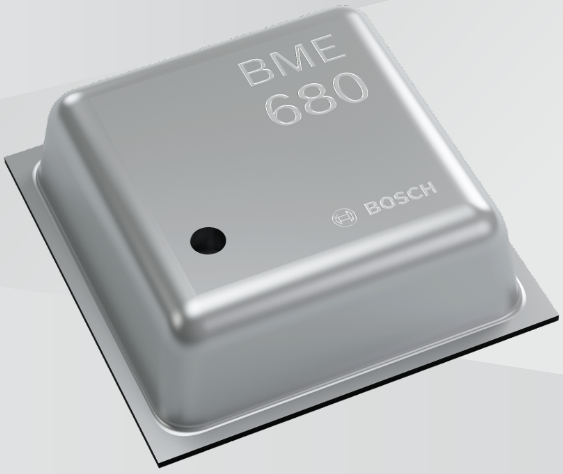
Integration Guide
Bosch Software Environmental Cluster (BSEC)



Modifications reserved | Data subject to change without notice | Document number: BST-BME680-Integration-Guide-AN008-49 | Date 2022-06-13

Contents

1 BSEC Integration Guideline

1.1 Overview of BME Family Sensors

The BME sensor family has been designed to enable pressure, temperature, humidity and gas measurements. The sensors can be operated in different modes specified in supplied header files. For example, ULP mode offers output data at slow rate thereby minimizing power consumption. In general, higher data rate corresponds to higher power consumption.

This section will provide information about the integrated sensors which are used by the BSEC library and also a brief overview of them.

Temperature Sensor

In order to guarantee fast response times, the temperature sensor within BME680 is expected to be mounted at a location in the device that enables good air and temperature exchange. The integrated temperature sensor has been optimized for very low noise and high resolution. It is primarily used for estimating ambient temperature and for temperature compensation of the other sensors present. The temperature measurement accuracy is specified in the corresponding data sheet of the used hardware.

Pressure Sensor

The pressure sensor within BME680 is an absolute barometric pressure sensor featuring exceptionally high accuracy and resolution at very low noise. The pressure measurement accuracy is specified in the corresponding data sheet of the used hardware.

Relative Humidity Sensor

The humidity sensor within BME680 measures relative humidity from 0 to 100 percent across a temperature range from -40 degrees centigrade to +85 degrees centigrade. The humidity measurement accuracy is specified in the corresponding data sheet of the used hardware.

Gas Sensor

The gas sensor within BME680 can detect a broad range of gases to measure indoor air quality for personal well being. Gases that can be detected by the BME680 include volatile organic compounds (VOC) from paints (such as formaldehyde), lacquers, paint strippers, cleaning supplies, furnishings, office equipment, glues, adhesives and alcohol. The gas measurement accuracy is specified in the corresponding data sheet of the used hardware.

1.2 The Environmental Fusion Library BSEC

General Description

BSEC fusion library has been conceptualized to provide higher-level signal processing and fusion for the BME sensor. The library receives compensated sensor values from the sensor API. It processes the BME sensor signals (in combination with the additional optional device sensors) to provide the requested sensor outputs. Inputs to BSEC signals are commonly called signals from *physical sensors*. For the outputs of BSEC, several denominations are coined for the name of the sensors providing the respective signal: composite sensors, synthetic sensors, software-based sensors and virtual sensors. For BSEC, only the denomination *virtual sensors* shall be used.

Prior to probing into BSEC Library, the entire BSEC system can be understood as a combination of the below mentioned system architecture components

- BME680 sensor (pressure, temperature, humidity and gas)
- Device with BME680 integrated
- Sensor driver API
- BSEC fusion library
- *Optional*: Additional device sensors (i.e., temperature of other heat sources in the device or position sensors)

1.2.1 BSEC Library Solutions

A BSEC solution can be chosen from a set of pre-defined and tested solutions that have a fixed set of features. Based on customer requests it is technically possible to further customize BSEC to meet specific customer demands.

Available BSEC solutions are

Solution	Included features
IAQ*	Index for air quality**, sensor heating compensated temperature/humidity, raw signals

*Lite version also available: Abbreviated version of BSEC. The code size is reduced, because it does not include the bsec_set_configuration() and the bsec_get_state() functions. As a result, it will not be possible to configure the solution based on customer specific needs or to save the state of BSEC, if the device powers down.

**The Indoor Air Quality feature of BSEC estimates the Index for Air Quality

1.2.2 BSEC Configuration Settings

BSEC offers the flexibility to configure the solution based on customer specific needs. The configuration can be loaded to BSEC via bsec_set_configuration(). The following settings can be configured

- Supply voltage of the BME680. The supply voltage influences the self-heating of the sensor.
  - 1.8V
  - 3.3V
- Different power modes for the gas sensor and corresponding data rates are supported by the software solution:

- Ultra low power (ULP) mode is designed for battery-powered and/or frequency-coupled devices over extended periods of time. This mode features an update rate of 300 seconds and an average current consumption of <0.1 mA
- Low power (LP) mode that is designed for interactive applications where the air quality is tracked and observed at a higher update rate of 3 seconds with a current consumption of <1 mA
- Continuous(CONT) mode provides an update rate if 1 Hz and shall only be used short-term for use cases that incorporate very fast events or stimulus. This mode has an average current consumption of <12 mA

- The history BSEC considers for the automatic background calibration of the IAQ is in days. That means changes in this time period will influence the IAQ value.
  - 4days, means BSEC will consider the last 4 days of operation for the automatic background calibration.
  - 28days, means BSEC will consider the last 28 days of operation for the automatic background calibration.

Configuration for IAQ solution

For the IAQ solution, the following configuration sets are available:

Configuration	Supply voltage of BME680	Maximum time between bsec_sensor_control() calls	Time considered for background calibration
generic_33v_300s_28d	3.3V	300s	28 days
generic_33v_300s_4d	3.3V	300s	4 days
generic_33v_3s_28d	3.3V	3s	28 days
generic_33v_3s_4d	3.3V	3s	4 days
generic_18v_300s_28d	1.8V	300s	28 days
generic_18v_300s_4d	1.8V	300s	4 days
generic_18v_3s_28d	1.8V	3s	28 days
generic_18v_3s_4d	1.8V	3s	4 days

The built-in configuration settings in BSEC library when no external configuration file is used can be "generic_18v_300s_4d" settings.

1.2.3 Key Features

- Precise calculation of ambient air temperature outside the device
- Precise calculation of ambient relative humidity outside the device
- Precise calculation of atmospheric pressure outside the device
- Precise calculation of Indoor Air Quality (IAQ) level outside the device

Applications

- Health monitoring/ well-being (warning regarding dehydration / heat stroke)
- Home automation control

- Control heating, venting, air conditioning (HVAC) applications
- Gaming applications like flying toys
- Internet of things
- Context awareness
- The pressure sensor provides the following features
  - Enhancement of GPS navigation (e.g., time-to-first-fix improvement, dead-reckoning, slope detection)
  - Indoor navigation (floor detection, elevator detection)
  - Outdoor navigation, leisure and sports applications
  - Weather forecast
  - Health care applications (e.g., spirometry)
  - Vertical velocity indication (e.g., rise/sink speed)

Advantages

- Hardware and software co-design for optimal performance
- Complete software fusion solution
- Eliminates need for developing fusion software in customer's side
- Robust virtual sensor outputs optimized for the application

1.2.4 Supported Virtual Sensor Output Signals

BSEC provides the output signals given in the table below. All signals from virtual sensor outputs are time-continuous signals sampled in equidistant time intervals.

Signal name	Min	Max	Unit	Acc.? [1]	Supporting Modes.? [2]
BSEC_OUTPUT_RAW_PRESSURE	300	110000	Pa	no	ULP,LP,CONT
BSEC_OUTPUT_RAW_TEMPERATURE	-40	85	deg C	no	ULP,LP,CONT
BSEC_OUTPUT_RAW_HUMIDITY	0	100	%	no	ULP,LP,CONT
BSEC_OUTPUT_RAW_GAS	170	12800000	Ohm	no	ULP,LP,CONT
BSEC_OUTPUT_IAQ	0	500		yes	ULP,LP,CONT
BSEC_OUTPUT_STATIC_IAQ	0	-		yes	ULP,LP,CONT
BSEC_OUTPUT_CO2_EQUIVALENT	400	-	ppm	yes	ULP,LP,CONT
BSEC_OUTPUT_BREATH_VOC_EQUIVALENT	0	1000	ppm	yes	ULP,LP,CONT
BSEC_OUTPUT_SENSOR_HEAT_COMPENSATED_TEMPERATURE	-45	85	deg C	no	ULP,LP,CONT
BSEC_OUTPUT_SENSOR_HEAT_COMPENSATED_HUMIDITY	0	100	%	no	ULP,LP,CONT
BSEC_OUTPUT_STABILIZATION_STATUS	False	True		no	ULP,LP,CONT
BSEC_OUTPUT_RUN_IN_STATUS	False	True		no	ULP,LP,CONT
BSEC_OUTPUT_GAS_PERCENTAGE	0	100	%	yes	ULP,LP,CONT

To achieve best gas sensor performance, the user shall not switch between different modes during the lifetime of a given sensor. Please note that the above table presents the BSEC solutions which can be made available for customer, on specific request.

[1] Accuracy status available (see bsec_output_t::accuracy).

The IAQ accuracy indicator will notify the user when she/he should initiate a calibration process. Calibration is performed in the background if the sensor is exposed to clean or polluted air for approximately 30 minutes each.

[2] The sample rate of ULP, LP and CONT mode are 1/300Hz, 1/3Hz and 1Hz respectively.

1.3 Requirements for Integration

1.3.1 Hardware

BSEC was specifically designed to work together with Bosch environmental sensor of the BMExxx family. No other sensors are supported. To ensure a consistent performance, the sensors shall be configured by BSEC itself by the use of the bsec_sensor_control() interface.

1.3.2 Software Framework

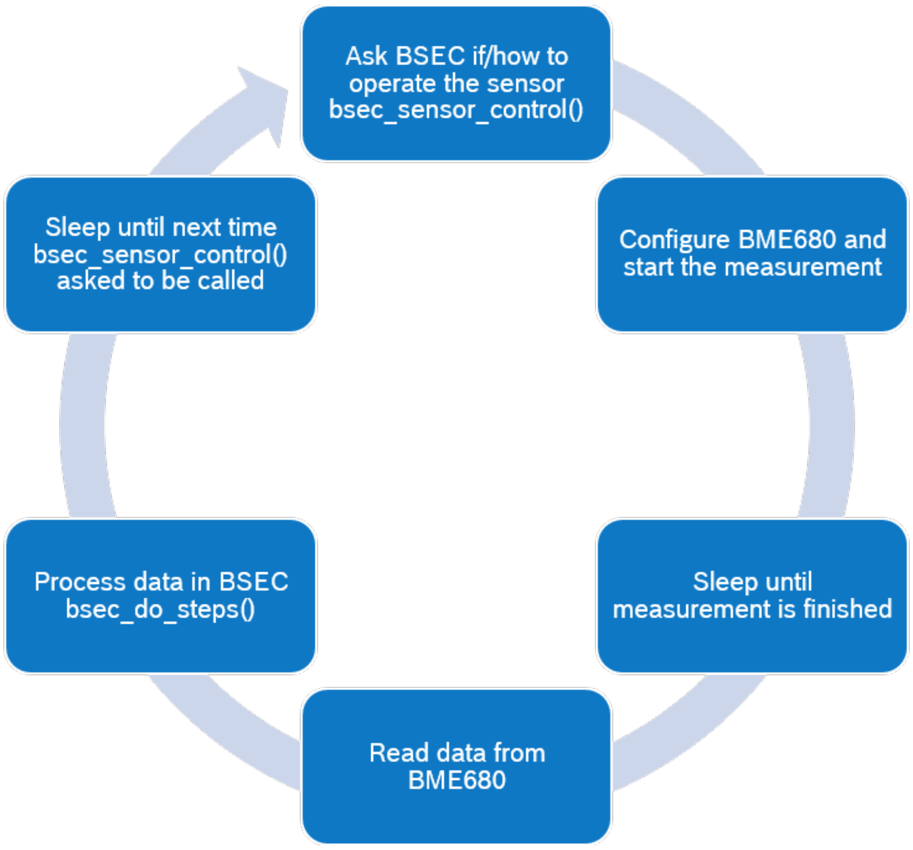

Ask BSEC if/how to operate the sensor bsec_sensor_control()
Configure BME680 and start the measurement
Sleep until measurement is finished
Read data from BME680
Process data in BSEC bsec_do_steps()
Sleep until next time bsec_sensor_control() asked to be called

Figure 1.1: BSEC Overview

The framework must provide the sample rates requested by the user for the virtual sensors to BSEC via bsec_update_subscription(), e.g., using an application on the end-user graphical interface like an Android application. BSEC internally configures itself according to the requested output sample rates. The framework must then use bsec_sensor_control() periodically to configure the BMExxx sensor. After every call to bsec_sensor_control(), the next call to bsec_sensor_control() should be scheduled by the framework as specified in the returned sensor settings structure.

