1599     if (prevR.length == 9) {
1600         pri0 = prevR[0];
1601         pri1 = prevR[1];
1602         pri2 = prevR[2];
1603         pri3 = prevR[3];
1604         pri4 = prevR[4];
1605         pri5 = prevR[5];
1606         pri6 = prevR[6];
1607         pri7 = prevR[7];
1608         pri8 = prevR[8];
1609     } else if (prevR.length == 16) {
1610         pri0 = prevR[0];
1611         pri1 = prevR[1];
1612         pri2 = prevR[2];
1613         pri3 = prevR[4];
1614         pri4 = prevR[5];
1615         pri5 = prevR[6];
1616         pri6 = prevR[8];
1617         pri7 = prevR[9];
1618         pri8 = prevR[10];
1619     }
```

```

1620
1621 // calculate the parts of the rotation difference matrix we need
1622 // rd[i][j] = pri[0][i] * ri[0][j] + pri[1][i] * ri[1][j] + pri[2][i] * ri[2][j];
1623
1624 rd1 = pri0 * ri1 + pri3 * ri4 + pri6 * ri7; //rd[0][1]
1625 rd4 = pri1 * ri1 + pri4 * ri4 + pri7 * ri7; //rd[1][1]
1626 rd6 = pri2 * ri0 + pri5 * ri3 + pri8 * ri6; //rd[2][0]
1627 rd7 = pri2 * ri1 + pri5 * ri4 + pri8 * ri7; //rd[2][1]
1628 rd8 = pri2 * ri2 + pri5 * ri5 + pri8 * ri8; //rd[2][2]
1629
1630 angleChange[0] = (float) Math.atan2(rd1, rd4);
1631 angleChange[1] = (float) Math.asin(-rd7);
1632 angleChange[2] = (float) Math.atan2(-rd6, rd8);
1633
1634 }
1635
1636 /** Helper function to convert a rotation vector to a rotation matrix.
1637  * Given a rotation vector (presumably from a ROTATION_VECTOR sensor), returns a
1638  * 9 or 16 element rotation matrix in the array R. R must have length 9 or 16.
1639  * If R.length == 9, the following matrix is returned:
1640  * <pre>
1641  * / R[ 0] R[ 1] R[ 2] \
1642  * | R[ 3] R[ 4] R[ 5] |
1643  * \ R[ 6] R[ 7] R[ 8] /
1644  *</pre>
1645  * If R.length == 16, the following matrix is returned:
1646  * <pre>
1647  * / R[ 0] R[ 1] R[ 2] 0 \
1648  * | R[ 4] R[ 5] R[ 6] 0 |
1649  * | R[ 8] R[ 9] R[10] 0 |
1650  * \ 0      0      0      1 /
1651  *</pre>
1652  * @param rotationVector the rotation vector to convert
1653  * @param R an array of floats in which to store the rotation matrix
1654  */
1655 public static void getRotationMatrixFromVector(float[] R, float[] rotationVector) {
1656

```

```
1657     float q0;
1658     float q1 = rotationVector[0];
1659     float q2 = rotationVector[1];
1660     float q3 = rotationVector[2];
1661
1662     if (rotationVector.length >= 4) {
1663         q0 = rotationVector[3];
1664     } else {
1665         q0 = 1 - q1 * q1 - q2 * q2 - q3 * q3;
1666         q0 = (q0 > 0) ? (float) Math.sqrt(q0) : 0;
1667     }
1668
1669     float sq_q1 = 2 * q1 * q1;
1670     float sq_q2 = 2 * q2 * q2;
1671     float sq_q3 = 2 * q3 * q3;
1672     float q1_q2 = 2 * q1 * q2;
1673     float q3_q0 = 2 * q3 * q0;
1674     float q1_q3 = 2 * q1 * q3;
1675     float q2_q0 = 2 * q2 * q0;
1676     float q2_q3 = 2 * q2 * q3;
1677     float q1_q0 = 2 * q1 * q0;
1678
1679     if (R.length == 9) {
1680         R[0] = 1 - sq_q2 - sq_q3;
1681         R[1] = q1_q2 - q3_q0;
1682         R[2] = q1_q3 + q2_q0;
1683
1684         R[3] = q1_q2 + q3_q0;
1685         R[4] = 1 - sq_q1 - sq_q3;
1686         R[5] = q2_q3 - q1_q0;
1687
1688         R[6] = q1_q3 - q2_q0;
1689         R[7] = q2_q3 + q1_q0;
1690         R[8] = 1 - sq_q1 - sq_q2;
1691     } else if (R.length == 16) {
1692         R[0] = 1 - sq_q2 - sq_q3;
1693         R[1] = q1_q2 - q3_q0;
```

