Sensor Guide

Cross-Domain Development Kit XDK110
Platform for Application Development

Bosch Connected Devices and Solutions

BOSCH
Invented for life

XDK110: Data Sheet

Document revision	1.2
Document release date	09.06.17
Document number	BCDS-XDK110-SENSOR-GUIDE
Technical reference code(s)	
Notes	Data in this document is subject to change without notice. Product photos and pictures are for illustration purposes only and may differ from the real product's appearance. **Subject to change without notice**

XDK Sensor Guide

Platform for Application Development

The Sensors Guide provides the first steps to using the XDK sensors. This includes a general overview of the features provided by the sensors and how to get started with sensor defaults, presettings, initializations and sensor data reading algorithms. Additionally, some useful features of the virtual sensors in the sensor toolbox are described.

Table of Contents

This guide postulates a basic understanding of the XDK and according Workspace. For new users, we recommend going through the following guides at xdk.io/guides first:

- *Workbench Installation*
- *Workbench First Steps*
- *XDK Guide FreeRTOS*

1. API Overview

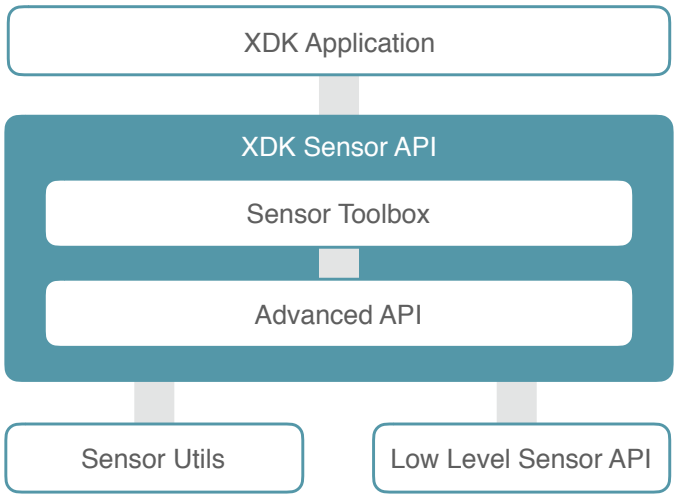
1.1 XDK Sensor API

It is generally recommended to develop an application based on the highest API level the XDK framework supports, although it is possible to access deeper API levels if this isn't enough for the distinct purpose.

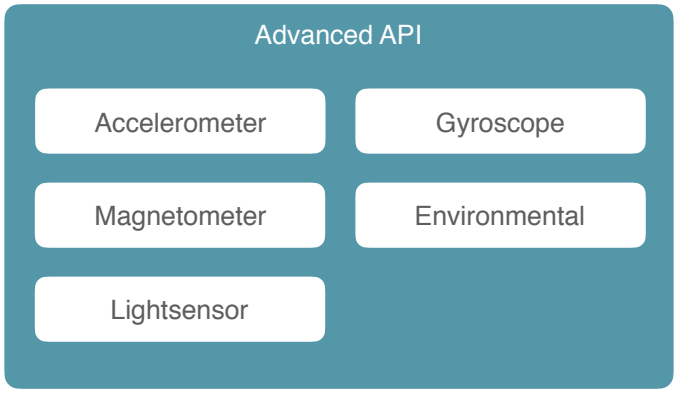
The XDK Sensor API is a major part of the XDK platform. It implements seven different physical sensors which are able to measure eight different ambient values. These sensors are accessible in detail via the Advanced API, except the acoustic sensor, which has no implemented interface yet.

Also, various advanced sensor features are available in the sensor toolbox which is on the top of the XDK Sensor API. If this API level is not sufficient for desired application use cases, then it is also possible to access the complete functionality of all XDK sensors via the Low Level Sensor API.

Picture 1: API Overview



Picture 2: Advanced API Overview



This guide provides a chapter for every sensor of the Advanced API of the XDK. Each chapter can be seen as a standalone guide for each specific sensor. They follow the same structure and show how to implement sensor specific applications with custom presettings, and explain how to read the corresponding data from the sensor. The sixth chapter contains information about the Sensor Toolbox, the highest level of the XDK Sensor API.

2. General Introduction of the XDK Sensors

This section will give an overview of each of the seven sensors included in the XDK, except the acoustic sensor AKU340. The content is created with the help of the data sheets of every sensor which are linked in this guide for further details.

2.1 Accelerometer BMA280

The BMA280 is a triaxial, low-g acceleration sensor with digital output for consumer applications. It allows measurements of acceleration in three perpendicular axes. An evaluation circuitry (ASIC) converts the output of a micro mechanical acceleration-sensing structure (MEMS) that works according to the differential capacitance principle.

The BMA280 senses tilt, motion, inactivity and shock vibration in cell phones, handhelds, computer peripherals, human-machine interfaces, virtual reality features and game controllers.

For more information please refer to the BMA280 data sheet you can find **here**.

2.2 Gyroscope BMG160

The BMG160 is a small, digital 3-axis angular rate sensor with a measurement range up to 2000°/s and a 16-bit digital resolution for consumer electronic applications.

The BMG160 allows low-noise measurement of angular rates in 3 perpendicular axes and is designed for use in mobile phones, handhelds, computer peripherals, human-machine interfaces, virtual reality features, remote and game controllers.

For more information please refer to the BMG160 data sheet you can find **here**.

2.3 Environmental Sensor BME280

The BME280 is a combined digital humidity, pressure and temperature sensor.

The BME280 is designed for high performance especially in all applications requiring humidity and pressure measurement. The emerging applications of home automation control, in-door navigation, fitness as well as GPS refinement require high accuracy and a low TCO (temperature coefficient of offset) at the same time.

The humidity sensor provides extraordinary response times for highly dynamic context awareness applications and high overall accuracy over a wide temperature range.

The pressure sensor is an absolute barometric pressure sensor with extremely high accuracy and resolution.

The integrated temperature sensor has been optimized for lowest noise and highest resolution. Its output is used for temperature compensation of the pressure and humidity sensors and can also be used to estimate the ambient temperature.

For more information please refer to the BME280 data sheet you can find **here**.

2.4 Geomagnetic Sensor BMM150

The BMM150 is a standalone geomagnetic sensor for consumer market applications. It allows measurements of the magnetic field in three perpendicular axes. Based on Bosch's proprietary FlipCore technology, performance and features of BMM150 are carefully tuned and perfectly match the demanding requirements of all 3-axis mobile applications such as electronic compass, navigation or augmented reality.

The BMM150 senses the three axis of the terrestrial field in cell phones, handhelds, computer peripherals, human-machine interfaces, virtual reality features and game controllers.

For more information please refer to the BMM150 data sheet you can find **here**.

2.5 Ambient Light Sensor MAX44009

The MAX44009 ambient light sensor features an I2C digital output that is ideal for a number of portable applications such as smartphones, notebooks, and industrial sensors. Low-light operation allows convenient application in dark-glass environments

The on-chip photodiode's spectral response is optimized to mimic the human eye's perception of ambient light and incorporates IR and UV blocking capability. The adaptive gain block automatically selects the correct lux range to optimize the lux data that is written in every byte (see table 8 in data sheet MAX44009).

For more information please refer to the MAX44009 data sheet you can find **here**.

2.6 Inertial Measurement Unit BMI160

The BMI160 is an inertial measurement unit (IMU) consisting of a state-of-the-art 3-axis, low-g accelerometer and a low power 3-axis gyroscope. It provides high precision sensor data together with the accurate timing of the corresponding data.

It has been designed for low power, high precision 6-axis and 9-axis applications in mobile phones, tablets, wearable devices, remote controls, game controllers, head-mounted devices and toys. Due to its built-in hardware synchronization of the inertial sensor data and its ability to synchronize data of external devices such as geomagnetic sensors, BMI160 is ideally suited for augmented reality, gaming and navigation applications which require highly accurate sensor data fusion.

For more information please refer to the BMI160 data sheet you can find **here**.

2.7 Limitations of the I2C Bus and sampling rate jitter

This short chapter introduces the limitations coming with the usage of multiple sensors at once via the I2C bus. This chapter also presents important information about the jitter of the sampling rate for sensor XDK applications.

Each physical sensor of the XDK is connected to a I2C bus. This bus transmits all data read from the sensors to the XDK MCU. The I2C bus is capable of transmitting data in frequencies of up to 1 kHz. This depends on the amount of sensors used. The total transmission rate is shared equally among all active sensors.

