SensorTag User Guide

From Texas Instruments Wiki

Back to *Bluetooth* SensorTag main page (http://processors.wiki.ti.com/index.php/Bluetooth_SensorTag)

Contents

SensorTag Hardware Overview

The Bluetooth SensorTag packs the following sensor:

- Contactless IR temperature sensor (Texas Instruments TMP006)
- Humidity Sensor (Sensirion SHT21)
- Gyroscope (Invensense IMU-3000)
- Accelerometer (Kionix KXTJ9)
- Magnetometer (Freescale MAG3110)
- Barometric pressure sensor (Epcos T5400)

- On-chip temperature sensor (Built into the CC2541)
- Battery/voltage sensor (Built into the CC2541)

The reference design is fully tested and certified for by the following authorities: - FCC/IC for use of radio equipment in US and Canada - ETSI/CE for use of radio equipment in EU - Bluetooth SIG approved as Bluetooth 4.0 accessories For more information about approvals and certification of your own product please see How to Certify your Bluetooth product

Sensor Suppliers

TMP006 Contactless IR Temperature Sensor, Texas Instruments (http://www.ti.com/tmp006)
SHT21 Humidity Sensor, Sensirion (http://www.sensirion.com)
IMU-3000 Gyroscope, Invensense (http://www.invensense.com)
KXTJ9 Accelerometer, Kionix (http://www.kionix.com)
MAG3110 Magnetometer, Freescale (http://www.freescale.com)
T5400(C953H) Barometric Pressure Sensor, Epcos (http://www.epcos.com)

Other Components

CC2541 Bluetooth Low Energy Radio SoC, Texas Instruments (http://www.ti.com/product/cc2541)
TPS62730 Ultra Low Power DC/DC Converter, Texas Instruments (http://www.ti.com/product/tps62730)


SensorTag

SensorTag Kit Content

The SensorTag kit includes one SensorTag with enclosure and battery. The plastic enclosure can easily be removed to access the internals of the SensorTag. To use the SensorTag a smartphone with Bluetooth 4.0 is required. A list of supported devices is updated here Supported App Development Platforms

Optional Accessories

CC Debugger (Purchased Separately)

USB Debug Interface for programming and debugging SensorTag firmware The CC Debugger is not required for normal testing and development of SensorTag applications. The SensorTag is configurable from the smartphone app example or the PC tool BLE Device Monitor (http://processors.wiki.ti.com/index.php/BLE_Device_Monitor_User_Guide), provided as a stand-alone tool.

If you want to make custom versions of the SensorTag firmware the CC debugger can be used to debug applications or download hex files. See CC debugger product page (http://www.ti.com/tool/cc-debugger).

CC Debugger

CC2540 USB dongle(Purchased Seperately)

The USB dongle can be used for packet sniffer of RF activity and it is needed for the BLE Device Monitor (http://processors.wiki.ti.com/index.php/BLE_Device_Monitor_User_Guide).
CC2540 USB dongle product page (http://www.ti.com/tool/cc2540emk-usb)

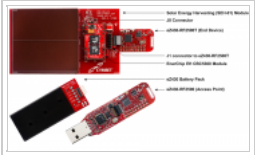

CC2540 USB dongle

Battery Pack (Purchased Seperately)

Used for battery power for the SensorTag for applications that requires longer battery life. Can be purchased as part of several EZ430 development kits or CC2530ZNP ZigBee kits.Note that a connector(not included) for the battery pack must be soldered onto the SensorTag board

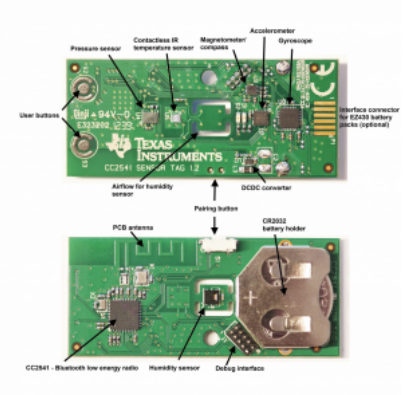

Battery Board

EZ430-RF2500 development tool (http://www.ti.com/tool/ez430-rf2500t)
CC2530 ZigBee Network Processor development tool (http://www.ti.com/tool/cc2530zdk-znp-mini)
There is an solar powered option as well:
Solar Energy Harvesting kit (http://www.ti.com/tool/ez430-rf2500-seh)


Energy Harvesting Kit

SensorTag Hardware

The key components of the SensorTag is shown in the picture below.



The SensorTag reference design files including schematic and layout can be found here (http://www.ti.com/lit/zip/swrr109)

Insert Battery

Note! When inserting battery the first time, the PCB contact point surface may have a thin layer of solder residue that may prevent contact to the battery. If the SensorTag is not working as expected please try to remove and insert the battery a few times to power the SensorTag.


Inserting batteries

Side Button

When not connected, this button is used to toggle advertising on and off. In pushed when in a connection, the connection is dropped, sensors reset and all data values are set to zero (Soft reset).

SensorTag side button

LEDs

The D1 LED (Green) blinks when the device is advertising. (If lit, it is hung and a power cycle is needed)

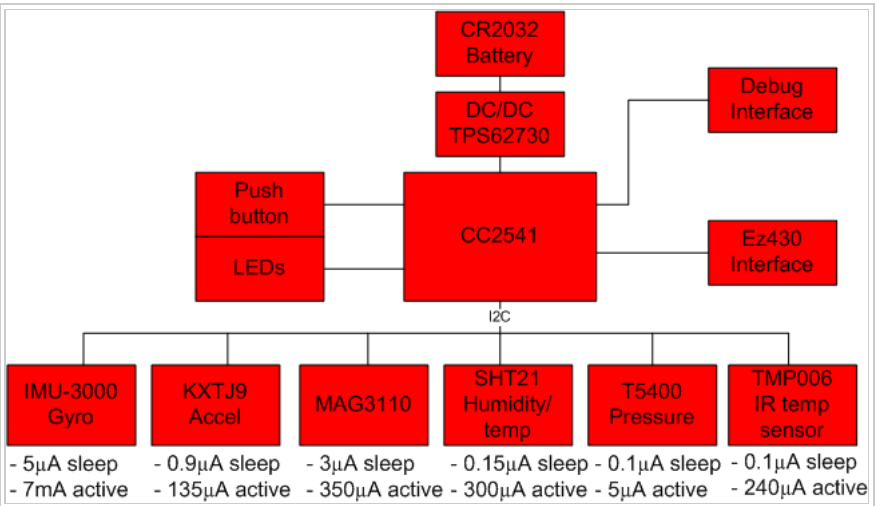
Bluetooth Low Energy Radio

The SensorTag includes a CC2541 Bluetooth Low Energy radio System on Chip with 256KB flash and 8KB RAM. The devices is preprogrammed with Bluetooth Low Energy software stack firmware for configuring the sensors and communicating with Bluetooth 4.0 enabled smartphones. The firmware is included in the BLE stack v1.30 (http://www.ti.com/tool/ble-stack) and newer.

