

# Generic API User manual

## Generic API UserManual

Document revision 1.7

Document release date August 2020

Document number BST-DHW-SD016-00

Note Data and descriptions in this document are subject to change without notice. Product photos and pictures are for illustration purposes only and may differ from the real product appearance.

## 1. About this user manual

This manual describes the usage of GenericAPI for interfacing with sensors using Bosch Sensortec Application/Development boards.

### Who should read this manual

This information is intended for users who want to interface with sensors using Bosch Sensortec boards for their Analytical, Demonstrative and Engineering applications.

## Abbreviations

SID	Shuttle ID
HWID	Hardware ID
SWID	Software ID
Old DB/AB	Development Board/Application Board
APP2.0	Application Board

## 2. Generic API

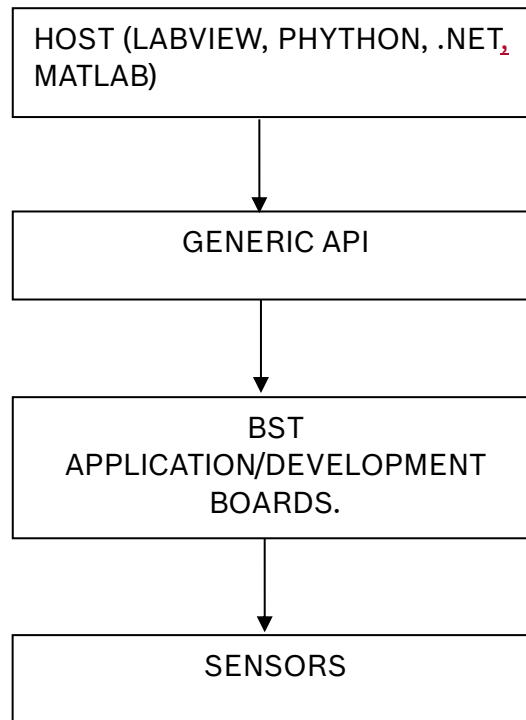
### 2.1 Overview

GenericAPI are group of API used for interfacing with sensor boards using the Bosch Sensortec Application/Development boards. It offers a flexible solution for developing a host independent Wrapper interface for the sensors with robust error handling mechanism.

### 2.2 Key Features

- Supports USB 3.0 interface.
- Supports RS 232/ Bluetooth interface.
- Single User Library Interface for all functionalities.
- Synchronous Programming Model.
- Robust error determining mechanisms.
- Enable Platform Independent Developments

## 2.3 Block Diagram of GenericAPI Application



## Table of Contents

<b>1. About this user manual</b>	<b>2</b>
Who should read this manual	2
<b>2. Generic API</b>	<b>2</b>
2.1 Overview	2
2.2 Key Features	2
2.3 Block Diagram of GenericAPI Application	3
<b>3. Installation</b>	<b>8</b>
3.1 System requirements	8
3.2 Installing the software	8
3.3 Usage of UserApplication DLL in Python	8
3.4 Usage of UserApplicationBoard dll in MATLAB	9
<b>4. Supported API</b>	<b>10</b>
4.1 API Function	10
4.2 Function Description & Examples	14
4.2.1 ClosePCInterface	14
4.2.2 GetBoardInfo	14
4.2.3 GetLED	15
4.2.4 GetPinConfig	15
4.2.5 LCDWrite	15
4.2.6 PCInterfaceConfig	16
4.2.7 PinConfig	16
4.2.8 Read	16
4.2.9 SensorI2CConfig	17
4.2.10 SensorSPICongfig	19
4.2.11 CustomSPICongfig	21
4.2.12 SetLED	21
4.2.13 SetVDD	22
4.2.14 SetVDDIO	22
4.2.15 Write (To configure a word)	23
4.2.16 Write (To configure a byte)	24
4.2.17 Streaming	26
InterruptStreamingSettings( 3 parameters)	26
4.2.18 TriggerStreaming (2 parameters)	27

- 4.2.19 **Time stamp** .....28
- 4.2.20 **ADCConfigure (1 parameter)** .....29
- 4.2.21 **ADCRead** .....30
- 4.2.22 **ReadSensorData** .....30
- 4.2.23 **ConfigSystemClock**.....31
  
- 5. DATA TRANSFER MECHANISM** ..... **32**
  
- 6. Error Codes**..... **32**
  - 6.1 **General Error Codes:** .....32
  - 6.2 **Pinconfig Specific Error Codes** .....33
  - 6.3 **LCD Specific Error Codes** .....33
  - 6.4 **Read/Write Specific Error Codes** .....33
  - 6.5 **Streaming specific Error codes** .....33
  - 6.6 **ADC/ConfigSystemClockSpecificErrorCode:**.....34
  
- 7. Enum Section** ..... **34**
  - 7.1 **EONOFF** .....34
  - 7.2 **EHIGHLOW** .....34
  - 7.3 **PINMODE** .....35
  - 7.4 **EINOUT** .....35
  - 7.5 **PINLEVEL** .....35
  - 7.6 **EBOARDTYPE**.....35
  - 7.7 **PCINTERFACE** .....35
  - 7.8 **I2CSPEED** .....35
  - 7.9 **SPISPEED**.....36
  - 7.10 **ADCCHANNELS**.....36
  - 7.11 **SYSTEMCLOCK**.....36
  
- 8. Property**..... **37**
  - 8.1 **TimeStamp** .....37
  
- 9. Structures** ..... **37**
  - 9.1 **BoardInformationDetails** .....37
  - 9.2 **PinConfigInfo** .....37

<b>10.</b>	<b>BNO I2C – SDO handling</b> .....	<b>37</b>
<b>11.</b>	<b>Legal disclaimer</b> .....	<b>37</b>
11.1	<b>Engineering samples</b> .....	37
11.2	<b>Product use</b> .....	38
11.3	<b>Application examples and hints</b> .....	38
<b>12.</b>	<b>Document history and modifications</b> .....	<b>38</b>

## List of figures

Figure 1: Running Gen.API in IronPython .....	9
Figure 2: UART/BLUETOOTH Communication .....	32

## List of tables

Table 1: Revision History .....	39
---------------------------------	----

## 3. Installation

The procedure describes the system requirements for using Generic API, prerequisites for using GenericAPI with an example to include the GenericAPI in the Application.

### 3.1 System requirements

- Operating system: Windows XP, Windows 7 and Windows 8.
- Both 32 bit and 64 bit operating systems are supported.
- Required software: Microsoft .NET Framework 4.0 or higher
- Memory: 1 GB
- Processor: 1 GHz or higher (Recommended)
- USB 2.0 host controllers

### 3.2 Installing the software

- Install the Development Desktop 2.0 Software.  
(Refer Development Desktop 2.0 User Manual.doc)

### 3.3 Usage of UserApplication DLL in Python

- Connect Application Board or Development Board or APP2.0 with the loaded firmware and switch ON.
- Open **cmd.exe** in Administrator mode ( windows-Start >> type *cmd* >> right-click and *Run as Administrator* )
- Change directory to the location where "IronPython" is installed. For Ex: C:\ProgramFiles(x86)\IronPython2.7 ( type *cd C:\*...* and enter )
- Start the iron python console ( type **ipy** and hit enter)
- Load the .NET library:
  - `import clr`
  - `import sys`
  - `sys.path.append(r" path of the folder which holds the dll")`
  - `clr.AddReference ("UserApplicationBoard.dll")`
- Import the library
  - `import BST`
  - `from BST import * # here the UserApplicationBoard is imported as well as the rest from BST`
- Create Instantiation from Class "UserApplicationBoard"
  - `myBoard = UserApplicationBoard()`
- Use some methods to start the communication, setup board voltage, pins, and read some values from sensor:
  - `myBoard.PCInterfaceConfig(PCINTERFACE.USB)`
  - `myBoard.SetVDD(1)`
  - `myBoard.SetVDDIO(1)`
  - `myBoard.SensorI2CConfig(0x18, I2CSPEED.STANDARDMODE)`
  - `myBoard.PinConfig(9, EONOFF.ON, PINMODE.OUTPUT, PINLEVEL.HIGH)`
  - `myBoard.Read(0x00, 10)`



```

C:\Windows\system32\cmd.exe - ipy.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\mis1cob>cd C:\Program Files (x86)\IronPython 2.7

C:\Program Files (x86)\IronPython 2.7>ipy.exe
IronPython 2.7.4 (2.7.0.40) on .NET 4.0.30319.34209 (32-bit)
Type "help", "copyright", "credits" or "license" for more information.
>>> import clr
>>> import sys
>>> sys.path.append(r"C:\Program Files\Bosch Sensortec\Development Desktop 2.0\U
serApplicationBoard")
>>> clr.AddReference("UserApplicationBoard.dll")
>>> import BST
>>> from BST import *
>>> myBoard = UserApplicationBoard()
>>> myBoard.PCInterfaceConfig(PCINTERFACE.USB)
>>> myBoard.SetUDD(1)
3.6
>>> myBoard.SetUDDIO(1)
3.6
>>> myBoard.SensorI2CConfig(0x18, I2CSPEED.STANDARDMODE)
>>> myBoard.PinConfig(9, EONOFF.ON, PINMODE.OUTPUT, PINLEVEL.HIGH)
>>> myBoard.Read(0x00, 10)
Array[int]<<250, 32, 193, 255, 209, 252, 209, 63, 1, 0>>
>>>