How data measurement is influenced by the configuration of a sensor in respect to the sampling rate requires a more detailed explanation. The sampling rate configuration of every sensor will be described in more detail in the following sections. Furthermore, setting the sampling rate of a sensor only influences how often data is written into the sensor's own data buffer. Afterwards the data from the buffer is requested by the MCU and sent to the MCU via the I2C bus. In practice, that means that an application which tries to read data from two sensors, which both are set to sample at a rate of 1kHz, for example, cannot be read by the MCU with 1kHz each, because the MCU can only read data with a frequency of 1kHz, and multiple sensor data sets cannot be read at the same time.

This means that a programmer has to be very mindful when setting the sample rate of every initialized sensor, because there might be a discrepancy with how often you want to read and how often you can read. These differences are also influenced by the inaccuracy resulting from the temperature of the internal crystal clock, which is responsible for the clock generation. Additionally, the sampling rate is also influenced by interrupts, which are performed by the freeRTOS operating system.

Those limitations are responsible for the jitter of the sampling rate in which sensor values are measured and further processed. Furthermore, all these explained causes lead to the fact that the jitter always varies and as such, the jitter of the overall sampling rate of the XDK sensor application is unquantifiable.

3. Accelerometer

This chapter introduces the configuration and use of the accelerometer BMA280, as well as the accelerometer component of the BMI160 on the XDK via the accelerometer interface. It shows how to initialize and read the measured sensor data, and it also explains sensor specific presettings.

3.1 Introduction of the Accelerometer Sensor Initialization Handler

This section describes the sensor init handlers for the accelerometer on the XDK. For every physical and virtual sensor, one specific sensor init handler is reserved. This allows the use of only one interface for more than one specific sensor. The XDK provides two sensors, the BMA280 and the BMI160 which share the same interface. Code 1 shows which initialization handlers are available to configure the accelerometer.

Code 1: Sensor Init Handler for the Accelerometer

```
/* Sensor Handler for the BMA280 Accelerometer */
extern Accelerometer_HandlePtr_T xdkAccelerometers_BMA280_Handle;

/* Sensor Handler for the BMI160 Accelerometer */
extern Accelerometer_HandlePtr_T xdkAccelerometers_BMI160_Handle;
```

3.2 Specific Accelerometer Parameters

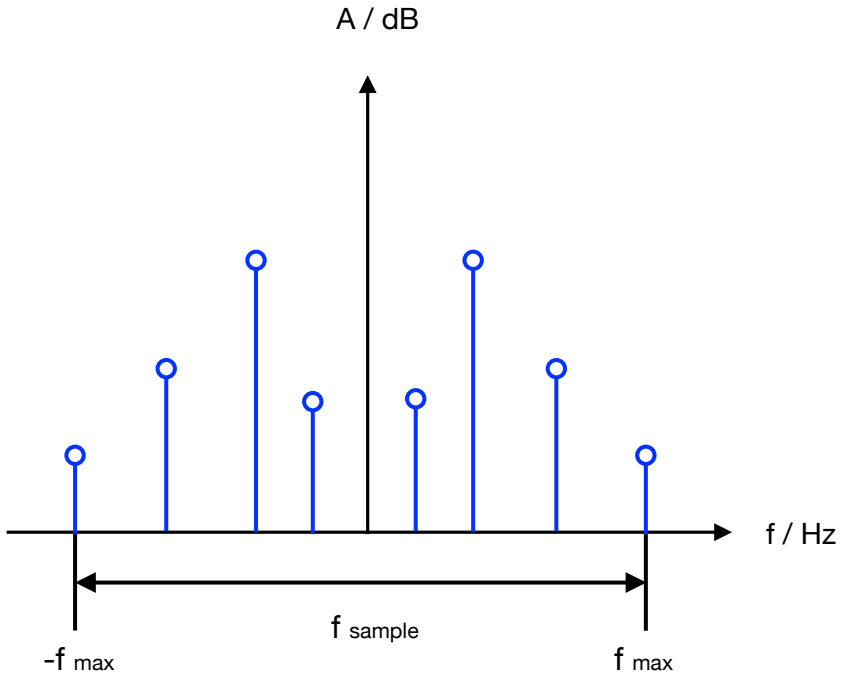
This chapter introduces accelerometer specific parameters and how they can be set after initialization of the sensor to affect the behavior of the accelerometer. This section also includes an overview and an explanation of what each parameter does.

Bandwidth

This subchapter gives a short explanation on how the bandwidth parameter works and how it affects the sampling frequency of the BMA280 and the accelerometer component of the BMI160, which are on the XDK. Please note that the sampling frequency is still limited by the I2C bus by up to 1 kHz, which was described in 2.7. As such, even if a sensor can be configured with a bandwidth of at least 1kH (according to the respective data sheet), the maximum bandwidth is still 1kHz, due to the previously mentioned limitation of the I2C bus.

The bandwidth itself is used to set a filter option, which will filter higher frequencies of the input signal than the configured bandwidth. The sensor on the other hand will always operate with the doubled configured bandwidth as sampling frequency. This is the result of the Shannon / Nyquist theorem. The theorem states that the sampling frequency always needs to be at least two times higher than the highest frequency of the input signal to be able to reproduce the input signal. The following picture shows the frequency spectrum of possible signal frequencies and how the sampling frequency needs to be configured to sample the complete spectrum of a given input signal.

Picture 3: Graphical representation of the Shannon / Nyquist theorem



The sampling frequency describes, as shown in the previous picture, the frequency range from the highest positive signal frequency to the highest negative frequency. The bandwidth parameter is set respectively to the highest frequency within the input signal and filters all higher frequencies, which are for example included in sensor noise, which affect the sensor values.

The sampling frequency for the two sensors can only be set by configuring the bandwidth parameter. It will always result in the doubled bandwidth.

Sensor Range

The sensor range configures the range in which sensor values can be measured. This means that the highest measurable value can be manually adjusted by this parameter. Appropriate values depend on the use case of the sensor application. For every sensor on the XDK, different sensor ranges are available and configurable.

Power Mode

The power mode is a feature almost every XDK sensor has. The power mode sets the power consumption of a device, or in the case of the XDK, of a peripheral like a sensor. Multiple power mode configurations are available. These can send the sensor into sleep mode right after measuring to lower overall power consumption, for example, but can also reduce accuracy and noise suppression.

Sleep Duration

The sleep duration describes how long a sensor is inactive and stays in a sleep state. This is beneficial because the sensor has a very low power consumption in the sleep state. The sleep duration defines the duration in which the sensor is in sleep state and does not measure anything.

The following table shows the function calls to set the presettings described above.

Table 1: Accelerometer Presetting Functions

Function	Description
`Accelerometer_setBandwidth()`	Sets a certain bandwidth for the accelerometer
`Accelerometer_getBandwidth()`	Returns the defined bandwidth that was set for the accelerometer
`Accelerometer_setRange()`	Sets a certain range for the accelerometer
`Accelerometer_getRange()`	Returns the defined range that was set for the accelerometer
`Accelerometer_setMode()`	Sets a certain power mode for the accelerometer
`Accelerometer_getMode()`	Returns the defined power mode that was set for the accelerometer
`Accelerometer_setSleepDuration()`	Sets a certain sleep duration for the accelerometer
`Accelerometer_getSleepDuration()`	Returns the defined sleep duration that was set for the accelerometer

The following example shows how to use the listed presetting functions listed in table 1 and the initialization handler of the accelerometer correctly. To do so, the sensor handling header file has to be included, which is `XdkSensorHandle.h`. This header file provides the interfaces for every sensor type.

Code 2: Interface of the Accelerometer

```
/* Interface for all  sensors on the XDK */
#include "XdkSensorHandle.h"
```

After including the interfaces for the sensors, the accelerometer sensor can be initialized with the specific presetting parameters as shown in Code 3.

Code 3: Initialization of the Accelerometer with Presettings

```
Retcode_T accelerometerInitReturnValue = RETCODE_FAILURE;
Retcode_T returnBandwidthValue = RETCODE_FAILURE;
Retcode_T returnRangeValue = RETCODE_FAILURE;

accelerometerInitReturnValue =
Accelerometer_init(xdkAccelerometers_BMA280_Handle);

if (RETCODE_OK != accelerometerInitReturnValue) {
    // do something
}

returnBandwidthValue =
Accelerometer_setBandwidth(xdkAccelerometers_BMA280_Handle,
ACCELEROMETER_BMA280_BANDWIDTH_125HZ);

if (RETCODE_OK != returnBandwidthValue) {
    // do something
}

returnRangeValue =
Accelerometer_setRange(xdkAccelerometers_BMA280_Handle,ACCELEROMETER_BMA280_RAN
GE_2G);

if (RETCODE_OK != returnRangeValue) {
    // do something
}
```

Note: All settings, such as the bandwidth or sensor range, have to take place after the sensor is initialized. Otherwise the sensor will be initialized with the default values and the application may not behave as intended.