The datasheet for CC2541, reference designs, and appnotes can be found on the CC2541 product page (http://www.ti.com/product/cc2541). This page also includes complete schematic and design files for the SensorTag.

Sensors

All sensors on the SensorTag are chosen to be small and low cost surface mount devices. The sensors use I2C interface and are connected to the same interface bus with seperate enable signals. To minimize current consumption all sensors are by default disabled and they are in sleep mode between measurements. Apps developers can enable and read data from one or several sensors and use the data to develop creative apps. The block diagram below shows the sensors and lists some potential apps that can be enabled using the sensors



Bluetooth SensorTag block diagram

Contactless IR Temperature Sensor

The IR temp sensor temperature can measure the temperature of objects in front of the SensorTag with typical +/- 1C accuracy. The measurement distance depends on the size of the object to be measured. The datasheet recommends maximum distance of the radius/2. This means if the object has a radius of 1 meter, the distance from the object should be less than 50cm.
Note that polished and shiny metal surface objects cannot accurately be measured with the IR temperature sensor. To understand the theory of operation and how to use IR temperature sensor please see TMP006 user guide (http://www.ti.com/product/tmp006) and this document describing different lens materials.

Accelerometer

The accelerometer measures acceleration in 3 axis with programmable resolution up to 14 bit. The accelerometer can also measure direction of gravity unlike a gyroscope that can only measure change of direction. More information about the KXTJ9 accelerometer can be found on the Kionix web site (http://www.kionix.com/accelerometers/kxtj9)

Humidity Sensor

The humidity sensor reports the relative humidity (%RH) and temperature. This is an integrated 12-bit relative humidity sensor with a 14-bit temperature sensor. In order to ensure the best performance of relative humidity and temperature measurement, there are some basic rules of design-in. The sensor requires for direct contact with ambient air (like e.g. provided by a hole in the enclosure), while at the same time the sensor should be airtight towards the inner volume of the device. Another basic rule is the thermal decoupling from hot spots on the electronic board, while allowing optimal thermal coupling towards the outside

Please be aware that the sensor, despite its robustness and long term stability, requires for some basic knowledge of handling while production and storage. More information about the optimal design-in “Design Guide Humidity Sensors”, the basic rules of handling for the sensor “Handling Instructions Humidity Sensors” and the SHT21 humidity sensor datasheet can be found on the Sensirion web site (http://www.sensirion.com/en/products/humidity-temperature/humidity-sensor-sht21) and Download center (http://www.sensirion.com/en/products/humidity-temperature/download-center/)

Magnetometer/Compass

The Magnetometer measures the magnetic field in 3 axis. The data from the magnetometer is usually combined with a accelerometer to make a compass. Magnetometers can be used to measure the earth's magnetic field but also to measure local magnetic field created from electronics. More information about the MAG3110 magnetometer can be found on the Freescale web site (http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=MAG3110)

Barometric Pressure Sensor

The pressure sensor measures the barometric air pressure. This is a 16-bit pressure sensor with 16-bit temperature reading. More information about the T5400 pressure sensor can be found in the T5400 data sheet (http://www.epcos.com/inf/57/ds/T5400.pdf) and Pressure Sensor appnote (http://www.epcos.com/web/generator/Web/Sections/ProductCatalog/Sensors/PressureSensors/T5400-ApplicationNote,property=Data__en.pdf;/T5400_ApplicationNote.pdf)

Gyroscope

The gyroscope measures the rate of rotation in all 3 axes (X, Y, Z) with up to 16-bit resolution. Readings from the gyroscope can be combined with readings from the accelerometer to determine the orientation of the SensorTag in 3-D space.” More information about the IMU-3000 gyroscope can be found on the Invensense web site (http://www.invensense.com/mems/gyro/imu3000.html)

EZ430 Battery Pack Interface

The P1 connector can be used to interface to EZ430 battery packs and USB sticks. This interface is primarily designed as a battery pack connector and not as a data interface. The UART interface is not supported in the SensorTag firmware. To use the interface a connector must be soldered to the board. The connector is type is BL SMD 91-06 U from MPE Gerry (http://www.mpe-connector.com)

P1 Connector Pinout

P1 pin number	Signal name
1	TX/D0(Not supported in firmware)
2	GND
3	RESET
4	TEST(Not supported in firmware)
5	VDD_EXT
6	RX/D0(Not supported in firmware)

Debug interface

For normal operation the SensorTag is configured by the smartphone App examples. If you are an advanced user you can use your own firmware on the SensorTag. A 2x5 pin 1.27 mm pitch header (J1) is used for programming and debugging the SensorTag. The pinout of this connector is shown below. The CC2541 can be programmed with a [CC debugger (http://http://focus.ti.com/docs/toolsw/folders/print/cc-debugger.html)]. When using the CC debugger, notice the small pin 1 marking on the connector.
Note that while using the CC debugger the SensorTag is powered from the coin cell battery. During debugging the current consumption is higher than many batteries can support. It is recommended to use external power while debugging

Debug Connector Pinout

Signal name	Pin #	Pin #	Signal name
GND	1	2	VDD
DC	3	4	DD
CSN	5	6	SCLK
EM_RESET	7	8	MOSI
NC	9	10	MISO

Applying external power to the SensorTag

There are two ways to power the SensorTag with external power:

External power through the P1 connector

The SensorTag can be powered through the EZ430 battery pack pinrow P1 by connecting GND to pin 2 and VDD to pin 5. Note that there are several resistors which are not mounted by default in this interface and a 0 ohm resistor needs to be soldered on (R2 in the reference design) to use VDD through this interface.

External power through the debug connector

To power the SensorTag through the CC Debugger you will need to connect pin 2 and pin 9 in the CC Debugger. This will power the SensorTag through pin 9 and at the same time power the level shifter inputs inside the Debugger connected to pin 9.

Do not use this option while having a battery inserted into the SensorTag as this will apply power to a non-rechargeable coin cell battery!

SensorTag Software

The SensorTag is a BLE peripheral slave device based on CC2541 hardware platform including five peripheral sensors with a complete software solution for sensor drivers interfaced to a GATT server running on TI BLEv1.4 stack. The GATT server contains a primary service for each sensor for configuration and data collection.

Operation

On start-up, the SensorTag is advertising with a 100ms interval. The connection is established by a Central Device and the sensors can then be configured to provide measurement data. The Central Device could be any BLE compliant device and the main focus is on BLE compliant mobile phones, running either Android (4.3+) or iOS. The central device should be able to

- Scan and discover the Sensor Tag. (Scan response contain name “SensorTag”)
- Establish connection based on user defined Connection Parameters
- Perform Service Discovery – Discover Characteristic by UUID
- Operate as a GATT Client - Write to and read from Characteristic Value

The Central Device shall initiate the connection and thereby become the Master.

To obtain the data, the corresponding sensor must first be activated, which is done via a Characteristic Value write to appropriate service.

