This application note describes the usage of I2C on Talaria TWO using sensors integrated in the EVB board.

# I2C

Talaria TWO modules include I2C bus interface that can serve as an I2C master or slave. The SCL and SDA lines can be individually programmed for use on any GPIO. Internal pull-up resistors are available for SCL/SDA on all GPIOs except for GPIO18.

**Note**: GPIO18 only has internal pull-down resistors.

|  |  |
| --- | --- |
| **I2C Specification** | **Details** |
| Data Rates | 100Kbps, 400Kbps, 1Mbps |
| Address Modes | 7-bit, 10-bit |
| Other Features | Send STOP at End  NOSTART Before Msg  IGNORE NAK From Slave |

Table 1: I2C specifications

# I2C Sensors

The INP301x board has the following sensors available on board for quick prototyping/testing:

1. Temperature/Humidity (Sensirion SHTC3)
2. Pressure (Bosch BMP388)
3. Light (TI OPT3002)

A picture containing text, electronics, circuit

Description automatically generated

Figure : On-board I2C sensors

**Note :** To use the sensors on I2C bus the jumpers J7, J8 and pins 1 & 2 of J1 should be connected as shown in the picture below. This enables the I2C clock, I2C data and power connection to the sensors on board.

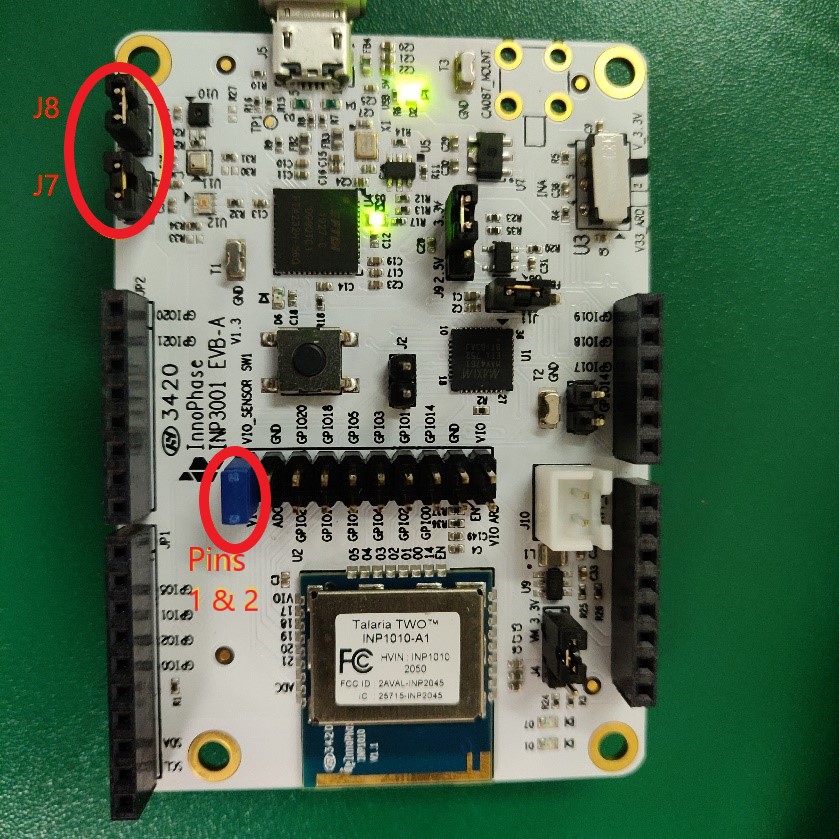


Figure 2: I2C sensor jumper connection

This application enables I2C supported sensors available on the EVB. It measures and displays the real-time values of pressure, temperature, humidity and light using the on-board sensors.

# Source Code Walkthrough

## Directory Structure

Figure 3: File directory tree

* 1. **i2c\_sensor**: The i2c\_sensor.c file present in this directory contains the logic to configure the i2c bus, read and display the readings from the sensors periodically.
  2. **include**: contains header files with the structure variables, unions, and prototypes of the functions to initialize, read and write corresponding each of the sensors.
  3. **sensor**:
     1. **bmp388**

The bmp388.c file in this directory contains the function definitions to initialize, configure and read the data from bmp388 pressure sensor.

* + 1. **sensor\_delay**

It contains routines to generate delay in milli seconds.