The first function initializes the sensor with the correct sensor specific handler. The second and third functions set a specific bandwidth and sensor range for the accelerometer. Please refer to the corresponding data sheet for supported bandwidth and sensor range configurations.

Note: This section describes some of the basic `BCDS_Accelerometer.h` interface features, but interrupt handling is not described. If this is necessary for the implementation, please refer to `BCDS_Accelerometer.h` for further information. The interface can be found in the XDK-Workbench by navigating to the following directory:

SDK > xdk110 > Platform > Sensors > include

3.3 Reading Accelerometer Data

This section describes how to read data from the accelerometer and print the data to the console. This also includes a short initialization of the accelerometer with default values.

Code 4: Initialization and Reading of the Accelerometer Data

```
 /* initialize accelerometer */

Retcode_T returnValue = RETCODE_FAILURE;

 returnValue = Accelerometer_init(xdkAccelerometers_BMA280_Handle);

 if ( RETCODE_OK != returnValue) {
    // do something
 }

 /* read and print BMA280 accelerometer data */

Accelerometer_XyzData_T bma280 = {INT32_C(0), INT32_C(0), INT32_C(0)};
memset(&bma280, 0, sizeof(Accelerometer_XyzData_T));
returnValue =
Accelerometer_readXyzGValue(xdkAccelerometers_BMA280_Handle,&bma280);

if (RETCODE_OK == returnValue) {
    printf("BMA280 Acceleration Data \n\r: %f \n\r %f \n\r %f \n\r", (float)
bma280.xAxisData, (float) bma280.yAxisData, (float) bma280.zAxisData);
}
```

Note: This code snippet displayed in Code 4 initializes and reads data from the accelerometer on the XDK. It does not contain the complete implementation of an example application that is required to make the code work on the XDK.

The first line of code 4 initializes the accelerometer with the specific sensor initialization handler for the BMA280. The if-condition checks if the initialization was successful. If so, the storage struct for every axis' acceleration data is declared. The data is then read by the `Accelerometer_readXyzGValue()` function. A reference to the acceleration data struct is passed to the function. An additional if-condition checks if data is successfully read by the accelerometer and prints the accelerometer data onto the console output.

3.4 Accelerometer Data Reading Algorithms

Accelerometer data can be read by using two different functions. These functions provide data which differs in its display unit. The first of the two functions provides data that is measured in earth acceleration g. Additionally the physical data is stored in 32 bit integers and therefore measured in milli g. The second one provides the sensor data in bytes.

The codes snippets 5 and 6 show how to access the accelerometer to receive the measured data as physical or byte values.

Code 5: Reading Physical Accelerometer Data

```
Retcode_T returnDataValue = RETCODE_FAILURE;

Accelerometer_XyzData_T getAccelGData = { INT32_C(0), INT32_C(0), INT32_C(0) };

returnDataValue = Accelerometer_readXyzGValue(xdkAccelerometers_BMA280_Handle,
&getAccelGData);
```

Code 6: Reading Digital Accelerometer Data

```
Retcode_T returnDataValue = RETCODE_FAILURE;

Accelerometer_XyzData_T getAccelLsbData = { INT32_C(0), INT32_C(0),
INT32_C(0) };

returnDataValue =
Accelerometer_readXyzLsbValue(xdkAccelerometers_BMA280_Handle,
&getAccelLsbData);
```

Both code snippets have the same structure, only the function calls differ as described above in their function and names. The first line declares the storage struct for the measured data of every accelerometer axis and initializes it with zero values. The second line calls the function that reads the data from the accelerometer in either physical or byte values.

4. Gyroscope

This chapter introduces the configuration and use of the gyroscope BMG160 and the gyroscope component of the BMI160 on the XDK via the gyroscope interface. It shows how to initialize and read the measured sensor data and provides an explanation for some specific presettings.

4.1 Introduction of the Gyroscope Sensor Initialization Handler

This section describes the sensor init handlers for the gyroscope on the XDK. For every physical and virtual sensor, one specific sensor init handler is reserved. This allows the use of only one interface for more then one specific sensor. The XDK provides two sensors, the BMG160 and the BMI160 which share the same interface. Code 7 shows which initialization handlers are available to configure the gyroscope.

Code 7: Sensor Init Handler for the Gyroscope

```
/* Sensor Handler for the BMG160 Gyroscope */
extern Gyroscope_HandlePtr_T xdkGyroscope_BMG160_Handle;

/* Sensor Handler for the BMI160 Gyroscope */
extern Gyroscope_HandlePtr_T xdkGyroscope_BMI160_Handle;
```

4.2 Specific Gyroscope Parameters

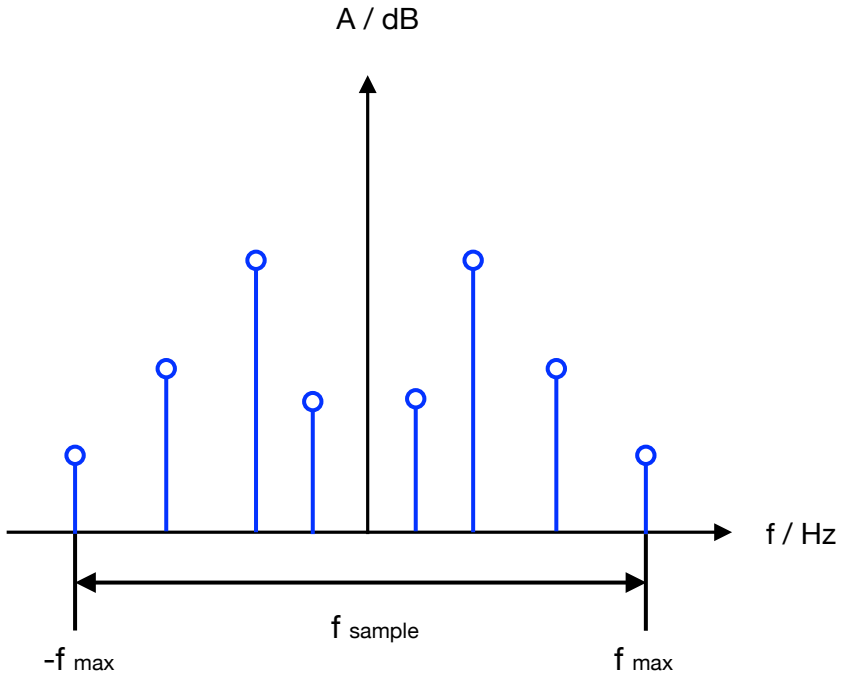
This chapter introduces gyroscope specific parameters and how they can be set after initialization of the sensor to affect the behavior of the gyroscope. This section also includes an overview and an explanation of what each parameter does.

Bandwidth

This subchapter gives a short explanation on how the bandwidth parameter works and how it affects the sampling frequency of the BMG160 and the gyroscope component of the BMI160, which are on the XDK. Please note that the sampling frequency is still limited by the I2C bus by up to 1 kHz, which was described in 2.7. As such, even if a sensor can be configured with a bandwidth of at least 1kH (according to the respective data sheet), the maximum bandwidth is still 1kHz, due to the previously mentioned limitation of the I2C bus.

The bandwidth itself is used to set a filter option, which will filter higher frequencies of the input signal than the configured bandwidth. The sensor on the other hand will always operate with the doubled configured bandwidth as sampling frequency. This is the result of the Shannon / Nyquist theorem. The theorem states that the sampling frequency always needs to be at least two times higher than the highest frequency of the input signal to be able to reproduce the input signal. The following picture shows the frequency spectrum of possible signal frequencies and how the sampling frequency needs to be configured to sample the complete spectrum of a given input signal.

Picture 4: Graphical representation of the Shannon / Nyquist theorem



The sampling frequency describes, as shown in the previous picture, the frequency range from the highest positive signal frequency to the highest negative frequency. The bandwidth parameter is set respectively to the highest frequency within the input signal and filters all higher frequencies, which are for example included in sensor noise, which affect the sensor values.