The most power efficient way to obtain measurements for a sensor is to

1. Enable notification
2. Enable Sensor
3. When notification with data is obtained at the Master side, disable the sensor (notification still on though)

Alternative do not use notifications at all, then simply

1. Enable sensor
2. Read data and verify
3. Disable sensor

For the latter alternative please keep in mind that sensor take different amount of time to perform measurement. Depending on the connection interval (7.5 – 4000 ms) set by the Central Device, the time for achieving measurement data can vary. The individual sensors require varying delays to complete measurements. Recommended setting is 100ms but for fast accelerometer and Magnetometer data updates, a lower is necessary. Notifications can be stopped and the sensors turned on/off. Note that when using iOs or Android the application has no direct control of the connection parameters. The SensorTag does however provide a Connection Control Service to circumvent this limitation.

Sensors

The SensorTag has support for the following sensors:

- IR Temperature, both object and ambient temperature
- Accelerometer, 3 axis
- Humidity, both relative humidity and temperature
- Magnetometer, 3 axis
- Barometer, both pressure and temperature
- Gyroscope, 3 axis

Sensor Overview

Sensor	Name	Update	Size per Update	I2C Address	I2C Speed
IR Temperature	TMP006	>250 ms	2 x 16bit	0x44	<3.4 MHz
Accelerometer	KXTJ9	>20 ms	3 x 8bit	0x0F	<400 kHz
Gyroscope	IMU-3000	>0.125 ms	3 x 16 bit	0x68	<3.4 MHz
Humidity	SHT21	N/A	2 x 16bit	0x40	<400 kHz
Pressure	T5400(C953A)	>2 ms	2 x 16bit	0x77	<3.4 MHz
Compass	MAG3110	>12 ms	3 x 16bit	0x0E	<400 kHz

The SensorTag uses I2C to interface to all sensors.
The SensorTag can be configured to send notifications for every sensor by writing “01 00” to the characteristic configuration < GATT_CLIENT_CHAR_CFG_UUID> for the corresponding sensor data, the data is then sent as soon as the data has been updated. The sensors are enabled by writing 0x01 (NB: Gyroscope has a different code) to the corresponding Configuration and then disabled by writing 0x00
The SensorTag does not use any interrupt features provided by the sensors

Gatt Server

The GATT server contains several services and following are used:

- GAP
- GATT
- Device Information
- IR Temperature
- Accelerometer
- Humidity
- Magnetometer
- Barometer
- Gyroscope
- Simple Keys
- Test
- OAD Service
- Connection Control Service

Device Information Service is aligned to official SIG profiles, whilst the other serves as examples of profile service implementation. The complete Attribute Table for the SensorTag can be found in BLE_SensorTag_GATT_Server.pdf. The sensor profile is a custom profile with 128 bit unique UUIDs. The profile enables a device that has a set of specific sensors to expose status and data measurements to another device. The profile also provides configuration interface to the individual sensors, each supporting a configuration characteristic (enable/disable) and a period characteristic.

The TI Base 128-bit UUID is: **F0000000-0451-4000-B000-000000000000**. All sensor services use 128-bit UUIDs, but for practical reasons only the 16-bit part is listed in this document. It is embedded in the 128-bit UUID as shown by example. Example: 0xAA01 maps as F000**AA01**-0451-4000-B000-000000000000. All UUIDs that are mapped to 128-bit values are marked *.

IR Temperature Sensor

Texas Instruments TMP006 @ U5 Two types of data are obtained from the IR Temperature sensor: object temperature and ambient temperature.

IR Temperature Sensor

Type	UUID	Read/Write	Format
<Data>	AA01 *	Read/Notify	ObjLSB ObjMSB AmbLSB AmbMSB (4 bytes)
<Data Notification>		R/W	2 bytes
<Configuration>	AA02 *	R/W	1 byte
<Period>	AA03 *	R/W	1 byte

When the enable command is issued, the sensor starts to perform measurements each second (average over four measurements) and the data is stored in the <Data> each second as well. When the disable command is issued, the sensor is put in stand-by mode. To obtain data OTA either use notifications or read the data directly. The period range varies from 300 ms to 2.55 seconds. The unit is 10 ms. i.e. writing 0x32 gives 500 ms, 0x64 1 second etc. The default value is 1 second.

For more information please refer to TI TMP006 User's Guide (http://www.ti.com/lit/ug/sbou107/sbou107.pdf)

The raw data value read from this sensor are two unsigned 16 bit values, one for die temperature and one for object temperature. To convert to temperature in degrees (Celsius), use the algorithm below:

C	Java

	`public void onCharacteristicChanged(BluetoothGattCharacteristic c){`
	
	`/* The IR Temperature sensor produces two measurements;`
	`* Object ( AKA target or IR) Temperature,`
	`* and Ambient ( AKA die ) temperature.`
	`*`
	`* Both need some conversion, and Object temperature is dependent on Ambient temperature.`
	`*`
	`* They are stored as [ObjLSB, ObjMSB, AmbLSB, AmbMSB] (4 bytes)`

```
/*  Conversion algorithm for die temperature */
double calcTmpLocal(uint16 rawT)
{
  //-- calculate die temperature [°C] --
  m_tmpAmb = (double)((qint16)rawT)/128.0; // Used in also in the calc. below
  return m_tmpAmb;
}


import static java.lang.Math.pow;

/* Conversion algorithm for target temperature */
double calcTmpTarget(uint16 rawT)
{
  //-- calculate target temperature [°C] -
  double Vobj2 = (double)(qint16)rawT;
  Vobj2 *= 0.00000015625;

  double Tdie2 = m_tmpAmb + 273.15;
  const double S0 = 6.4E-14;            // Calibration factor

  const double a1 = 1.75E-3;
  const double a2 = -1.678E-5;
  const double b0 = -2.94E-5;
  const double b1 = -5.7E-7;
  const double b2 = 4.63E-9;
  const double c2 = 13.4;
  const double Tref = 298.15;
  double S = S0*(1+a1*(Tdie2 - Tref)+a2*pow((Tdie2 - Tref),2));
  double Vos = b0 + b1*(Tdie2 - Tref) + b2*pow((Tdie2 - Tref),2);
  double fObj = (Vobj2 - Vos) + c2*pow((Vobj2 - Vos),2);
  double tObj = pow(pow(Tdie2,4) + (fObj/S),.25);
  tObj = (tObj - 273.15);

  return tObj;
}
```

```
     * Which means we need to shift the bytes around to get the correct values.
     */

    double ambient = extractAmbientTemperature(c);
    double target = extractTargetTemperature(c, ambient);

    model.setAmbientTemperature(ambient);
    model.setTargetTemperature(target);
}

private double extractAmbientTemperature(BluetoothGattCharacteristic c) {
    int offset = 2;
    return shortUnsignedAtOffset(c, offset) / 128.0;
}

private double extractTargetTemperature(BluetoothGattCharacteristic c, double ambient) {
    Integer twoByteValue = shortSignedAtOffset(c, 0);

    double Vobj2 = twoByteValue.doubleValue();
    Vobj2 *= 0.00000015625;

    double Tdie = ambient + 273.15;

    double S0 = 5.593E-14;      // Calibration factor
    double a1 = 1.75E-3;
    double a2 = -1.678E-5;
    double b0 = -2.94E-5;
    double b1 = -5.7E-7;
    double b2 = 4.63E-9;
    double c2 = 13.4;
    double Tref = 298.15;
    double S = S0*(1+a1*(Tdie - Tref)+a2*pow((Tdie - Tref),2));
    double Vos = b0 + b1*(Tdie - Tref) + b2*pow((Tdie - Tref),2);
    double fObj = (Vobj2 - Vos) + c2*pow((Vobj2 - Vos),2);
    double tObj = pow(pow(Tdie,4) + (fObj/S),.25);

    return tObj - 273.15;
}


/**
 * Gyroscope, Magnetometer, Barometer, IR temperature
 * all store 16 bit two's complement values in the awkward format
 * LSB MSB, which cannot be directly parsed as getIntValue(FORMAT_SINT16, offset)
 * because the bytes are stored in the "wrong" direction.
 *
 * This function extracts these 16 bit two's complement values.
 * */
private static Integer shortSignedAtOffset(BluetoothGattCharacteristic c, int offset) {
    Integer lowerByte = c.getIntValue(FORMAT_UINT8, offset);
    Integer upperByte = c.getIntValue(FORMAT_SINT8, offset + 1); // Note: interpret MSB as sig

    return (upperByte << 8) + lowerByte;
}

private static Integer shortUnsignedAtOffset(BluetoothGattCharacteristic c, int offset) {
    Integer lowerByte = c.getIntValue(FORMAT_UINT8, offset);
    Integer upperByte = c.getIntValue(FORMAT_UINT8, offset + 1); // Note: interpret MSB as uns

    return (upperByte << 8) + lowerByte;
}
```

