Position sensors

On this page

The Android platform provides two sensors that let you determine the position of a device: the geomagnetic field sensor and the accelerometer. The Android platform also provides a sensor that lets you determine how close the face of a device is to an object (known as the *proximity sensor*). The geomagnetic field sensor and the proximity sensor are hardware-based. Most handset and tablet manufacturers include a geomagnetic field sensor. Likewise, handset manufacturers usually include a proximity sensor to determine when a handset is being held close to a user's face (for example, during a phone call). For determining a device's orientation, you can use the readings from the device's accelerometer and the geomagnetic field sensor.

★ **Note:** The orientation sensor was deprecated in Android 2.2 (API level 8), and the orientation sensor type was deprecated in Android 4.4W (API level 20).

Position sensors are useful for determining a device's physical position in the world's frame of reference. For example, you can use the geomagnetic field sensor in combination with the accelerometer to determine a device's position relative to the magnetic north pole. You can also use these sensors to determine a device's orientation in your application's frame of reference. Position sensors are not typically used to monitor device movement or motion, such as shake, tilt, or thrust (for more information, see Motion sensors).

The geomagnetic field sensor and accelerometer return multi-dimensional arrays of sensor values for each `SensorEvent`. For example, the geomagnetic field sensor provides geomagnetic field strength values for each of the three coordinate axes during a single sensor event. Likewise, the accelerometer sensor measures the acceleration applied to the device during a sensor event. For more information about the coordinate systems that are used by sensors, see Sensor coordinate systems. The proximity sensor provides a single value for each sensor event. Table 1 summarizes the position sensors that are supported on the Android platform.

Table 1. Position sensors that are supported on the Android platform.

Sensor	Sensor event data	Description	Units of measure
`TYPE_GAME_ROTATION_VECTOR`	`SensorEvent.values[0]`	Rotation vector component along the x axis (x * sin(θ/2)).	Unitless

	`SensorEvent.values[1]`	Rotation vector component along the y axis (y * sin(θ/2)).	
	`SensorEvent.values[2]`	Rotation vector component along the z axis (z * sin(θ/2)).	
`TYPE_GEOMAGNETIC_ROTATION_VECTOR`	`SensorEvent.values[0]`	Rotation vector component along the x axis (x * sin(θ/2)).	Unitless
	`SensorEvent.values[1]`	Rotation vector component along the y axis (y * sin(θ/2)).	
	`SensorEvent.values[2]`	Rotation vector component along the z axis (z * sin(θ/2)).	
`TYPE_MAGNETIC_FIELD`	`SensorEvent.values[0]`	Geomagnetic field strength along the x axis.	μT
	`SensorEvent.values[1]`	Geomagnetic field strength along the y axis.	
	`SensorEvent.values[2]`	Geomagnetic field strength along the z axis.	
`TYPE_MAGNETIC_FIELD_UNCALIBRATED`	`SensorEvent.values[0]`	Geomagnetic field strength (without hard iron calibration) along the x axis.	μT

	SensorEvent.values[1]	Geomagnetic field strength (without hard iron calibration) along the y axis.	
	SensorEvent.values[2]	Geomagnetic field strength (without hard iron calibration) along the z axis.	
	SensorEvent.values[3]	Iron bias estimation along the x axis.	
	SensorEvent.values[4]	Iron bias estimation along the y axis.	
	SensorEvent.values[5]	Iron bias estimation along the z axis.	
TYPE_ORIENTATION[1]	SensorEvent.values[0]	Azimuth (angle around the z-axis).	Degrees
	SensorEvent.values[1]	Pitch (angle around the x-axis).	
	SensorEvent.values[2]	Roll (angle around the y-axis).	
TYPE_PROXIMITY	SensorEvent.values[0]	Distance from object.[2]	cm

★ [1]This sensor was deprecated in Android 2.2 (API level 8), and this sensor type was deprecated in Android 4.4W (API level 20). The sensor framework provides alternate methods for acquiring device orientation, which are discussed in Compute the device's orientation.

★ [2] Some proximity sensors provide only binary values representing near and far.

Use the game rotation vector sensor

The game rotation vector sensor is identical to the Rotation vector sensor, except it does not use the geomagnetic field. Therefore the Y axis does not point north but instead to some other reference. That reference is allowed to drift by the same order of magnitude as the gyroscope drifts around the Z axis.