The sampling frequency for the two sensors can only be set by configuring the bandwidth parameter. It will always result in the doubled bandwidth.

Sensor range

The sensor range configures the range in which sensor values can be measured. This means that the highest measurable value can be manually adjusted. Appropriate values depend on the use case of the sensor application. For every sensor on the XDK, different sensor ranges are available and configurable.

Power Mode

The power mode is a feature almost every XDK sensor has. The power mode sets the power consumption of a device, or in the case of the XDK, of a peripheral like a sensor. Multiple power mode configurations are available. These can send the sensor into sleep mode right after measuring to lower overall power consumption, for example, but can also reduce accuracy and noise suppression.

Sleep Duration

The sleep duration describes how long a sensor is inactive and stays in a sleep state and does not measure anything. This is beneficial because the sensor has a very low power consumption in the sleep state.

The following table shows the function calls to set the presettings described above.

Table 2: Gyroscope Presetting Functions

Function	Description
`Gyroscope_setBandwidth()`	Sets a certain bandwidth for the gyroscope
`Gyroscope_getBandwidth()`	Returns the defined bandwidth that was set for the gyroscope
`Gyroscope_setRange()`	Sets a certain range for the gyroscope
`Gyroscope_getRange()`	Returns the defined range that was set for the gyroscope
`Gyroscope_setMode()`	Sets a certain power mode for the gyroscope
`Gyroscope_getMode()`	Returns the defined power that was set for the gyroscope
`Gyroscope_setSleepDuration()`	Sets a certain sleep duration for the gyroscope
`Gyroscope_setAutoSleepDuration()`	Returns the defined sleep duration that was set for the gyroscope

The following example shows how to use the listed presetting functions listed in table 1 and the initialization handler of the gyroscope correctly. To do so, the sensor handling header file has to be included, which is `XdkSensorHandle.h`. This header file provides the interfaces for every sensor type.

Code 8: Interface of the Gyroscope

```
/* Interface for all sensors on the XDK */
#include "XdkSensorHandle.h"
```

Code 9: Initialization of the Gyroscope with Presettings

```
Retcode_T gyroscopeInitReturnValue = RETCODE_FAILURE;
Retcode_T returnBandwidthValue = RETCODE_FAILURE;
Retcode_T returnRangeValue = RETCODE_FAILURE;

gyroscopeInitReturnValue = Gyroscope_init(xdkGyroscope_BMG160_Handle);

if (RETCODE_OK != gyroscopeInitReturnValue) {
    // do something
}

returnBandwidthValue = Gyroscope_setBandwidth(xdkGyroscope_BMG160_Handle,
GYROSCOPE_BMG160_BANDWIDTH_116HZ);

if (RETCODE_OK != returnBandwidthValue) {
    // do something
}

returnRangeValue =
Gyroscope_setRange(xdkGyroscope_BMG160_Handle,GYROSCOPE_BMG160_RANGE_500s);

if (RETCODE_OK != returnRangeValue) {
    // do something
}
```

Note: All settings, such as the bandwidth or sensor range, have to take place after the sensor is initialized. Otherwise the sensor will be initialized with the default values and the application may not behave as intended.

The first function initialises the sensor with the correct sensor specific handler. The second and third functions set a specific bandwidth and sensor range for the gyroscope. Please refer to the corresponding data sheet for supported bandwidth and sensor range configurations.

Note: This section describes some of the basic `BCDS_Gyroscope.h` interface features, but interrupt handling is not described. If this is necessary for the implementation, please refer to `BCDS_Gyroscope.h` for further information. The interface can be found in the XDK-Workbench by navigating to the following directory:

SDK > xdk110 > Platform > Sensors > include

4.3 Reading Gyroscope Data

This section describes how to read measured data from the gyroscope and print the data to the console. This also includes a short initialization of the gyroscope with default values.

Code 10: Initialization and Reading of the Gyroscope Data

```
/* initialize gyroscope */

Retcode_T returnValue = RETCODE_FAILURE;

returnValue = Gyroscope_init(xdkGyroscope_BMG160_Handle);

if(RETCODE_OK != returnValue) {
    // do something
}

/* read and print BMG160 gyroscope data */

Gyroscope_XyzData_T bmg160 = {INT32_C(0), INT32_C(0), INT32_C(0)};

returnValue = Gyroscope_readXyzDegreeValue(xdkGyroscope_BMI160_Handle,
&bmg160);

if (RETCODE_OK == returnValue) {
    printf("BMG160 Gyro Data :\n\rx =%ld mDeg\n\ry =%ld mDeg\n\rz =%ld"
             "mDeg\n\r", (long int) bmg160.xAxisData,
             (long int) bmg160.yAxisData, (long int) bmg160.zAxisData);
}
```

Note: Code 10 only describes how to initialize and read data from the gyroscope on the XDK. It does not contain the complete implementation of an example application that is required to make the code work on the XDK.

`Gyroscope_init()` initializes all required modules for the gyroscope with the specific sensor handle for the BMG160. An if-condition checks if the initialization succeeded. after that the data storage struct for the gyroscope data of every axis is declared. A reference to the allocated data storage is passed to the function `Gyroscope_readXyzDegreeValue()`, which will be further described in the following section 4.4. Additionally, a second if-condition checks if the measured data was read correctly from the gyroscope.

4.4 Gyroscope specific data reading algorithms

The data from the gyroscope can be received using two different functions. These function calls provide data which differs in the unit in which it is displayed. One of the two functions provides data which is calculated in its physical unit degree. Additionally the physical data is stored in 32 bit integers and therefore represented in milli degree. The function `Gyroscope_readXyzValue()` provides the sensor data in the digital representation as bytes.

The code snippets 11 and 12 show how to access the gyroscope to receive the measured data as physical or byte values.

Code 11: Reading of the physical Gyroscope Data

```
Retcode_T returnDataValue = RETCODE_FAILURE;

Gyroscope_XyzData_T getGyroDegreeData = { INT32_C(0), INT32_C(0), INT32_C(0) };

returnDataValue = Gyroscope_readXyzDegreeValue(xdkGyroscope_BMG160_Handle,
&getGyroDegreeData);
```

Code 12: Reading of the digital Gyroscope Data

```
Retcode_T returnDataValue = RETCODE_FAILURE;

Gyroscope_XyzData_T getGyroLsbData = { INT32_C(0), INT32_C(0), INT32_C(0) };

returnDataValue = Gyroscope_readXyzValue(xdkGyroscope_BMG160_Handle,
&getGyroLsbData);
```

Both code snippets have basically the same structure, only the function calls differ as described in their function and names. The first line declares the storage struct for the measured data of every gyroscope axis and initializes it with zeroes. The second line calls the function that reads the data from the gyroscope, depending on whether the data should be received as physical or byte values.

5. Magnetometer

This chapter introduces the configuration and use of the magnetometer BMM150 also known as the geomagnetic sensor BMM150 on the XDK via the magnetometer interface. It shows not only how to initialize and read the sensor data, but also provides an explanation of some presetting parameters.

5.1 Introduction of the Magnetometer Sensor Initialization handler

This section introduces the sensor init handler for the magnetometer on the XDK. For every physical and virtual sensor one specific sensor init handler is reserved. This allows the use of one interface for more than one specific sensor. Code 13 shows which initialization handler is available to configure the magnetometer.

Code: 13 Sensor Init Handler for the Magnetometer

```
/* Sensor Handler for the BMM150 Sensor */
extern Magnetometer_HandlePtr_T xdkMagnetometer_BMM150_Handle;
```

5.2 Specific Magnetometer Parameters

This chapter introduces magnetometer specific parameters and how they can be set after initialization of the sensor to affect the behavior of the magnetometer. This section also includes an overview and an explanation of what each parameter does.

Preset Mode

The preset mode is a specialty of the magnetometer BMM150. It does not differ very much from the parameter called power mode described below. In case of the magnetometer, it allows the direct adjustment of power consumption, output data rate and the accuracy of the sensor. Be aware that every configured mode with lower power consumption also affects the accuracy and the output data rate.

Data Rate

The data rate configures the sample rate for incoming data of the magnetometer BMM150, similar to the bandwidth of the accelerometer and gyroscope. To be conform with the data sheet of the BMM150, the data rate is also known as output data rate.

Power Mode

The power mode is a feature almost every XDK sensor has. The power mode sets the power consumption of a device, or in the case of the XDK, of a peripheral like a sensor. Multiple power mode configurations are available. These can send the sensor into sleep mode right after measuring to lower overall power consumption, for example, but can also reduce accuracy and noise suppression.