Accelerometer

Kionix KXTJ9 @ U7 3 axis data are read from the accelerometer

Accelerometer

Type	UUID	Handle	Read/Write	Format	Description
<Data>	F000AA11 *	0x2D	Read/Notify	X : Y : Z (3 bytes)	
<Data Notification>		0x2E	R/W	2 bytes	
<Configuration>	F000AA12 *	0x31	R/W	1 byte	
<Period>	F000AA13 *	0x34	R/W	1 byte	Period = [Input*10]ms

When the enable command is issued, the sensor starts to perform measurements each second and the data is stored in each second as well. When the disable command is issued, the sensor is put to sleep. To obtain data OTA either use notifications or read the data directly. The default period for the data is one second, which can be changed by writing to the <Period> (unit 10 ms). Range is set to +/-2g. For more information please refer to Kionix KXTJ9 Data sheet (http://www.kionix.com/accelerometers/kxtj9).

For conversion from raw data to g, use this algorithm:

C

```
float calcAccel(int8 rawX)
{
  float v;
  //-- calculate acceleration, unit g, range -2, +2
  v = (rawX * 1.0) / (64);
  return v;
}
```

Java

```
public void onCharacteristicChanged(final BluetoothGattCharacteristic c){
    /*
     * The accelerometer has the range [-2g, 2g] with unit (1/64)g.
     *
     * To convert from unit (1/64)g to unit g we divide by 64.
     *
     * (g = 9.81 m/s^2)
     *
     * The z value is multiplied with -1 to coincide
     * with how we have arbitrarily defined the positive y direction.
     * (illustrated by the apps accelerometer image)
     * */

    Integer x = c.getIntValue(FORMAT_SINT8, 0);
    Integer y = c.getIntValue(FORMAT_SINT8, 1);
    Integer z = c.getIntValue(FORMAT_SINT8, 2) * -1;

    double scaledX = x / 64.0;
    double scaledY = y / 64.0;
    double scaledZ = z / 64.0;
```

```
    model.setAccelerometer(scaledX , scaledY, scaledZ);
}
```

Humidity Sensor

Sensirion SHT21 @ U6 Two types of data are obtained from the Humidity sensor, relative humidity and ambient temperature

Humidity Sensor

Type	UUID	Read/Write	Format
<Data>	AA21 *	Read/Notify	TempLSB TempMSB HumLSB HumMSB (4 bytes)
<Data Notification>		R/W	2 bytes
<Configuration>	AA22 *	R/W	1 byte
<Period>	AA23 *	R/W	1 byte

The driver for this sensor is using a state machine so when the enable command is issued, the sensor starts to perform one measurements and the data is stored in the <Data>. To obtain data OTA either use notifications or read the data directly. The update rate ranges from 100 ms to 2.55 seconds.

The humidity and temperature data in the sensor is issued and measured explicitly where the humidity data takes ~64ms to measure. For more information please refer to Sensirion SHT21 data sheet (http://www.sensirion.com/en/pdf/product_information/Datasheet-humidity-sensor-SHT21.pdf)

Conversion to temperature and relative humidity is done as follows:

C

```
/* Conversion algorithm, temperature */

double calcHumTmp(uint16 rawT)
{
  double v;

  //-- calculate temperature [deg C] --
  v = -46.85 + 175.72/65536 *(double)(qint16)rawT;

  return v;
}

/*  Conversion algorithm, humidity */

double calcHumRel(uint16 rawH)
{
  double v;

  rawH &= ~0x0003; // clear bits [1..0] (status bits)
  //-- calculate relative humidity [%RH] --
  v = -6.0 + 125.0/65536 * (double)rawH; // RH= -6 + 125 * SRH/2^16

  return v;
}
```

Java

```
public void onCharacteristicChanged(BluetoothGattCharacteristic c) {
    int a = shortUnsignedAtOffset(c, 2);
    // bits [1..0] are status bits and need to be cleared according
    // to the userguide, but the iOS code doesn't bother. It should
    // have minimal impact.
    a = a - (a % 4);

    model.setHumidity((-6f) + 125f * (a / 65535f));
}

private static Integer shortUnsignedAtOffset(BluetoothGattCharacteristic c, int offset) {
    Integer lowerByte = c.getIntValue(FORMAT_UINT8, offset);
    Integer upperByte = c.getIntValue(FORMAT_UINT8, offset + 1); // Note: interpret MSB as unsigned.

    return (upperByte << 8) + lowerByte;
}
```

Magnetometer

Freescale MAG3110 @ U3 3 axis data are obtained from the magnetometer

Magnetometer

Type	UUID	Read/Write	Format	Description
<Data>	AA31 *	Read/Notify	XLSB XMSB YLSB YMSB ZLSB ZMSB	
<Data Notification>		R/W	2 bytes	
<Configuration>	AA32 *	R/W	1 byte	
<Period>	AA33 *	R/W	1 byte	Period = [Input*10]ms

When the enable command is issued, the sensor starts to perform measurements each second and the data is stored in <Data> each second as well. When the disable command is issued, the sensor is put to sleep. To obtain data OTA either use notifications or read the data directly.

The default period for the data is 2 seconds, which can be changed by writing to the <Period> characteristic. The period has a resolution of 10 milliseconds and the range is 100 ms to 2.55 seconds.