Because the game rotation vector sensor does not use the magnetic field, relative rotations are more accurate, and not impacted by magnetic field changes. Use this sensor in a game if you do not care about where north is, and the normal rotation vector does not fit your needs because of its reliance on the magnetic field.

The following code shows you how to get an instance of the default game rotation vector sensor:

Kotlin Java

```
private lateinit var sensorManager: SensorManager
private var sensor: Sensor? = null
...
sensorManager = getSystemService(Context.SENSOR_SERVICE) as SensorManager
sensor = sensorManager.getDefaultSensor(Sensor.TYPE_GAME_ROTATION_VECTOR)
```

Use the geomagnetic rotation vector sensor

The geomagnetic rotation vector sensor is similar to the rotation vector sensor, but it doesn't use the gyroscope. The accuracy of this sensor is lower than the normal rotation vector sensor, but the power consumption is reduced. Use this sensor only if you want to collect rotation information in the background without using too much battery. This sensor is most useful when used in conjunction with batching.

The following code shows you how to get an instance of the default geomagnetic rotation vector sensor:

Kotlin Java

```
private lateinit var sensorManager: SensorManager
private var sensor: Sensor? = null
...
sensorManager = getSystemService(Context.SENSOR_SERVICE) as SensorManager
sensor = sensorManager.getDefaultSensor(Sensor.TYPE_GEOMAGNETIC_ROTATION_VECTOR)
```

Compute the device's orientation

By computing a device's orientation, you can monitor the position of the device relative to the earth's frame of reference (specifically, the magnetic north pole). The following code shows you how to compute a device's orientation:

Kotlin Java

```
private lateinit var sensorManager: SensorManager
...
// Rotation matrix based on current readings from accelerometer and magnetometer.
val rotationMatrix = FloatArray(9)
SensorManager.getRotationMatrix(rotationMatrix, null, accelerometerReading, magnetometerReading)

// Express the updated rotation matrix as three orientation angles.
val orientationAngles = FloatArray(3)
SensorManager.getOrientation(rotationMatrix, orientationAngles)
```

The system computes the orientation angles by using a device's geomagnetic field sensor in combination with the device's accelerometer. Using these two hardware sensors, the system provides data for the following three orientation angles:

- **Azimuth (degrees of rotation about the -z axis).** This is the angle between the device's current compass direction and magnetic north. If the top edge of the device faces magnetic north, the azimuth is 0 degrees; if the top edge faces south, the azimuth is 180 degrees. Similarly, if the top edge faces east, the azimuth is 90 degrees, and if the top edge faces west, the azimuth is 270 degrees.

- **Pitch (degrees of rotation about the x axis).** This is the angle between a plane parallel to the device's screen and a plane parallel to the ground. If you hold the device parallel to the ground with the bottom edge closest to you and tilt the top edge of the device toward the ground, the pitch angle becomes positive. Tilting in the opposite direction— moving the top edge of the device away from the ground—causes the pitch angle to become negative. The range of values is -180 degrees to 180 degrees.
- **Roll (degrees of rotation about the y axis).** This is the angle between a plane perpendicular to the device's screen and a plane perpendicular to the ground. If you hold the device parallel to the ground with the bottom edge closest to you and tilt the left edge of the device toward the ground, the roll angle becomes positive. Tilting in the opposite direction— moving the right edge of the device toward the ground— causes the roll angle to become negative. The range of values is -90 degrees to 90 degrees.

★ **Note:** The sensor's roll definition has changed to reflect the vast majority of implementations in the geosensor ecosystem.

Note that these angles work off of a different coordinate system than the one used in aviation (for yaw, pitch, and roll). In the aviation system, the x axis is along the long side of the plane, from tail to nose.

The orientation sensor derives its data by processing the raw sensor data from the accelerometer and the geomagnetic field sensor. Because of the heavy processing that is involved, the accuracy and precision of the orientation sensor is diminished. Specifically, this sensor is reliable only when the roll angle is 0. As a result, the orientation sensor was deprecated in Android 2.2 (API level 8), and the orientation sensor type was deprecated in Android 4.4W (API level 20). Instead of using raw data from the orientation sensor, we recommend that you use the `getRotationMatrix()` method in conjunction with the `getOrientation()` method to compute orientation values, as shown in the following code sample. As part of this process, you can use the `remapCoordinateSystem()` method to translate the orientation values to your application's frame of reference.