```

1694         R[2] = q1_q3 + q2_q0;
1695         R[3] = 0.0f;
1696
1697         R[4] = q1_q2 + q3_q0;
1698         R[5] = 1 - sq_q1 - sq_q3;
1699         R[6] = q2_q3 - q1_q0;
1700         R[7] = 0.0f;
1701
1702         R[8] = q1_q3 - q2_q0;
1703         R[9] = q2_q3 + q1_q0;
1704         R[10] = 1 - sq_q1 - sq_q2;
1705         R[11] = 0.0f;
1706
1707         R[12] = R[13] = R[14] = 0.0f;
1708         R[15] = 1.0f;
1709     }
1710 }
1711
1712 /** Helper function to convert a rotation vector to a normalized quaternion.
1713  * Given a rotation vector (presumably from a ROTATION_VECTOR sensor), returns a normalized
1714  * quaternion in the array Q. The quaternion is stored as [w, x, y, z]
1715  * @param rv the rotation vector to convert
1716  * @param Q an array of floats in which to store the computed quaternion
1717  */
1718 public static void getQuaternionFromVector(float[] Q, float[] rv) {
1719     if (rv.length >= 4) {
1720         Q[0] = rv[3];
1721     } else {
1722         Q[0] = 1 - rv[0] * rv[0] - rv[1] * rv[1] - rv[2] * rv[2];
1723         Q[0] = (Q[0] > 0) ? (float) Math.sqrt(Q[0]) : 0;
1724     }
1725     Q[1] = rv[0];
1726     Q[2] = rv[1];
1727     Q[3] = rv[2];
1728 }
1729
1730 /**

```

```
1731 * Requests receiving trigger events for a trigger sensor.
1732 *
1733 * <p>
1734 * When the sensor detects a trigger event condition, such as significant motion in
1735 * the case of the {@link Sensor#TYPE_SIGNIFICANT_MOTION}, the provided trigger listener
1736 * will be invoked once and then its request to receive trigger events will be canceled.
1737 * To continue receiving trigger events, the application must request to receive trigger
1738 * events again.
1739 * </p>
1740 *
1741 * @param listener The listener on which the
1742 *         {@link TriggerEventListener#onTrigger(TriggerEvent)} will be delivered.
1743 * @param sensor The sensor to be enabled.
1744 *
1745 * @return true if the sensor was successfully enabled.
1746 *
1747 * @throws IllegalArgumentException when sensor is null or not a trigger sensor.
1748 */
1749 public boolean requestTriggerSensor(TriggerEventListener listener, Sensor sensor) {
1750     return requestTriggerSensorImpl(listener, sensor);
1751 }
1752
1753 /**
1754 * @hide
1755 */
1756 protected abstract boolean requestTriggerSensorImpl(TriggerEventListener listener,
1757     Sensor sensor);
1758
1759 /**
1760 * Cancels receiving trigger events for a trigger sensor.
1761 *
1762 * <p>
1763 * Note that a Trigger sensor will be auto disabled if
1764 * {@link TriggerEventListener#onTrigger(TriggerEvent)} has triggered.
1765 * This method is provided in case the user wants to explicitly cancel the request
1766 * to receive trigger events.
1767 * </p>
```



```

1768      *
1769      * @param listener The listener on which the
1770      *      {@link TriggerEventListener#onTrigger(TriggerEvent)}
1771      *      is delivered. It should be the same as the one used
1772      *      in {@link #requestTriggerSensor(TriggerEventListener, Sensor)}
1773      * @param sensor The sensor for which the trigger request should be canceled.
1774      *      If null, it cancels receiving trigger for all sensors associated
1775      *      with the listener.
1776      *
1777      * @return true if successfully canceled.
1778      *
1779      * @throws IllegalArgumentException when sensor is a trigger sensor.
1780      */
1781 public boolean cancelTriggerSensor(TriggerEventListener listener, Sensor sensor) {
1782     return cancelTriggerSensorImpl(listener, sensor, true);
1783 }
1784
1785 /**
1786  * @hide
1787  */
1788 protected abstract boolean cancelTriggerSensorImpl(TriggerEventListener listener,
1789     Sensor sensor, boolean disable);
1790
1791
1792 /**
1793  * For testing purposes only. Not for third party applications.
1794  *
1795  * Initialize data injection mode and create a client for data injection. SensorService should
1796  * already be operating in DATA_INJECTION mode for this call succeed. To set SensorService into
1797  * DATA_INJECTION mode "adb shell dumpsys sensorservice data_injection" needs to be called
1798  * through adb. Typically this is done using a host side test. This mode is expected to be used
1799  * only for testing purposes. If the HAL is set to data injection mode, it will ignore the input
1800  * from physical sensors and read sensor data that is injected from the test application. This
1801  * mode is used for testing vendor implementations for various algorithms like Rotation Vector,
1802  * Significant Motion, Step Counter etc. Not all HALs support DATA_INJECTION. This method will
1803  * fail in those cases. Once this method succeeds, the test can call
1804  * {@link injectSensorData(Sensor, float[], int, long)} to inject sensor data into the HAL.

```