The sensor is set to an output rate of 10 Hz with an over sampling ratio of 1. Through the source code, the sensor is configurable from 0.08-80 Hz with an over sampling rate of 1-8. For more information please refer to Freescale MAG3110 datasheet (http://cache.freescale.com/files/sensors/doc/fact_sheet/MAG3110FS.pdf)

Use the following algorithm top convert from raw values to uTesla (uT):

C

Java

```
public void onCharacteristicChanged(BluetoothGattCharacteristic c) {
    // Multiply x and y with -1 so that the values correspond with our pretty pictures in the app.
    float x = shortSignedAtOffset(c, 0) * (2000f / 65536f) * -1;
    float y = shortSignedAtOffset(c, 2) * (2000f / 65536f) * -1;
```

```
/* Converting to microTeslas */

float calcMagn(int16 rawX)
{
  float v;

  //-- calculate magnetic-field strength, unit uT, range -1000, +1000
  v = (rawX * 1.0) / (65536/2000);

  return v;
}
```

```
   float z = shortSignedAtOffset(c, 4) * (2000f / 65536f);

   model.setMagnetometer(x, y, z);
}

/**
* Gyroscope, Magnetometer, Barometer, IR temperature
* all store 16 bit two's complement values in the awkward format
* LSB MSB, which cannot be directly parsed as getIntValue(FORMAT_SINT16, offset)
* because the bytes are stored in the "wrong" direction.
*
* This function extracts these 16 bit two's complement values.
* */
private static Integer shortSignedAtOffset(BluetoothGattCharacteristic c, int offset) {
   Integer lowerByte = c.getIntValue(FORMAT_UINT8, offset);
   Integer upperByte = c.getIntValue(FORMAT_SINT8, offset + 1); // Note: interpret MSB as signed.

   return (upperByte << 8) + lowerByte;
}
```

Barometric Pressure Sensor

Epcos T5400-C953 @ U4 Two types of data are obtained from the Barometric Pressure Sensor: pressure and ambient temperature

Barometric Pressure Sensor

Type	UUID	Read/Write	Format
<Data>	AA41 *	Read only	TempLSB TempMSB PressLSB PressMSB (4 bytes)
<Data Notification>		R/W	2 bytes
<Configuration>	AA42 *	R/W	1 byte
<Calibration>	AA43 *	Read only	C1LSB C1MSB…C8LSB C8MSB (16 bytes)
<Period>	AA44 *	R/W	1 Byte

The driver for this sensor is using a state machine so when the enable command is issued, the sensor starts to perform one measurements and the data is stored in the <Data>. To obtain data OTA either use notifications or read the data directly. The period ranges for 100 ms to 2.55 seconds, resolution 10 ms.

For calculation, the 8 16-bit pairs of calibration values are needed which is read from sensor and stored in the <Calibration> by writing “02” to <Configuration>. The Calibration values are obtained by either read from <Calibration> or if notification has been enabled automatically sent when available.

Conversion from raw data to temperature and hP (hecto-pascal) is achieved with the following conversion:

C

```
/*  Conversion algorithm for barometer temperature
 *
 *  Formula from application note, rev_X:
 *  Ta = ((c1 * Tr) / 2^24) + (c2 / 2^10)
 *
 *  c1 - c8: calibration coefficients the can be read from the sensor
 *  c1 - c4: unsigned 16-bit integers
 *  c5 - c8: signed 16-bit integers
 */

double calcBarTmp(uint16 rawT)
{
  uint16 c1, c2;

  c1 = m_barCalib.c1;
  c2 = m_barCalib.c2;
  m_raw_temp = rawT;

  int64 temp, val;
  val = ((int64)(c1 * m_raw_temp) * 100);
  temp = (val >> 24);
  val = ((int64)c2 * 100);
  temp += (val >> 10);

  return ((double)temp) / 100;
}

/* Conversion algorithm for barometer pressure (hPa)
 *
 * Formula from application note, rev_X:
 * Sensitivity = (c3 + ((c4 * Tr) / 2^17) + ((c5 * Tr^2) / 2^34))
 * Offset = (c6 * 2^14) + ((c7 * Tr) / 2^3) + ((c8 * Tr^2) / 2^19)
 * Pa = (Sensitivity * Pr + Offset) / 2^14
 */

double TcalcBarPress(uint16 rawT)
{
  int64 s, o, pres, val;
  uint16 c3, c4;
  int16 c5, c6, c7, c8;
  uint16 Pr;
  int16 Tr;

  Pr = rawT;
  Tr = m_raw_temp;
  c3 = m_barCalib.c3;
  c4 = m_barCalib.c4;
  c5 = m_barCalib.c5;
  c6 = m_barCalib.c6;
  c7 = m_barCalib.c7;
  c8 = m_barCalib.c8;

  // Sensitivity
  s = (int64)c3;
  val = (int64)c4 * Tr;
```

Java

```
import static java.lang.Math.pow;
public void onCharacteristicChanged(BluetoothGattCharacteristic characte
    // c holds the calibration coefficients

    final Integer t_r;  // Temperature raw value from sensor
    final Integer p_r;  // Pressure raw value from sensor
    final Double t_a;   // Temperature actual value in unit centi degrees
    final Double S;     // Interim value in calculation
    final Double O;     // Interim value in calculation
    final Double p_a;   // Pressure actual value in unit Pascal.

    t_r = shortSignedAtOffset(characteristic, 0);
    p_r = shortUnsignedAtOffset(characteristic, 2);

    t_a = (100 * (c[0] * t_r / pow(2,8) + c[1] * pow(2,6))) / pow(2,16);
    S = c[2] + c[3] * t_r / pow(2,17) + ((c[4] * t_r / pow(2,15)) * t_r)
    O = c[5] * pow(2,14) + c[6] * t_r / pow(2,3) + ((c[7] * t_r / pow(2,1
    p_a = (S * p_r + O) / pow(2,14);

    model.setBarometer(p_a);
}
```

```
  s += (val >> 17);
  val = (int64)c5 * Tr * Tr;
  s += (val >> 34);

  // Offset
  o = (int64)c6 << 14;
  val = (int64)c7 * Tr;
  o += (val >> 3);
  val = (int64)c8 * Tr * Tr;
  o += (val >> 19);

  // Pressure (Pa)
  pres = ((int64)(s * Pr) + o) >> 14;

  return (double)pres/100;
}

#define BUILD_UINT16(loByte, hiByte) ((uint16)(((loByte) & 0x00FF) + (((hiByte) & 0x00FF) << 8)))

void storeCalibrationData(uint8 *pData)
{
  m_barCalib.c1 = BUILD_UINT16(pData[0],pData[1]);
  m_barCalib.c2 = BUILD_UINT16(pData[2],pData[3]);
  m_barCalib.c3 = BUILD_UINT16(pData[4],pData[5]);
  m_barCalib.c4 = BUILD_UINT16(pData[6],pData[7]);
  m_barCalib.c5 = BUILD_UINT16(pData[8],pData[9]);
  m_barCalib.c6 = BUILD_UINT16(pData[10],pData[11]);
  m_barCalib.c7 = BUILD_UINT16(pData[12],pData[13]);
  m_barCalib.c8 = BUILD_UINT16(pData[14],pData[15]);
}
```