Typical durations for the "Sleep until measurement is finished" are 0.190 seconds for LP mode, 2 seconds for ULP mode and 0.9 seconds for CONT mode. Typical durations for the "Sleep until next time bsec_sensor_control() asked to be called" are 2.8 seconds for LP mode, 298 seconds for ULP mode and 0.1seconds for CONT mode

For each input data, an exact time stamp shall be provided synchronized to each other when they belong to the same instant in time, i.e., they are "aligned". The processing function requires at least one input signal.

1.3.3 Physical Input Sensor Signals

BSEC is designed to be used exclusively together with sensors of the BMExxx family, such as the BME680/BME688.

Moreover, ambient temperature and humidity estimation may require additional inputs from the host system to compensate for self-heating effects caused by the operation of the host device. This may include information such as supply voltage, charging status or display status.

1.3.4 Build the Solution

BSEC is delivered as a pre-compiled static library to be linked against the host integration code. The library includes the following header files which need to be included along with BSEC library package.

Header file	Description
bsec_datatypes.h	Data types and defines used by interface functions
bsec_interface.h	Declaration of interface functions

2 BSEC Step-by-step Example

Temperature, humidity, and the presence of certain gases all influence the quality of the air we are breathing. In this walk-through, we will see how to use Bosch Sensortec BME680 sensors together with the BSEC software package to measure indoor-air-quality (IAQ).

2.1 Prerequisites

First you will need a BME680 sensor that is connected to a microcontroller (MCU). The MCU will be used to control the operation of the sensor and to process the sensor signals in order to derive index for air quality in the end. Of course, you will also need a development environment for the MCU of your choice. In this example, I will use the Arduino-based Octopus board with BME680 already included on-board. If you plan on following this tutorial with the same hardware, you can find instructions on how to setup Arduino IDE for this board here. If you have trouble downloading the board support package via the board manager, you can manually get the required files from GitHub directly.

Once we are set with our hardware and development environment, we need two pieces of software to get the most out of our BME680 sensor:

1. BME68x API (available on GitHub) deals with the low-level communication and basic compensation of sensor data. It saves us from having to fiddle with individual registers on the BME680 ourselves.
2. BSEC (available from Bosch Sensortec) will be used by us to control the sensor operation and it will provide us with an index for air quality output as well as compensated temperature and relative humidity data. For this it interfaces with the BME68x API we just downloaded. In case you are wondering, BSEC stands for "Bosch Software Environmental Cluster".

While the above might sound somewhat intimidating, we are lucky that BSEC comes with a ready made example code that only requires a small number of modifications to get it running on a new platform.

2.2 Setting Everything Up

To get started, we will first see which files, from the packages we just downloaded, need to be added to our project. From the BME68x API, we need to add all included .h and .c files. In case of BSEC, we need to add the BSEC interface headers as well the example that we want to extend:

- `inc`
  - `bsec_interface.h`
  - `bsec_datatypes.h`
- `examples`
  - `bsec_integration.h`
  - `bsec_integration.c`

- bsec_iot_example.c

As BSEC is made available as a pre-compiled binary, we should also get the correct .a file. Since the Octopus board uses an ESP8266 MCU, we need to use the library file found in `algo/bin/ESP8266/libalgobsec.a` of the BSEC release package.

To use our code in an Arduino sketch, we should copy all the above mentioned files into a folder named `bsec_iot_example`.

2.3 The Example Code

Once we are set up, let's have a look at `bsec_iot_example.c` which is the only file we will have to modify to get our project up and running. You will see that it contains a `main()` function as shown below.

```
int main()
{
    return_values_init ret;

    /* Call to the function which initializes the BSEC library
     * Switch on low-power mode and provide no temperature offset */
    ret = bsec_iot_init(BSEC_SAMPLE_RATE_LP, 0.0f, bus_write, bus_read, sleep_n,
      state_load, config_load);
    if (ret.bme68x_status)
    {
        /* Could not intialize BME68x */
        return (int)ret.bme68x_status;
    }
    else if (ret.bsec_status)
    {
        /* Could not intialize BSEC library */
        return (int)ret.bsec_status;
    }

    /* Call to endless loop function which reads and processes data based on sensor settings */
    /* State is saved every 10.000 samples, which means every 10.000 * 3 secs = 500 minutes  */
    bsec_iot_loop(sleep_n, get_timestamp_us, output_ready, state_save, 10000);

    return 0;
}
```

Here, we first initialize both the API and the BSEC library. For this purpose, we provide 3 function pointers `bus_write`, `bus_read`, and `sleep` to `bsec_iot_init()`. These pointers are used by BSEC and the BME68x API to communicate with the sensor and to put the system to sleep to control timings. Moreover, we provide the desired operation mode, in this case, we use low-power (LP) mode. The numerical argument allows us to subtract a temperature offset from the temperature reading and correct the humidity accordingly. More on this later. Next, pointer to a state_load function is provided. This is optional and can be used to load a previous BSEC state from non-volative memory to keep the internal calibration status of the library. Last, pointer to config_load function is provided. This is optional and can be used to load BSEC configuration string.

Next, `bsec_iot_loop()` is called to enter an endless loop which periodically reads out sensor data, processes the signals, and calls the provided function pointer `output_ready` whenever new data is available. Additionally, we provide a function pointer `get_timestamp_us` that is used to get the system time stamps in microseconds. The second to last argument is a function pointer `state_save` that is called periodically to allow our system to save the current BSEC state for later use with `state_load`. The desired period between the calls is passed as the last argument.

Inside `bsec_iot_example.c`, we already find empty implementations of these five functions pointers. All we have to do to get our basic example up and running is to fill in some code into these functions and add a little bit of MCU initialization to the beginning of `main()`.

2.4 Hello "Indoor-Air-Quality"

Since we want to use the example with Arduino IDE, we will have to convert the example C code into an Arduino-compatible sketch file. For this, we change the file name of `bsec_iot_example.c` to `bsec_iot_example.ino` and rename the `int main()` function into `void setup()`. At the top of the function, we add 2 lines to initialize the I2C port used to talk with our sensor as well as the serial line we want to use to report IAQ values back to our host PC. At the end of the function, we need to remove the `return 0`. We also add an empty `loop()` to create a complete Arduino sketch.

```
void setup()
{
    return_values_init ret;
    pinMode(LED_BUILTIN, OUTPUT);
    /* Init I2C and serial communication */
    Wire.begin();
    Serial.begin(115200);
    delay(1000);
    /* Call to the function which initializes the BSEC library
     * Switch on low-power mode and provide no temperature offset */
    ret = bsec_iot_init(BSEC_SAMPLE_RATE_LP, 0.0f, bus_write, bus_read, sleep_n,
      state_load, config_load);
    if (ret.bme68x_status)
    {
        /* Could not initialize BME68x */
        Serial.println("Error while initializing BME68x:"+String(ret.bme68x_status));
        return;
    }
    else if (ret.bsec_status)
    {
        /* Could not initialize BSEC library */
        Serial.println("Error while initializing BSEC library:"+String(ret.bsec_status));
        return;
    }

    String file_header = "\nTime(ms), IAQ, IAQ accuracy, Static IAQ, Raw_Temp(degC), Raw_Humidity(%rH),
       Comp_Temp(degC),  Comp_Humidity(%rH), Pressure(Pa), Gas Resistance(ohms), Gas percentage, CO2, bVOC,
       Stabilization status, Run in status, Bsec status";
    Serial.println(file_header);
    /* Call to endless loop function which reads and processes data based on sensor settings */
    /* State is saved every 10.000 samples, which means every 10.000 * 3 secs = 500 minutes  */
    bsec_iot_loop(sleep_n, get_timestamp_us, output_ready, state_save, 10000);
}

void loop()
{
    /* We do not need to put anything here as we enter our own loop function in setup() */
}
```

To be able to compile to above code, we also need to add the following include at the top of the file.

```
#include <Wire.h>
```

With the setup done, we can now move to implementing the communication with the sensor. The write function takes an array of bytes as well as a register address to start writing to.

```
int8_t bus_write(uint8_t reg_addr, const uint8_t *reg_data_ptr, uint32_t data_len, void *intf_ptr)
{
    uint8_t dev_addr = *(uint8_t*)intf_ptr;

    Wire.beginTransmission(dev_addr);
    Wire.write(reg_addr);    /* Set register address to start writing to */

    /* Write the data */
    for (int index = 0; index < data_len; index++) {
        Wire.write(reg_data_ptr[index]);
    }

    return (int8_t)Wire.endTransmission();
}
```

In case of the read function, we burst read from the BME680 register map.

```
int8_t bus_read(uint8_t reg_addr, uint8_t *reg_data_ptr, uint32_t data_len, void *intf_ptr)
{
    int8_t comResult = 0;
    uint8_t dev_addr = *(uint8_t*)intf_ptr;
    Wire.beginTransmission(dev_addr);
    Wire.write(reg_addr);                    /* Set register address to start reading from */
    comResult = Wire.endTransmission();

    delayMicroseconds(150);                 /* Precautionary response delay */
    Wire.requestFrom(dev_addr, (uint8_t)data_len);    /* Request data */

    int index = 0;
    while (Wire.available())  /* The slave device may send less than requested (burst read) */
    {
        reg_data_ptr[index] = Wire.read();
        index++;
    }

    return comResult;
}
```

To implement sleep functionality, we make use of the Arduino `delay()` function.

```
void sleep_n(uint32_t t_us, void *intf_ptr)
{
  delay(t_us / 1000);
}
```

Getting a timestamp is just as easy. Here, we use the `millis()` function to get a timestamp in milliseconds and multiply by 1000 to get the required microsecond resolution.

```
int64_t get_timestamp_us()
{
    int64_t timeMs = millis() * 1000;

    if (lastTimeMS > timeMs) /* An overflow occurred */
    {
        overflowCounter++;
    }
    lastTimeMS = timeMs;

    return timeMs + (overflowCounter * INT64_C(0xFFFFFFFF));
}
```

Finally, we need to do something with the measurement data we get. The simplest thing is to print out the something on the serial interface and look at it on your host machine. In this case, we print out the IAQ, IAQ accuracy, static IAQ, temperature, humidity, pressure, CO2 equivalent and BVOC equivalent returned by BSEC. In the future, we could even return the pressure as it is also measured.

```
void output_ready(int64_t timestamp, float iaq, uint8_t iaq_accuracy, float temperature, float humidity,
     float pressure, float raw_temperature, float raw_humidity, float gas, float gas_percentage,
     bsec_library_return_t bsec_status, float static_iaq, float stabStatus, float
      runInStatus, float co2_equivalent,
     float breath_voc_equivalent)
```

```
{
    digitalWrite(LED_BUILTIN, LOW);
    float timestamp_ms = timestamp/1e6;
    output = String(timestamp_ms) + ", ";
    output += String(iaq) + ", ";
    output += String(iaq_accuracy) + ", ";
    output += String(static_iaq) + ", ";
    output += String(raw_temperature) + ", ";
    output += String(raw_humidity) + ", ";
    output += String(temperature) + ", ";
    output += String(humidity) + ", ";
    output += String(pressure) + ", ";
    output += String(gas) + ", ";
    output += String(gas_percentage) + ", ";
    output += String(co2_equivalent) + ", ";
    output += String(breath_voc_equivalent) + ", ";
    output += String(stabStatus) + ", ";
    output += String(runInStatus) + ", ";
    output += String(bsec_status);
    Serial.println(output);
    digitalWrite(LED_BUILTIN, HIGH);
}
```

The last step we need to do is to ensure that the pre-build `libalgobsec.a` library is linked when we compile our project. Unfortunately, this process is somewhat tricky when it comes to Arduino IDE. We first need to find where the board support package for our board is installed. On Windows, this could be for example in `<USER_HOME>\AppData\Local\Arduino15\packages\esp8266\hardware` or in `<ARDUINO_ROOT>\hardware`. Once we found the location, we need to perform the following steps. Please keep in mind that the target paths might differ slightly depending on the ESP8266 package version you are using.