* + 1. **opt3002**

The opt3002.c file in this directory contains function definitions to initialize, configure and read opt3002 Light to Digital Sensor.

* + 1. **sensor.h**

This header file contains structure definitions required to initialize and read the sensor readings.

* + 1. **shtc1-4.1.0**

The shtc1-4.1.0.c file in this directory contains function definitions for i2c abstraction layer and commonly shared code.

## Application Flow

In this application, Talaria TWO is programmed to enable I2C communication with the sensors integrated in the EVB board. This application read the on-board sensor readings and print it to the console.

Following are the steps to achieve this as per the i2c\_sensor.c:

1. This application creates a thread to initializes i2c bus.
2. Initializes the sensors available in EVB.
3. Reads the sensor ids from the sensors.
4. Starts fetching the readings from sensors.
5. Prints the sensor readings in console.

## I2C APIs

1. i2c\_bus\_init - Return a handle for the specified bus.

This function is the first one to call when working with I2C devices. This call will initialize the bus driver and returns a handle for the new bus. The Talaria TWO device provides a single I2C interface, therefor the bus no must be set to 0.

1. i2c\_acquire\_bus() - Take ownership of the bus.

Called to claim ownership of the I2C bus. If another thread is currently operating the bus, the function will block until the bus becomes available.

1. i2c\_release\_bus() - Release ownership of the bus.

Not normally used since the i2c\_transfer() function handles this. See rationale in i2c\_acquire\_bus().

1. i2c\_create\_device() - Create a new I2 C device on the specified bus.

This function will create an object representing an I2C device attached to the specified bus. The device is specified using its address on this bus. The maximum frequency this device can handle is also specified in this call.

1. i2c\_destroy\_device() - Free an I2 C device.

This function will release and free the resources claimed by a previously initialized i2c\_device.

1. i2c\_set\_address() - Change the I2 C address for an i2c\_device.

Some devices require certain messages to be address to a different address (For example: while performing a device reset). This function will assist in temporarily changing the device address.

1. i2c\_transfer()- Perform one I2 C transfer.

An I2C transfer consists of several messages. Each message has a direction, READ (from slave to master) or WRITE (from master to slave). The msg parameter is a pointer to an array of messages and the len parameter specifies the number of messages that make up the transfer.

## Source Files

### i2c\_sensor.c

This section describes the sample application used to read the sensor values over I2C. The sensor\_app\_init() creates a thread called sensor\_app\_main that initializes the I2C bus. Sensors fetch the sensor IDs, reads and prints the sensor data.

|  |
| --- |
| /\* Create a thread to handle the i2c sensor \*/  xTaskCreate(sensor\_app\_main, /\* The function that implements the task. \*/  "sens\_thread", /\* The text name assigned to the task - for debug only as  \* it is not used by the kernel. \*/  APP\_THREAD\_STACK\_SIZE  / 4, /\* The size of the stack to allocate to the task. \*/  NULL, /\* The parameter passed to the task - not used in this case. \*/  (APP\_THREAD\_PRIO), /\* The priority assigned to the task. \*/  &app\_thread);  if (app\_thread == NULL) {  os\_printf(" thread creation failed\n");  return;  }  vTaskSuspend(NULL); |

xTaskCreate routine initializes the given thread pointed to the argument and puts the thread on an active queue. This app thread allows the user to implement concurrent functions at the same time.

The sensor\_app\_main thread initiates the I2C bus by calling the init\_i2c thread post which it starts fetching the sensor ID and prints the readings to the console.

init\_i2c will initialize the bus driver after enabling the internal pull-ups on SCL and SDA pins of Talaria TWO module. It routes the SCL and SDA pins to the corresponding GPIOs.

|  |
| --- |
| os\_gpio\_set\_pull(GPIO\_PIN(SCL\_PIN) | GPIO\_PIN(SDA\_PIN));  os\_gpio\_mux\_sel(GPIO\_MUX\_SEL\_SCL, SCL\_PIN);  os\_gpio\_mux\_sel(GPIO\_MUX\_SEL\_SDA, SDA\_PIN);  return i2c\_bus\_init(0); |

It then begins to initialize the I2C bus with corresponding GPIO pins after which it initiates the sensors by init\_sensors().