Gyroscope

Invensense IMU-3000 @ U8 3 axis data are obtained from the gyroscope

Gyroscope

Type	UUID	Read/Write	Format
<Data>	AA51 *	Read Only	XLSB XMSB YLSB YMSB ZLSB ZMSB
<Data Notification>		R/W	2 bytes
<Configuration>	AA52 *	R/W	1 byte
<Period>	AA53 *	R/W	1 byte

NB: Gyroscope is special as it has a different "enable-code" from the other sensors. Gyroscope is unique in that you can enable any combination of the the 3 axes when you write to the configuration characteristic. Write 0 to turn off gyroscope, 1 to enable X axis only, 2 to enable Y axis only, 3 = X and Y, 4 = Z only, 5 = X and Z, 6 = Y and Z, 7 = X, Y and Z. The period ranges from 100 ms to 2.55 seconds, default 1 second.

The driver for the Gyroscope sensor produces data every second. Data are obtained over the air either by direct read or by notifications. For more information please see Invensense IMU-3000 datasheet (http://invensense.com/mems/gyro/imu3000.html).

C

```
/* Converting from raw data to degrees/second.

float calcGyro(int16 rawX)
{
  float v;

  //-- calculate rotation, unit deg/s, range -250, +250
  v = (rawX * 1.0) / (65536/ 500);

  return v;
}
```

Java

```
public void onCharacteristicChanged(BluetoothGattCharacteristic c) {
    // NB: x,y,z has a weird order.
    float y = shortSignedAtOffset(c, 0) * (500f / 65536f) * -1;
    float x = shortSignedAtOffset(c, 2) * (500f / 65536f);
    float z = shortSignedAtOffset(c, 4) * (500f / 65536f);

    model.setGyroscope(x, y, z);
}

/**
 * Gyroscope, Magnetometer, Barometer, IR temperature
 * all store 16 bit two's complement values in the format
 * LSB MSB.
 *
 * This function extracts these 16 bit two's complement values.
 * */
private static Integer shortSignedAtOffset(BluetoothGattCharacteristic c, int offset) {
    Integer lowerByte = c.getIntValue(FORMAT_UINT8, offset);
    Integer upperByte = c.getIntValue(FORMAT_SINT8, offset + 1); // Note: interpret MSB as signed.

    return (upperByte << 8) + lowerByte;
}
```

Test Service

Test Service

Type	UUID	Read/Write	Format
<Data>	AA61 *	Read Only	2 bytes, power on self-test result
<Configuration>	AA62 *	R/W	1 byte

At start-up a communication test is performed on each sensor. Its purpose is to test the communication between the CC2541 and the sensor over the I2C-bus. This is not a functional test; it checks the value of selected registers against known values and also verifies that writes to registers are performed correctly. The test result is available as a 16-bit wide characteristic. One bit is set for each self-test that passes.

Test Service

Bit Number	Description
0	IR temperature sensor test
1	Humidity sensor test
2	Magnetometer sensor test
3	Accelerometer sensor test
4	Pressure sensor test
6	Gyroscope test
6-15	Reserved for future use

The test service offers two distinct functionalities.

Power on Self Test

The self-test result is available over the air in the DATA field of the Test Service. It is accessed by direct reading. Note that this is the value of the power-on self test. It should always show the value **0x3F**.

Test Mode

The Sensor Tag can also be put in a Test Mode. This functionality is used access the two LEDs on the board and also to test the three keys. The device is put in test mode by setting bit 7 of the configuration characteristic of the Test Service. In this mode the normal operation of the keys and LEDs are disabled. The LEDs are turned on by setting bit 0 and 1 respectively

Testing the keys is done by enabling notifications in the Simple Key service. The side key sets bit 2, the left key bit 1 and the right key bit 0.

Typically sequence for testing the keys and LEDs.

1. Enable test mode by writing the value 0x80 to the 0xAA62 (configuration) characteristic.
2. Enable Simple keys notification
3. Observe the notification data that results from a sequence of key presses, releases
4. Write 0x81 to 0xAA62 to blink LED D1
5. Write 0x82 to 0xAA62 to blink LED D2
6. Write 0x00 to 0xAA62 to turn of LEDs and exit test mode

Simple Key Service

Simple Key Service

Type	UUID	Notify	Format
<Data>	FFE1	Notification Only	Bit 2: side key, Bit 1- right key, Bit 0 –left key

The Simple Key service is used to enable notifications for key hits on the sensor tag. Note that the side key will only provided notifications in test mode, as it is normally used to disconnect a connected device or toggle advertising on/off.

Java

```
public void onCharacteristicChanged(BluetoothGattCharacteristic c) {
    /*
     * The key state is encoded into 1 unsigned byte.
     * bit 0: right key.
     * bit 1: left key.
     * bit 2: side key (test mode only).
     */
    Integer encodedInteger = c.getIntValue(FORMAT_UINT8, 0);

    SimpleKeysStatus newValue = SimpleKeysStatus.values()[encodedInteger % 4];
    model.setSimpleKeysStatus(newValue);
}

static enum SimpleKeysStatus {
    // Warning: The order in which these are defined matters.
    OFF_OFF, OFF_ON, ON_OFF, ON_ON;
}
```

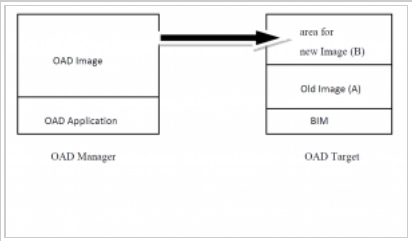
OAD (Over-the-Air Download) Service

Test Service

Type	UUID	Read/Write	Format
<OAD Image Identify>	FFC1 *	Write / Notify	1 byte write (0 or 1). 8 bytes notification.
<OAD Image Block>	FFC2 *	18 bytes write (2 bytes block number, 16 bytes OAD data)	3 bytes notification (2 bytes block number, 1 byte status)

The SensorTag features an Over-the-air firmware download (OAD). This allows a peer device or OAD manager (which could be a central BT Smart device such as a smartphone) to push a new firmware image onto the SensorTag (OAD target) and update the firmware. This feature uses the Boot Image Manger (BIM), in which case the downloaded image gets stored in the alternate flash half as the currently running image. For example, if the "A" image is the current image and is used to perform the download, then the downloaded image becomes the "B" image. Upon reset, the "B" image with the updated firmware would be loaded.

The OAD service consists of two characteristics: OAD Image Identify and OAD Image Block. To start OAD it is first off all necessary to find out what type of image is currently active in the device. This is done by writing '0' or '1' to the 'Identify' characteristic. If the image is of type 'A' it will respond with a notification, and an image of type 'B' will respond similarly if '1' is written. The central device is now able to select the alternate image for download and programming may start. Programming is done in 16 byte blocks, and these are preceded by a two 'block ID' bytes. A notification is send after each block has been received, but experience shows that OAD is faster without notifications. This approach however assumes a fixed block interval and this MUST be at least as long as the connection interval, otherwise the OAD transfer is likely to stall.