1. We need to copy the file `algo\normal_version\bin\esp\esp8266\libalgobsec.a` from the BSEC package into the `hardware\esp8266\3.0.2\tools\sdk\lib` folder.
2. The linker file found at `hardware\esp8266\3.0.2\tools\sdk\ld\eagle.app.v6.common.ld` needs to be modified by inserting the line `*libalgobsec.a:(.literal .text .literal.* .text.*)` after the line `*libm.a:(.literal .text .literal.* .text.*)`.
3. Modify the platform.txt file found in `hardware\esp8266\3.0.2\platform.txt`.

If you have already used the previous example code and hack guide, remove the linker flag `-libalgobsec` in the platform.txt file and reference to the `compiler.c.elf.extra_flags`.

The standard arduino-builder now passes the linker flags under `compiler.libraries.ldflags`. Most platform.txt files do not already include this new optional variable. You will hence need to declare this variable's default and add it to the end of the combine recipe. It is recommended to declare it in the following section like below,

```
# These can be overridden in platform.local.txt
compiler.c.extra_flags=
compiler.c.elf.extra_flags=
compiler.S.extra_flags=
compiler.cpp.extra_flags=
compiler.ar.extra_flags=
compiler.objcopy.eep.extra_flags=
compiler.elf2hex.extra_flags=
compiler.libraries.ldflags=-lalgobsec
```

and add it in the combine recipe like the below example

ESP8266 community forum's ESP8266 core Original line 122,

```
## Combine gc-sections, archives, and objects
recipe.c.combine.pattern="{compiler.path}{compiler.c.elf.cmd}" {build.exception_flags} -Wl,-Map
       "-Wl,{build.path}/{build.project_name}.map" {compiler.c.elf.flags} {compiler.c.elf.extra_flags} -o
       "{build.path}/{build.project_name}.elf" -Wl,--start-group {object_files} "{archive_file_path}" {compiler.c.elf.libs}
       -Wl,--end-group  "-L{build.path}"
```

should become

```
## Combine gc-sections, archives, and objects
recipe.c.combine.pattern="{compiler.path}{compiler.c.elf.cmd}" {build.exception_flags} -Wl,-Map
       "-Wl,{build.path}/{build.project_name}.map" {compiler.c.elf.flags} {compiler.c.elf.extra_flags} -o
       "{build.path}/{build.project_name}.elf" -Wl,--start-group {object_files} "{archive_file_path}" {compiler.c.elf.libs}
       {compiler.libraries.ldflags} -Wl,--end-group  "-L{build.path}"
```

If we now run our project and check with the Serial Monitor (found under Tools menu in Arduino IDE), we can see a new measurement come in every 3 seconds as shown below.

```
Time(ms), IAQ, IAQ accuracy, Static IAQ, Raw_Temp(degC), Raw_Humidity(%rH), Comp_Temp(degC),
       Comp_Humidity(%rH), Pressure(Pa), Gas Resistance(ohms), CO2, bVOC, Stabilization status, Run in status, Bsec status
1086.00,  50.00, 0, 50.00, 31.87, 62.80, 31.87, 62.80, 96037.65, 10528.67, 600.00, 0.50, 1.00, 0.00, 0
4086.00,  50.00, 0, 50.00, 31.90, 62.60, 31.83, 62.75, 96038.34, 8957.93,  600.00, 0.50, 1.00, 0.00, 0
7086.00,  50.00, 0, 50.00, 31.93, 62.36, 31.86, 62.45, 96038.34, 10993.99, 600.00, 0.50, 1.00, 0.00, 0
10087.00, 50.00, 0, 50.00, 31.94, 62.19, 31.88, 62.27, 96038.27, 13525.52, 600.00, 0.50, 1.00, 0.00, 0
13087.00, 50.00, 0, 50.00, 31.95, 62.10, 31.88, 62.19, 96038.41, 16344.65, 600.00, 0.50, 1.00, 0.00, 0
...
```

Success!

2.5 Reducing power consumption

You will notice that we now get IAQ, temperature and humidity data once every 3 seconds. This is because the example code is pre-configured to use what is called low-power (LP) mode.

For certain applications, we may want to reduce the power consumption (and data rate) and use ultra-low-power mode. Since it only takes around 0.08 mA current on average, this mode is ideal for long-term battery powered operation. But let's see if we can change the code to lower the data rate.

In order to make this change, we can simply change the first argument we pass to `bsec_iot_init()` to ULP mode:

```
/* Call to the function which initializes the BSEC library
 * Switch on ultra_low-power mode and provide no temperature offset */
ret = bsec_iot_init(BSEC_SAMPLE_RATE_ULP, 0.0f, bus_write, bus_read, sleep_n,
      state_load, config_load);
```

When we run our project now again, we can see the data coming in much slower and with greatly reduced current consumption.

```
 Time(ms), IAQ, IAQ accuracy, Static IAQ, Raw_Temp(degC), Raw_Humidity(%rH), Comp_Temp(degC),
       Comp_Humidity(%rH), Pressure(Pa), Gas Resistance(ohms), CO2, bVOC, Stabilization status, Run in status, Bsec status
1086,    50, 0, 50, 32.07, 61.45, 32.07, 61.45, 96040.05, 467083.5,  600, 0.5, 1, 0, 0
301086,  50, 0, 50, 31.8,  63.55, 31.8,  63.56, 96034.35, 411389.44, 600, 0.5, 1, 0, 0
601086,  50, 0, 50, 31.85, 63.44, 31.85, 63.44, 96032.02, 397228.84, 600, 0.5, 1, 0, 0
901087,  50, 0, 50, 31.78, 63.65, 31.78, 63.65, 96027.02, 397228.84, 600, 0.5, 1, 0, 0
1201087, 50, 1, 50, 33.73, 66.38, 33.73, 66.38, 96018.21, 27378.78,  600, 0.5, 1, 1, 0
...
```

2.6 Trigger ULP plus

Maybe the ULP sample rate is suitable for the most of your use cases, but you still want to trigger an extra measurement when it's needed, and keep the power consumption as low as possible at the same time. For this, you can use the extended feature in BSEC: ULP plus. In order to trigger it, you need a button or other interrupt source to allow users to force an extra gas measurement between two regular ULP measurements. The example function `ulp_plus_button_press()` shows how to handle such an interrupt to trigger an extra measurement.

```
void ICACHE_RAM_ATTR ulp_plus_button_press()
{
    /* We call bsec_update_subscription() in order to instruct BSEC to perform an extra measurement at the
       next
     * possible time slot
     */

    bsec_sensor_configuration_t requested_virtual_sensors[1];
    uint8_t n_requested_virtual_sensors = 1;
    bsec_sensor_configuration_t required_sensor_settings[
      BSEC_MAX_PHYSICAL_SENSOR];
    uint8_t n_required_sensor_settings = BSEC_MAX_PHYSICAL_SENSOR;
    bsec_library_return_t status = BSEC_OK;

    /* To trigger a ULP plus, we request the IAQ virtual sensor with a specific sample rate code */
    requested_virtual_sensors[0].sensor_id = BSEC_OUTPUT_IAQ;
    requested_virtual_sensors[0].sample_rate =
      BSEC_SAMPLE_RATE_ULP_MEASUREMENT_ON_DEMAND;

    /* Call bsec_update_subscription() to enable/disable the requested virtual sensors */
    status = bsec_update_subscription(requested_virtual_sensors,
      n_requested_virtual_sensors, required_sensor_settings,
        &n_required_sensor_settings);
    /* The status code would tell is if the request was accepted. It will be rejected if the sensor is not
       already in
     * ULP mode, or if the time difference between requests is too short, for example. */
}
```

Note that ULP plus is only possible in ULP mode. Also it's necessary to load a ULP plus specific config string, which guarantees a sufficient calling rate of `bsec_sensor_control()` for ULP plus. A config file with 3s maximum time between `bsec_sensor_control()` calls shall be used, as described in this chapter. There are different formats of the config files. Let's use the .c/.h files. Copy the config files to the project and include it:

```
#include "bsec_serialized_configurations_iaq.h"
```

Afterwards, we need to link it to config_load.

```
uint32_t config_load(uint8_t *config_buffer, uint32_t n_buffer)
{
    // ...
    // Load a library config from non-volatile memory, if available.
    //
    // Return zero if loading was unsuccessful or no config was available,
    // otherwise return length of loaded config string.
    // ...

    memcpy(config_buffer, bsec_config_iaq, sizeof(bsec_config_iaq));
    return sizeof(bsec_config_iaq);
}
```

So we should initialize BSEC in ULP mode and run our project like this:

```
void setup()
{
    return_values_init ret;
    pinMode(LED_BUILTIN, OUTPUT);
    /* Init I2C and serial communication */
    Wire.begin();
    Serial.begin(115200);
    delay(1000);
    /* Setup button interrupt to trigger ULP plus */
    pinMode(2, INPUT_PULLUP);
    attachInterrupt(digitalPinToInterrupt(2), ulp_plus_button_press, FALLING);

    /* Call to the function which initializes the BSEC library
     * Switch on ultra_low-power mode and provide no temperature offset */
    ret = bsec_iot_init(BSEC_SAMPLE_RATE_ULP, 0.0f, bus_write, bus_read, sleep_n,
      state_load, config_load);
    if (ret.bme68x_status)
    {
        /* Could not initialize BME68x */
        Serial.println("Error while initializing BME68x:"+String(ret.bme68x_status));
        return;
    }
    else if (ret.bsec_status)
    {
        /* Could not initialize BSEC library */
        Serial.println("Error while initializing BSEC library:"+String(ret.bsec_status));
        return;
    }

    String file_header = "\nTime(ms), IAQ, IAQ accuracy, Static IAQ, Raw_Temp(degC), Raw_Humidity(%rH),
       Comp_Temp(degC),  Comp_Humidity(%rH), Pressure(Pa), Gas Resistance(ohms), Gas percentage, CO2, bVOC,
       Stabilization status, Run in status, Bsec status";
    Serial.println(file_header);
    /* Call to endless loop function which reads and processes data based on sensor settings */
    /* State is saved every 10.000 samples, which means every 100 * 300 secs = 500 minutes  */
    bsec_iot_loop(sleep_n, get_timestamp_us, output_ready, state_save, 100);
}
```

We can see now trigger a measurement in between the regular once every 5 minutes ULP measurements.

```
Timestamp [ms], IAQ, IAQ accuracy, Static IAQ, CO2 equivalent, breath VOC equivalent, raw temp[°C],
       pressure [hPa], raw relative humidity [%], gas [Ohm], Stab Status, run in status, comp temp[°C], comp humidity [%],
       gas percentage
1142,   177.12, 0, 425.13, 4251.29, 1000.00, 32.16, 95920.84, 59.05, 417028.44, 1.00, 0.00, 32.16, 59.05,
       88.61
301142, 178.81, 0, 430.11, 4301.08, 1000.00, 32.25, 95931.66, 58.47, 374851.59, 1.00, 0.00, 32.25, 58.47,
       100.00
403142, 180.97, 0, 436.49, 4364.95, 1000.00, 32.31, 95934.16, 58.29, 361281.94, 1.00, 0.00, 32.31, 58.29,
       100.00 <--
601142, 184.63, 0, 447.30, 4472.95, 1000.00, 32.31, 95936.52, 58.07, 369800.84, 1.00, 0.00, 32.31, 58.07,
       100.00
...
```

2.7 Temperature offsets due to heat sources on the board

Let's have a look at the temperature and humidity values we are receiving from the board. A temperature of over thirty degrees and such a low relative humidity seems off. Looking at a reference thermometer, we can see that our temperature is indeed a few degrees to high. Does that mean the temperature sensor inside the BME680 is inaccurate?

Actually no, it very accurately measures the temperature exactly where it is located on the board. But there also is the issue: our board as most devices contains some heat sources (e.g., MCU, WiFi chip, display, ...). This means the temperature on our board is actually higher than the ambient temperature. Since the absolute amount of water in the air is approximately constant, this also causes the relative humidity to be lower on our board than elsewhere in the room.

As BSEC cannot know in which kind of device the sensor is integrated, we have provide some information to the algorithm to enable it to compensate this offset. In the simplest case, we have to deal with an embedded device with a constant workload and approximately constant self-heating. In such a case, we can simply supply a temperature offset to BSEC which will be subtracted from the temperature and will be used to correct the humidity reading as well.