```

Figure 1: Running Gen.API in IronPython

### 3.4 Usage of UserApplicationBoard dll in MATLAB

```

%% Basic setup
% Generic API
dllPath = 'C:\Program Files\Boasc sensortec\Development Desktop2.0\
UserpApplicationBoard\UserApplicationBoard.dll';
assemblyInfo = NET.addAssembly(dllPath);
import BST.*
typBstProtoSet
=assemblyInfo.AssemblyHandle.GetType("BST.UserApplicationBoard");
myBoard = System.Activator.CreateInstance(typBstProtoSet);
myBoard.PCInterfaceConfig(PCINTERFACE.USB);
chipSelectPin=7; % select 7 for BMI160 and 8 for BMI162
boardDetails = myBoard.GetBoardInfo();
pininfo = myBoard.GetPinConfig(14);
StatusLED = myBoard.GetLed();
ValueLED=63; % Turn ON all LEDs
myBoard.SetLED(ValueLED);

```

**Note: Commands should be adapted to the matlab syntax**

## 4. Supported API

### 4.1 API Function

Commands	Arguments(s)
ClosePCInterface	<p><b>Description:</b></p> <ul style="list-style-type: none"> <li>Method to close the communication between PC and board.</li> </ul> <p><b>Return value:</b> None.</p>
GetBoardInfo	<p><b>Description:</b></p> <ul style="list-style-type: none"> <li>Method to get the Board information like SID, HWID, SWID Boardtype.</li> <li>No Arguments is needed for this method.</li> </ul> <p><b>Return value:</b> Structure which holds information of SID, HWID and board type.</p>
GetLed	<p><b>Description:</b></p> <ul style="list-style-type: none"> <li>Method to get the LED's status from the Boards</li> <li>No Arguments is needed for this method.</li> </ul> <p><b>Return value:</b> Returns a byte value of LED status.</p>
GetPinConfig	<p><b>a. With 2 Arguments:</b> pinNumber: Pin number for which the configured information need to be obtained.</p> <p><b>Description:</b></p> <ul style="list-style-type: none"> <li>Method to get the Pin configuration information.</li> </ul> <p><b>Return value:</b> Structure which holds the information like Pinstate,Switchstate and Direction.</p>
LCDWrite	<p><b>a. With 2 Arguments:</b> Row: Row in the LCD to write. Column: Column in the LCD to write.</p> <p><b>Description:</b></p> <ul style="list-style-type: none"> <li>Method to write the string to LCD display in the Development board.</li> <li>LCD display has three rows and sixteen columns each row and column represents one character.</li> </ul> <p><b>Return value:</b> None.</p>
PCInterfaceConfig	<p><b>a. With Arguments:</b> Communication channel: USB or UART. portName : COM99, COM1....</p> <p><b>b. No Argument :</b> Default Interface of USB is selected.</p> <p><b>Description:</b></p> <ul style="list-style-type: none"> <li>This method will set the Interface to the PC side.</li> <li>The argument is optional.</li> <li>The "pc_interface" set the interface to USB or to Bluetooth for Bluetooth port name should be given as argument otherwise errocode(0x8F) will return.</li> </ul> <p><b>Return value:</b> None.</p>
PinConfig	<p><b>a. With Arguments:</b> pinNumber: MultiIO Pin to be configured. switchState : Analog Switch state(ON or OFF)</p>

	<p>direction : Input or Output outputState : HIGH or LOW</p> <p><b>Description:</b></p> <ul style="list-style-type: none"> <li>Configures the MultiIO pins.</li> </ul> <p><b>Return value:</b> None.</p>
Read	<p><b>a. With 1 Argument:</b> registerAddress: Register address to read from the sensor.</p> <p><b>b. With 2 Arguments:</b> registerAddress: Register address to read from the sensor. numberOfRead : Number of reads to be done from sensor.</p> <p><b>c. With 3 Arguments:</b> registerAddress: Register address to read from the sensor. numberOfRead : Number of reads to be done from sensor. sensorInterfaceDetail: I2C address or Pin No of SPI CS.</p> <p><b>Description:</b></p> <ul style="list-style-type: none"> <li>This method is a blocking read method.</li> <li>With 1 Argument call with <b>registerAddress</b>, a single read is performed from the specified register address.</li> <li>With 2 Argument, a burst read for the specified <b>numberOfRead</b> is done.</li> <li>With 3 Argument, a burst read is done for the specified <b>numberOfRead</b> with the <b>sensorInterfaceDetail</b>.</li> </ul> <p><b>Return value:</b> Array of read data.</p>
SensorI2CConfig	<p><b>a. With 2 Arguments:</b> i2c_addr: i2c address. speed: Set the clock speed(KHz) (STANDARDMODE or FASTMODE) In case of no parameter “speed”, default speed of FASTMODE will be set</p> <p><b>b. With 3 Argument:</b> sensorID: sensor Identifier i2c_addr: i2c address. Speed: Set the clock speed(KHz) (STANDARDMODE or FASTMODE)</p> <p><b>Description:</b></p> <ul style="list-style-type: none"> <li>This method will set the Interface to communicate with the sensor through I2C protocol.</li> <li>Argument Speed is Optional.</li> <li>In case of no parameter “speed”, default speed of FASTMODE will be set.</li> </ul> <p><b>Return value:</b> None.</p>
SensorSPICongig	<p><b>a. With 4 Arguments</b> chipSelectPin: Selects the chip select pin spiSpeed : SPI speed selected(SPI250KBIT, SPI300KBIT, SPI400KBIT, SPI500KBIT, SPI600KBIT, SPI750KBIT, SPI1000KBIT, SPI1200KBIT, SPI1250KBIT, SPI1500KBIT, SPI2000KBIT, SPI2500KBIT, SPI3000KBIT, SPI3750KBIT, SPI5000KBIT, SPI6000KBIT, SPI7500KBIT, SPI10000KBIT, SPI15000KBIT, SPI30000KBIT)</p> <p>spiMode: Mode used for SPI(MODE0 and MODE3) spiLegnth: denotes spiLength (8BIT, 16BIT and 32 BIT)</p> <p><b>b. With 5 Argument:</b> sensorID: sensor Identifier</p>

	<p>chipSelectPin: Selects the chip select pin  spiSpeed : SPI speed selected(SPI250KBIT, SPI300KBIT, SPI400KBIT, SPI500KBIT, SPI600KBIT, SPI750KBIT, SPI1000KBIT, SPI1200KBIT, SPI1250KBIT, SPI1500KBIT, SPI2000KBIT, SPI2500KBIT, SPI3000KBIT, SPI3750KBIT, SPI5000KBIT, SPI6000KBIT, SPI7500KBIT, SPI10000KBIT, SPI15000KBIT, SPI30000KBIT)</p> <p>spiMode: Mode used for SPI(MODE0 and MODE3)  spiLength: 8BIT, 16BIT and 32BIT</p> <p><b>Description:</b></p> <ul style="list-style-type: none"> <li>This method will set the Interface to communicate with the sensor through SPI protocol.</li> <li>The arguments "spiSpeed" and spiMode and spiLength are optional, default speed of SPI1000KBIT and MODE0 and 8BIT will be applied.</li> </ul> <p><b>Return value:</b> None.</p>
SetLEDS	<p><b>a. With Arguments:</b>  Value: Each bit in argument "value" controls the state(ON or OFF) of the LEDs in the board.</p> <p><b>Description:</b></p> <ul style="list-style-type: none"> <li>Method to control the state of the LEDs in the board.</li> </ul> <p>Return value: None.</p>
SetVDD	<p><b>a. With Arguments:</b>  Value: Voltage value to set.</p> <p><b>b. No Argument:</b>  Default voltage value is set (3.3v).</p> <p><b>Return value:</b> Returns the voltage set.</p>
SetVDDIO	<p><b>a. With Arguments:</b>  Value: Voltage value to set.</p> <p><b>b. No Argument:</b>  Default voltage value is set(3.3v).</p> <p><b>Return value:</b> Returns the voltage set.</p>
Write (for configuring a byte)	<p><b>a. With 2 Arguments:</b>  registerAddress: Register address where the value needs to be written.  registerValue: be tn.</p> <p><b>b. With 3 Arguments:</b>  registerAddress:.  registerValue: Value to write in the sensor register.  sensorInterfaceDetail: Pin No of SPI CS.</p> <p><b>Description:</b></p> <ul style="list-style-type: none"> <li>This method is a blocking write method.</li> </ul> <p><b>Return value:</b> None.</p>
Write (for configuring a Word)	<p><b>a. With 3 Arguments: (To configure a word)</b>  addressOfAWord: address of the word where the value needs to be written.  Value: value to be written in the word.  sensorInterfaceDetail = chipselect pin number.</p> <p><b>b. With 3 Arguments: (To configure more than one word)</b>  addressOfAWord:.  Values: values to be written in the word.  sensorInterfaceDetail: Pin No of SPI CS.</p> <p><b>Description:</b></p>