Kotlin Java

```kotlin
class SensorActivity : Activity(), SensorEventListener {

    private lateinit var sensorManager: SensorManager
    private val accelerometerReading = FloatArray(3)
    private val magnetometerReading = FloatArray(3)

    private val rotationMatrix = FloatArray(9)
    private val orientationAngles = FloatArray(3)

    public override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.main)
        sensorManager = getSystemService(Context.SENSOR_SERVICE) as SensorManager
    }

    override fun onAccuracyChanged(sensor: Sensor, accuracy: Int) {
        // Do something here if sensor accuracy changes.
        // You must implement this callback in your code.
    }

    override fun onResume() {
        super.onResume()

        // Get updates from the accelerometer and magnetometer at a constant rate.
        // To make batch operations more efficient and reduce power consumption,
        // provide support for delaying updates to the application.
        //
        // In this example, the sensor reporting delay is small enough such that
        // the application receives an update before the system checks the sensor
        // readings again.
        sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER)?.also { accelerometer ->
            sensorManager.registerListener(
                    this,
                    accelerometer,
                    SensorManager.SENSOR_DELAY_NORMAL,
                    SensorManager.SENSOR_DELAY_UI
            )
        }
        sensorManager.getDefaultSensor(Sensor.TYPE_MAGNETIC_FIELD)?.also { magneticField ->
            sensorManager.registerListener(
                    this,
                    magneticField,
                    SensorManager.SENSOR_DELAY_NORMAL,
                    SensorManager.SENSOR_DELAY_UI
            )
        }
    }

    override fun onPause() {
        super.onPause()

        // Don't receive any more updates from either sensor.
        sensorManager.unregisterListener(this)
    }

    // Get readings from accelerometer and magnetometer. To simplify calculations,
    // consider storing these readings as unit vectors.
    override fun onSensorChanged(event: SensorEvent) {
        if (event.sensor.type == Sensor.TYPE_ACCELEROMETER) {
            System.arraycopy(event.values, 0, accelerometerReading, 0, accelerometerReading.size)
        } else if (event.sensor.type == Sensor.TYPE_MAGNETIC_FIELD) {
            System.arraycopy(event.values, 0, magnetometerReading, 0, magnetometerReading.size)
        }
    }

    // Compute the three orientation angles based on the most recent readings from
    // the device's accelerometer and magnetometer.
    fun updateOrientationAngles() {
        // Update rotation matrix, which is needed to update orientation angles.
        SensorManager.getRotationMatrix(
                rotationMatrix,
                null,
                accelerometerReading,
                magnetometerReading
        )

        // "rotationMatrix" now has up-to-date information.

        SensorManager.getOrientation(rotationMatrix, orientationAngles)

        // "orientationAngles" now has up-to-date information.
    }
}
```

You don't usually need to perform any data processing or filtering of the device's raw orientation angles other than translating the sensor's coordinate system to your application's frame of reference.

Use the geomagnetic field sensor

The geomagnetic field sensor lets you monitor changes in the earth's magnetic field. The following code shows you how to get an instance of the default geomagnetic field sensor:

Kotlin Java

```kotlin
private lateinit var sensorManager: SensorManager
private var sensor: Sensor? = null
...
sensorManager = getSystemService(Context.SENSOR_SERVICE) as SensorManager
sensor = sensorManager.getDefaultSensor(Sensor.TYPE_MAGNETIC_FIELD)
```

★ **Note:** If your app targets Android 12 (API level 31) or higher, this sensor is rate-limited.

This sensor provides raw field strength data (in μT) for each of the three coordinate axes. Usually, you do not need to use this sensor directly. Instead, you can use the rotation vector sensor to determine raw rotational movement or you can use the accelerometer and geomagnetic field sensor in conjunction with the `getRotationMatrix()` method to obtain the rotation matrix and the inclination matrix. You can then use these matrices with the `getOrientation()` and `getInclination()` methods to obtain azimuth and geomagnetic inclination data.

★ **Note:** When testing your app, you can improve the sensor's accuracy by waving the device in a figure-8 pattern.