To achieve this, we simply provide a non-zero temperature offset as the second argument to `bsec_iot_init()` as shown below. Here, we subtract a 5 degrees offset, for example.

```
/* Call to the function which initializes the BSEC library
 * Switch on ultra-low-power mode and subtract a 5 degrees temperature offset */
ret = bsec_iot_init(BSEC_SAMPLE_RATE_ULP, 5.0f, bus_write, bus_read, sleep_n,
      state_load, config_load);
```

2.8 Simulate multiple sensors using single BSEC instance

Last but not least, it is possible to simulate data collected from multiple sensors with one BSEC instance by following below integration pseudo-code:

```
retrieve_state_file() // Get default state string
call_dummy_do_step()  // do_step needs to be called for proper initialization of the library. Populate
       input struct with time stamp equal to zero, sensor id of BSEC_INPUT_TEMPERATURE and signal equal to 25
for (i_sensor = 0; i_sensor < n_sensors; i_sensor++){ // For loop for all sensors
    load_state_file(i_sensor) // Load state string for the particular sensor. In case that the sensor was
       not used before, use default state string values from the library
    call_update_subscription() // Same sampling period and outputs for all sensors
    set_input(i_sensor, input) // Populate input struct using recorded data-point
    call_do_steps(input) // Call do_steps
    retrieve_state_file(i_sensor) // Retrieve state string for the particular sensor
}
```

2.9 Quick Ultra-low Power mode (q-ULP)

Q-ULP is a variant of ULP mode, which provides additional temperature/pressure/humidity measurement w/ 1/3 Hz ODR in ULP mode (gas measurement w/ 1/300 Hz ODR). You could find the example code in the BSEC release folders,i.e. \Arduino\BSEC\examples\basic_config_state_ULP_LP.

2.10 Conclusion

As you can see, it is easy to integrate BSEC and BME68x API into an Arduino platform, as well as to measure indoor-air-quality, temperature, and humidity. We also did some modifications to the existing example code in order to change the sampling rate to ULP and to subtract a temperature offset.

3 FAQ

3.1 No Output From bsec_do_steps()

Possible reasons:

- The virtual sensor was not enabled via bsec_update_subscription()
- The input signals required for that virtual sensor were not provided to bsec_do_steps()
- The timestamps passed to bsec_do_steps() are duplicated or are not in nanosecond resolution

3.2 IAQ output does not change or accuracy remains 0

Possible reason:

- Timing of gas measurements is not within 6.25% of the target values. For example, when running the sensor in low-power mode the intended sample period is 3 s. In this case the difference between two consecutive measurements must not exceed 106.25% of 3 s which is 3.1875 s. When integrating BSEC, it is crucial to strictly follow the timing between measurements as returned by bsec_sensor_control() in the bsec_sensor_↩ settings_t::next_call field.

Correction method:

- Ensure accurate timestamps with ns resolution, especially avoid overflows of the timer or issues with 64-bit arrithmetic
- Ensure that the bsec_sensor_control() and bsec_do_steps() loop is correctly implemented and the bsec_↩ sensor_settings_t::next_call field is used to determine the frequency between measurements.

3.3 Error Codes and Corrective Measures

This chapter will give possible possible correction methods in order to fix the issues mentioned below. An overview of the error codes is given in bsec_library_return_t.

3.3.1 Errors Returned by bsec_do_steps()

3.3.1.1 BSEC_E_DOSTEPS_INVALIDINPUT

Possible reasons:

- General description: BSEC_E_DOSTEPS_INVALIDINPUT
- The sensor id of the input (physical) sensor passed to bsec_do_steps() is not in valid range or not valid for the requested output (virtual) sensor.

- The number of inputs passed to bsec_do_steps() is greater than the actual number of populated input structs.

  E.g:

```
inputs[0].sensor_id = 100;
inputs[0].signal = 25;
inputs[0].time_stamp= ts;
n_inputs = 3;
status = bsec_do_steps(inputs, n_inputs, outputs, &n_outputs);
```

Correction methods:

- The sensor_id field in the inputs structure passed to bsec_do_steps() should be one among the input (physical) sensors ids returned from bsec_update_subscription() stored in required_sensor_settings array.
- The sensor_id field in the inputs structure passed to bsec_do_steps() should be in the range of bsec_physical↩ _sensor_t enum.
- n_inputs should be equal to the number of inputs passed to bsec_do_steps(),i.e. size of inputs structure array.

3.3.1.2 BSEC_E_DOSTEPS_VALUELIMITS

Possible reasons:

- General description: BSEC_E_DOSTEPS_VALUELIMITS
- The value of the input (physical) sensor signal passed to bsec_do_steps() is not in the valid range.

  E.g:

```
inputs[0].sensor_id = BSEC_INPUT_TEMPERATURE;
inputs[0].signal = 250;
inputs[0].time_stamp= ts;
n_inputs = 1;
status = bsec_do_steps(inputs, n_inputs, outputs, &n_outputs);
```

Correction methods:

- The value of signal field in the inputs structure passed to bsec_do_steps() should be in a valid range. The allowed input value range for the sensors is listed below.
  - TEMPERATURE (-65 to 125)
  - HUMIDITY (0 to 100)
  - PRESSURE (0 to 2000000)
  - GASRESISTOR (170 to 13200000)
  - Other Sensors (-3.4028e+38 to +3.4028e+38)

3.3.1.3 BSEC_E_DOSTEPS_DUPLICATEINPUT

Possible reasons:

- General description: BSEC_E_DOSTEPS_DUPLICATEINPUT

- Duplicate input (physical) sensor ids are passed to bsec_do_steps().

  E.g:

```
inputs[0].sensor_id = BSEC_INPUT_TEMPERATURE;
inputs[0].signal = 25;
inputs[0].time_stamp= ts;
inputs[1].sensor_id = BSEC_INPUT_TEMPERATURE;
inputs[1].signal = 30;
inputs[1].time_stamp= ts;
n_inputs = 2;
status = bsec_do_steps(inputs, n_inputs, outputs, &n_outputs);
```

Correction methods:

- Each input (physical) sensor id passed to bsec_do_steps() should be unique.

3.3.1.4 BSEC_I_DOSTEPS_NOOUTPUTSRETURNABLE

Possible reasons:

- General description: BSEC_I_DOSTEPS_NOOUTPUTSRETURNABLE
- Some virtual sensors are requested, but no memory is allocated to hold the returned output values corresponding to these virtual sensors from bsec_do_steps().

  E.g:

```
n_outputs=0; /*Requested number of virtual sensor is 5*/
status = bsec_do_steps(inputs, n_inputs, outputs, &n_outputs);
```

Correction methods:

- n_outputs should be assigned the value equal to the maximum number of virtual sensors defined in bsec_↩ virtual_sensor_t enum.

3.3.1.5 BSEC_W_DOSTEPS_EXCESSOUTPUTS

Possible reasons:

- General description: BSEC_W_DOSTEPS_EXCESSOUTPUTS
- Some virtual sensors are requested, but not enough memory is allocated to hold the returned output values corresponding to these virtual sensors from bsec_do_steps().

  E.g:

```
n_outputs = 2 ; /*Requested number of virtual sensor is 5*/
status=bsec_do_steps(inputs, n_inputs, outputs, &n_outputs);
```

Correction methods:

- n_outputs should be assigned the value equal to the maximum number of virtual sensors defined in bsec_↩ virtual_sensor_t enum.

3.3.1.6 BSEC_W_DOSTEPS_TSINTRADIFFOUTOFRANGE

Possible reasons:

- General description: BSEC_W_DOSTEPS_TSINTRADIFFOUTOFRANGE

- Current timestamp of the inputs passed to bsec_do_steps() is same as the previous one stored for the same inputs.

Correction methods:

- time_stamp field of the inputs structure passed to bsec_do_steps() should be unique.

3.3.2 Errors Returned by bsec_update_subscription()

3.3.2.1 BSEC_E_SU_WRONGDATARATE

Possible reasons:

- General description: BSEC_E_SU_WRONGDATARATE
- Virtual sensors are requested with a sampling rate which is not allowed, e.g. 100 Hz.

  E.g:

```
requested_virtual_sensors[virtual_sensor_count].sample_rate = 100;
requested_virtual_sensors[virtual_sensor_count].sensor_id = BSEC_OUTPUT_RAW_GAS;
status = bsec_update_subscription(requested_virtual_sensors,
      n_requested_virtual_sensors, required_sensor_settings, &n_required_sensor_settings);
```

Correction methods:

- The sample_rate field in the requested_virtual_sensors structure passed to bsec_update_subscription() should match with the sampling rate defined for that sensor. The allowed sampling rate(s) in hertz for each sensor is listed in this table.

3.3.2.2 BSEC_E_SU_SAMPLERATELIMITS

Possible reasons:

- General description: BSEC_E_SU_SAMPLERATELIMITS
- Virtual sensors are requested with an incorrect sampling rate.

  E.g:

```
requested_virtual_sensors[virtual_sensor_count].sample_rate = 5;
requested_virtual_sensors[virtual_sensor_count].sensor_id = BSEC_OUTPUT_RAW_GAS;
 status=bsec_update_subscription(requested_virtual_sensors,
      n_requested_virtual_sensors, required_sensor_settings, &n_required_sensor_settings);
```

Correction methods:

- The sample_rate field in the requested_virtual_sensors structure passed to bsec_update_subscription() should match with the sampling rate defined for that sensor. The allowed sampling rate(s) in hertz for each sensor is listed in this table.

3.3.2.3 BSEC_E_SU_DUPLICATEGATE

Possible reasons:

- General description: BSEC_E_SU_DUPLICATEGATE
- Duplicate virtual sensors are requested through bsec_update_subscription() function.

  E.g:

```
requested_virtual_sensors[virtual_sensor_count].sample_rate = 1;
requested_virtual_sensors[virtual_sensor_count].sensor_id = BSEC_OUTPUT_RAW_GAS;
virtual_sensor_count++;
requested_virtual_sensors[virtual_sensor_count].sample_rate = 1;
requested_virtual_sensors[virtual_sensor_count].sensor_id = BSEC_OUTPUT_RAW_GAS;
status = bsec_update_subscription(requested_virtual_sensors,
      n_requested_virtual_sensors, required_sensor_settings ,&n_required_sensor_settings);
```

Correction methods:

- The sensor_id field in the requested_virtual_sensors structure passed to bsec_update_subscription() should be unique.

3.3.2.4 BSEC_E_SU_INVALIDSAMPLERATE

Possible reasons:

- General description: BSEC_E_SU_INVALIDSAMPLERATE
- The sampling rate of the requested virtual sensors in not within the valid limit.

  E.g:

```
requested_virtual_sensors[virtual_sensor_count].sample_rate = 100;
requested_virtual_sensors[virtual_sensor_count].sensor_id = BSEC_OUTPUT_RAW_GAS;
status = bsec_update_subscription(requested_virtual_sensors,
      n_requested_virtual_sensors, required_sensor_settings, &n_required_sensor_settings);
```

Correction methods:

- The sample_rate field in the requested_virtual_sensors structure passed to bsec_update_subscription() should match with the sampling rate defined for that sensor. The allowed sampling rate(s) in hertz for each sensor is listed in this table.

3.3.2.5 BSEC_E_SU_GATECOUNTEXCEEDSARRAY

Possible reasons:

- General description: BSEC_E_SU_GATECOUNTEXCEEDSARRAY
- Enough memory is not allocated to hold the returned physical sensor data from bsec_update_subscription().

  E.g:

```
n_required_sensor_settings = 0;
status = bsec_update_subscription(requested_virtual_sensors,
      n_requested_virtual_sensors, required_sensor_settings, &n_required_sensor_settings);
```

Correction methods:

- n_required_sensor_settings passed to bsec_update_subscription() should be equal to BSEC_MAX_PHYSIC↩ AL_SENSOR.

3.3.2.6 BSEC_E_SU_SAMPLINTVLINTEGERMULT

Possible reasons:

- General description: BSEC_E_SU_SAMPLINTVLINTEGERMULT
- The sampling rate of an output requested via bsec_update_subscription() is not an integer multiple of the other active sampling rates.

Correction methods:

- Use one of the sampling rates listed in this table.

3.3.2.7 BSEC_E_SU_MULTGASSAMPLINTVL

Possible reasons:

- General description: BSEC_E_SU_MULTGASSAMPLINTVL
- The sampling rate of the requested output requires the gas sensor, which is currently running at a different sampling rate.

Correction methods:

- The outputs that require the gas sensor must have equal sampling rates.

3.3.2.8 BSEC_E_SU_HIGHHEATERONDURATION

Possible reasons:

- General description: BSEC_E_SU_HIGHHEATERONDURATION
- The sampling period of the requested output is lower than the duration of a complete measurement.

Correction methods:

- Use a slower sampling rate.