	<ul style="list-style-type: none"> <li>This method is a blocking write method.</li> </ul> <p><b>Return value:</b> None.</p>
PollingStreamingSettings	<p><b>a. With 3 Arguments:</b>                  sensorId: Identifier of a sensor whose register values need to be obtained.                  dataRate: Rate at which sensor data is read[in terms of Hz].                  params int[] blockOfRegisters :Integer array to hold the block of register details.</p> <p><b>Description:</b></p> <ul style="list-style-type: none"> <li>Sends a command to firmware to obtain sensor data through polling method.</li> </ul> <p><b>Return value:</b> None.                  Note: The block of registers array contains the register start address and the number of bytes to read.</p>
InterruptStreamingSettings	<p><b>a. With 3 Arguments:</b>                  sensor Id: Identifier of a sensor whose register values need to be obtained.                  Pin information: Holds information regarding the hardware pin used for interrupt streaming.                  params int[] blockOfRegisters :Integer array to hold the block of register details</p> <p><b>Description:</b></p> <ul style="list-style-type: none"> <li>Sends a command to firmware to obtain sensor data through interrupt method.</li> </ul> <p><b>Return value:</b> None.                  Note: The block of registers array contains the register start address and the number of bytes to read.</p>
InterruptStreamingSettings	<p><b>a. With 9 arguments</b>                  Start address1, number of registers,                  Start address2, number of registers,                  Start address3, number of registers,                  Start address4, number of registers,                  Start address5, number of registers,                  Mask1, mask2</p> <p><b>Description:</b> The main aim of this API is to read the FIFO memory block from different memory location.  <b>Return value:</b> None  <b>Note:</b> This is to enable the user to read data from BHY2 sensors.</p>
TriggerStreaming	<p><b>a. With 2 Arguments:</b>                  Streaming feature: Holds information whether Interrupt (or) polling type streaming is required.                  Number of samples: Holds the below information                  Start: 0xFF,                  Stop: 0x00,                  One Sample: 0x01                  Two Sample: 0x02</p> <p><b>Description:</b></p> <ul style="list-style-type: none"> <li>Sends a command to firmware to start/stop the streaming.</li> </ul> <p><b>Return value:</b> None.</p>
Configure Timer	<p><b>a. With 1 Arguments:</b>                  Timer configure: Enumerator holds the below information.</p>

	Stop: 0x00 Start: 0x01 Reset: 0x02 <b>Description:</b> <ul style="list-style-type: none"> <li>Sends a command to firmware to start/stop/reset the timer in firmware.</li> </ul> <b>Return value:</b> None.
Configure Timestamp	<b>a. With 1 Arguments:</b> TimeStamp: Enumerator holds the below information Enable: 0x00 Disable: 0x01 <b>Description:</b> <ul style="list-style-type: none"> <li>Sends a command to firmware to enable/disable the time stamp.</li> </ul> <b>Return value:</b> None.
ERRORCODE	Every function call will update the ERRORCODE value, for error codes please refer section <a href="#">6.Error codes</a>

## 4.2 Function Description & Examples

### 4.2.1 ClosePCInterface

Disposes the resources used by the USB/serial communication.

**Functional Call:**

ClosePCInterface()

- Output: void

**Example:**

```
UserApplicationBoard.ClosePCInterface()
```

**Return:** void

### 4.2.2 GetBoardInfo

Obtains the information like SID, HWID, SWID and Board type.

**Functional Call:**

GetBoardInfo()

- Output: BoardInformationDetails structure holds the information related to SID, HWID, SWID and Board type

**Example:**

```
UserApplicationBoard.BoardInformationDetails boardDetails =
    UserApplicationBoard.GetBoardInfo()
```

**Return:** BoardInformationDetails

boardDetails object holds the below information :

```
boardDetails.Boardtype = EBOARDTYPE.ApplicationBoardV2
boardDetails.HardwareId = 2.1; (hardware version Id)
boardDetails.ShuttleID = 0xB1; (Shuttle Id)
boardDetails.SoftwareID = 1.7; (Software Version Id)
```

For EBOARDTYPE enum please refer [Enum](#) section

Note: The Boardtype, ShuttleId and SoftwareId may change based on the type of board, sensor connected and the software version.

### 4.2.3 GetLED

Gets the state of the leds .

**Functional Call:**

GetLED()

- Output: data type is byte which holds the LED's status ON/OFF.
  - In APP2.0 "6" LED's are available
  - Returns the "6" LED's status

**Example:**

```
byte LEDSTATUS = UserApplicationBoard.GetLed()
```

**Return:** 0x00 (LED's in OFF state)

### 4.2.4 GetPinConfig

Obtains information regarding the Pin's state, level and direction.

**Functional Call:**

GetPinConfig(ushort pinNumber)

- Output: PinConfigInfo structure holds the information pinstate, pindirection and pin level.

**Example:**

```
UserApplicationBoard.PinConfigInfo pininfo =  
  
UserApplicationBoard.GetPinConfig(14)
```

**Return:**

pininfo.Direction = IN (Pin is in INPUT state)

pininfo.SwitchState = ON (Pin Switch State)

pininfo.Value = LOW (Pin Value) – This indicates the voltage level of pin.

### 4.2.5 LCDWrite

Writes the string to LCD display in the Development board. For Application board/APP2.0 boards an Error code (refer Section 5.3) will be returned.

**Functional Call:**

LCDWrite(byte row, byte column, string data)

- row(1-3): data type is byte. Holds the row number in LCD display and it has 3 rows and each row holds each character location
- column(1-16): data type is byte. Holds the column number in LCD display and it has 16 columns, holds each character location
- data: data type is string which holds the characters to display in the LCD display
- Output: void

**Example:**

```
UserApplicationBoard.LCDWrite(2, 4, "Name: " + "BST")
```

**Return:** void

#### 4.2.6 PCInterfaceConfig

Sets the communication interface between board and PC to USB or Serial.

##### Functional Call:

PCInterfaceConfig(PCINTERFACE communicationChannel, string portName)

- communicationChannel(Optional): data type is Enum PCINTERFACE holds USB/SERIAL
- portName(Optional): PCInterface selected to Serial then the portName should be given as argument otherwise error code(0x8F) will return
- Output: void

##### Example:

For USB communication interface

```
UserApplicationBoard.PCInterfaceConfig(PCINTERFACE.USB)
```

For Serial communication interface, comport number should be given

```
UserApplicationBoard.PCInterfaceConfig(PCINTERFACE.SERIAL, "COM83")
```

**Return:** void

**Note:** In case no argument is provided, the default communication will be USB

#### 4.2.7 PinConfig

Configures the pin's state, level and direction

##### Functional Call:

PinConfig(int pinNumber, EONOFF switchState, PINMODE direction, PINLEVEL outputState)

- pinNumber: data type is integer and represents the MultiIO pin number
- switchState: data type is Enum EONOFF holds the switch state ON/OFF
- direction: data type is Enum PINMODE holds the direction state INPUT/OUTPUT
- outputState: data type is Enum PINLEVEL holds the status LOW/HIGH
- Output: void.

##### Example:

```
UserApplicationBoard.PinConfig(9, EONOFF.ON, PINMODE.OUTPUT,
```

```
PINLEVEL.HIGH)
```

(Pin 9 is Switched ON and is configured as Output with Pinvalue as High)

**Return:** void

**Note:** Pin 7 state cannot be changed from the External input (after configuring the pin as input) in the AB/DB board due to HW limitations. There are no such hardware limitations in APP2.0 board.

#### 4.2.8 Read

##### Read(1 paramater)

Reads a particular register's value from the sensor.



**Functional Call:**

Read(int registerAddress)

- registerAddress: data type is int. Holds the register address whose value needs to be read.
- Output: data from the register.

**Example:**

```
int chipid = UserApplicationBoard.Read(0x00)
```

**Return:** 0x00(Reads Chip Id)

**Read(2 paramater)**

Reads the value of block of registers from the sensor.

**Functional Call:**

Read (int registerAddress, ushort numberOfReads)

- registerAddress: data type is int. Holds the register start address from where the data needs to be read.
- numberOfReads: data type is ushort[Unsigned 16 bit integer]. Holds the number of registers to be read from the start address
- Output: Integer array to hold the register values.

**Example:**

```
int []regarr = UserApplicationBoard.Read(0x00,10)
```

**Return:** regarr{0x00, 0x01,..... Up to size of 10} (Read Data)

Note: If the numberOfReads exceeds 2KB, an errorcode is updated. For more info Refer

[6.4 Read/Write specific error code](#)

**Read(3 parameter)**

Reads the value of block of registers from the sensor using the sensorInterfaceDetails and numberOfReads.

**Functional Call:**

Read (int registerAddress, ushort numberOfReads, int sensorInterfaceDetail);

- registerAddress: data type is int. Holds the register start address to be read
- numberOfReads: data type is ushort[Unsigned 16 bit integer]. Holds the number of registers to be read from the start address
- sensorInterfaceDetail: data type is integer holds either I2C device address or SPI CS pin number of sensor whose data is to be read
- Output: data from the registers in a buffer data type is Integer

**Example:**

SPI: SensorInterfacedetail = 0x05; (SPIChipSelectMultiIO3)

I2C: SensorInterfacedetail = 0x18; (Device address)

```
int[]data = UserApplicationBoard.Read(0x00,10,
                                     SensorInterfacedetail)
```

**Return:** data {0x00, 0x01... Up to size of 10} (Read Data)

**Note:** If the numberOfReads exceeds 2KB, an errorcode is updated.

For more info Refer [6.4 Read/Write specific error code](#)

**4.2.9 SensorI2CConfig**

Sets the Interface to I2C and sets the I2C speed.