|  |
| --- |
| int rc;  struct i2c\_bus \*bus = NULL;  sensor\_id\_t ids = {};  bus = init\_i2c();  init\_sensors(bus);  get\_sensor\_ids(&ids);  print\_sensor\_ids(&ids);  os\_printf("\n");  sensor\_reading\_t \*readings = NULL;  readings = osal\_zalloc(sizeof(\*readings)); |

Sensor readings are read using the poll\_sensors() function and is printed on the console for every 2 seconds.

|  |
| --- |
| poll\_sensors(readings);  print\_sensor\_readings(readings, 1);  poll\_sensors(readings);  vTaskDelay (2000); |

### sensor.c

init\_sensors() contains all three sensors initialization part. All the sensors are initiated from here by the following functions:

BMP388 (Pressure sensor) - bmp388\_init() initiates the pressure sensor.

|  |
| --- |
| bmp388\_init(&pres\_sen,&dev,bus,0x76); |

Opt3002 (Light sensor) - opt3002\_init() initiates the light sensor.

|  |
| --- |
| opt3002\_init(&opt\_sen, bus, 0x44); |

shtc3 (Temperature/Humidity sensor) - sensirion\_i2c\_init() initializes the temp/hum sensor.

|  |
| --- |
| sensirion\_i2c\_init(bus);  shtc1\_probe();  shtc1\_enable\_low\_power\_mode(1);  #else  sensirion\_i2c\_init(bus);  shtc1\_probe();  sensirion\_i2c\_release();  #endif |

In the humidity sensor also there is a need to implement the mode of operation. shtc1\_probe() enables or disables sleep in the driver based on product code and will put the device in sleep mode if supported.

The get\_sensor\_ids() function reads the sensor IDs from each of the sensor.

BMP388 (Pressure sensor) - bmp3\_get\_device\_ID () API reads the device ID of bmp388 pressure sensor. The mode is set using set\_normal\_mode().

|  |
| --- |
| ids->bmp388\_id = bmp3\_get\_device\_ID(&dev);  set\_normal\_mode(&dev); |

opt3002 (Light sensor) - opt3002\_readManufacturerID() reads the manufacturing ID of light sensor.

|  |
| --- |
| ids->opt3002\_id = opt3002\_readManufacturerID(&opt\_sen); |

shtc3 (Temperature/Humidity sensor) - shtc1\_read\_serial() API reads the sensor ID of shtc3 sensor.

|  |
| --- |
| ids->shtc3\_serial = 0;  shtc1\_read\_serial(&ids->shtc3\_serial); |

Poll\_sensor() function reads the sensor readings of all three sensors.

The get\_sensor\_data()reads the sensor data. The temperature and pressure value of sensors are assigned to temp\_bmp and pressure variables of this structure sensor\_reading\_t readings.

|  |
| --- |
| reading->pressure = 0;  reading->temp\_bmp = 0;  /\* Read pressure and temperature recorded by bmp388 \*/  float \*sensor\_data;  sensor\_data = get\_sensor\_data(&dev);  reading->temp\_bmp = (sensor\_data[0]/100);  reading->pressure = (sensor\_data[1]/100); |

opt\_config\_trigger assigns the sensor mode, conversion time and latch operation. The opt3002\_config\_t opt\_config\_read() function reads the raw data. The Memset() function stores the light sensor data in a memory.

|  |
| --- |
| opt3002\_config\_t opt\_config\_trigger = {  .RangeNumber = 0xC, // Automatic full-scale mode  .ConversionTime = 0, // 100 ms conversion time  .ModeOfConversionOperation = 0x1, // Single-shot mode  .Latch = 0x1 // Latched operation  };  opt3002\_config\_t opt\_config\_read = {.rawData = 0};  memset(&reading->light, 0, sizeof(reading->light));  opt3002\_writeConfig(&opt\_sen, opt\_config\_trigger);  sensor\_delay(100);  do  { opt\_config\_read = opt3002\_readConfig(&opt\_sen);  }  while(!opt\_config\_read.ConversionReady);  reading->light = opt3002\_readResult(&opt\_sen); |

opt3002\_write\_config() triggers the reading of sensor data. The sensor reading is assigned to the light member.