3.3.2.9 BSEC_W_SU_UNKNOWNOUTPUTGATE

Possible reasons:

- General description: BSEC_W_SU_UNKNOWNOUTPUTGATE
- Requested virtual sensor id is not valid.
- Number of virtual sensors passed to bsec_update_subscription() is greater than the actual number of output(virtual) sensors requested.

  E.g:

```
requested_virtual_sensors[virtual_sensor_count].sample_rate =  1;
requested_virtual_sensors[virtual_sensor_count].sensor_id = 100;

n_requested_virtual_sensors = 3;
status = bsec_update_subscription(requested_virtual_sensors,
      n_requested_virtual_sensors, required_sensor_settings, &n_required_sensor_settings);
```

Correction methods:

- The sensor_id field in the requested_virtual_sensors structure passed to bsec_update_subscription() should be in the range of bsec_virtual_sensor_t enum.
- n_requested_virtual_sensors should be equal to the number of output (virtual) sensors requested through bsec_update_subscription() i.e. size of requested_virtual_sensors structure array.

3.3.2.10 BSEC_I_SU_SUBSCRIBEDOUTPUTGATES

Possible reasons:

- General description: BSEC_I_SU_SUBSCRIBEDOUTPUTGATES
- No output (virtual) sensor requested through bsec_update_subscription()
- Number of requested output (virtual) sensors passed to bsec_update_subscription() is zero even when there are some output (virtual) sensors requested.

  E.g:

```
requested_virtual_sensors[virtual_sensor_count].sample_rate =  1/300;
requested_virtual_sensors[virtual_sensor_count].sensor_id = BSEC_OUTPUT_RAW_GAS;
n_requested_virtual_sensors = 0;
status = bsec_update_subscription(requested_virtual_sensors,
      n_requested_virtual_sensors, required_sensor_settings, &n_required_sensor_settings);
```

Correction methods:

- requested_virtual_sensors structure to bsec_update_subscription() should be populated with the data of the required output (virtual) sensors. It should not be empty.
- n_requested_virtual_sensors should be equal to the number of output (virtual) sensors requested via bsec_update_subscription(), i.e., size of requested_virtual_sensors structure array. It should not be zero.

3.3.2.11 BSEC_W_SU_MODINNOULP

Possible reasons:

- General description: BSEC_W_SU_MODINNOULP
- Triggering measurement on demand (MOD) in non-ULP mode

Correction methods:

- Ensure that sensors are running in ULP mode or enable ULP mode using bsec_update_subscription() before triggering MOD

3.3.3 Errors Returned by bsec_set_configuration() / bsec_set_state()

3.3.3.1 BSEC_E_CONFIG_VERSIONMISMATCH

Possible reasons:

- General description: BSEC_E_CONFIG_VERSIONMISMATCH
- Version mentioned in the configuration string or state string passed to bsec_set_configuration() or bsec_set_state(), respectively, does not match with the current version.

Correction methods:

- Obtain a compatible string.
- A call to bsec_get_version() would give the current version information.

3.3.3.2 BSEC_E_CONFIG_FEATUREMISMATCH

Possible reasons:

- General description: BSEC_E_CONFIG_FEATUREMISMATCH
- Enabled features encoded in configuration/state strings passed to bsec_set_configuration() or bsec_set_state() does not match with current library implementation.

Correction methods:

- Ensure the configuration/state strings generated for current library implementation is passed to bsec_set_↩ configuration() or bsec_set_state().

3.3.3.3 BSEC_E_CONFIG_CRCMISMATCH

Possible reasons:

- General description: BSEC_E_CONFIG_CRCMISMATCH
- Difference in configuration/state strings passed to bsec_set_configuration() or bsec_set_state() from what is generated for current library implementation.
- String was corrupted during storage or loading.
- String was truncated.
- String is shorter than the size argument provided to the setter function.

Correction methods:

- Ensure the configuration/state strings generated for current library implementation is passed to bsec_set_↩ configuration() or bsec_set_state().

3.3.3.4 BSEC_E_CONFIG_EMPTY

Possible reasons:

- General description: BSEC_E_CONFIG_EMPTY
- String passed to the setter function is too short to be able to be a valid string.

Correction methods:

- Obtain a valid config string.

3.3.3.5 BSEC_E_CONFIG_INSUFFICIENTWORKBUFFER

Possible reasons:

- General description: BSEC_E_CONFIG_INSUFFICIENTWORKBUFFER
- Length of work buffer passed to bsec_set_configuration() or bsec_set_state() is less than required value.
- Length of work buffer passed to bsec_get_configuration() or bsec_get_state() is less than required value.

Correction methods:

- Value of n_work_buffer_size passed to bsec_set_configuration() and bsec_set_state() should be assigned the maximum value BSEC_MAX_PROPERTY_BLOB_SIZE.

- Value of n_work_buffer passed to bsec_get_configuration() and bsec_get_state() should be assigned the maximum value BSEC_MAX_PROPERTY_BLOB_SIZE.

3.3.3.6 BSEC_E_CONFIG_INVALIDSTRINGSIZE

Possible reasons:

- General description: BSEC_E_CONFIG_INVALIDSTRINGSIZE
- String size encoded in configuration/state strings passed to bsec_set_configuration() or bsec_set_state() does not match with the actual string size n_serialized_settings/n_serialized_state passed to these functions.

Correction methods:

- Ensure the configuration/state strings generated for current library implementation is passed to bsec_set_configuration() or bsec_set_state().

3.3.3.7 BSEC_E_CONFIG_INSUFFICIENTBUFFER

Possible reasons:

- General description: BSEC_E_CONFIG_INSUFFICIENTBUFFER
- Value of n_serialized_settings_max/n_serialized_state_max passed to bsec_get_configuration() or bsec_get_state() is insufficient to hold serialized data from BSEC library.

Correction methods:

- Value of n_serialized_settings_max/n_serialized_state_max passed to bsec_get_configuration() or bsec_get_state() should be equal to BSEC_MAX_PROPERTY_BLOB_SIZE.

3.3.4 Errors Returned by bsec_sensor_control()

3.3.4.1 BSEC_W_SC_CALL_TIMING_VIOLATION

Possible reasons:

- General description: BSEC_W_SC_CALL_TIMING_VIOLATION
- The timestamp at which bsec_sensor_control(timestamp) is called differs from the target timestamp which was returned during the previous call in the .next_call struct member by more than 6.25%.

Correction methods:

- Ensure that your system calls bsec_sensor_control() at the time instructed in the previous call.
- To ensure proper operation, a maximum jitter of 6.25% is allowed.

3.3.4.2 BSEC_W_SC_MODINSUFFICIENTWAITTIME

Possible reasons:

- General description: BSEC_W_SC_MODINSUFFICIENTWAITTIME
- Insufficient wait time between two MODs, at least 60s is needed.

Correction methods:

- Make sure that there is sufficient wait time between two MODs

3.3.4.3 BSEC_W_SC_MODEXCEEDULPTIMELIMIT

Possible reasons:

- General description: BSEC_W_SC_MODEXCEEDULPTIMELIMIT
- Insufficient time between MOD and the last regular ULP measurement.
- Insufficient time between MOD and the next regular ULP measurement, at least 60s is needed.

Correction methods:

- Make sure to trigger MOD within the defined time range between two regular ULP measurements.

4 Module Documentation

4.1 BSEC C Interface

4.1.1 Interface Usage

Interfaces of BSEC signal processing library.

Interface usage The following provides a short overview on the typical operation sequence for BSEC.

- Initialization of the library

Steps	Function
Initialization of library	bsec_init()
Update configuration settings (optional)	bsec_set_configuration()
Restore the state of the library (optional)	bsec_set_state()

- The following function is called to enable output signals and define their sampling rate / operation mode.

Steps	Function
Enable library outputs with specified mode	bsec_update_subscription()

- This table describes the main processing loop.

Steps	Function
Retrieve sensor settings to be used	bsec_sensor_control()
Configure sensor and trigger measurement	See BME680 API and example codes
Read results from sensor	See BME680 API and example codes
Perform signal processing	bsec_do_steps()

- Before shutting down the system, the current state of BSEC can be retrieved and can then be used during re-initialization to continue processing.

Steps	Function
Retrieve the current library state	bsec_get_state()
Retrieve the current library configuration	bsec_get_configuration()

Configuration and state Values of variables belonging to a BSEC instance are divided into two groups:

- Values **not updated by processing** of signals belong to the **configuration group**. If available, BSEC can be configured before use with a customer specific configuration string.

- Values **updated during processing** are member of the **state group**. Saving and restoring of the state of BSEC is necessary to maintain previously estimated sensor models and baseline information which is important for best performance of the gas sensor outputs.

Note

> BSEC library consists of adaptive algorithms which models the gas sensor which improves its performance over the time. These will be lost if library is initialized due to system reset. In order to avoid this situation library state shall be stored in non volatile memory so that it can be loaded after system reset.

4.1.2 Interface Functions

4.1.2.1 bsec_do_steps()

```
bsec_library_return_t bsec_do_steps (
             const bsec_input_t *const inputs,
             const uint8_t n_inputs,
             bsec_output_t * outputs,
             uint8_t * n_outputs )
```

Main signal processing function of BSEC.

Processing of the input signals and returning of output samples is performed by bsec_do_steps().

- The samples of all library inputs must be passed with unique identifiers representing the input signals from physical sensors where the order of these inputs can be chosen arbitrary. However, all input have to be provided within the same time period as they are read. A sequential provision to the library might result in undefined behavior.
- The samples of all library outputs are returned with unique identifiers corresponding to the output signals of virtual sensors where the order of the returned outputs may be arbitrary.
- The samples of all input as well as output signals of physical as well as virtual sensors use the same representation in memory with the following fields:
- Sensor identifier:
  - For inputs: required to identify the input signal from a physical sensor
  - For output: overwritten by bsec_do_steps() to identify the returned signal from a virtual sensor
  - Time stamp of the sample

Calling bsec_do_steps() requires the samples of the input signals to be provided along with their time stamp when they are recorded and only when they are acquired. Repetition of samples with the same time stamp are ignored and result in a warning. Repetition of values of samples which are not acquired anew by a sensor result in deviations of the computed output signals. Concerning the returned output samples, an important feature is, that a value is returned for an output only when a new occurrence has been computed. A sample of an output signal is returned only once.

Parameters

in	*inputs*	Array of input data samples. Each array element represents a sample of a different physical sensor.
in	*n_inputs*	Number of passed input data structs.

Parameters

out	*outputs*	Array of output data samples. Each array element represents a sample of a different virtual sensor.
in,out	*n_outputs*	[in] Number of allocated output structs, [out] number of outputs returned

Returns

Zero when successful, otherwise an error code

```
// Example //

// Allocate input and output memory
bsec_input_t input[3];
uint8_t n_input = 3;
bsec_output_t output[2];
uint8_t  n_output=2;

bsec_library_return_t status;

// Populate the input structs, assuming the we have timestamp (ts),
// gas sensor resistance (R), temperature (T), and humidity (rH) available
// as input variables
input[0].sensor_id = BSEC_INPUT_GASRESISTOR;
input[0].signal = R;
input[0].time_stamp= ts;
input[1].sensor_id = BSEC_INPUT_TEMPERATURE;
input[1].signal = T;
input[1].time_stamp= ts;
input[2].sensor_id = BSEC_INPUT_HUMIDITY;
input[2].signal = rH;
input[2].time_stamp= ts;


// Invoke main processing BSEC function
status = bsec_do_steps( input, n_input, output, &n_output );

// Iterate through the BSEC output data, if the call succeeded
if(status == BSEC_OK)
{
    for(int i = 0; i < n_output; i++)
    {
        switch(output[i].sensor_id)
        {
            case BSEC_OUTPUT_IAQ:
                // Retrieve the IAQ results from output[i].signal
                // and do something with the data
                break;
            case BSEC_OUTPUT_STATIC_IAQ:
                // Retrieve the static IAQ results from output[i].signal
                // and do something with the data
                break;

        }
    }
}
```

4.1.2.2 bsec_get_configuration()

```
bsec_library_return_t bsec_get_configuration (
               const uint8_t config_id,
               uint8_t * serialized_settings,
               const uint32_t n_serialized_settings_max,
               uint8_t * work_buffer,
               const uint32_t n_work_buffer,
               uint32_t * n_serialized_settings )
```

Retrieve the current library configuration.

BSEC allows to retrieve the current configuration using bsec_get_configuration(). Returns a binary blob encoding the current configuration parameters of the library in a format compatible with bsec_set_configuration().

Note

The function bsec_get_configuration() is required to be used for debugging purposes only.
A work buffer with sufficient size is required and has to be provided by the function caller to decompose the serialization and apply it to the library and its modules.