Now that all adjustable parameters of the magnetometer are described in their functionality, the function calls necessary to set them are described as an overview in table 3.

Table 3: Magnetometer Pre Setting Functions

Function	Description
`Magnetometer_setDataRate()`	Sets a certain data rate for the magnetometer
`Magnetometer_getDataRate()`	Returns a defined data rate that was set for the magnetometer
`Magnetometer_setPowerMode()`	Sets a certain power mode for the magnetometer
`Magnetometer_getPowerMode()`	Returns a defined power mode that was set for the magnetometer
`Magnetometer_setPresetMode()`	Sets a certain preset mode for the magneto meter
`Magnetometer_getPresetMode()`	Returns the defined preset mode that was set for the magnetometer

The following short example demostrates how to use the previously listed presetting functions and the initialization handler of the magnetometer correctly. To do so, the sensor handling header file has to be included, which is `XdkSensorHandle.h`. This header file provides the interfaces for every sensor type.

Code 14: Interface of the Magnetometer

```
/* Interface for all sensors on the XDK */
#include "XdkSensorHandle.h"
```

Code 15 shows the initialization with the pre-defined parameters of the magnetometer.

Code 15: Initialization of the Magnetometer with Presettings

```
Retcode_T magnetometerInitReturnValue = RETCODE_FAILURE;
Retcode_T returnDataRateValue = RETCODE_FAILURE;
Retcode_T returnPresetModeValue = RETCODE_FAILURE;

magnetometerInitReturnValue = Magnetometer_init(xdkMagnetometer_BMM150_Handle);

if (RETCODE_OK != magnetometerInitReturnValue) {
    // do something
}

returnDataRateValue = Magnetometer_setDataRate(xdkMagnetometer_BMM150_Handle,
MAGNETOMETER_BMM150_DATARATE_10HZ);

if (RETCODE_OK != returnDataRateValue) {
    // do something
}

returnPresetModeValue =
Magnetometer_setPresetMode(xdkMagnetometer_BMM150_Handle,MAGNETOMETER_BMM150_PR
ESETMODE_REGULAR);

if (RETCODE_OK != returnPresetModeValue) {
    // do something
}
```

Note: All settings, such as the data rate or the preset mode, have to take place after the sensor is initialized. Otherwise the sensor will be initialized with the default values and the application may not behave as intended.

The first function initialises the sensor with the correct sensor specific handler. The second and third functions set a specific data rate and preset mode for the magnetometer. Please refer to the corresponding data sheet for supported data rate and preset mode configurations.

Note: This section describes some of the basic `BCDS_Magnetometer.h` interface features, but interrupt handling is not described. If this is necessary for the implementation, please refer to `BCDS_Magnetometer.h` for further information. The interface can be found in the XDK-Workbench by navigating to the following directory:

SDK > xdk110 > Platform > Sensors > include

5.3 Reading Magnetometer Data

This section describes how to read measured data from the magnetometer and print the data to the console. This also includes a short initialization of the magnetometer with default values.

Code 16: Initialization and Reading of the Magnetometer Data

```
/* initialize magnetometer */

Retcode_T returnValue = RETCODE_FAILURE;

returnValue = Magnetometer_init(xdkMagnetometer_BMM150_Handle);

if (RETCODE_OK != returnValue){
    // do something
}

/* read and print BMM150 magnetometer data */

Magnetometer_XyzData_T bmm150 = {INT32_C(0), INT32_C(0), INT32_C(0),
INT32_C(0)};

returnValue = Magnetometer_readXyzTeslaData(xdkMagnetometer_BMM150_Handle,
&bmm150);

if (RETCODE_OK == returnValue){
    printf("BMM150 Magnetic Data :\n\rx =%ld mT\n\ry =%ld mT\n\rz =%ld mT\n\r",
    (long int) bmm150.xAxisData, (long int) bmm150.yAxisData, (long int)
       bmm150.zAxisData);
}
```

Note: This code snippet only describes how to initialize and read data from the magnetometer on the XDK. It does not contain the complete implementation of an example application that is required to make the code work on the XDK.

`Magnetometer_init()` initializes all required modules for the magnetometer with the specific sensor handle for the BMM150. An if-condition checks if the initialization succeeded. after that the data storage struct for the magnetometer data is declared. A reference to the allocated data storage is passed to the function `Magnetometer_readXyzTeslaData()`, which will be further described in the following section 5.4. Additionally, a second if-condition checks if the measured data was read correctly from the magnetometer.

5.4 Magnetometer Specific data reading algorithms

Measurement data from the magnetometer can be received using two different functions. These function calls provide data which differs in the unit in which it is displayed. One of the two functions provides data which is calculated in its physical unit tesla. Additionally the physical data is stored in 32 bit integers and therefore represented in milli tesla. The function `Magnetometer_readXyzLsbValue()` provides the digital sensor data in bytes.

The codes snippets 17 and 18 shows how to access the gyroscope to receive the measured data as physical or byte values.

Code 17: Reading of the physical Magnetometer Data

```
Retcode_T returnDataValue = RETCODE_FAILURE;

Magnetometer_XyzData_T getMagnetTeslaData = { INT32_C(0), INT32_C(0),
INT32_C(0) };

returnDataValue = Magnetometer_readXyzTeslaValue(xdkMagnetometer_BMM150_Handle,
&getMagnetTeslaData);
```

Code 18: Reading of the digital Magnetometer Data

```
Retcode_T returnDataValue = RETCODE_FAILURE;

Magnetometer_XyzData_T getMagnetLsbData = { INT32_C(0), INT32_C(0),
INT32_C(0) };

returnDataValue = Magnetometer_readXyzLsbValue(xdkMagnetometer_BMM150_Handle,
&getMagnetLsbData);
```

Both code snippets have basically the same structure, only the function calls differ as described in their function and names. The first line declares the storage struct for the measured data of every magnetometer axis and initializes it with zero values. The second line calls the function that reads the data from the magnetometer depending on if the data should be received as physical or byte values.

6. Environmental Sensor

This chapter introduces how to configure and use the environmental sensor BME280 on the XDK via the environmental interface. Therefore it includes the measurement of temperature, pressure and humidity. It provides not only how to initialize and read the sensor data, but also an explanation of some specific presettings.

6.1 Introduction of the Environmental Sensor Initialization Handler

This section introduces the sensor init handler for the environmental sensor on the XDK. For every physical and virtual sensor one specific sensor init handler is reserved. This allows the use of one interface for more than one specific sensor. Code 19 shows which initialization handler is available to use to configure the environmental sensor.

Code: 19 Sensor Init Handler for the Environmental Sensor

```
/* Sensor Handler for the BME280 Sensor */
extern Environmental_HandlePtr_T xdkEnvironmental_BME280_Handle;
```

6.2 Specific Environmental Sensor Parameters

This chapter introduces environmental specific parameters and how they can be set after initialization of the sensor to affect the behavior of the environmental sensor. This section also includes an overview and an explanation of what each parameter does.

Oversampling

Oversampling is a configurable feature of the environmental sensor BME280. it allows to set the sampling frequency much higher than necessarily needed to measure pressure, temperature and humidity. This has the advantage that noise or other disturbances, that can affect the measurements, can be easily removed.

Filtering

Filtering is a configurable feature of the environmental sensor BME280. The advantage of filtering is the enhancement in the measurement of pressure and temperature. Doing so allows the suppression of ambient disturbance. In terms of the pressure measurements, this could be the removal of short term fluctuations caused by slamming a door for example.

Standby Duration

The standby duration is a time interval for the environmental sensor BME280. It is the configurable parameter that directly affects the sampling rate of the BME280. The standby duration sets the time intervals in which BME280 does not sample anything. In addition with the measuring time, the sampling period is configured. The standby duration has basically the same effect as the sleep duration of the accelerometer.

Power Mode

The power mode is a feature almost every XDK sensor has. The power mode sets the power consumption of a device, or in the case of the XDK, of a peripheral like a sensor. Multiple power mode configurations are available. These can send the sensor into sleep mode right after measuring to lower overall power consumption, for example, but can also reduce accuracy and noise suppression.

Now that all adjustable parameters of the environmental sensor are described in their functionality. The overview, including which functions calls are necessary to set them, is represented in table 4.