Brief Overview of OAD Systems

For more detailed information check out the Developer's Guide for Over Air Download for CC254x (http://processors.wiki.ti.com/images/8/82/OAD_for_CC254x.pdf).

OAD using the SensorTag iOs App

Open the profiles section and select TI OAD Profile, when selecting OAD all the other profiles will be turned off.


Select OAD profile


OAD profile selected

The OAD profile window will open when the OAD profile is selected. Click on Upload Image and select 'Internal Image' or 'Shared files'. Internal Image is the SensorTag images that are delivered with the app. If you want to upload your own image, from for instance an e-mail, select shared files and select the appropriate image to upload to the SensorTAg.

TI OAD profile

Select location

If 'Internal Images' is selected, an image for location A and an image for location B is provided. By default the image that is loaded to the SensorTag is stored in location A, therefore select SensorTagImgB to upload a new image to the SensorTag. If you select an image that corresponds to the same location as the existing image on the SensorTag an error message will appear and you are asked to select another image.

Select image

Wrong flash location

When a correct image is selected the SensorTag app will upload the new image to the SensorTag. Due to the limited packet size in Bluetooth low energy and the Smartphone connection interval settings the OAD process takes approximately 3 minutes. Make sure to keep the SensorTag in range of the Smartphone during the OAD process.

Uploading of a new image

When a new firmware has been successfully uploaded to the SensorTag it will need a few seconds to install the firmware. When ready the LED on the SensorTag will blink once and the SensorTag is reset and ready to use with the new firmware. If the OAD procedure, for some reason, should be aborted the SensorTag continues to use the old firmware and the OAD procedure needs to be re-started.

OAD using BLE Device Monitor

A new firmware can be uploaded using the BLE Device Monitor. Follow the connection set up in the BLE Device Monitor User Guide (http://processors.wiki.ti.com/index.php/BLE_Device_Monitor_User_Guide). When the SensorTag is connected to the BLE Device Monitor select 'File' and 'Program (OAD)'.

OAD - BLE Device Monitor

In the SensorTag packaged two different SensorTag images are available, the default location is \\Program Files\Texas Instruments\BLE Device Monitor\firmware\cc254x. Select SensorTagImg*.bin and click 'start' to start uploading the new image to the SensorTag.

Selecting image

Uploading image

Back to *Bluetooth* SensorTag main page (http://processors.wiki.ti.com/index.php/Bluetooth_SensorTag)

Connection Control Service

Test Service

Type	UUID	Read/Write	Format
<Connection Parameters>	CCC1*	Read / Notify	6 bytes: connection interval, slave latency, supervision timeout (2 bytes each)
<Request connection parameters>	CCC2*	Write	8 bytes write: max. connection interval, min. connection interval, slave latency, supervision timeout (2 bytes each)
<Request disconnect>	CCC3*	Write	1 byte write: change the value to disconnect

This service is a work-around for limitations in the BLE stacks of iOs and Android. Those do not allow the mobile application to modify the connection parameters which in turn places severe limitations on OAD. On iOs the connection interval is not optimal for speedy OAD and on Android it is in some cases too long, and others so short the OAD fails due to flash access timing limitations. The optimal connection interval for OAD is around 15-25 ms which allows enough time for flash write to complete and is also short enough to not slow down the transfer unduly.

The connection parameters are changed by writing four values (8 bytes) to characteristic 0xCCC2. The byte order is little endian. Characteristic 0xCCC1 will give a notification when the values have been updated. After this OAD can start.

Android

Android devices that support BLE and use Android 4.3 or newer can communicate with the SensorTag.

SensorTag Android Development

If you are new to programming in general then Android BLE development is not a recommended first start as it involves advanced topics like threads, callbacks, message passing, the complex Android framework etc. If you are new to Android development it is recommended that you visit the android development site (https://developer.android.com/) before continuing. If you are new to BLE Android development it is recommended that you visit the android BLE developers tutorial (https://developer.android.com/guide/topics/connectivity/bluetooth-le.html) before continuing.

Resources

This wiki should give you all the information you need about the SensorTag. It includes information about what the sensors physically can and cannot do, specifies the SensorTag-specific UUIDs for services and characteristics, and more.

The best resource for application development with Android and SensorTag is the application Java code itself:

SensorTag.java defines the UUIDs for the SensorTag's services, and characteristics. Sensor.java shows how you can get your data from the BluetoothGattCharacteristic after getting a notification. LeController.java is an implementation of a Bluetooth low energy controller. It includes code for scanning, connection management, enabling sensors and enabling notifications.

The TI Bluetooth low energy forums (http://e2e.ti.com/support/low_power_rf/f/538.aspx) is also recommended.

The SensorTag Android App source code can be found here (http://www.ti.com/tool/sensortag-sw).

Reading sensor values

With the aforementioned resources you should be able to write an app that scans, connects, and does service discovery on a SensorTag. Since the how-to on getting that far is not SensorTag-specific it will not be covered here. So, assuming you have a connection to the SensorTag and have received your onServicesDiscovered callback, the following describes how to get your precious data.

All sensors are turned off when you disconnect, and they need to be explicitly turned on again when you reconnect. Turning on a sensor is as simple as writing 0x01 to the "config" characteristic of the corresponding service. See the below code snippet for an example.

```
/**
* This is a self-contained function for turning on the magnetometer
* sensor. It must be called AFTER the onServicesDiscovered callback
* is received.
*/
private static void turnOnMagnetometer(BluetoothGatt bluetoothGatt) {
    UUID magnetServiceUuid = UUID.fromString("f000aa30-0451-4000-b000-000000000000");
    UUID magnetConfigUuid = UUID.fromString("f000aa32-0451-4000-b000-000000000000");

    BluetoothGattService magnetService = bluetoothGatt.getService(magnetServiceUuid);
    BluetoothGattCharacteristic config = magnetService.getCharacteristic(magnetConfigUuid);
    config.setValue(new byte[]{1}); //NB: the config value is different for the Gyroscope
    bluetoothGatt.writeCharacteristic(config);
}
```

See src/com/ti/sensortag/ble/LeController:changeSensorStatus() for how the SensorTag app turns on sensors.