**Functional Call:**

1. SensorI2CConfig (ushort i2cAddress, I2CSPEED speed)

- i2cAddress: data type is ushort[Unsigned 16 bit integer]. Holds the address of the device to communicate through I2C protocol
- speed(Optional) : data type is Enum I2CSPEED holds the STANDARDMODE, FASTMODE, HSMODE and HSMODE2 default will be FASTMODE
- Output: void

## 2. SensorI2CConfig(byte sensorId, ushort i2cAddress, I2CSPEED speed)

When this function is called, a dictionary is maintained internally at the wrapper side. This dictionary holds the sensorId and its corresponding i2c address. In polling and interrupt streaming APIs, the corresponding i2cAddress is obtained from this dictionary based on the sensorId.

- sensorId: Identifier for each sensor.
- i2cAddress: data type is ushort[Unsigned 16 bit integer]. Holds the address of the device to communicate through I2C protocol
- speed(Optional) : data type is Enum I2CSPEED holds the STANDARDMODE, FASTMODE, HSMODE and HSMODE2 default will be FASTMODE
- Output: void

### Example:

#### Scenario 1:

Deviceaddress = 0x76

`UserApplicationBoard.SensorI2CConfig(Deviceaddress, I2CSPEED.STANDARDMODE)`

`UserApplicationBoard.SensorI2CConfig (Deviceaddress)`

by default the I2CSPEED will be in Fastmode. In case the user wants to configure the speed, they can mention in this API.

(Deviceaddress 0x76 is configured for I2C communication interface)

#### Scenario 2:

SensorId = 1

Deviceaddress = 0x76

`UserApplicationBoard.SensorI2CConfig(sensorId,Deviceaddress, I2CSPEED.STANDARDMODE)`

`UserApplicationBoard.SensorI2CConfig (sensorId, Deviceaddress)`

by default the I2CSPEED will be in Fastmode. In case the user wants to configure the speed, they can mention in this API.

The sensorId and the device address mentioned here would be stored in a dictionary at the wrapper side.

PollingStreamingSettings(1, 1000, 2, 10) Please refer [4.2.15.2](#)

[PollingStreamingSettings 4.2.14 Streaming](#)

While configuring polling streaming settings, the sensorId mentioned in streaming API, would be used to obtain the corresponding i2cAddress from the dictionary. This i2c address is used while sending command to firmware.

**Scenario 3:** [This explains the error scenario]:

SensorId = 1

Deviceaddress = 0x76

`UserApplicationBoard.SensorI2CConfig(sensorId,Deviceaddress, I2CSPEED.STANDARDMODE)`

SensorId = 2

Deviceaddress = 0x77

```
UserApplicationBoard.SensorI2CConfig(sensorId, Deviceaddress,
I2CSPEED.STANDARDMODE)
```

```
PollingStreamingSettings(3, 1000, 2, 10)
```

In this case the sensorId 3 will not be present in the dictionary, since sensorId 1 and 2 only are present.

**Note:** For scenario3, If the sensor Id is not present in the dictionary below error would be thrown in case of IronPython.

**“KeyError: The given key is not present in the dictionary.”**

**Return:** void

#### 4.2.10 SensorSPIConfig

Set the Interface to SPI and sets the SPI speed and mode. Support only 60MHz clock.

**Functional Call:**

##### 1. SensorSPIConfig(int chipSelectPin, SPISPEED spiSpeed, SPIMODE spiMode, SPILENGTH spiLength)

- chipSelectPin: data type is int. Holds the CS pin number to communicate through SPI protocol
- spiSpeed (Optional): sets the clock speed used in KHz. Default speed is 200KHz ( Data type is Enum SPISPEED holds the SPI250KBIT, SPI300KBIT, SPI400KBIT, SPI500KBIT, SPI600KBIT, SPI750KBIT, SPI1000KBIT, SPI1200KBIT, SPI1250KBIT, SPI1500KBIT, SPI2000KBIT, SPI2500KBIT, SPI3000KBIT, SPI3750KBIT, SPI5000KBIT, SPI6000KBIT, SPI7500KBIT, SPI10000KBIT, SPI15000KBIT, and SPI30000KBIT)
- spiMode(Optional): Default mode is MODE0, Data type is Enum SPIMODE holds MODE0, MODE3
- spiLength(Optional): Default length is 8 Bit, Data type is Enum SPILENGTH holds 8BIT, 16BIT, 32BIT.

This spiLength parameter decides the functionality of the following read APIs.

For ex: If the length of SPI is 8 bit, the following Read APIs' functionality changes

- Read(int registerAddress) -> reads a Word(16bit)
- Read(int registerAddress, ushort numberOfRead) -> reads more than one Word
- Read(int registerAddress, ushort numberOfRead, int sensorInterfaceDetail) -> reads more than one Word
- Output: void

##### 2. SensorSPIConfig(byte sensorId, int chipselectpin, SPISPEED spispeed, SPIMODE spimode, SPILENGTH spiLength)

When this function is called, a dictionary is maintained internally at the wrapper side. This dictionary holds the sensorId and its corresponding chipselectpin. In polling and interrupt streaming APIs, the corresponding chipselectpin is obtained from this dictionary based on the sensorId.

- sensorId: Identifier for each sensor
- chipSelectPin: data type is int. Holds the CS pin number to communicate through SPI protocol
- spiSpeed (Optional): sets the clock speed used in KHz. Default speed is 200KHz ( Data type is Enum SPISPEED holds the SPI250KBIT, SPI300KBIT, SPI400KBIT, SPI500KBIT, SPI600KBIT, SPI750KBIT, SPI1000KBIT, SPI1200KBIT, SPI1250KBIT, SPI1500KBIT, SPI2000KBIT, SPI2500KBIT, SPI3000KBIT, SPI3750KBIT, SPI5000KBIT, SPI6000KBIT, SPI7500KBIT, SPI10000KBIT, SPI15000KBIT, and SPI30000KBIT)
- spiMode(Optional): Default mode is MODE0, Data type is Enum SPIMODE holds MODE0, MODE3
- spiLength(Optional): Default length is 8 Bit, Data type is Enum SPILENGTH holds 8BIT, 16BIT, 32BIT.

This spiLength parameter decides the functionality of the following read APIs.

For ex: If the length of SPI is 8 bit, the following Read APIs' functionality changes

- Read(int registerAddress) -> reads a Word(16bit)
- Read(int registerAddress, ushort numberOfRead) -> reads more than one Word
- Read(int registerAddress, ushort numberOfRead, int sensorInterfaceDetail) -> reads more than one Word
- Output: void

### Example:

Scenario 1:

Chipselectpin = 8;

`UserApplicationBoard.SensorSPIConfig(Chipselectpin, SPI SPEED.SPI1000KBIT, SPI MODE.MODE3, SPI LENGTH.16BIT)`

`UserApplicationBoard.SensorSPIConfig(Chipselectpin)`

(Configures the Chipselect pin 8 for SPI communication interface)

by default the SPI SPEED will be 1000Kbit and SPI MODE will be MODE0. In case the user wants to configure the speed and mode, they can mention in this API.

Scenario 2:

sensorId = 1

Chipselectpin = 8;

`UserApplicationBoard.SensorSPIConfig(sensorId,Chipselectpin, SPI SPEED.SPI1000KBIT, SPI MODE.MODE3, SPI LENGTH.8BIT)`

`UserApplicationBoard.SensorSPIConfig(sensorId,Chipselectpin)`

(Configures the Chipselect pin 8 for SPI communication interface)

by default the SPI SPEED will be 1000Kbit and SPI MODE will be MODE0. In case the user wants to configure the speed and mode, they can mention in this API.

The sensorId and the chipselectpin mentioned here would be stored in a dictionary at the wrapper side.

`PollingStreamingSettings(1, 1000, 2, 10)`. Please refer [4.2.15.2 PollingStreamingSettings](#)

While configuring polling streaming settings/Interrupt streaming settings, the sensorId mentioned in streaming API, would be used to obtain the corresponding chipselectpin from the dictionary. This chipselectpin is used while sending command to firmware.

Scenario 3:

sensorId = 1

Chipselectpin = 8;

`UserApplicationBoard.SensorSPIConfig(sensorId,Chipselectpin, SPI SPEED.SPI1000KBIT, SPI MODE.MODE3)`

`UserApplicationBoard.SensorSPIConfig(sensorId,Chipselectpin)`

(Configures the Chipselect pin 8 for SPI communication interface)

sensorId = 2

Chipselectpin = 14;

`UserApplicationBoard.SensorSPIConfig(sensorId,Chipselectpin, SPI SPEED.SPI1000KBIT, SPI MODE.MODE3)`

`UserApplicationBoard.SensorSPIConfig(sensorId,Chipselectpin)`

(Configures the Chipselect pin 8 for SPI communication interface)

`PollingStreamingSettings(3, 1000, 2, 10)` – In this case the sensorId 3 will not be present in the dictionary, since sensorId 1 and 2 are only present. Please refer

**Note:** For scenario3, If the sensor Id is not present in the dictionary below error would be thrown in case of IronPython.

**“KeyError: The given key is not present in the dictionary.”**

**Return:** void

#### 4.2.11 CustomSPIConfig

Configures the SPI and set the speed and mode. Supports both 60MHz and 100 MHz clock.