The function initiates the humidity and temperature of shtc3 sensor and shtc1\_measure\_blocking\_read reads the sensor temperature and humidity readings. The sensor readings are assigned to the humidity and temp\_shtc members.

|  |
| --- |
| int32\_t humidity\_x1000 = 0, temp\_shtc\_x1000 = 0;  shtc1\_measure\_blocking\_read(&temp\_shtc\_x1000, &humidity\_x1000);  reading->humidity = humidity\_x1000 / 1000.0;  reading->temp\_shtc = temp\_shtc\_x1000 / 1000.0; |

Also, the printing functions are here to print the readings of the sensors to the console.

### sensor\_delay.c

The sensor\_delay.c file contains the routines to generate the delay in milliseconds.

### bmp388.c (Pressure sensor)

**Configuring the sensor**

To configure the pressure sensor, select the power mode and sensor setting. In addition to this, output data rate and oversampling settings for pressure and temperature are selected using the following function:

**Note**: Here, BMP3\_NO\_OVERSAMPLING is selected, and the mode of operation is chosen to be normal.

|  |
| --- |
| int8\_t set\_normal\_mode(struct bmp3\_dev \*dev) |

Enable the pressure and temperature sensor:

|  |
| --- |
| dev->settings.press\_en = BMP3\_ENABLE;  dev->settings.temp\_en = BMP3\_ENABLE; |

Select the output data rate and oversampling settings for pressure and temperature:

|  |
| --- |
| dev->settings.odr\_filter.press\_os = BMP3\_NO\_OVERSAMPLING;  dev->settings.odr\_filter.temp\_os = BMP3\_NO\_OVERSAMPLING;  dev->settings.odr\_filter.odr = BMP3\_ODR\_200\_HZ; |

Set the power mode to normal:

|  |
| --- |
| ev->settings.op\_mode = BMP3\_NORMAL\_MODE;  rslt = bmp3\_set\_op\_mode(dev); |

**Initialize the sensor**

To initiate the sensor, select the digital interface as I2C and instance is created of the structure bpm388 by bmp388\_init(). Read and write instances are also created inside this function.

|  |
| --- |
| bmp388->dev = i2c\_create\_device(bus, address, I2C\_CLK\_400K);  dev->dev\_id = bmp388->dev;  dev->intf = BMP3\_I2C\_INTF;  dev->read = bmp3\_read\_data;  dev->write = bmp3\_write\_data;  dev->delay\_ms = sensor\_delay;  bmp3\_init(dev); |

**Reading sensor data**

To read the sensor data, create a readData()function. This defines the length, flag, and data fields of the sensor. This function read sensor data and store it in a buffer.

|  |
| --- |
| uint8\_t buf[1];  int ret = 0;  uint16\_t length = 0;  while(length < len){  if((ret = read\_reg(dev\_id, buf, 1))){  os\_printf("I2C read error");  return ret;  }  data[length] = \*buf;  length++;  }  return ret; |

This executes read transaction on the I2C. The function reads I2C data and stores it in buffer. It reads a given number of bytes. If the device does not acknowledge the read command, an error will be returned. To read I2C data, initialize read\_reg()function. This permits reading of the I2C data and storing it in msg. This function will be reading the I2C data.

|  |
| --- |
| struct i2c\_msg msg;  int i2c\_result = 0;  if( !dev\_id){  os\_printf("no device\n");  return -ENODEV;  }  msg.im\_len = count;  msg.im\_flags = I2C\_M\_RD | I2C\_M\_STOP;  msg.im\_buf = data;  if ((i2c\_result = i2c\_transfer(dev\_id, &msg, 1))){  os\_printf("bmp388 i2c read error %d: %s\n", i2c\_result, strerror(-i2c\_result));  }  return i2c\_result; |

**Writing sensor data**

To write the sensor data, create writeData()instance. This defines the length, flag, and data fields of the sensor. This function writes the command data on a register.

|  |
| --- |
| uint8\_t command\_byte = command;  write\_reg( dev\_id,&command\_byte, 1);  return 0; |

This permits writing of I2C data in msg buffer. The write\_reg()function reads the I2C data and stores it in msg buffer. This executes write transaction on the I2C bus, which sends a given number of bytes. The bytes in the supplied buffer must be sent to the given address. If the slave device does not acknowledge any of the bytes, an error will be returned.