Please use BSEC_MAX_PROPERTY_BLOB_SIZE for allotting the required size.

Parameters

in	*config_id*	Identifier for a specific set of configuration settings to be returned; shall be zero to retrieve all configuration settings.
out	*serialized_settings*	Buffer to hold the serialized config blob
in	*n_serialized_settings_max*	Maximum available size for the serialized settings
in,out	*work_buffer*	Work buffer used to parse the binary blob
in	*n_work_buffer*	Length of the work buffer available for parsing the blob
out	*n_serialized_settings*	Actual size of the returned serialized configuration blob

Returns

Zero when successful, otherwise an error code

```
// Example //

// Allocate variables
uint8_t serialized_settings[BSEC_MAX_PROPERTY_BLOB_SIZE];
uint32_t n_serialized_settings_max = BSEC_MAX_PROPERTY_BLOB_SIZE;
uint8_t work_buffer[BSEC_MAX_WORKBUFFER_SIZE];
uint32_t n_work_buffer = BSEC_MAX_WORKBUFFER_SIZE;
uint32_t n_serialized_settings = 0;

// Configuration of BSEC algorithm is stored in 'serialized_settings'
bsec_get_configuration(0, serialized_settings, n_serialized_settings_max, work_buffer
      , n_work_buffer, &n_serialized_settings);
```

4.1.2.3 bsec_get_state()

```
bsec_library_return_t bsec_get_state (
            const uint8_t state_set_id,
            uint8_t * serialized_state,
            const uint32_t n_serialized_state_max,
            uint8_t * work_buffer,
            const uint32_t n_work_buffer,
            uint32_t * n_serialized_state )
```

Retrieve the current internal library state.

BSEC allows to retrieve the current states of all signal processing modules and the BSEC module using bsec_get_state(). This allows a restart of the processing after a reboot of the system by calling bsec_set_state().

Note

> A work buffer with sufficient size is required and has to be provided by the function caller to decompose the serialization and apply it to the library and its modules.

Please use BSEC_MAX_STATE_BLOB_SIZE for allotting the required size.

Parameters

in	*state_set_id*	Identifier for a specific set of states to be returned; shall be zero to retrieve all states.
out	*serialized_state*	Buffer to hold the serialized config blob
in	*n_serialized_state_max*	Maximum available size for the serialized states
in,out	*work_buffer*	Work buffer used to parse the blob
in	*n_work_buffer*	Length of the work buffer available for parsing the blob
out	*n_serialized_state*	Actual size of the returned serialized blob

Returns

> Zero when successful, otherwise an error code

```
// Example //

// Allocate variables
uint8_t serialized_state[BSEC_MAX_STATE_BLOB_SIZE];
uint32_t n_serialized_state_max = BSEC_MAX_STATE_BLOB_SIZE;
uint32_t  n_serialized_state = BSEC_MAX_STATE_BLOB_SIZE;
uint8_t work_buffer_state[BSEC_MAX_WORKBUFFER_SIZE];
uint32_t  n_work_buffer_size = BSEC_MAX_WORKBUFFER_SIZE;

// Algorithm state is stored in 'serialized_state'
bsec_get_state(0, serialized_state, n_serialized_state_max, work_buffer_state,
      n_work_buffer_size, &n_serialized_state);
```

4.1.2.4 bsec_get_version()

```
bsec_library_return_t bsec_get_version (
               bsec_version_t * bsec_version_p )
```

Return the version information of BSEC library.

Parameters

out	*bsec_version_p*	pointer to struct which is to be populated with the version information

Returns

Zero if successful, otherwise an error code

See also: bsec_version_t

```
// Example //
bsec_version_t  version;
bsec_get_version(&version);
printf("BSEC version: %d.%d.%d.%d",version.major, version.minor, version.
      major_bugfix, version.minor_bugfix);
```

4.1.2.5 bsec_init()

```
bsec_library_return_t bsec_init (
            void  )
```

Initialize the library.

Initialization and reset of BSEC is performed by calling bsec_init(). Calling this function sets up the relation among all internal modules, initializes run-time dependent library states and resets the configuration and state of all BSEC signal processing modules to defaults.

Before any further use, the library must be initialized. This ensure that all memory and states are in defined conditions prior to processing any data.

Returns

Zero if successful, otherwise an error code

```
// Initialize BSEC library before further use
bsec_init();
```

4.1.2.6 bsec_reset_output()

```
bsec_library_return_t bsec_reset_output (
            uint8_t sensor_id )
```

Reset a particular virtual sensor output.

This function allows specific virtual sensor outputs to be reset. The meaning of "reset" depends on the specific output. In case of the IAQ output, reset means zeroing the output to the current ambient conditions.

Parameters

in	*sensor↩ _id*	Virtual sensor to be reset

Returns

Zero when successful, otherwise an error code

```
// Example //
bsec_reset_output(BSEC_OUTPUT_IAQ);
```

4.1.2.7 bsec_sensor_control()

```
bsec_library_return_t bsec_sensor_control (
            const int64_t time_stamp,
            bsec_bme_settings_t * sensor_settings )
```

Retrieve BMExxx sensor instructions.

The bsec_sensor_control() interface is a key feature of BSEC, as it allows an easy way for the signal processing library to control the operation of the BME sensor. This is important since gas sensor behaviour is mainly determined by how the integrated heater is configured. To ensure an easy integration of BSEC into any system, bsec_sensor_control() will provide the caller with information about the current sensor configuration that is necessary to fulfill the input requirements derived from the current outputs requested via bsec_update_↩ subscription().

In practice the use of this function shall be as follows:

- Call bsec_sensor_control() which returns a bsec_bme_settings_t struct.
- Based on the information contained in this struct, the sensor is configured and a forced-mode measurement is triggered if requested by bsec_sensor_control().
- Once this forced-mode measurement is complete, the signals specified in this struct shall be passed to bsec_do_steps() to perform the signal processing.
- After processing, the process should sleep until the bsec_bme_settings_t::next_call timestamp is reached.

Parameters

in	*time_stamp*	Current timestamp in [ns]
out	*sensor_settings*	Settings to be passed to API to operate sensor at this time instance

Returns

Zero when successful, otherwise an error code

4.1.2.8 bsec_set_configuration()

```
bsec_library_return_t bsec_set_configuration (
            const uint8_t *const serialized_settings,
            const uint32_t n_serialized_settings,
            uint8_t * work_buffer,
            const uint32_t n_work_buffer_size )
```

Update algorithm configuration parameters.

BSEC uses a default configuration for the modules and common settings. The initial configuration can be customized by bsec_set_configuration(). This is an optional step.

Note

A work buffer with sufficient size is required and has to be provided by the function caller to decompose the serialization and apply it to the library and its modules.

Please use BSEC_MAX_PROPERTY_BLOB_SIZE for allotting the required size.

Parameters

in	*serialized_settings*	Settings serialized to a binary blob
in	*n_serialized_settings*	Size of the settings blob
in,out	*work_buffer*	Work buffer used to parse the blob
in	*n_work_buffer_size*	Length of the work buffer available for parsing the blob

Returns

Zero when successful, otherwise an error code

```
// Example //

// Allocate variables
uint8_t serialized_settings[BSEC_MAX_PROPERTY_BLOB_SIZE];
uint32_t n_serialized_settings_max = BSEC_MAX_PROPERTY_BLOB_SIZE;
uint8_t work_buffer[BSEC_MAX_WORKBUFFER_SIZE];
uint32_t n_work_buffer = BSEC_MAX_WORKBUFFER_SIZE;

// Here we will load a provided config string into serialized_settings

// Apply the configuration
bsec_set_configuration(serialized_settings, n_serialized_settings_max, work_buffer,
      n_work_buffer);
```

4.1.2.9 bsec_set_state()

```
bsec_library_return_t bsec_set_state (
              const uint8_t *const serialized_state,
              const uint32_t n_serialized_state,
              uint8_t * work_buffer,
              const uint32_t n_work_buffer_size )
```

Restore the internal state of the library.

BSEC uses a default state for all signal processing modules and the BSEC module. To ensure optimal performance, especially of the gas sensor functionality, it is recommended to retrieve the state using bsec_get_state() before unloading the library, storing it in some form of non-volatile memory, and setting it using bsec_set_state() before resuming further operation of the library.

Note

A work buffer with sufficient size is required and has to be provided by the function caller to decompose the serialization and apply it to the library and its modules.

Please use BSEC_MAX_STATE_BLOB_SIZE for allotting the required size.

Parameters

in	*serialized_state*	States serialized to a binary blob
in	*n_serialized_state*	Size of the state blob
in,out	*work_buffer*	Work buffer used to parse the blob
in	*n_work_buffer_size*	Length of the work buffer available for parsing the blob

Returns

Zero when successful, otherwise an error code

```
// Example //

// Allocate variables
uint8_t serialized_state[BSEC_MAX_PROPERTY_BLOB_SIZE];
uint32_t  n_serialized_state = BSEC_MAX_PROPERTY_BLOB_SIZE;
uint8_t work_buffer_state[BSEC_MAX_WORKBUFFER_SIZE];
uint32_t  n_work_buffer_size = BSEC_MAX_WORKBUFFER_SIZE;

// Here we will load a state string from a previous use of BSEC

// Apply the previous state to the current BSEC session
bsec_set_state(serialized_state, n_serialized_state, work_buffer_state, n_work_buffer_size);
```

4.1.2.10 bsec_update_subscription()

```
bsec_library_return_t bsec_update_subscription (
            const bsec_sensor_configuration_t *const requested_virtual_sensors,
            const uint8_t n_requested_virtual_sensors,
            bsec_sensor_configuration_t * required_sensor_settings,
            uint8_t * n_required_sensor_settings )
```

Subscribe to library virtual sensors outputs.

Use bsec_update_subscription() to instruct BSEC which of the processed output signals are requested at which sample rates. See bsec_virtual_sensor_t for available library outputs.

Based on the requested virtual sensors outputs, BSEC will provide information about the required physical sensor input signals (see bsec_physical_sensor_t) with corresponding sample rates. This information is purely informational as bsec_sensor_control() will ensure the sensor is operated in the required manner. To disable a virtual sensor, set the sample rate to BSEC_SAMPLE_RATE_DISABLED.

The subscription update using bsec_update_subscription() is apart from the signal processing one of the the most important functions. It allows to enable the desired library outputs. The function determines which physical input sensor signals are required at which sample rate to produce the virtual output sensor signals requested by the user. When this function returns with success, the requested outputs are called subscribed. A very important feature is the retaining of already subscribed outputs. Further outputs can be requested or disabled both individually and group-wise in addition to already subscribed outputs without changing them unless a change of already subscribed outputs is requested.

Note

The state of the library concerning the subscribed outputs cannot be retained among reboots.

The interface of bsec_update_subscription() requires the usage of arrays of sensor configuration structures. Such a structure has the fields sensor identifier and sample rate. These fields have the properties:

- Output signals of virtual sensors must be requested using unique identifiers (Member of bsec_virtual_sensor_t)
- Different sets of identifiers are available for inputs of physical sensors and outputs of virtual sensors
- Identifiers are unique values defined by the library, not from external
- Sample rates must be provided as value of
  - An allowed sample rate for continuously sampled signals
  - 65535.0f (BSEC_SAMPLE_RATE_DISABLED) to turn off outputs and identify disabled inputs

Note

The same sensor identifiers are also used within the functions bsec_do_steps().

The usage principles of bsec_update_subscription() are:

- Differential updates (i.e., only asking for outputs that the user would like to change) is supported.
- Invalid requests of outputs are ignored. Also if one of the requested outputs is unavailable, all the requests are ignored. At the same time, a warning is returned.
- To disable BSEC, all outputs shall be turned off. Only enabled (subscribed) outputs have to be disabled while already disabled outputs do not have to be disabled explicitly.