**FunctionalCall:**

CustomSPIConfig(byte sensorID, int chipSelectPin, int speed, SPIMODE spiMode =SPIMODE.MODE0)

- sensorId: Identifier for each sensor.
- chipSelectPin: data type is 32 bit signed int. Holds the CS pin number to communicate through SPI protocol.
- Speed: Clock speed in terms of Hz.
- Mode: Default mode is MODE0, Data type is Enum SPIMODE holds MODE0, MODE3.
- In this API, user is allowed to configure the SPI speed. All SPI speed values which is possible with the supported core clock is implemented.
- Our SPI frequency is supported by the following relationship:
  - SCBR: Serial Clock Bit Rate = Core Clock/ SPI Clock.
  - SCBR values range from 1 to 255 and can have positive integer value only.
- Example Scenario (SPI speed not possible):

UserApplicationBoard.CustomSPIConfig(1, 7, 14000000, SPIMODE.MODE0);

Consider the following:

Core Clock – 60 MHz (Default core clock supported and core clock can be changed using Change Clock API)

SPI Clock – 14 MHz

SCBR = Peripheral Clock (Core Clock)/SPI Clock.

SCBR = 60 MHz /14 MHz = 4.28

In this case, SCBR value will be round off to 5 and SPI clock speed of 12 MHz will be set for SPI interface.

- Example Scenario (SPI speed possible):

UserApplicationBoard.CustomSPIConfig(1, 7, 15000000, SPIMODE.MODE0);

Consider the following:

Core Clock – 60 MHz (Default core clock supported)

SPI Clock – 15 MHz

SCBR = Peripheral Clock (Core Clock)/SPI Clock.

SCBR = 60 MHz /15 MHz = 4

In this case, SCBR value be 4 and SPI clock speed of 15 MHz will be set for SPI interface.

**Return:** void

#### 4.2.12 SetLED

Control the state of the leds in the boards.

**Functional Call:**

SetLED(byte value)

- Value: data type is byte. Holds the LED’s Status and maximum number of combinations can be  $2^6$  to change the state of LED to ON/OFF
  - Old Application and Development boards has four LEDs

- Maximum Input would be 0x0F, if input exceeds, the exceeded bits will be considered as don't care bits.
- New Application board has six LED's
  - Maximum Input would be 0x3F, if input exceeds, the exceeded bits will be considered as don't care bits.
- Output: void

**Example:**

```
UserApplicationBoard.SetLED(0x3f)
(Sets all the LED's status to ON)
```

**Return:** void**4.2.13 SetVDD**

Set the VDD.

**Functional Call:**

SetVDD(double value) and the default call will be SetVDD(double value = 3.3)

- value: data type is double and holds the value of VDD
- OFF : value = 0v
- ON:
  - APP2.0:
    - value > 0v, voltage(VDD) value will always be set to 3.6v
  - AB BOARD:
    - Value <1.2v, voltage(VDD) value will be set to 1.2v
    - 1.2v<value<3.3v, voltage (VDD) value will be set to value.
    - Value>3.3v, voltage(VDD) value will be set to 3.3v
  - DB BOARD:
    - Value <0.8v, voltage (VDD) value will be set to 0.8v.
    - Value > 3.6v, voltage(VDD) value will be set to 3.6v
    - 0.8v<value<3.6v, voltage (VDD) value will be set to value.
- Output: Returns the assigned voltages in volts

**Example:**

```
double vdd = UserApplicationBoard.SetVDD(3.3)
double vdd = UserApplicationBoard.SetVDD()
```

**Return:**

- APP2.0:       3.6v  
                  3.6v
- AB       :       3.3v  
                  3.3v
- DB       :       3.3v  
                  3.3v

**4.2.14 SetVDDIO**

Set the VDDIO.

**Functional Call:**

SetVDDIO(double value) and the default call will be SetVDDIO(double value = 3.3)

- value: data type is double and holds the value of VDDIO
- OFF : value = 0v
- ON:
  - APP2.0:
    - value > 0v, voltage(VDDIO) value will always be set to 3.6v
  - AB BOARD:
    - Value <1.2v, voltage(VDDIO) value will be set to 1.2v
    - 1.2v<value<3.3v, voltage (VDDIO) value will be set to value.
    - Value>3.3v, voltage(VDDIO) value will be set to 3.3v
  - DB BOARD:
    - Value <0.8v, voltage (VDDIO) value will be set to 0.8v.
    - Value > 3.6v, voltage(VDDIO) value will be set to 3.6v
    - 0.8v<value<3.6v, voltage (VDDIO) value will be set to value.
- Output: Returns the assigned voltages in volts

**Example:**

```
double _vddio = UserApplicationBoard.SetVDDIO(3.3)
double _vddio = UserApplicationBoard.SetVDDIO()
```

**Return:**

```
APP2.0: 3.6V, 3.6V
AB: 3.3V, 3.3V
DB: 3.3V, 3.3V
```

**4.2.15 Write (To configure a word)****Write (3 parameters)**

Writes a data into a particular word

**Functional call:**

Write (**ushort** address, **ushort** value, **int** sensorInterfaceDetail = 0) – This function takes care of writing a value to a Word.

- address: indicates the address of a word
- value: datatype is 16 bit signed integer
- sensorInterfaceDetail = SPI chip select pin number.

**Example:**

```
Address = 0x0001
Data = 0x0000
sensorInterfaceDetail = 7
UserApplicationBoard.Write(Address, Data, sensorInterfacedetail)
```

**Write (3 parameters)**

Writes data into a bulk of word)

**Functional call:**

Write (**ushort** address, **ushort[]** value, **int** sensorInterfaceDetail = 0) – This function takes care of writing a value to a Word.

- address: indicates the address of a word
- value: datatype is 16 bit signed integer. This is an array to hold 16bit signed integer values
- sensorInterfaceDetail = SPI chip select pin number.

**Example:**

```
Address = 0x0001
```

```
Data = [0x0000, 0x0001, 0x0002, 0x0003...]
sensorInterfaceDetail = 7
UserApplicationBoard.Write(Address, Data, sensorInterfacedetail)
```

**In IronPython:**

```
Data =[0x0020,0x0021,0x0022,0x0023....]. This needs to be cast to “Array” type
before being passed as an argument.
from System import Array, Byte
UserApplicationBoard.Write(Address, Array[Byte](Data), sensorInterfacedetail)
```

**4.2.16 Write (To configure a byte)****4.2.16.1 Write (2 parameter)**

Writes a data to a particular register.

**Functional Call:**

Write (int registerAddress, int registerValue) – This function supports write operation in normal mode.

- registerAddress: data type is int. Holds the register address where the data needs to be written.
- registerValue: data type is int. Holds the register value to be written in the address provided
- Output: void

**Example:**

```
Register = 0x7e
Data = 0x11
UserApplicationBoard.Write(Register, Data)
```

**Burst Write**

Write (int registerAddress, byte[] registerValue) – This function supports write operation in burst mode.

- registerAddress: data type is int. Holds the register address where the data needs to be written.
- registerValue: data type is int. Byte array to hold the values that need to be written into the register map.
- Output: void

**Example**

```
Register = 0x00
Byte[] register value = The maximum allowed size of this array
is 2KB.
```

**In IronPython:**

```
Data =[0x20,0x21,0x22,0x23,0x24....]. This needs to be cast to “Array” type
before being passed as an argument.
from System import Array, Byte
UserApplicationBoard.Write(Register, Array[Byte](Data))
```

**Note:**

For APP2.0 the number of write allowed for burst operation is 2kB based on the RAM requirements. If the number of write exceeds 2kB, an error code is updated. Refer section [6.4 Read/Write specific error code](#)  
For old AB/DB, due to RAM size, the number of write for burst operation is 46 bytes.



If the number of write exceeds 46 bytes, an error code is updated. Refer [6.4 Read/Write specific error code](#)

**Return:** void

#### 4.2.16.2 Write (3 parameter)

Writes a data to a particular register using the Sensor Interface details.

##### Functional Call:

Write (int registerAddress, int registerValue, int sensorInterfaceDetail) – This function supports write operation in normal mode.

- registerAddress: data type is int. Holds the register start address to be read
- registerValue: data type is int. Holds the register value to be write in the address provided
- sensorInterfaceDetail: data type is int. Holds either I2C device address or SPI CS pin number of sensor.
- Output: void

##### Example:

SPI: SensorInterfacedetail = 0x05 (SPIChipSelectMultiIO3)

I2C: SensorInterfacedetail = 0x18 (Deviceaddress)

Register = 0x7e

Data = 0x11

`UserApplicationBoard.Write(Register, Data, SPIChipSelectMultiIO3)`

`UserApplicationBoard.Write(Register, Data, Deviceaddress)`

##### Burst Write

Write (int registerAddress, byte[] registerValue, int sensorInterfaceDetail) – This function supports write operation in burst mode.

- registerAddress: data type is int. Holds the register start address to be read
- registerValue: Byte array to hold the values that need to be written into the register map.
- sensorInterfaceDetail: data type is int. Holds either I2C device address or SPI CS pin number of sensor.
- Output: void

##### Example

SPI: SensorInterfacedetail = 0x05 (SPIChipSelectMultiIO3)

I2C: SensorInterfacedetail = 0x18 (Deviceaddress)

Register = 0x7e

Byte[] register value = The maximum allowed size of this array is 2KB.