|  |
| --- |
| struct i2c\_msg msg;  int i2c\_result = 0;  if( !dev\_id){  os\_printf("no device\n");  return -ENODEV;  }  msg.im\_len = count;  msg.im\_flags = I2C\_M\_STOP;  msg.im\_buf = data;  if ((i2c\_result = i2c\_transfer(dev\_id, &msg, 1))){  os\_printf("bmp388 i2c write error in write reg %d: %s\n", i2c\_result, strerror(-i2c\_result));  }  return i2c\_result; |

### Opt3002.c (Optical sensor)

**Initializing the sensor**

To initialize the sensor, select the digital interface as I2C and create an instance of structure:

opt3002\_init() function enables the I2C device, clock signals with frequency of 400khz.

|  |
| --- |
| opt3002->dev = i2c\_create\_device(bus, address, I2C\_CLK\_400K). |

The function readManufacturerID()reads the manufacturing ID of the device. This reads the manufacturing ID. If sensor is detected, the opt3002\_write data exports the manufacturing ID.

|  |
| --- |
| uint16\_t result = 0;  int error = opt3002\_writeData(opt3002, MANUFACTURER\_ID);  if (!error)  error = opt3002\_readData(opt3002, &result);  return result; |

This enables the configuration of the read and write functions of optical sensor opt3002. The function pt3002\_readConfig()defines the configuration of reading.

|  |
| --- |
| opt3002\_config\_t config = {.rawData = 0};  int error = opt3002\_writeData(opt3002, CONFIG);  if (!error)  error = opt3002\_readData(opt3002, &config.rawData);  return config; |

The opt3001\_i2c\_write writes the configuration of opt3002 sensor.

|  |
| --- |
| uint8\_t buf[3] = {CONFIG, config.rawData >> 8, config.rawData & 0x00FF};  return opt3002\_i2c\_write(opt3002, buf, ARRAY\_SIZE(buf)); |

Post initiating, read and write instances are created to read sensor data stored in buffer and sent to the I2C bus. The following function reads data from opt3002 to the I2C bus. The function opt3002\_light\_t opt3002\_readRegister()reads data from sensor in a raw format and makes the required calculations by using formula:

|  |
| --- |
| (lux = (1.2)\*(powr(2, er.Exponent)\*er.Result)) |

The calculated data value will be stored lux variable.

|  |
| --- |
| int error = opt3002\_writeData(opt3002, command);  if (!error) {  opt3002\_light\_t result;  result.lux = 0;  result.raw.rawData = 0;  result.error = 0;  opt3002\_ER\_t er;  error = opt3002\_readData(opt3002, &er.rawData);  if (!error) {  result.raw = er;  if(!raw){  result.lux = (1.2)\*(powr(2, er.Exponent)\*er.Result);  }  }  else {  result.error = error;  }  return result;  }  else {  return opt3002\_returnError(error);} |

**Reading sensor data**

The opt3002\_i2c\_readData()function executes the read transaction on the I2C bus, reads data from the sensor through I2C and stores it in buffer. If the device does not acknowledge the read command, an error will be returned.

|  |
| --- |
| struct i2c\_msg msg;  int i2c\_result = 0;  if(!opt3002 || !opt3002->dev)  return -ENODEV;  msg.im\_len = count;  msg.im\_flags = I2C\_M\_RD | I2C\_M\_STOP;  msg.im\_buf = data;  if((i2c\_result = i2c\_transfer(opt3002->dev, &msg, 1)))  os\_printf("opt3002 i2c read error %d: %s\n", i2c\_result, strerror(-i2c\_result));  return i2c\_result; |

The opt3002\_readData() reads the sensor data and OPT3002 transmits data in Big-Endian format.

|  |
| --- |
| uint8\_t buf[2];  int ret = 0;  if((ret = opt3002\_i2c\_read(opt3002, buf, 2)))  return ret;  \*data = (buf[0] << 8) | buf[1];  return ret; |

**Writing sensor data**

The opt3002\_writeData() writes the command data to the I2C.

|  |
| --- |
| return opt3002\_i2c\_write(opt3002, &command\_byte, 1); |

The int opt3002\_i2c\_write() executes write transaction on the I2C bus and sends a given number of bytes. The bytes in the supplied buffer must be sent to the given address. If the slave device does not acknowledge any of the bytes, an error will be returned.

|  |
| --- |
| struct i2c\_msg msg;  int i2c\_result = 0;  if(!opt3002 || !opt3002->dev)  return -ENODEV;  msg.im\_len = count;  msg.im\_flags = I2C\_M\_STOP;  msg.im\_buf = data;  if((i2c\_result = i2c\_transfer(opt3002->dev, &msg, 1)))  os\_printf("opt3002 i2c write error %d: %s\n", i2c\_result, strerror(-i2c\_result));  return i2c\_result; |

### sensirion\_hw\_i2c\_implementation.c (Temperature/Humidity)

The sensirion\_hw\_i2c\_implementation.c contains the routines required to perform the I2C initialization, read and write operations of Sensirion temperature/humidity sensor.