```
1805      *
1806      * @param enable True to initialize a client in DATA_INJECTION mode.
1807      *               False to clean up the native resources.
1808      *
1809      * @return true if the HAL supports data injection and false
1810      *         otherwise.
1811      * @hide
1812      */
1813 @SystemApi
1814 public boolean initDataInjection(boolean enable) {
1815     return initDataInjectionImpl(enable);
1816 }
1817
1818 /**
1819  * @hide
1820  */
1821 protected abstract boolean initDataInjectionImpl(boolean enable);
1822
1823 /**
1824  * For testing purposes only. Not for third party applications.
1825  *
1826  * This method is used to inject raw sensor data into the HAL. Call {@link
1827  * initDataInjection(boolean)} before this method to set the HAL in data injection mode. This
1828  * method should be called only if a previous call to initDataInjection has been successful and
1829  * the HAL and SensorService are already operating in data injection mode.
1830  *
1831  * @param sensor The sensor to inject.
1832  * @param values Sensor values to inject. The length of this
1833  *              array must be exactly equal to the number of
1834  *              values reported by the sensor type.
1835  * @param accuracy Accuracy of the sensor.
1836  * @param timestamp Sensor timestamp associated with the event.
1837  *
1838  * @return boolean True if the data injection succeeds, false
1839  *         otherwise.
1840  * @throws IllegalArgumentException when the sensor is null,
1841  *         data injection is not supported by the sensor, values
```

```

1842     *         are null, incorrect number of values for the sensor,
1843     *         sensor accuracy is incorrect or timestamps are
1844     *         invalid.
1845     * @hide
1846     */
1847     @SystemApi
1848     public boolean injectSensorData(Sensor sensor, float[] values, int accuracy,
1849         long timestamp) {
1850         if (sensor == null) {
1851             throw new IllegalArgumentException("sensor cannot be null");
1852         }
1853         if (!sensor.isDataInjectionSupported()) {
1854             throw new IllegalArgumentException("sensor does not support data injection");
1855         }
1856         if (values == null) {
1857             throw new IllegalArgumentException("sensor data cannot be null");
1858         }
1859         int expectedNumValues = Sensor.getLengthValuesArray(sensor, Build.VERSION_CODES.M);
1860         if (values.length != expectedNumValues) {
1861             throw new IllegalArgumentException("Wrong number of values for sensor "
1862                 + sensor.getName() + " actual=" + values.length + " expected="
1863                 + expectedNumValues);
1864         }
1865         if (accuracy < SENSOR_STATUS_NO_CONTACT || accuracy > SENSOR_STATUS_ACCURACY_HIGH) {
1866             throw new IllegalArgumentException("Invalid sensor accuracy");
1867         }
1868         if (timestamp <= 0) {
1869             throw new IllegalArgumentException("Negative or zero sensor timestamp");
1870         }
1871         return injectSensorDataImpl(sensor, values, accuracy, timestamp);
1872     }
1873
1874     /**
1875     * @hide
1876     */
1877     protected abstract boolean injectSensorDataImpl(Sensor sensor, float[] values, int accuracy,
1878         long timestamp);

```

```
1879
1880     private LegacySensorManager getLegacySensorManager() {
1881         synchronized (mSensorListByType) {
1882             if (mLegacySensorManager == null) {
1883                 Log.i(TAG, "This application is using deprecated SensorManager API which will "
1884                     + "be removed someday. Please consider switching to the new API.");
1885                 mLegacySensorManager = new LegacySensorManager(this);
1886             }
1887             return mLegacySensorManager;
1888         }
1889     }
1890
1891     private static int getDelay(int rate) {
1892         int delay = -1;
1893         switch (rate) {
1894             case SENSOR_DELAY_FASTEST:
1895                 delay = 0;
1896                 break;
1897             case SENSOR_DELAY_GAME:
1898                 delay = 20000;
1899                 break;
1900             case SENSOR_DELAY_UI:
1901                 delay = 66667;
1902                 break;
1903             case SENSOR_DELAY_NORMAL:
1904                 delay = 200000;
1905                 break;
1906             default:
1907                 delay = rate;
1908                 break;
1909         }
1910         return delay;
1911     }
1912
1913     /** @hide */
1914     public boolean setOperationParameter(SensorAdditionalInfo parameter) {
1915         return setOperationParameterImpl(parameter);
1916     }
1917 }
```

```
1916     }
1917
1918     /** @hide */
1919     protected abstract boolean setOperationParameterImpl(SensorAdditionalInfo parameter);
1920 }
```