##### In IronPython:

Data =[0x20,0x21,0x22,0x23,0x24...]. This needs to be cast to “Array” type before being passed as an argument.

from System import Array, Byte

`UserApplicationBoard.Write(Register, Array[Byte](Data),`

`SPIChipSelectMultiIO3)`

`UserApplicationBoard.Write(Register, Array[Byte](Data), Deviceaddress)`

##### Note:

For APP2.0 the number of write allowed for burst operation is 2kB based on the RAM requirements. If the number of write exceeds 2kB, an error code is updated. Refer section [6.4 Read/Write specific error code](#)

For old AB/DB, due to RAM size, the number of write for burst operation is 46 bytes.

If the number of write exceeds 46 bytes, an error code is updated. Refer [6.4 Read/Write specific error code](#)

**Return:** void

#### 4.2.17 Streaming

##### InterruptStreamingSettings( 3 parameters)

Sends commands to firmware to read sensor data through interrupt method

##### Functional Call:

InterruptStreamingSettings(byte sensorId, MULTIIO pinInformation,  
params int[] blockOfRegisters)

- sensorId: Identifier for each sensor
- pinInformation: holds the value of Multi IO pin  
(Data type is MULTIIO – Enumerator to hold the values for Multi IO pins  
MULTIIO\_0, MULTIIO\_1, MULTIIO\_2, MULTIIO\_3, MULTIIO\_4, MULTIIO\_5,  
MULTIIO\_6, MULTIIO\_7, MULTIIO\_8)
- blockOfRegisters: integer array to hold register start address and the number of bytes to read. .

##### Example:

```
sensorId = 1
pinInformation = BST.MULTIIO.MULTIIO_6
params int[] blockOfRegisters - 2, 11, 3, 10
2 = StartAddress.
11 = BytesToRead.
3 = StartAddress.
10 = BytesToRead.
```

[UserApplicationBoard](#).InterruptStreamingSettings(1, BST.MULTIIO.MULTIIO\_6, 2, 11, 3, 10);  
Here 2, 11 is one block of register. 3, 10 is another block of register.

**Return:** Void

**Note:**

- 1) It is possible to read more than 256 bytes through interrupt streaming.
- 2) At wrapper side a dictionary is maintained to hold the i2caddress or chipselectpin for the corresponding sensorId based on the sensor interface[I2C/SPI]. If this sensor Id is not present in the dictionary the below error would be thrown in case of IronPython. For more info, please refer [4.2.9 SensorI2CConfig](#) and [4.2.10 SensorSPICongig](#) and [4.2.11 CustomSPICongig](#)

**“KeyError – the given key is not present in the dictionary.”**

##### InterruptStreamingSettings (9 parameters)

Sends commands to firmware to read FIFO data through interrupt method, from different memory location.

**Note:** This is mostly used for BHY2 sensors

##### Functional Call:

```
InterruptStreamingSettings(byte chunk1StartAddress, ushort chunk1Registers,
byte chunk2StartAddress, ushort chunk2Registers, byte
chunk3StartAddress, ushort chunk3Registers,
byte chunk4StartAddress, ushort chunk4Registers, byte
chunk5StartAddress, ushort chunk5Registers, byte mask1, byte
mask2)
```

- ChunkStartAddress: Start address of the each chunk. This indicates the memory address of the FIFO block
- ChunkRegisters: Indicates the number of registers to be read from the FIFO.

**Example:**

```
UserApplicationBoard.InterruptStreamingSettings(0x2D, 1, 0x01, 2, 0x01, 0, 0x02, 2, 0x02, 0,
0x06, 0x18);
```

**Return:** Void

**PollingStreamingSettings(3 parameters)**

Sends commands to firmware to read sensor data through polling method

**Functional Call:**

```
PollingStreamingSettings(byte sensorId, dataRate,
params int[] blockOfRegisters)
```

- sensorId: Identifier for each sensor
- dataRate: rate at which sensor data is read [in terms of HZ]
- blockOfRegisters: integer array to hold the block of registers details array contains the register start address and bytes to read.

**Example:**

```
sensorId = 1
dataRate = 1000
params int[] blockOfRegisters - 2, 11, 3, 10
2 = StartAddress.
11 = BytesToRead.
3 = StartAddress.
10 = BytesToRead.
```

```
UserApplicationBoard.PollingStreamingSettings(1, 1000, 2, 11, 3, 10);
```

Here 2, 11 is one block of register. 3, 10 is another block of register.

**Return:** Void

Note:

- 1) Using polling streaming method, the maximum number of bytes that shall be read is 255 bytes [including the 8 bytes for time stamp].
- 2) The maximum data rate supported is 4000Hz.
- 3) At wrapper side a dictionary is maintained to hold the i2caddress or chipselectpin for the corresponding sensorId based on the sensor interface[I2C/SPI]. If this sensor Id is not present in the dictionary the below error would be thrown in case of IronPython. For more info, please refer [4.2.9 SensorI2CConfig](#) and [4.2.10 SensorSPICongig](#) and [4.2.11 CustomSPICongig](#)  
**“KeyError – the given key is not present in the dictionary.”**

**4.2.18 TriggerStreaming (2 parameters)**

Sends a command to firmware to start/stop the streaming

**Functional Call:**

```
TriggerStreaming(ESTREAMINGFEATURE streamingType, ESAMPLES
numberOfSamples)
```

- streamingType: holds the value indicating whether polling type or interrupt type streaming is required.  
(Data type is ESTREAMINGFEATURE: enumerator to hold the value for polling type or interrupt type or FIFO type).
- numberOfSamples: holds the value to indicate whether infinite samples are required or no samples are required[stop the streaming].  
(ESAMPLES. Enumerator to hold the below information  
STOP = 0x00,

```
INFINITE = 0xFF,
ONESAMPLE = 0x01,
TWO SAMPLES = 0x02).
```

**Example:**

```
streamingType = ESTREAMINGFEATURE.POLLINGSTREAMING
numberOfSamples = ESAMPLES.INFINITE
UserApplicationBoard.TriggerStreaming(ESTREAMINGFEATURE.POLLINGSTREAMING,
ESAMPLES.INFINITE);
```

**Return:** Void

Note: Error code is updated for the below scenarios

- 1) Polling streaming command is sent. But during trigger streaming, if the streaming feature is interrupt streaming, error code -15 is updated.
- 2) Interrupt streaming command is sent. But during trigger streaming, if the streaming feature is polling streaming, error code -16 is updated.  
Refer section [6.5 Streaming specific error codes](#).
- 3) Start streaming command is not sent for the following scenario.

**Scenario:**

```
UserApplicationBoard.TriggerStreaming(ESTREAMINGFEATURE.POLLINGSTREAMING,
ESAMPLES.INFINITE);
```

```
UserApplicationBoard.TriggerStreaming(ESTREAMINGFEATURE.POLLINGSTREAMING,
ESAMPLES.INFINITE);
```

In this case the second start streaming command will not be sent. But the data would be obtained.

4)

- a) An error code is updated if stop streaming commands are sent more than once.

```
UserApplicationBoard.TriggerStreaming(ESTREAMINGFEATURE.POLLINGSTREAMING,
ESAMPLES.STOP);
```

```
UserApplicationBoard.TriggerStreaming(ESTREAMINGFEATURE.POLLINGSTREAMING,
ESAMPLES.STOP);
```

Error code -17 is updated for this scenario. Refer section [6.5 Streaming specific error codes](#)

- b) when stop streaming command is sent without sending the start streaming command.

Error code -17 is updated for this scenario. Refer section [6.5 Streaming specific error codes](#)

No data would be obtained for the above two scenarios.

#### 4.2.19 Time stamp

**Configure Timer(1 parameter)**

Sends a command to firmware to start/stop/reset the timer in firmware.

**Function Call:**

```
ConfigureTimer(TIMERCONFIGURE timerConfig)
```

- timerConfig: holds a value for start/stop/reset the timer  
(TIMERCONFIGURE: Enumerator holds the below information  
STOP = 0, START = 1, RESET = 2)

**Example:**

```
timerConfig = TIMERCONFIGURE.START
UserApplicationBoard.ConfigureTimer(TIMERCONFIGURE.START);
```

**Return:** Void

**Configure TimeStamp(1 parameter)**

Sends a command to firmware to enable/disable the timestamp from firmware.

**Function Call:**

ConfigureTimeStamp(TIMESTAMP timeStampConfig)

- timeStampConfig: holds a value to enable/disable the timestamp  
(Data type is TIMESTAMP: Enumerator holds the below information  
ENABLE = 0, DISABLE = 1)

**Example:**

```
timeStampConfig = TIMESTAMP.ENABLE
UserApplicationBoard.ConfigureTimeStamp(TIMESTAMP.ENABLE);
```

**Return: Void**

Note:

- For Non-streaming API: The time stamp is updated in the property called "TimeStamp" refer section [8.1.TimeStamp](#). User shall obtain the time stamp from this property for non-streaming APIs.
- For streaming API: The time stamp is returned with the number of samples requested. For more information please refer [4.2.18 ReadSensorData](#) API.