The i2c\_create\_device() function creates the I2C device with the clock frequency of 400KHz. The sensirion\_i2c\_init()initializes all hardware and software components of the Sensirion for I2C.

|  |
| --- |
| dev = i2c\_create\_device(bus, SHTC1\_ADDRESS, I2C\_CLK\_400K); |

It executes one read transaction on the I2C bus through the function sensirion\_i2c\_read(), which reads a given number of bytes. If the device does not acknowledge the read command, an error will be returned.

|  |
| --- |
| struct i2c\_msg msg;  int i2c\_result = 0;  if(!dev)  return -ENODEV;  msg.im\_len = count;  msg.im\_flags = I2C\_M\_RD | I2C\_M\_STOP;  msg.im\_buf = data;  i2c\_set\_address(dev, address);  if((i2c\_result = i2c\_transfer(dev, &msg, 1)))  os\_printf("shtc3 i2c read error %d: %s\n", i2c\_result, strerror(-i2c\_result));  return i2c\_result; |

The sensirion\_i2c\_write()executes one write transaction on the I2C bus which sends a given number of bytes. The bytes in the supplied buffer must be sent to the given address. If the slave device does not acknowledge any of the bytes, an error will be returned.

|  |
| --- |
| struct i2c\_msg msg;  int i2c\_result = 0;  if(!dev)  return -ENODEV;  msg.im\_len = count;  msg.im\_flags = I2C\_M\_STOP;  msg.im\_buf = (uint8\_t\*)data; /\* Data pointed to won't be modified \*/  i2c\_set\_address(dev, address);  if((i2c\_result = i2c\_transfer(dev, &msg, 1)))  os\_printf("shtc3 i2c write error %d: %s\n", i2c\_result, strerror(-i2c\_result));  return i2c\_result; |

### shtc1.c (Temperature/Humidity)

The SHTC3 Humidity and Temperature Sensor from Sensirion is a highly accurate digital humidity and temperature sensor that communicates using the I2C protocol.

**Note**: SHTC1 compatible sensors: SHTW1, SHTW2, SHTC3.

**Configuring mode of operation**

The SHTC3 provides a low power measurement mode with a specific set of commands. Using the low power mode significantly shortens the measurement duration and thus minimizes the energy consumption per measurement. The following functions define the power mode of the shtc3. Low power mode is being implemented here.

To initiate the measurement, the following function is created:

|  |
| --- |
| shtc1\_measure(void) |

This is meant to awaken the sensor from sleep mode, begin measuring the sensor data and write the data through I2C.

Function shtc1\_measure() starts a measurement in high precision mode. Use shtc1\_read() to read out the values once the measurement is done. The duration of the measurement depends on the sensor in use. Refer datasheet for more details.

|  |
| --- |
| int16\_t ret;  return PM\_WAKE(ret,sensirion\_i2c\_write\_cmd(SHTC1\_ADDRESS, shtc1\_cmd\_measure)); |

Function shtc1\_probe()detects if a sensor is connected by reading out the ID register. If the sensor does not answer or if the answer is not the expected value, the function returns error. If the sensor is detected, 0 is returned.

|  |
| --- |
| uint16\_t id;  int16\_t ret;  supports\_sleep = 1;  sleep\_enabled = 1;  (void)shtc1\_wakeup();  ret= sensirion\_i2c\_delayed\_read\_cmd(SHTC1\_ADDRESS, SHTC1\_CMD\_READ\_ID\_REG,  SHTC1\_CMD\_DURATION\_USEC, &id, 1);  if (ret)  return ret;  if ((id & SHTC3\_PRODUCT\_CODE\_MASK) == SHTC3\_PRODUCT\_CODE)  return shtc1\_sleep();  if ((id & SHTC1\_PRODUCT\_CODE\_MASK) == SHTC1\_PRODUCT\_CODE) {  supports\_sleep = 0;  return STATUS\_OK;  }  return STATUS\_UNKNOWN\_DEVICE; |