Table 4: Environmental Pre Setting Functions

Function	Description
`Environmental_setPowerMode()`	Sets a certain power mode for the environmental sensor
`Environmental_getPowerMode()`	Returns a defined power mode that was set for the environmental sensor
`Environmental_setStandbyDuration()`	Sets a certain sleep duration for the environmental sensor
`Environmental_getStandbyDuration()`	Returns a defined sleep duration that was set for the environmental sensor
`Environmental_setFilterCoefficient()`	Sets the filter coefficient for the environmental sensor
`Environmental_setOverSamplingTemperature()`	Sets the oversampling temperature for the environmental sensor
`Environmental_setOverSamplingPressure()`	Sets the oversampling pressure for the environmental sensor
`Environmental_setOverSamplingHumidity()`	Sets the oversampling humidity for the environmental sensor

Additionally follows a short example on how to use the previous listed presetting functions and the initialization handler of the environmental sensor correctly. To do so, the sensor handling header file has to be included, which is `XdkSensorHandle.h`. This header file provides the interfaces for every sensor type.

Code 20: Interface of the Environmental Sensor

```
/* Interface for all sensors on the XDK */
#include "XdkSensorHandle.h"
```

Code 21: Initialization of the Environmental Sensor with Presettings

```
Retcode_T environmentalInitReturnValue = RETCODE_FAILURE;
Retcode_T returnOverSamplingValue = RETCODE_FAILURE;
Retcode_T returnFilterValue = RETCODE_FAILURE;

environmentalInitReturnValue =
Environmental_init(xdkEnvironmental_BME280_Handle);

if (RETCODE_OK != environmentalInitReturnValue) {
    // do something
}

returnOverSamplingValue =
Environmental_setOverSamplingPressure(xdkEnvironmental_BME280_Handle,
ENVIRONMENTAL_BME280_OVERSAMP_2X);

if (RETCODE_OK != returnOverSamplingValue) {
    // do something
}

returnFilterValue =
Environmental_setFilterCoefficient(xdkEnvironmental_BME280_Handle,ENVIRONMENTAL
_BME280_FILTER_COEFF_2);

if (RETCODE_OK != returnFilterValue) {
    // do something
}
```

Note: All settings, such as the bandwidth or sensor range, have to take place after the sensor is initialized. Otherwise the sensor will be initialized with the default values and the application may not behave as intended.

The first function calls the sensor initialization with the correct sensor specific handler takes place. The second and third functions set a specific oversampling rate and filter coefficient for the pressure measurement of the environmental sensor. Please refer to the corresponding data sheet for supported oversampling rate and filter coefficient configurations. The presettings for humidity and temperature can be set the same way as for pressure.

Note: This section described some of the features of the `BCDS_Environmental.h` interface, but interrupt handling was not described. If this is necessary to implement, it is recommended to refer to `BCDS_Environmental.h` for further information. The interface can be found in the XDK-Workbench by navigating to the following directory:

SDK > xdk110 > Platform > Sensors > include

6.3 Reading Environmental Sensor Data

This section describes how to read measured data from the environmental sensor and print the data to the console. This also includes a short initialization of the environmental sensor with default values.

Code 22: Initialization and Reading of the Environmental Data

```
/* initialize environmental sensor */

Retcode_T returnValue = RETCODE_FAILURE;

returnValue = Environmental_init(xdkEnvironmental_BME280_Handle);

if ( RETCODE_OK != returnValue) {
    // do something
}

/* read and print BME280 environmental sensor data */

Environmental_Data_T bme280 = { INT32_C(0), UINT32_C(0), UINT32_C(0) };
returnValue = Environmental_readData(xdkEnvironmental_BME280_Handle, &bme280);

if ( RETCODE_OK == returnValue) {
    printf("BME280 Environmental Conversion Data :\n\rp =%ld Pa\n\rt =%ld mDeg
       \n\rh =%ld %%rh\n\r", (long int) bme280.pressure, (long int)
       bme280.temperature, (long int) bme280.humidity);
}
```

Note: This code snippet only describes how to initialize and read data from the environmental sensor on the XDK. It does not contain the complete implementation of an example application that is required to make the code work on the XDK.

First an initialization of the environmental sensor has to be done. This is achieved by calling the function `Environmental_init()` with the specific initialization handler of the BME280. Furthermore a storage struct is declared and initialized to hold the measured sensor data of the BME280. In the next step the function `Evironmetal_readData()` reads the measured sensor data from the BME280. The if-condition checks if sensor data was received and, if this is the case, the data is displayed by accessing the parameters of the `Environmental_Data_T` struct.

6.4 Environmental Specific data reading algorithms

The data of the environmental sensor can be received by using multiple different functions. These function calls provide data which differs in the unit in which the data is displayed. It also differs if the data is stored in the `Environmental_Data_T` struct or stored separately in own specific declared variables.

Depending on this the measured values can be accessed by calling multiple functions of the `environmental.h` interface.

Table 5 shows which functions are available for this special purpose.

Table 5: Environmental Data Access Functions

Function	Description
`Environmental_readTemperature()`	Returns the temperature as physical value by passing in a storage variable as reference
`Environmental_readTemperatureLsb()`	Returns the temperature as digital value by passing in a storage variable as reference
`Environmental_readPressure()`	Returns the pressure as physical value by passing in a storage variable as reference
`Environmental_readPressureLsb()`	Returns the pressure as digital value by passing in a storage variable as reference
`Environmental_readHumidity()`	Returns the humidity as physical value by passing in a storage variable as reference
`Environmental_readHumidityLsb()`	Returns the humidity as digital value by passing in a storage variable as reference
`Environmental_readData()`	Returns the environmental data as physical values by passing a storage struct to hold the data as reference
`Environmental_readDataLsb()`	Returns the environmental data as digital values by passing a storage struct to hold the data as reference

In addition, the environmental API provides a convenient way to read data, depending on the way how the data should be received(physical or digital). Physical data are received by calculating the digital data which received by converting the sensor data back to its physical data. The physical data are stored in 32 bit integer and measured in corresponding milli unit (e.g. milligrams instead of grams).

The following code snippet 23, 24 show how to receive data as physical based value or as digital based value by using the storage struct `Environmental_Data_T`. The code snippets also contain how to receive the measured temperature value exclusively.

Code 23: Reading the physical Environmental Data

```
Retcode_T returnDataValue = RETCODE_FAILURE;

/* Read all physical data of the environmental sensor */

Environmental_Data_T getAllData = { INT32_C(0), INT32_C(0), INT32_C(0) };

returnDataValue = Environmental_readData(xdkEnvironmental_BME280_Handle,
&getAllData);

/* Read only the physical temperature value of the environmental sensor */

int32_t getTemperature = INT32_C(0);

returnDataValue = Environmental_readTemperature(xdkEnvironmental_BME280_Handle,
&getTemperature);
```

Code 24: Reading of the digital Environmental Data

```
Retcode_T returnDataValue = RETCODE_FAILURE;

/* Read all digital data of the environmental sensor */

Environmental_LsbData_T getAllLsbData = { INT32_C(0), INT32_C(0), INT32_C(0) };

returnDataValue = Environmental_readDataLsb(xdkEnvironmental_BME280_Handle, &
getAllLsbData);

/* Read only the digital temperature value of the environmental sensor */

int32_t getLsbTemperature = INT32_C(0);

returnDataValue =
Environmental_readTemperatureLSB(xdkEnvironmental_BME280_Handle,
&getLsbTemperature);
```

Note: Reading the pressure and humidity only differs in the function calls that provide the physical and byte value. Anything else is the same as reading the temperature value of the environmental sensor

Both code snippets have basically the same structure. In case of the environmental sensor, not only the functions calls differ as described in their functionality and names, but also the storage structs. The first line declares the storage struct for the measured data of every data set of the environmental sensor and initializes it with zero values. The second line calls the function that reads the data of the environmental sensor depending on if the data should be received as physical or byte values.

The second two code lines describe how to access only one measured sensor value of the environmental sensor. Furthermore the only difference between the two functions is the unit of the measured data values provided by the sensor. The storage variable for the temperature is declared in both cases with the same data type.

7. Ambient Light Sensor

This chapter introduces how to configure and use the ambient light sensor MAX44009 on the XDK via the light sensor interface. It provides not only an example on how to initialize and read the sensor data, but also an explanation of some specific presettings.

7.1 Introduction of the Ambient Light Sensor Initialization Handler

This section introduces the sensor init handler for the ambient light sensor on the XDK. For every physical and virtual sensor one specific sensor init handler is reserved. This allows the use of one interface for more then one specific sensor. Code 25 shows which initialization handler is available to configure the ambient light sensor.