**4.2.20 ADCConfigure (1 parameter)**

Configures the ADC pins of APP2.0 board

**ADC Pins Mapping in APP2.0**

PIN	SIGNAL	EXPANSION INTERFACE
PA19	AD2	22
PA20	AD3	23
PB0	AD4	5
PB1	AD5	6
PB2	AD6	1
PB3	AD7	2

**Function Call:**

ADCConfigure (int pinNumber)

Configures the ADC pins of APP2.0

- pinNumber[32 bit integer type]: indicates the ADC pin number to be configured and each bit of the pinNumber corresponds to ADC Channel as described in the following:
- Currently 9 ADC pins in APP2.0 can be configured.
  - CHANNEL\_A(AD2) - 0000 0000 0001
  - CHANNEL\_B(AD3) - 0000 0000 0010
  - CHANNEL\_C(AD4) - 0000 0000 0100
  - CHANNEL\_D(AD5) - 0000 0000 1000
  - CHANNEL\_E(AD6) - 0000 0001 0000
  - CHANNEL\_F(AD7) - 0000 0010 0000
  - CHANNEL\_AB(AD2 &AD3) - 0000 0100 0000
  - CHANNEL\_CD(AD4 &AD5) - 0000 1000 0000

- CHANNEL\_EF(AD6 &AD7) - 0001 0000 0000
- OFF - 0000 0000 0000
- Pins for ADC are available in the Expansion interface.

**Example:**

```
pinNumber = ADCCHANNELS.CHANNEL_B | ADCCHANNELS.CHANNEL_F
```

For more info refer [7.10 ADCCHANNELS](#)

```
UserApplicationBoard.ADCConfigure (pinNumber);
```

Configure the ADC channel B and channel F.

**Error Conditions:**

- Simultaneous enabling of ADC channel in both single ended and Differential Ended Mode.
- This ADCConfigure API is not supported for old Application and development Board. Error code -19 is updated for this scenario. Refer [6.6 ADC/SystemClockSpecificErrorCode](#)

**Example:**

```
pinNumber = CHANNEL_A | CHANNEL_AB.
```

```
ADCConfigure (pinNumber);
```

**Result of Error:**

Error code will be updated in [ERRORCODE](#) property

**4.2.21 ADCRead**

Read the ADC data from the channel.

**Function Call:**

```
ADCRead (ADCCHANNELS channel)
```

channel – Channel from which ADC data has to be read.

For more info refer [7.10 ADCCHANNELS](#)

**Example:**

```
UserApplicationBoard.ADCRead (ADCCHANNELS.CHANNEL_D);
```

Reads the ADC data from the channel D.

**Error Conditions:**

- Reading the ADC channel without configuring the ADC.
- This ADCConfigure API is not supported for old Application and development Board. Error code -19 is updated for this scenario.

Refer [6.6 ADC/SystemClockSpecificErrorCode](#)

**Example:**

```
pinNumber = ADCCHANNELS.CHANNEL_B | ADCCHANNELS.CHANNEL_F.
```

```
ADCConfigure (pinNumber);
```

```
ADCRead (ADCCHANNELS.CHANNEL_D);
```

**Result of Error:**

Error code will be updated in [ERRORCODE](#) property

**4.2.22 ReadSensorData****ReadSensorData(2 parameters)**

Reads the sensor data from buffer

**Function Call:**

```
ReadSensorData(sensorId, numberOfSamples)
```

- sensorId[32 bit integer]: identifier for each sensor.
- numberOfSamples[32 bit integer]: numberOfSamples needed .

**Example:**

```

sensorId = 1
numberOfSamples = 100
streamingData = UserApplicationBoard.ReadSensorData(1, 100);

```

**Scenario 1:**

The user enables time stamp through time stamp APIs. Refer [4.2.17 TimeStamp](#). If the user reads 6 bytes of data from sensor through streaming APIs[Polling/Interrupt] and requests 10 samples through ReadSensorData API, the number of bytes expected is 60[1 sample = 6 bytes. 10 samples = 60 bytes]. For every sample, 8 bytes of time stamp are received. These 8 bytes are converted to a timestamp value[int64 format]. For every sample, 1 time stamp value[int64 format] is stored in this structure.

**In python:**

streamingData = UserApplicationBoard.ReadSensorData(1,10). The ReadSensorData API returns a structure. Here streamingData is a structure which holds a byte array and a int64 array.  
streamingData.sensorData -> will return the data bytes[60 bytes]  
streamingData.timestamp -> will return the int64 array containing the time stamp [10 time stamp values – since 10 samples are requested by the user].

**Scenario 2:**

The user does not enable time stamp. The user reads 6 bytes of data from sensor through streaming APIs[Polling/Interrupt] and requests 10 samples through ReadSensorData API, the number of bytes expected is 60[1 sample = 6 bytes. 10 samples = 60 bytes]. Since the time stamp is not enabled, 8 bytes of time stamp are not received. Only the sensor data bytes are stored in the structure.

**In python:**

streamingData = UserApplicationBoard.ReadSensorData(1,10). The ReadSensorData API returns a structure. Here streamingData is a structure which holds a byte array and a int64 array.  
streamingData.sensorData -> will return the data bytes[60 bytes]  
streamingData.timestamp -> will return null, since the time stamp is not enabled by the user.

**Note:**

In PC side, a buffer of size 10MB is maintained to obtain the streaming data.

In firmware side, a USB buffer size of 4kB is maintained.

**Return:** streamingData = UserApplicationBoard.ReadSensorData(1,10).

streamingData.sensorData -> will return the data bytes[60 bytes]

streamingData.timestamp -> will return the int64 array containing the time stamp. [if the time stamp is enabled, else null is received]

### 4.2.23 ConfigSystemClock

Configure the System Clock of the microcontroller.

**Function Call:****ConfigSystemClock (SYSTEMCLOCK Clock)**

- **Clock:** data type is Enum [SYSTEMCLOCK](#) holds the MHZ60 and MHZ100.

**Example****UserApplicationBoard.ConfigSystemClock (SYSTEMCLOCK.100MHZ\_CLOCK)**

- Above example will set the core clock as 100MHZ.

**Return:** void.

**Note:**

For OldAB/DB, this API is not supported. If this API is used for Old AB/DB, “CommandNotSupported” error code is updated.

Refer section [6.6 ADC/ConfigSystemClock specific error Codes](#)

For 100MHZ clock, only CustomSPIConfig is supported. SensorSPIConfig API is not

Supported for 100MHz clock.

## 5. DATA TRANSFER MECHANISM

USB:

- Compliant with USB 2.0 full speed of Standard USB specification.
- Uses Bulk Mode of Transport.
- Embedded Dual-port RAM for Endpoints.
- PC buffer of 1MB is used.
- FW maintains 4KB ring buffer for data integrity.
- USB endpoint/channel buffer is 64 bytes.

UART/BLUETOOTH:

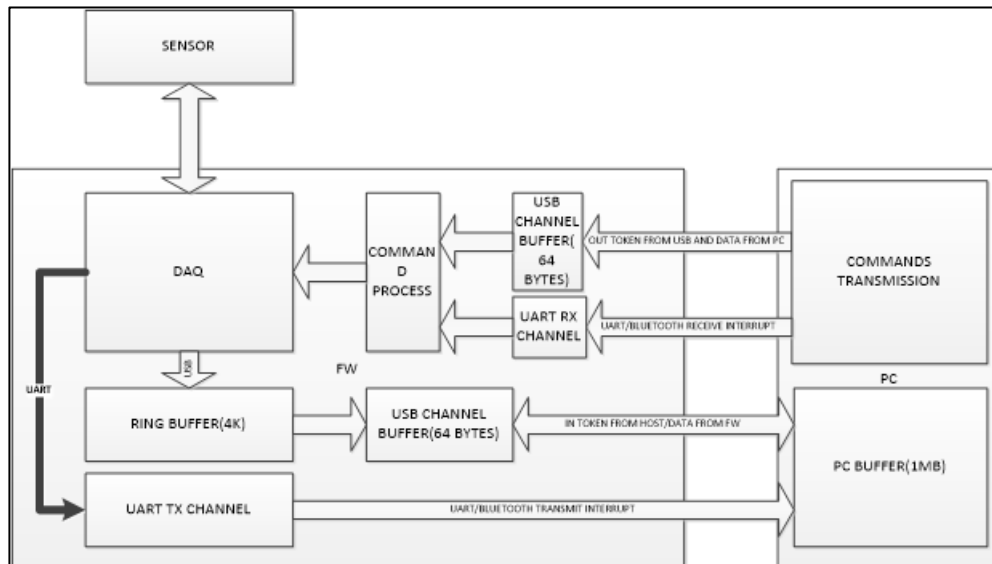


Figure 2: UART/BLUETOOTH Communication

- Support 115200bps data transfer rate.
- PC buffer of 1MB is used.
- Streaming API performance is assured for 250hz and 45 bytes.

## 6. Error Codes

Once the API call has been done user can call the `ERRORCODE` to check the error code return by the API.

### Example:

`long errorcode = UserApplicationBoard.ERRORCODE`

**Return:** 0x00(Success)

### 6.1 General Error Codes:

Error Values	Description
--------------	-------------



0	No Error response/Success
-1	Failure.
-2	Length Error.
-4	Configuration is Unsuccessful
-5	Invalid Instruction
-6	Memory Error.
-100	Timeout.

## 6.2 Pinconfig Specific Error Codes

Error Values	Description
1	Analog Switch is turned ON/OFF.
-10	Invalid Pin
-19	Invalid ADC Pin.

## 6.3 LCD Specific Error Codes

Error Values	Description
-9	NO LCD

## 6.4 Read/Write Specific Error Codes

Error Values	Description
2	Default read of 128 bytes is done. Requested bytes of read not supported. For APP2.0 Board read more than 128 bytes is possible and up to 1024 bytes
-3	The number of bytes that shall be read is 2KB. If this exceeds, error code is updated.
-18	1) For APP2.0 the maximum number of bytes that shall be written for burst operation is 2KB based on RAM requirements. 2) For AB/DB, due to RAM size the maximum number of bytes that shall be written for burst operation is 46 bytes.