Function shtc1\_measure\_blocking\_read() starts reading the sensor data. This function blocks while the measurement is in progress. Temperature is returned in [°C], multiplied by 1000 and relative humidity in [percent relative humidity], multiplied by 1000.

|  |
| --- |
| int16\_t ret;  PM\_WAKE(ret, shtc1\_measure());  #if !defined(USE\_SENSIRION\_CLOCK\_STRETCHING) || !USE\_SENSIRION\_CLOCK\_STRETCHING  sensirion\_sleep\_usec(SHTC1\_MEASUREMENT\_DURATION\_USEC);  #endif  /\* USE\_SENSIRION\_CLOCK\_STRETCHING \*/  ret = shtc1\_read(temperature, humidity);  return PM\_SLEEP(ret); |

**Reading the sensor data**

To read the sensor data, the function shtc1\_read() is used . It reads the sensor data and calculates temperature (Temperature = 175 \* S\_T / 2^16 – 45), humidity (Relative Humidity = 100 \* S\_RH / 2^16) using the formulae. It reads out the results of a measurement that was previously started by shtc1\_measure(). If the measurement is still in progress, this function returns an error. Temperature is returned in [°C], multiplied by 1000, and relative humidity [in percent relative humidity], multiplied by 1000.

|  |
| --- |
| uint16\_t words[2];  int16\_t ret = sensirion\_i2c\_read\_words(SHTC1\_ADDRESS, words,  SENSIRION\_NUM\_WORDS(words));  \*temperature = ((21875 \* (int32\_t)words[0]) >> 13) - 45000;  \*humidity = ((12500 \* (int32\_t)words[1]) >> 13);  return PM\_SLEEP(ret); |

The function shtc1\_disable\_sleep()enables or disables the SHT's sleep mode between measurements, if supported. Sleep mode is enabled by default if supported.

|  |
| --- |
| if (!supports\_sleep)  return STATUS\_FAIL;  sleep\_enabled = !disable\_sleep;  if (disable\_sleep)  return shtc1\_wakeup();  return shtc1\_sleep(); |

Enable or disable the SHT's low power mode.

|  |
| --- |
| shtc1\_cmd\_measure =enable\_low\_power\_mode ? SHTC1\_CMD\_MEASURE\_LPM : SHTC1\_CMD\_MEASURE\_HPM; |

The function shtc1\_read\_serial() is implemented to read out the serial number.

|  |
| --- |
| int16\_t shtc1\_read\_serial(uint32\_t \*serial) |

# Building

To build the sample application, execute the following commands:

|  |
| --- |
| cd examples/i2c  make |

The make command generates the i2c\_sensor.elf in the out directory.

# Running the Application

## Programming Talaria TWO

Program i2c\_sensor.elf *(freertos\_sdk\_x.y\examples\i2c\bin)* using the Download tool:

1. Launch the Download tool provided with InnoPhase Talaria TWO SDK.
2. In the GUI window:
   1. Boot Target: Select the appropriate EVK from the drop-down.
   2. ELF Input: Load the i2c\_sensor.elf by clicking on Select ELF File.
   3. Programming: Prog RAM or Prog Flash as per requirement.

## Expected Output

|  |
| --- |
| Y-BOOT 208ef13 2019-07-22 12:26:54 -0500 790da1-b-7  ROM yoda-h0-rom-16-0-gd5a8e586  FLASH:PWWWAE  Build $Id: git-b3777d5 $  Flash detected. flash.hw.uuid: 39483937-3207-0039-002c-ffffffffffff  Initializing bmp388...  Initializing opt3002...  Initializing shtc3...  bmp388 ID: 0x50  opt3002 ID: 0x5449  shtc3 ID: 0x2B5A0069  -----Timestamp: 30130 uS-----  Pressure: 91235.0 Pa  Temperature (bmp): 27.9200 C  Optical power: 28416.0 nW/cm2  Humidity: 71.7419 %  Temperature (shtc): 28.6650 C  -----Timestamp: 2286954 uS-----  Pressure: 91228.0625 Pa  Temperature (bmp): 28.0599 C  Optical power: 30528.0 nW/cm2  Humidity: 71.4260 %  Temperature (shtc): 28.6650 C |