Parameters

in	*requested_virtual_sensors*	Pointer to array of requested virtual sensor (output) configurations for the library
in	*n_requested_virtual_sensors*	Number of virtual sensor structs pointed by requested_virtual_sensors
out	*required_sensor_settings*	Pointer to array of required physical sensor configurations for the library
in,out	*n_required_sensor_settings*	[in] Size of allocated required_sensor_settings array, [out] number of sensor configurations returned

Returns

Zero when successful, otherwise an error code

See also

bsec_sensor_configuration_t
bsec_physical_sensor_t
bsec_virtual_sensor_t

```
// Example //

// Change 3 virtual sensors (switch IAQ and raw temperature -> on / pressure -> off)
bsec_sensor_configuration_t requested_virtual_sensors[3];
uint8_t n_requested_virtual_sensors = 3;

requested_virtual_sensors[0].sensor_id = BSEC_OUTPUT_IAQ;
requested_virtual_sensors[0].sample_rate = BSEC_SAMPLE_RATE_ULP;
requested_virtual_sensors[1].sensor_id = BSEC_OUTPUT_RAW_TEMPERATURE;
requested_virtual_sensors[1].sample_rate = BSEC_SAMPLE_RATE_ULP;
requested_virtual_sensors[2].sensor_id = BSEC_OUTPUT_RAW_PRESSURE;
requested_virtual_sensors[2].sample_rate = BSEC_SAMPLE_RATE_DISABLED;

// Allocate a struct for the returned physical sensor settings
bsec_sensor_configuration_t required_sensor_settings[
      BSEC_MAX_PHYSICAL_SENSOR];
uint8_t  n_required_sensor_settings = BSEC_MAX_PHYSICAL_SENSOR;

// Call bsec_update_subscription() to enable/disable the requested virtual sensors
bsec_update_subscription(requested_virtual_sensors, n_requested_virtual_sensors,
      required_sensor_settings, &n_required_sensor_settings);
```

4.1.3 Enumerations

4.1.3.1 bsec_library_return_t

`enum bsec_library_return_t`

Enumeration for function return codes.

Enumerator

BSEC_OK	Function execution successful
BSEC_E_DOSTEPS_INVALIDINPUT	Input (physical) sensor id passed to bsec_do_steps() is not in the valid range or not valid for requested virtual sensor
BSEC_E_DOSTEPS_VALUELIMITS	Value of input (physical) sensor signal passed to bsec_do_steps() is not in the valid range
BSEC_W_DOSTEPS_TSINTRADIFFOUTOFRANGE	Past timestamps passed to bsec_do_steps()
BSEC_E_DOSTEPS_DUPLICATEINPUT	Duplicate input (physical) sensor ids passed as input to bsec_do_steps()
BSEC_I_DOSTEPS_NOOUTPUTSRETURNABLE	No memory allocated to hold return values from bsec_do_steps(), i.e., n_outputs == 0
BSEC_W_DOSTEPS_EXCESSOUTPUTS	Not enough memory allocated to hold return values from bsec_do_steps(), i.e., n_outputs < maximum number of requested output (virtual) sensors
BSEC_E_SU_WRONGDATARATE	The sample_rate of the requested output (virtual) sensor passed to bsec_update_subscription() is zero
BSEC_E_SU_SAMPLERATELIMITS	The sample_rate of the requested output (virtual) sensor passed to bsec_update_subscription() does not match with the sampling rate allowed for that sensor
BSEC_E_SU_DUPLICATEGATE	Duplicate output (virtual) sensor ids requested through bsec_update_subscription()
BSEC_E_SU_INVALIDSAMPLERATE	The sample_rate of the requested output (virtual) sensor passed to bsec_update_subscription() does not fall within the global minimum and maximum sampling rates
BSEC_E_SU_GATECOUNTEXCEEDSARRAY	Not enough memory allocated to hold returned input (physical) sensor data from bsec_update_subscription(), i.e., n_required_sensor_settings < BSEC_MAX_PHYSICAL_SENSOR
BSEC_E_SU_SAMPLINTVLINTEGERMULT	The sample_rate of the requested output (virtual) sensor passed to bsec_update_subscription() is not correct
BSEC_E_SU_MULTGASSAMPLINTVL	The sample_rate of the requested output (virtual), which requires the gas sensor, is not equal to the sample_rate that the gas sensor is being operated

Enumerator

BSEC_E_SU_HIGHHEATERONDURATION	The duration of one measurement is longer than the requested sampling interval
BSEC_W_SU_UNKNOWNOUTPUTGATE	Output (virtual) sensor id passed to bsec_update_subscription() is not in the valid range; e.g., n_requested_virtual_sensors > actual number of output (virtual) sensors requested
BSEC_W_SU_MODINNOULP	ULP plus can not be requested in non-ulp mode
BSEC_I_SU_SUBSCRIBEDOUTPUTGATES	No output (virtual) sensor data were requested via bsec_update_subscription()
BSEC_E_PARSE_SECTIONEXCEEDSWORKBU↩ FFER	n_work_buffer_size passed to bsec_set_[configuration/state]() not sufficient
BSEC_E_CONFIG_FAIL	Configuration failed
BSEC_E_CONFIG_VERSIONMISMATCH	Version encoded in serialized_[settings/state] passed to bsec_set_[configuration/state]() does not match with current version
BSEC_E_CONFIG_FEATUREMISMATCH	Enabled features encoded in serialized_[settings/state] passed to bsec_set_[configuration/state]() does not match with current library implementation
BSEC_E_CONFIG_CRCMISMATCH	serialized_[settings/state] passed to bsec_set_[configuration/state]() is corrupted
BSEC_E_CONFIG_EMPTY	n_serialized_[settings/state] passed to bsec_set_[configuration/state]() is to short to be valid
BSEC_E_CONFIG_INSUFFICIENTWORKBUFFER	Provided work_buffer is not large enough to hold the desired string
BSEC_E_CONFIG_INVALIDSTRINGSIZE	String size encoded in configuration/state strings passed to bsec_set_[configuration/state]() does not match with the actual string size n_serialized_[settings/state] passed to these functions
BSEC_E_CONFIG_INSUFFICIENTBUFFER	String buffer insufficient to hold serialized data from BSEC library
BSEC_E_SET_INVALIDCHANNELIDENTIFIER	Internal error code, size of work buffer in setConfig must be set to BSEC_MAX_WORKBUFFER_SIZE
BSEC_E_SET_INVALIDLENGTH	Internal error code
BSEC_W_SC_CALL_TIMING_VIOLATION	Difference between actual and defined sampling intervals of bsec_sensor_control() greater than allowed
BSEC_W_SC_MODEXCEEDULPTIMELIMIT	ULP plus is not allowed because an ULP measurement just took or will take place
BSEC_W_SC_MODINSUFFICIENTWAITTIME	ULP plus is not allowed because not sufficient time passed since last ULP plus

4.1.3.2 bsec_physical_sensor_t

`enum bsec_physical_sensor_t`

Enumeration for input (physical) sensors.

Used to populate bsec_input_t::sensor_id. It is also used in bsec_sensor_configuration_t::sensor_id structs returned in the parameter required_sensor_settings of bsec_update_subscription().

See also

bsec_sensor_configuration_t
bsec_input_t

Enumerator

BSEC_INPUT_PRESSURE	Pressure sensor output of BMExxx [Pa].
BSEC_INPUT_HUMIDITY	Humidity sensor output of BMExxx [%]. Note Relative humidity strongly depends on the temperature (it is measured at). It may require a conversion to the temperature outside of the device. See also bsec_virtual_sensor_t
BSEC_INPUT_TEMPERATURE	Temperature sensor output of BMExxx [degrees Celsius]. Note The BME680 is factory trimmed, thus the temperature sensor of the BME680 is very accurate. The temperature value is a very local measurement value and can be influenced by external heat sources. See also bsec_virtual_sensor_t
BSEC_INPUT_GASRESISTOR	Gas sensor resistance output of BMExxx [Ohm]. The resistance value changes due to varying VOC concentrations (the higher the concentration of reducing VOCs, the lower the resistance and vice versa).

Enumerator

BSEC_INPUT_HEATSOURCE	Additional input for device heat compensation. IAQ solution: The value is subtracted from BSEC_INPUT_TEMPERATURE to compute BSEC_OUTPUT_SENSOR_HEAT_COMPENSATED↩ _TEMPERATURE. ALL solution: Generic heat source 1 See also bsec_virtual_sensor_t
BSEC_INPUT_DISABLE_BASELINE_TRACKER	Additional input that disables baseline tracker. 0 - Normal, 1 - Event 1, 2 - Event 2

4.1.3.3 bsec_virtual_sensor_t

`enum bsec_virtual_sensor_t`

Enumeration for output (virtual) sensors.

Used to populate bsec_output_t::sensor_id. It is also used in bsec_sensor_configuration_t::sensor_id structs passed in the parameter requested_virtual_sensors of bsec_update_subscription().

See also

bsec_sensor_configuration_t
bsec_output_t

Enumerator

BSEC_OUTPUT_IAQ	Index for Air Quality estimate [0-500]. Index for Air Quality (IAQ) gives an indication of the relative change in ambient TVOCs detected by BME680. Note The IAQ scale ranges from 0 (clean air) to 500 (heavily polluted air). During operation, algorithms automatically calibrate and adapt themselves to the typical environments where the sensor is operated (e.g., home, workplace, inside a car, etc.).This automatic background calibration ensures that users experience consistent IAQ performance. The calibration process considers the recent measurement history (typ. up to four days) to ensure that IAQ=50 corresponds to typical good air and IAQ=200 indicates typical polluted air.
BSEC_OUTPUT_STATIC_IAQ	Unscaled Index for Air Quality estimate

Enumerator

BSEC_OUTPUT_CO2_EQUIVALENT	CO2 equivalent estimate [ppm]
BSEC_OUTPUT_BREATH_VOC_EQUIVALENT	Breath VOC concentration estimate [ppm]
BSEC_OUTPUT_RAW_TEMPERATURE	Temperature sensor signal [degrees Celsius]. Temperature directly measured by BME680 in degree Celsius. Note This value is cross-influenced by the sensor heating and device specific heating.
BSEC_OUTPUT_RAW_PRESSURE	Pressure sensor signal [Pa]. Pressure directly measured by the BME680 in Pa.
BSEC_OUTPUT_RAW_HUMIDITY	Relative humidity sensor signal [%]. Relative humidity directly measured by the BME680 in %. Note This value is cross-influenced by the sensor heating and device specific heating.
BSEC_OUTPUT_RAW_GAS	Gas sensor signal [Ohm]. Gas resistance measured directly by the BME680 in Ohm.The resistance value changes due to varying VOC concentrations (the higher the concentration of reducing VOCs, the lower the resistance and vice versa).
BSEC_OUTPUT_STABILIZATION_STATUS	Gas sensor stabilization status [boolean]. Indicates initial stabilization status of the gas sensor element: stabilization is ongoing (0) or stabilization is finished (1).
BSEC_OUTPUT_RUN_IN_STATUS	Gas sensor run-in status [boolean]. Indicates power-on stabilization status of the gas sensor element: stabilization is ongoing (0) or stabilization is finished (1).

Enumerator

BSEC_OUTPUT_SENSOR_HEAT_COMPENSATED_TEMPERATURE	Sensor heat compensated temperature [degrees Celsius]. Temperature measured by BME680 which is compensated for the influence of sensor (heater) in degree Celsius. The self heating introduced by the heater is depending on the sensor operation mode and the sensor supply voltage. Note IAQ solution: In addition, the temperature output can be compensated by an user defined value (BSEC_INPUT_HEATSOURCE in degrees Celsius), which represents the device specific self-heating. Thus, the value is calculated as follows: ▶ IAQ solution: `BSEC_OUTPUT_SENSOR_HEAT_COMPENSATED_TEMPERATURE = BSEC_INPUT_TEMPERATURE - function(sensor operation mode, sensor supply voltage) - BSEC_INPUT_HEATSOURCE` ▶ Other solutions: `BSEC_OUTPUT_SENSOR_HEAT_COMPENSATED_TEMPERATURE = BSEC_INPUT_TEMPERATURE - function(sensor operation mode, sensor supply voltage)` The self-heating in operation mode BSEC_SAMPLE_RATE_ULP is negligible. The self-heating in operation mode BSEC_SAMPLE_RATE_LP is supported for 1.8V by default (no config file required). If the BME680 sensor supply voltage is 3.3V, the corresponding config file shall be used.