## 6.5 Streaming specific Error codes

Error Values	Description
-11	Buffer Full. Indicates that the buffer containing the sensor streaming data is full. Buffer is an array maintained in the .NET wrapper side.
-12	Buffer empty. Indicates that the buffer containing the sensor streaming data is either fully read or new data is yet to be filled.
-13	Insufficient data in buffer.

	<p>Indicates that the number of samples requested by the user is partially available in the buffer.</p> <p>Ex: requested samples are 100. But only 50 samples are available in the buffer.</p>
-14	<p>Interrupt streaming is not supported.</p> <p>Indicates that interrupt streaming is not supported in old AB/DB. Interrupt streaming is supported only in APP2.0.</p>
-15	<p>Polling streaming is disabled.</p> <p>Ex: User sends polling based streaming commands but sends the interrupt streaming feature in trigger streaming API. So the polling based streaming will not be enabled. No data would be obtained in this scenario.</p>
-16	<p>Interrupt streaming is disabled.</p> <p>Ex: User sends Interrupt based streaming commands but sends the polling streaming feature in trigger streaming API. So the interrupt based streaming will not be enabled. No data would be obtained in this scenario.</p>
-17	<p>Streaming is not started.</p> <p>Ex:</p> <ul style="list-style-type: none"> <li>a) An error code is updated when the stop streaming command is sent more than once.</li> </ul> <p><b>Scenario:</b></p> <p>TriggerStreaming(ESTREAMINGFEATURE.POLLINGSTREAMING, ESAMPLES.STOP)</p> <p>TriggerStreaming(ESTREAMINGFEATURE.POLLINGSTREAMING, ESAMPLES.STOP). For the second stop streaming command an error code is updated.</p> <ul style="list-style-type: none"> <li>b) when stop streaming command is sent without sending the start streaming command, the same error code is updated for this scenario.</li> </ul>

## 6.6 ADC/ConfigSystemClockSpecificErrorCode:

Error Code	Description
-19	ADC/ConfigSystemClock command are not supported in AB/DB.

## 7. Enum Section

### 7.1 EONOFF

Enum holds value for ON and OFF

OFF = 0x00

ON = 0x01

### 7.2 EHIGLOW

Enum holds value for HIGH and LOW

LOW = 0x00  
HIGH = 0x01

### 7.3 PINMODE

Enum holds value for direction state of the pin sets to output or input

INPUT = 0x00  
OUTPUT = 0x01

### 7.4 EINOUT

Enum holds value for direction state of the pin sets to output or input

IN = 0x00  
OUT = 0x01

### 7.5 PINLEVEL

Enum holds the value for pin level status either high or low

LOW = 0x00  
HIGH = 0x01

### 7.6 EBOARDTYPE

Enum holds the board type

APPLICATIONBOARD = 1  
DEVELOPMENTBOARD = 2  
APPLICATIONBOARDV2 = 3  
BNOUSBSTICK = 4  
UNKNOWNBOARD = 5

### 7.7 PCINTERFACE

Enum to activate the communication channel

USB  
SERIAL

### 7.8 I2CSPEED

Enum holds the I2C Speed

I2C Standard mode 100 KHz SCL- frequencies  
STANDARADMODE = 0x00  
I2C Fast mode 400 KHz SCL- frequencies  
FASTMODE = 0x01  
I2C High-Speedmode ~3.4MHz SCL-frequency  
HSMODE = 0x02

I2C High-Speedmode ~1.7MHz SCL-frequency  
HSMODE2

## 7.9 SPI SPEED

Enum holds the SPI speed

SPI250KBIT  
SPI300KBIT  
SPI400KBIT  
SPI500KBIT  
SPI600KBIT  
SPI750KBIT  
SPI1000KBIT  
SPI1200KBIT  
SPI1250KBIT  
SPI1500KBIT  
SPI2000KBIT  
SPI2500KBIT  
SPI3000KBIT  
SPI3750KBIT  
SPI5000KBIT  
SPI6000KBIT  
SPI7500KBIT  
SPI10000KBIT  
SPI15000KBIT  
SPI30000KBIT

## 7.10 ADC CHANNELS

Enum holds the ADC channel.

OFF  
CHANNEL\_A  
CHANNEL\_B  
CHANNEL\_C  
CHANNEL\_D  
CHANNEL\_E  
CHANNEL\_F  
CHANNEL\_AB  
CHANNEL\_CD  
CHANNEL\_EF

## 7.11 SYSTEMCLOCK

Enum holds the core clock values.

MHZ60 – 60 MHz core clock in the microcontroller.

MHZ100 – 100 MHz core clock in the microcontroller

## 8. Property

### 8.1 TimeStamp

Once a non-streaming API is called, the TimeStamp property can be used to retrieve to the time stamp value[in terms of nano seconds] from the controller. This time stamp value indicates the time at which the command is executed at firmware side.

#### Example:

##### Non-streaming APIs:

```
Int64 timestamp = UserApplicationBoard.TimeStamp.
```

timestamp is in terms of nanoseconds.

timestamp = timestamp /10<sup>9</sup> will give you the timestamp in terms of nanoseconds.

**Return:** time stamp value in terms of nano seconds.

## 9. Structures

### 9.1 BoardInformationDetails

Structure holds the Board information details like SID, HWID, SWID and Board type

Boardtype: variable holds the Board type it is enum refer [EBOARDTYPE](#)

HardwareId: holds the hardware ID information

ShuttleID: holds the Shuttle ID information

SoftwareId: variable holds the software id information

### 9.2 PinConfigInfo

Structure holds the Pin configuration details like Pin SwitchState, value and Direction

Direction: holds the Direction state of the pin from the hardware and the Direction type is enum refer [EINOUT](#)

SwitchState: holds the switch state of the pin from the Hardware and the SwtichState type is enum refer [EONOFF](#)

Value: holds the value of the pin from the Hardware and the Value type is enum refer [EHIGHLOW](#)

## 10. BNO I2C – SDO handling

APP2.0 Board:

Switch OFF the Analog Switch of SDO pin which internally make the SDO pin as input.

OLD AB/DB Board:

Switch OFF the Analog Switch of SDO.

Functional Example:

```
UserApplicationBoard.PinConfig(4, EONOFF.OFF, PINMODE.OUTPUT, PINLEVEL.LOW)
```

## 11. Legal disclaimer

### 11.1 Engineering samples

Engineering Samples are marked with an asterisk (\*) or (e) or (E). Samples may vary from the valid technical specifications of the product series contained in this data sheet. They are therefore not intended or fit for resale to third parties or for use in end products. Their sole purpose is internal client testing. The

testing of an engineering sample may in no way replace the testing of a product series. Bosch Sensortec assumes no liability for the use of engineering samples. The Purchaser shall indemnify Bosch Sensortec from all claims arising from the use of engineering samples.

### 11.2 Product use

Bosch Sensortec products are developed for the consumer goods industry. They may only be used within the parameters of this product data sheet. They are not fit for use in life-sustaining or security sensitive systems. Security sensitive systems are those for which a malfunction is expected to lead to bodily harm or significant property damage. In addition, they are not fit for use in products which interact with motor vehicle systems.

The resale and/or use of products are at the purchaser's own risk and his own responsibility. The examination of fitness for the intended use is the sole responsibility of the Purchaser.

The purchaser shall indemnify Bosch Sensortec from all third party claims arising from any product use not covered by the parameters of this product data sheet or not approved by Bosch Sensortec and reimburse Bosch Sensortec for all costs in connection with such claims.

The purchaser must monitor the market for the purchased products, particularly with regard to product safety, and inform Bosch Sensortec without delay of all security relevant incidents.

### 11.3 Application examples and hints

With respect to any examples or hints given herein, any typical values stated herein and/or any information regarding the application of the device, Bosch Sensortec hereby disclaims any and all warranties and liabilities of any kind, including without limitation warranties of non-infringement of intellectual property rights or copyrights of any third party. The information given in this document shall in no event be regarded as a guarantee of conditions or characteristics. They are provided for illustrative purposes only and no evaluation regarding infringement of intellectual property rights or copyrights or regarding functionality, performance or error has been made

## 12. Document history and modifications

Rev. No	Chapter	Description of modification/changes	Date
1.0	All	Document creation	15 October 2014
1.1	All	Document modification	25 March 2015
1.2	All	Document modification	23 April 2015
1.3	All	Document modification(Burst Write and Error Code updated)	26 May 2015
1.4	All	Document modification(ADC)	21 March 2016
1.5	All	Document modification(Custom SPI Configuration and Core Clock change API is updated)	6 June 2016
1.6	All	Added APIs for configuring a word(16 bit)	July 2019
1.7	All	Updated the document for latest format	August 2020

Table 1: Revision History

**Bosch Sensortec GmbH**  
Gerhard-Kindler-Straße 9  
72770 Reutlingen / Germany

[contact@bosch-sensortec.com](mailto:contact@bosch-sensortec.com)  
[www.bosch-sensortec.com](http://www.bosch-sensortec.com)

Modifications reserved  
Preliminary - specifications subject to change without notice  
Document number: BST-DHW-SD016-00