The next step is enabling notifications. Enabling notifications is a two-part job, enabling it locally, and enabling it remotely. It can best be explained in code, see the following code snippet.

```
private static void enableMagnetometerNotifications(BluetoothGatt bluetoothGatt) {
    UUID magnetServiceUuid = UUID.fromString("f000aa30-0451-4000-b000-000000000000");
    UUID magnetDataUuid = UUID.fromString("f000aa31-0451-4000-b000-000000000000");
    UUID CCC = UUID.fromString("00002902-0000-1000-8000-00805f9b34fb");

    BluetoothGattService magnetService = bluetoothGatt.getService(magnetServiceUuid);
    BluetoothGattCharacteristic magnetDataCharacteristic = magnetService.getCharacteristic(magnetDataUuid);
    bluetoothGatt.setCharacteristicNotification(magnetDataCharacteristic, true); //Enabled locally

    BluetoothGattDescriptor config = magnetDataCharacteristic.getDescriptor(CCC);
    config.setValue(BluetoothGattDescriptor.ENABLE_NOTIFICATION_VALUE);
    bluetoothGatt.writeDescriptor(config); //Enabled remotely
}
```

See src/com/ti/sensortag/ble/LeController:changeNotificationStatus() for how the SensorTag app enables notifications.

One would hope that after executing those two functions the onCharacteristcChanged callbacks would start rolling in. But there is one major detail missing. You need to wait for your first onCharacteristicWrite callback before issuing a new write.

You can't do two writes immediately after each other. The Android BLE stack can only handle one "remote" request at a time, meaning that if you do a write immediately after another write the second request will be silently ignored. You can solve this by either creating a queue for the API calls that require remote callbacks, or sleep a short while between calls. See /src/com/ti/sensortag/ble/WriteQueue.java for how the SensorTag app handles this issue.

Now, finally you should be getting your onCharacteristicChanged callbacks. To get sensible data from the BluetoothGattCharacteristic objects, see src/com/ti/sensortag/ble/Sensor.java

Notification limit

In Google's Android BLE stack there is a hard limit of 4 unique notification subscriptions during a single connection's lifetime. This is not SensorTag specific. To get around this you must disconnect, and then reconnect with up to 4 new notifications. As an example:

```
You connect to X.
You subscribe to notifications for the humidity sensor.
You subscribe to notifications for the magnetometer.
You subscribe to notifications for the accelerometer.
You subscribe to notifications for the buttons.
// everything is good.
You subscribe to notifications for the Temperature sensor.
// temperature sensor notifications never show up.
You unsubscribe to notifications for the buttons.
You subscribe to notifications for the Temperature sensor.
// Temperature sensor notifications still don't show up
// because you have filled your quota of 4 unique notifications
// during this connections lifetime.
You disconnect from X.
You connect to X.
You subscribe to notifications for the Temperature sensor.
// Temperature sensor notifications finally show up,
// because this connection's quota is not used up.
```

Scan implementations

On most Android devices we get a callback for every advertisement the SensorTag broadcasts, every 0.1 second or so. But the Nexus 4 acts differently and will only give one callback for each BLE device. This would prevent you from implementing, say, a proximity sensor, since you want continuous feedback on the current signal strength. But you can get around this issue by restarting the scan in a loop. Note that this is not a SensorTag-specific issue, but applies to all BLE apps.

Android BLE issues

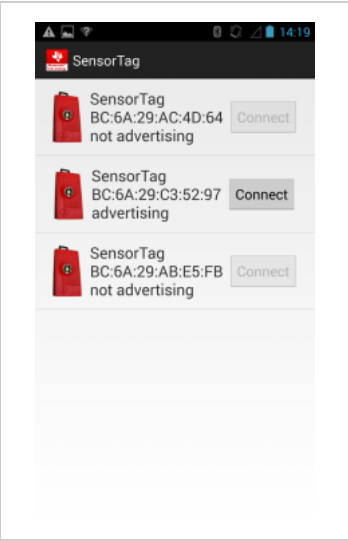
The Android BLE stack is relatively new on the market and stability issues are frequently reported on various forums. If applications that use the BLE stack stop working, there are a few ways to get started again:

- Restart the Bluetooth adapter by switching it off and on again
- Clear the applications cache and data (this can be done from Settings->Apps)

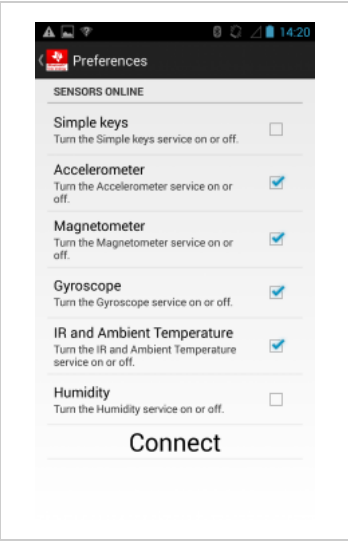
SensorTag App user guide

The SensorTag app can connect to devices and display live data of up to 4 sensors at a time. You can download it here (http://ti.com/tool/sensortag-sw), see the attached readme for help installing the app.

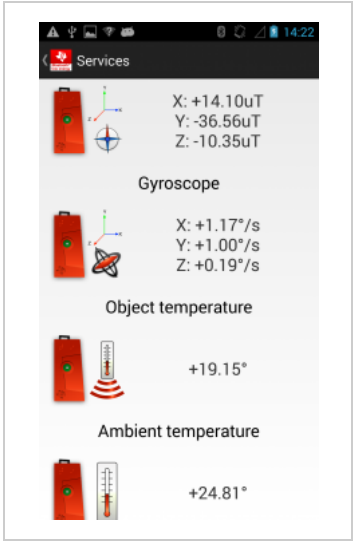
The app goes through three stages.



1. Select device



2. Select sensors



3. View data

Stage 1.

At stage 1 we check for BLE support and then start scanning. If a device hasn't sent an advertisement packet for the past second or so we consider it to be not advertising. After choosing an advertising device you move to stage 2.

Stage 2.

At this stage you are still not connected, but due to the notification limit mentioned earlier, you have to choose a subset of the sensors before connecting. Clicking connect will initiate a connection attempt, and if successful, move you to stage 3.

Stage 3.

At stage 3 we have a connection, and if you disconnect for any reason, you will be kicked back to stage 2. We immediately start service discovery, which is mandatory, even if you already know which service UUIDs you will be using. When we get the callback onServicesDiscovered we know that service discovery is complete and we start turning on sensors and subscribing to notifications.

Known issues and missing features

As of version 1.0, these features have not been implemented:

- Configurable polling period for sensors

Links

Amplifiers & Linear (http://www.ti.com/lsds/ti/analog/amplifier_and_linear.page)
Audio (http://www.ti.com/lsds/ti/analog/audio/audio_overview.page)
Broadband RF/IF & Digital Radio (http://www.ti.com/lsds/ti/analog/rfif.page)
Clocks & Timers (http://www.ti.com/lsds/ti/analog/clocksandtimers/clocks_and_timers.page)
Data Converters (http://www.ti.com/lsds/ti/analog/dataconverters/data_converter.page)
DLP & MEMS (http://www.ti.com/lsds/ti/analog/mems/mems.page)
High-Reliability (http://www.ti.com/lsds/ti/analog/high_reliability.page)
Interface (http://www.ti.com/lsds/ti/analog/interface/interface.page)
Logic (http://www.ti.com/lsds/ti/logic/home_overview.page)
Power Management (http://www.ti.com/lsds/ti/analog/powermanagement/power_portal.page)
Pro
(htt