Code: 25 Sensor Init Handler for the Ambient Light Sensor

```
/* Sensor Handler for the MAX44009 Sensor */
extern LightSensor_HandlePtr_T xdkLightSensor_MAX44009_Handle;
```

7.2 Specific Ambient Light Sensor Parameters

This chapter introduces ambient light sensor specific parameters and how they can be set after initialization of the sensor to affect the behavior of the ambient light sensor. This section also includes an overview and an explanation of what each parameter does.

Brightness

The brightness is the first configurable parameter of the ambient light sensor MAX44009. It allows a direct configuration of the light spectrum. The configuration can be done in two ways. One option is the configuration for the eye-visible spectrum. With this configuration, the measured ambient light values only contain the eye-visible spectrum. The intensity of ultra violet and infra red light is suppressed. The extended second configuration allows measuring of ultra violet and infra red light.

Integration Time

The integration time is also a configurable parameter of the ambient light sensor MAX44009. The parameters configures the maximum input voltage by setting the time intervals in which the sample and hold capacitor of the input ADC of the ambient light sensor is charged.
This parameter allows a fine adjusting of the internal sensor range.

Manual Mode / Continuous Mode

The manual and continuous mode are configurable modes of the ambient light sensor MAX44009. The integration time is set automatically if the continuous mode is used. The opposite is true for manual mode: It requires a configuration of the integration time and current division ratio. The brightness can still be set manually.

Now that all adjustable parameters of the ambient light sensor are described in their functionality you can see the following overview for the necessary functions calls in table 6.

Table 6: Ambient Light Sensor Pre Setting Functions

Function	Description
`LightSensor_setManualMode()`	Sets the manual mode for the light sensor, integration time and current devision ratio has to be set manually
`LightSensor_setContinuousMode()`	Sets the continuous mode for the ambient light sensor, integration time and current devision ratio are set automatically
`LightSensor_setIntegrationTime()`	Sets the integration time of the ambient light sensor
`LightSensor_setBrightness()`	Sets the used light spectrum via the brightness of the ambient light sensor

In addition to that, a short example on how to use the previous listed presetting functions and the initialization handler of the ambient light sensor correctly is supplied. To do so, the sensor handling header file has to be included, which is `XdkSensorHandle.h`. This header file provides the interfaces for every sensor type.

Code 26: Interface of the Ambient Light Sensor

```
/* Interface for all  sensors on the XDK */
#include "XdkSensorHandle.h"
```

Next, the initialization with the pre-defined parameters of the ambient light sensor is done in code 27.

Code 27: Initialization of the Ambient Light Sensor with Presettings

```
Retcode_T lightSensorInitReturnValue = RETCODE_FAILURE;
Retcode_T returnBrightnessValue = RETCODE_FAILURE;
Retcode_T returnIntegrationTimeValue = RETCODE_FAILURE;

lightSensorInitReturnValue = LightSensor_init(xdkLightSensor_MAX44009_Handle);

if (RETCODE_OK != lightSensorInitReturnValue) {
    // do something
}

returnBrightnessValue =
LightSensor_setBrightness(xdkLightSensor_MAX44009_Handle,
LIGHTSENSOR_NORMAL_BRIGHTNESS);

if (RETCODE_OK != returnBrightnessValue) {
    // do something
}

returnIntegrationTimeValue =
LightSensor_setIntegrationTime(xdkLightSensor_MAX44009_Handle,LIGHTSENSOR_200MS
);

if (RETCODE_OK != returnIntegrationTimeValue) {
    // do something
}
```

Note: All presettings such as the brightness or integration time have to be done after the sensor is initialized. Otherwise the sensor will be initialized with the default values and the application may show some unexpected behaviour.

The first function calls the sensor initialization with the correct sensor specific handler. The second and third functions set a specific brightness and integration time for the ambient light sensor. Please refer to the corresponding data sheet for supported brightness and integration time configurations.

7.3 Reading Ambient Light Sensor Data

This section describes how to read measured data from the ambient light sensor and print the data to the console. This also includes a short initialization of the ambient light sensor with default values.

Code 28: Initialization and Reading of the Ambient Light Sensor Data

```
/* initialize ambient light sensor */

Retcode_T returnValue = RETCODE_FAILURE;

returnValue = LightSensor_init(xdkLightSensor_MAX44009_Handle);
if ( RETCODE_OK != returnValue){
    // do something
}

uint32_t milliLuxData = UINT32_C(0);

/* read ambient light sensor */
returnValue = LightSensor_readLuxData(
 xdkLightSensor_MAX44009_Handle,&milliLuxData);

if (RETCODE_OK == returnValue){
    printf("Light sensor data obtained in milli lux :%d \n\r",
       (unsigned int) milliLuxData);
}
```

Note: This code snippet only describes how to initialize and read data from the ambient light sensor on the XDK. It does not contain the complete implementation of an example application that is required to make the code work on the XDK.

Lets take a look what happened here. The first part of the code initializes the ambient light sensor with its own sensor specific initialization handler. An if-condition checks if the initialization succeeded. Following this, reading the measured sensor data takes place. Therefore the variable `milliLuxData` is defined to store the data of the ambient light sensor. Furthermore, the data of the ambient light sensor is read by the function `LightSensor_readLuxData()`. A second if-condition checks if data has been read of the ambient light sensor correctly and if so, it prints the data to the XDK workbench console.

7.4 Ambient Light Sensor Specific data reading algorithms

The data of the ambient light sensor can be received by using two different types of function calls. These data variables differ in the unit in which they are displayed. One of the two functions provides data in its calculated physical unit lux. Additionally the physical data is stored in 32 bit integers and therefore measured in milli lux. The function `LightSensor_readRawData()` provides the data in bytes converted from the indirect raw data source. In case of the ambient light, the raw data source is an ambient light proportional current. For more information please refer to the data sheet of the ambient light sensor MAX44009.

The codes snippets 29 and 30 show how to access the data and receive the measured data as physical or byte values.

Code 29: Reading of the Ambient Light Lux Data

```
Retcode_T returnDataValue = RETCODE_FAILURE;

uint32 getLuxData = INT32_C(0);

returnDataValue = LightSensor_readLuxData(xdkLightSensor_MAX44009_Handle,
&getLuxData);
```

Code 30: Reading of the Ambient Light Raw Data

```
Retcode_T returnDataValue = RETCODE_FAILURE;

uint32 getRawData = INT32_C(0);

returnDataValue = LightSensor_readRawData(xdkLightSensor_MAX44009_Handle,
&getRawData);
```

Both code snippets have basically the same structure, only the function calls differ as described in their functionality and names. The first line declares the storage variable for the measured data of the ambient light sensor and initializes it with zero values. The second line calls the function that reads the data of the ambient light sensor depending on if the data should be received as lux or byte data.

8. Sensor Toolbox

This section introduces special features of the combined accelerometer and gyroscope BMI160, the magnetometer BMM150 and the environmental sensor BME280. These features can be implemented by using the virtual sensor initialization handler available in `XdkSensorHandle.h`.

8.1 Calibrated Sensors

The calibrated sensors include the calibrated accelerometer, gyroscope and magnetometer. These are special features of the BMI160 and the BMM150.

The calibrated sensors implement the same functionality as the uncalibrated sensors. but allow an automatic calibration of their measurements. This means, that constant offsets as the earth acceleration, for example, are omitted during the measurements.

These sensors can be initialized in the same way as the other sensors using the following initialization handler. For more information please refer to the interface `XdkSensorHandle.h`.

Code 31: Initialization Handler for the Calibrated Sensor

```
/* Sensor Handler for Calibrated accelerometer */
extern CalibratedAccel_HandlePtr_T xdkCalibratedAccelerometer_Handle;

/* Sensor Handler for Calibrated gyroscope */
extern CalibratedGyro_HandlePtr_T xdkCalibratedGyroscope_Handle;

/* Sensor Handler for Calibrated magnetometer */
extern CalibratedMag_HandlePtr_T xdkCalibratedMagnetometer_Handle;
```

To use the interfaces of the calibrated sensors, the header file `XdkSensorHandle.h` has to be included.

These interfaces provide a new function called `_getStatus()` which returns the current calibration status. Depending on which state is set, the calibration can be low or high. It is necessary for a properly working measurement to wait until the calibration state returns high.

The functions are similar for all calibrated sensors and all are listed in table 8.