Enumerator

BSEC_OUTPUT_SENSOR_HEAT_COMPENSATED_HUMIDITY	Sensor heat compensated humidity [%]. Relative measured by BME680 which is compensated for the influence of sensor (heater) in %. It converts the BSEC_INPUT_HUMIDITY from temperature BSEC_INPUT_TEMPERATURE to temperature BSEC_OUTPUT_SENSOR_HEAT_COMPENSATED_TEMPERATURE. Note IAQ solution: If BSEC_INPUT_HEATSOURCE is used for device specific temperature compensation, it will be effective for BSEC_OUTPUT_SENSOR_HEAT_COMPENSATED_HUMIDITY too.
BSEC_OUTPUT_GAS_PERCENTAGE	Percentage of min and max filtered gas value [%]

4.1.4 Defines

4.1.4.1 BSEC_MAX_PHYSICAL_SENSOR

```
#define BSEC_MAX_PHYSICAL_SENSOR (8)
```

Number of physical sensors that need allocated space before calling bsec_update_subscription()

4.1.4.2 BSEC_MAX_PROPERTY_BLOB_SIZE

```
#define BSEC_MAX_PROPERTY_BLOB_SIZE (462)
```

Maximum size (in bytes) of the data blobs returned by bsec_get_configuration()

4.1.4.3 BSEC_MAX_STATE_BLOB_SIZE

```
#define BSEC_MAX_STATE_BLOB_SIZE (155)
```

Maximum size (in bytes) of the data blobs returned by bsec_get_state()

4.1.4.4 BSEC_MAX_WORKBUFFER_SIZE

```
#define BSEC_MAX_WORKBUFFER_SIZE (2048)
```

Maximum size (in bytes) of the work buffer

4.1.4.5 BSEC_NUMBER_OUTPUTS

```
#define BSEC_NUMBER_OUTPUTS (14)
```

Number of outputs, depending on solution

4.1.4.6 BSEC_OUTPUT_INCLUDED

```
#define BSEC_OUTPUT_INCLUDED (1210863)
```

bitfield that indicates which outputs are included in the solution

4.1.4.7 BSEC_PROCESS_GAS

```
#define BSEC_PROCESS_GAS (1 << (BSEC_INPUT_GASRESISTOR-1))
```

process_data bitfield constant for gas sensor

See also

bsec_bme_settings_t

4.1.4.8 BSEC_PROCESS_HUMIDITY

```
#define BSEC_PROCESS_HUMIDITY (1 << (BSEC_INPUT_HUMIDITY-1))
```

process_data bitfield constant for humidity

See also

bsec_bme_settings_t

4.1.4.9 BSEC_PROCESS_PRESSURE

```
#define BSEC_PROCESS_PRESSURE (1 << (BSEC_INPUT_PRESSURE-1))
```

process_data bitfield constant for pressure

See also

bsec_bme_settings_t

4.1.4.10 BSEC_PROCESS_TEMPERATURE

```
#define BSEC_PROCESS_TEMPERATURE (1 << (BSEC_INPUT_TEMPERATURE-1))
```

process_data bitfield constant for temperature

See also

bsec_bme_settings_t

4.1.4.11 BSEC_SAMPLE_RATE_CONT

```
#define BSEC_SAMPLE_RATE_CONT (1.0f)
```

Sample rate in case of Continuous Mode

4.1.4.12 BSEC_SAMPLE_RATE_DISABLED

```
#define BSEC_SAMPLE_RATE_DISABLED (65535.0f)
```

Sample rate of a disabled sensor

4.1.4.13 BSEC_SAMPLE_RATE_LP

```
#define BSEC_SAMPLE_RATE_LP (0.33333f)
```

Sample rate in case of Low Power Mode

4.1.4.14 BSEC_SAMPLE_RATE_ULP

```
#define BSEC_SAMPLE_RATE_ULP (0.0033333f)
```

Sample rate in case of Ultra Low Power Mode

4.1.4.15 BSEC_SAMPLE_RATE_ULP_MEASUREMENT_ON_DEMAND

```
#define BSEC_SAMPLE_RATE_ULP_MEASUREMENT_ON_DEMAND (0.0f)
```

Input value used to trigger an extra measurment (ULP plus)

4.1.4.16 BSEC_STRUCT_NAME

```
#define BSEC_STRUCT_NAME Bsec
```

Internal struct name

5 Data Structure Documentation

5.1 bsec_bme_settings_t Struct Reference

Data Fields

- int64_t next_call
  Time stamp of the next call of the sensor_control.
- uint32_t process_data
  Bit field describing which data is to be passed to bsec_do_steps()
- uint16_t heater_temperature
  Heating temperature [degrees Celsius].
- uint16_t heating_duration
  Heating duration [ms].
- uint8_t run_gas
  Enable gas measurements [0/1].
- uint8_t pressure_oversampling
  Pressure oversampling settings [0-5].
- uint8_t temperature_oversampling
  Temperature oversampling settings [0-5].
- uint8_t humidity_oversampling
  Humidity oversampling settings [0-5].
- uint8_t trigger_measurement
  Trigger a forced measurement with these settings now [0/1].

5.1.1 Detailed Description

Structure returned by bsec_sensor_control() to configure BMExxx sensor.

This structure contains settings that must be used to configure the BMExxx to perform a forced-mode measurement. A measurement should only be executed if bsec_bme_settings_t::trigger_measurement is 1. If so, the oversampling settings for temperature, humidity, and pressure should be set to the provided settings provided in bsec_bme_settings_t::temperature_oversampling, bsec_bme_settings_t::humidity_oversampling, and bsec_bme_settings_t::pressure_oversampling, respectively.

In case of bsec_bme_settings_t::run_gas = 1, the gas sensor must be enabled with the provided bsec_bme_settings_t::heater_temperature and bsec_bme_settings_t::heating_duration settings.

5.1.2 Field Documentation

5.1.2.1 heater_temperature

`uint16_t bsec_bme_settings_t::heater_temperature`

Heating temperature [degrees Celsius].

5.1.2.2 heating_duration

`uint16_t bsec_bme_settings_t::heating_duration`

Heating duration [ms].

5.1.2.3 humidity_oversampling

`uint8_t bsec_bme_settings_t::humidity_oversampling`

Humidity oversampling settings [0-5].

5.1.2.4 next_call

`int64_t bsec_bme_settings_t::next_call`

Time stamp of the next call of the sensor_control.

5.1.2.5 pressure_oversampling

`uint8_t bsec_bme_settings_t::pressure_oversampling`

Pressure oversampling settings [0-5].

5.1.2.6 process_data

`uint32_t bsec_bme_settings_t::process_data`

Bit field describing which data is to be passed to bsec_do_steps()

See also

BSEC_PROCESS_GAS, BSEC_PROCESS_TEMPERATURE, BSEC_PROCESS_HUMIDITY, BSEC_P↩
ROCESS_PRESSURE

5.1.2.7 run_gas

`uint8_t bsec_bme_settings_t::run_gas`

Enable gas measurements [0/1].

5.1.2.8 temperature_oversampling

```
uint8_t bsec_bme_settings_t::temperature_oversampling
```

Temperature oversampling settings [0-5].

5.1.2.9 trigger_measurement

```
uint8_t bsec_bme_settings_t::trigger_measurement
```

Trigger a forced measurement with these settings now [0/1].

5.2 bsec_input_t Struct Reference

Data Fields

- int64_t time_stamp

  Time stamp in nanosecond resolution [ns].
- float signal

  Signal sample in the unit defined for the respective sensor_id.
- uint8_t signal_dimensions

  Signal dimensions (reserved for future use, shall be set to 1)
- uint8_t sensor_id

  Identifier of physical sensor.

5.2.1 Detailed Description

Structure describing an input sample to the library.

Each input sample is provided to BSEC as an element in a struct array of this type. Timestamps must be provided in nanosecond resolution. Moreover, duplicate timestamps for subsequent samples are not allowed and will results in an error code being returned from bsec_do_steps().

The meaning unit of the signal field are determined by the bsec_input_t::sensor_id field content. Possible bsec_input_t::sensor_id values and and their meaning are described in bsec_physical_sensor_t.

See also

bsec_physical_sensor_t

5.2.2 Field Documentation

5.2.2.1 sensor_id

```
uint8_t bsec_input_t::sensor_id
```

Identifier of physical sensor.

See also

bsec_physical_sensor_t

5.2.2.2 signal

`float bsec_input_t::signal`

Signal sample in the unit defined for the respective sensor_id.

See also

bsec_physical_sensor_t

5.2.2.3 signal_dimensions

`uint8_t bsec_input_t::signal_dimensions`

Signal dimensions (reserved for future use, shall be set to 1)

5.2.2.4 time_stamp

`int64_t bsec_input_t::time_stamp`

Time stamp in nanosecond resolution [ns].

Timestamps must be provided as non-repeating and increasing values. They can have their 0-points at system start or at a defined wall-clock time (e.g., 01-Jan-1970 00:00:00)

5.3 bsec_output_t Struct Reference

Data Fields

- int64_t time_stamp

  Time stamp in nanosecond resolution as provided as input [ns].
- float signal

  Signal sample in the unit defined for the respective bsec_output_t::sensor_id.
- uint8_t signal_dimensions

  Signal dimensions (reserved for future use, shall be set to 1)
- uint8_t sensor_id

  Identifier of virtual sensor.
- uint8_t accuracy

  Accuracy status 0-3.

5.3.1 Detailed Description

Structure describing an output sample of the library.

Each output sample is returned from BSEC by populating the element of a struct array of this type. The contents of the signal field is defined by the supplied bsec_output_t::sensor_id. Possible output bsec_output_t::sensor_id values are defined in bsec_virtual_sensor_t.

See also

bsec_virtual_sensor_t

5.3.2 Field Documentation

5.3.2.1 accuracy

`uint8_t bsec_output_t::accuracy`

Accuracy status 0-3.

Some virtual sensors provide a value in the accuracy field. If this is the case, the meaning of the field is as follows:

Name	Value	Accuracy description
UNRELIABLE	0	Sensor data is unreliable, the sensor must be calibrated
LOW_ACCURACY	1	Low accuracy, sensor should be calibrated
MEDIUM_ACCURACY	2	Medium accuracy, sensor calibration may improve performance
HIGH_ACCURACY	3	High accuracy

For example:

- IAQ accuracy indicator will notify the user when she/he should initiate a calibration process. Calibration is performed automatically in the background if the sensor is exposed to clean and polluted air for approximately 30 minutes each.

Virtual sensor	Value	Accuracy description
IAQ	0	Stabilization / run-in ongoing
	1	Low accuracy,to reach high accuracy(3),please expose sensor once to good air (e.g. outdoor air) and bad air (e.g. box with exhaled breath) for auto-trimming
	2	Medium accuracy: auto-trimming ongoing
	3	High accuracy

5.3.2.2 sensor_id

`uint8_t bsec_output_t::sensor_id`

Identifier of virtual sensor.

See also

bsec_virtual_sensor_t

5.3.2.3 signal

`float bsec_output_t::signal`

Signal sample in the unit defined for the respective bsec_output_t::sensor_id.

See also

bsec_virtual_sensor_t

5.3.2.4 signal_dimensions

`uint8_t bsec_output_t::signal_dimensions`

Signal dimensions (reserved for future use, shall be set to 1)

5.3.2.5 time_stamp

`int64_t bsec_output_t::time_stamp`

Time stamp in nanosecond resolution as provided as input [ns].

5.4 bsec_sensor_configuration_t Struct Reference

Data Fields

- float sample_rate

  Sample rate of the virtual or physical sensor in Hertz [Hz].
- uint8_t sensor_id

  Identifier of the virtual or physical sensor.

5.4.1 Detailed Description

Structure describing sample rate of physical/virtual sensors.

This structure is used together with bsec_update_subscription() to enable BSEC outputs and to retrieve information about the sample rates used for BSEC inputs.

5.4.2 Field Documentation

5.4.2.1 sample_rate

`float bsec_sensor_configuration_t::sample_rate`

Sample rate of the virtual or physical sensor in Hertz [Hz].

Only supported sample rates are allowed.

5.4.2.2 sensor_id

```
uint8_t bsec_sensor_configuration_t::sensor_id
```

Identifier of the virtual or physical sensor.

The meaning of this field changes depending on whether the structs are as the requested_virtual_sensors argument to bsec_update_subscription() or as the required_sensor_settings argument.

bsec_update_subscription() argument	**sensor_id field interpretation**
requested_virtual_sensors	bsec_virtual_sensor_t
required_sensor_settings	bsec_physical_sensor_t

See also

bsec_physical_sensor_t
bsec_virtual_sensor_t

5.5 bsec_version_t Struct Reference

Data Fields

- uint8_t major
  Major version.
- uint8_t minor
  Minor version.
- uint8_t major_bugfix
  Major bug fix version.
- uint8_t minor_bugfix
  Minor bug fix version.

5.5.1 Detailed Description

Structure containing the version information.

Please note that configuration and state strings are coded to a specific version and will not be accepted by other versions of BSEC.

5.5.2 Field Documentation

5.5.2.1 major

```
uint8_t bsec_version_t::major
```

Major version.

5.5.2.2 major_bugfix

```
uint8_t bsec_version_t::major_bugfix
```

Major bug fix version.

5.5.2.3 minor

```
uint8_t bsec_version_t::minor
```

Minor version.

5.5.2.4 minor_bugfix

```
uint8_t bsec_version_t::minor_bugfix
```

Minor bug fix version.