Use the uncalibrated magnetometer

The uncalibrated magnetometer is similar to the geomagnetic field sensor, except that no hard iron calibration is applied to the magnetic field. Factory calibration and temperature compensation are still applied to the magnetic field. The uncalibrated magnetometer is useful to handle bad hard iron estimations. In general, `geomagneticsensor_event.values[0]` will be close to `uncalibrated_magnetometer_event.values[0] - uncalibrated_magnetometer_event.values[3]`. That is,

`calibrated_x ~= uncalibrated_x - bias_estimate_x`

★ **Note:** Uncalibrated sensors provide more raw results and may include some bias, but their measurements contain fewer jumps from corrections applied through calibration. Some applications may prefer these uncalibrated results as smoother and more reliable. For instance, if an application is attempting to conduct its own sensor fusion, introducing calibrations can actually distort results.

In addition to the magnetic field, the uncalibrated magnetometer also provides the estimated hard iron bias in each axis. The following code shows you how to get an instance of the default uncalibrated magnetometer:

Kotlin Java

```kotlin
private lateinit var sensorManager: SensorManager
private var sensor: Sensor? = null
...
sensorManager = getSystemService(Context.SENSOR_SERVICE) as SensorManager
sensor = sensorManager.getDefaultSensor(Sensor.TYPE_MAGNETIC_FIELD_UNCALIBRATED)
```

Use the proximity sensor

The proximity sensor lets you determine how far away an object is from a device. The following code shows you how to get an instance of the default proximity sensor:

Kotlin Java

```kotlin
private lateinit var sensorManager: SensorManager
private var sensor: Sensor? = null
...
sensorManager = getSystemService(Context.SENSOR_SERVICE) as SensorManager
sensor = sensorManager.getDefaultSensor(Sensor.TYPE_PROXIMITY)
```

The proximity sensor is usually used to determine how far away a person's head is from the face of a handset device (for example, when a user is making or receiving a phone call). Most proximity sensors return the absolute distance, in cm, but some return only near and far values.

★ **Note:** On some device models, the proximity sensor is located underneath the screen, which can cause a blinking dot to appear on the screen if enabled while the screen is on.

The following code shows you how to use the proximity sensor:

Kotlin Java

```
class SensorActivity : Activity(), SensorEventListener {

    private lateinit var sensorManager: SensorManager
    private var proximity: Sensor? = null

    public override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.main)

        // Get an instance of the sensor service, and use that to get an instance of
        // a particular sensor.
        sensorManager = getSystemService(Context.SENSOR_SERVICE) as SensorManager
        proximity = sensorManager.getDefaultSensor(Sensor.TYPE_PROXIMITY)
    }

    override fun onAccuracyChanged(sensor: Sensor, accuracy: Int) {
        // Do something here if sensor accuracy changes.
    }

    override fun onSensorChanged(event: SensorEvent) {
        val distance = event.values[0]
        // Do something with this sensor data.
    }

    override fun onResume() {
        // Register a listener for the sensor.
        super.onResume()

        proximity?.also { proximity ->
            sensorManager.registerListener(this, proximity, SensorManager.SENSOR_DELAY_NORMAL)
        }
    }

    override fun onPause() {
        // Be sure to unregister the sensor when the activity pauses.
        super.onPause()
        sensorManager.unregisterListener(this)
    }
}
```

★ **Note:** Some proximity sensors return binary values that represent "near" or "far." In this case, the sensor usually reports its maximum range value in the far state and a lesser value in the near state. Typically, the far value is a value > 5 cm, but this can vary from sensor to sensor. You can determine a sensor's maximum range by using the `getMaximumRange()` method.

You should also read

- Sensors

- Sensors Overview
- Motion Sensors
- Environment Sensors

Was this helpful?

LinkedIn

MORE ANDROID

Android

Android for Enterprise

Security

Source

News

Blog

Podcasts

DISCOVER

Gaming

Machine Learning

Privacy

5G

ANDROID DEVICES

Large screens

Wear OS

ChromeOS devices

Android for cars

Android Things

Android TV

RELEASES

Android 13

Android 12

Android 11

Android 10

Pie

Oreo

Nougat

DOCUMENTATION AND DOWNLOADS

Android Studio guide

Developers guides

API reference

Download Studio

Android NDK

SUPPORT

Report platform bug

Report documentation bug

Google Play support

Join research studies