Table 8: Calibration functions of the calibrated sensors

Function	Description
`CalibratedAccel_getStatus()`	Calibration function that returns the current calibration state of the accelerometer
`CalibratedGyro_getStatus()`	Calibration function that returns the current calibration state of the gyroscope
`CalibratedMag()`	Calibration function that returns the current calibration state of the magnetometer

All other things are basically the same as the initialization and reading of sensor data by using the accelerometer, gyroscope and magnetometer. Furthermore the calibrated sensors only provide an additional function to read the measured sensor data.

Table 9 describes which additional functions to read sensor data are available.

Table 9: Additional reading functions of the calibrated sensors

Function	Description
`CalibratedAccel_readXyzMps2Value()`	Returns the measured accelerometer data in meters per square second
`CalibratedGyro_readXyzRpsValue()`	Return the measured gyroscope data in radiant per second
`CalibratedMag_readXyzGaussValue`	Returns the measured magnetometer data in gauss

To see how to use the calibrated sensors a short example follows how to use the calibrated accelerometer.

Code: 32 Usage of the calibrated Accelerometer

```
/* Initialization of the calibrated accelerometer */

Retcode_T calibratedAccelInitReturnValue = RETCODE_FAILURE;

calibratedAccelInitReturnValue =
CalibratedAccel_init(xdkCalibratedAccelerometer_Handle);

if (RETCODE_OK != calibratedAccelInitReturnValue) {
    // do something
}

CalibratedAccel_Status_T calibrationAccuracy = CALIBRATED_ACCEL_UNRELIABLE;
Retcode_T calibrationStatus = RETCODE_FAILURE;

calibrationStatus = CalibratedAccel_getStatus(&calibrationAccuracy);

if (CALIBRATED_ACCEL_HIGH == calibrationAccuracy && RETCODE_OK ==
calibrationStatus) {

    /* Reading of the data of the calibrated accelerometer */
    Retcode_T returnDataValue = RETCODE_FAILURE;

    CalibratedAccel_XyzMps2Data_T getAccelMpsData = { INT32_C(0), INT32_C(0),
INT32_C(0) };

    returnDataValue = CalibratedAccel_readXyzMps2Value(&getAccelMpsData);

    if (RETCODE_OK == returnDataValue) {
       printf("Calibrated Acceleration data: \n\r %f m/s2\n\r %f m/s2\n\r %f m/
s2 \n\r", (float) getAccelMpsData.xAxisData, (float) getAccelMpsData.yAxisData,
                  (float) getAccelMpsData.zAxisData);
    }
}
```

Note: This code snippet only describes how to initialize and read data of the calibrated accelerometer on the XDK. It does not contain the complete implementation of an example application that is required to make the code work on the XDK.

Let's take a look at the implementation. The first function call `CalibratedAccel_getStatus()` initializes the calibration of the calibrated accelerometer. There are four different calibration states available: `CALIBRATED_ACCEL_UNRELIABLE`, `CALIBRATED_ACCEL_LOW`, `CALIBRATED_ACCEL_MEDIUM`, `CALIBRATED_ACCEL_HIGH`. Every state represents the accuracy level of the calibration. Although the calibration of the particular sensor is not done directly by initializing the particular sensor and must be checked repeatedly (e.g in a timer task, of while loop) until the calibration state matches the desired behavior. Everything else works like in the implementation of the normal accelerometer.

8.2 Orientation Sensor

This chapter introduces another section of the virtual sensors, namely the orientation sensor feature. This feature allows a tracking of the orientation of the XDK in the room by using the sensors of the BMI160 and the BMM150. It tracks the direct outputs of the device in two or three dimensions.

To use this feature, a virtual sensor initialization handler is required.

Code: 33 Initialization Handler for the Orientation Sensor

```
/* Sensor Handler for the Orientation Sensor */
extern Orientation_HandlePtr_T xdkOrientationSensor_Handle;
```

To use the Orientation Sensor's Interface, the header file `XdkSensorHandle.h` has to be included.

The setup to use the orientation sensor is mostly the same as for the other sensors explained in this guide. The complete interface provides three types in which data of the current orientation of the device can be received. Table 10 shows which are available.

Table 10: Reading functions of the orientation sensor

Function	Description
`Orientation_readQuaternionValue()`	Returns the orientation data as quaternion values
`Orientation_readEulerRadianVal()`	Returns the orientation data as euler radiant values
`Orientation_readEulerDegreeVal()`	Returns the orientation data as euler degree values

To see how to use the orientation sensor works a short example is provided.

Code: 34 Usage of the orientation sensor

```
/* Initialization of the orientation sensor */

Retcode_T returnValue = RETCODE_FAILURE;

returnValue = Orientation_init(xdkOrientationSensor_Handle);
if ( RETCODE_OK != returnValue) {
    // do something
}

/* Reading of the data of the orientation sensor */

Orientation_QuaternionData_T QuaternionValues = { INT32_C(0), INT32_C(0),
INT32_C(0), INT32_C(0) };

Retcode_T returnQuatValue = Orientation_readQuaternionValue(&QuaternionValues);

if (RETCODE_OK == returnQuatValue) {
    printf("Quaternions : %f %f %f %f \n\r",
       QuaternionValues.x, QuaternionValues.y, QuaternionValues.z,
       QuaternionValues.w);
}
```

Note: This code snippet only describes how to initialize and read data of the orientation sensor on the XDK. It does not contain the complete implementation of an example application that is required to make the code work on the XDK.

The first lines of code initializes the orientation sensor with the correct virtual sensor handle. Next, the if-condition checks if the initialization succeeded. Follow this, the storage struct for the measured orientation data is declared and initialized with zeroes. The last lines use the function call `Orientation_readQuaternionValue()` to read the measured orientation values and, if this is successful, print them to the console of the XDK console.

8.3 Absolute Humidity Sensor

This section describes the absolute humidity feature of the environmental sensor BME280, that is included in the Sensor Toolbox of the XDK. The absolute humidity is calculated by measuring the humidity of two points. Therefore the XDK has to be moved to receive the required measurements and then a calculation can be done.

To implement this feature, the following initialization handler is necessary.

Code: 35 Initialization Handler for absolute humidity

```
/* Sensor Handler for the humidity Sensor */
extern AbsoluteHumidity_HandlePtr_T xdkHumiditySensor_Handle;
```

To use the Orientation Sensor's Interface, the header file `XdkSensorHandle.h` has to be included.

The sensor initialization and reading the data is the same as for the other sensors. The following example gives a short overview how to implement the absolute humidity sensor in code 36.

Code: 36 Usage of the absolute humidity sensor

```
/* Initialization of the absolute humidity sensor */

Retcode_T absoluteHumidityInitReturnValue = RETCODE_FAILURE;

absoluteHumidityInitReturnValue =
AbsoluteHumidity_init(xdkHumiditySensor_Handle);

if (RETCODE_OK != absoluteHumidityInitReturnValue) {
    // do something
}

/* Reading of the data of the absolute humidity sensor */

Retcode_T returnDataValue = RETCODE_FAILURE;

float getHumidityData = INT32_C(0);

returnDataValue =
AbsoluteHumidity_readValue(xdkHumiditySensor_Handle,&getHumidityData);

if (RETCODE_OK == returnDataValue) {
    printf("Absolute Humidity data:%f g/m3 \n\r",getHumidityData);
}
```

Note: This code snippet only describes how to initialize and read data of the absolute humidity sensor on the XDK. It does not contain the complete implementation of an example application that is required to make the code work on the XDK.

Lets take a look what the code snippet does. The first part is responsible for initializing the absolute humidity sensor. Furthermore, the sensor is initialized by the function call `AbsoluteHumidity_init()`. Additionally, the if-condition checks if the initialization succeeded.

The second passage reads the data of the absolute humidity sensor. Therefore, a storage pointer is declared and initialized to hold the data of the absolute humidity sensor. Additionally, the absolute humidity data is received by using the function call `AbsoluteHumidity_readValue()`. If the if-condition is fulfilled, the data is displayed in the console of the XDK Workbench.

8.4 Additional Features

The explained virtual sensors are only a part of the sensor toolbox of the XDK. There are far more features such as a fingerprint sensor and step counter available, but explaining them on the same level of detail would exceed the scope of this guide. For more information please refer to the interfaces `BCDS_FingerPrint.h` and `BCDS_StepCounter.h`. The interface can be found in the XDK-Workbench by navigating to the following directory:

SDK > xdk110 > Platform > SensorToolbox > include