IoTivity Programmer's Guide – Soft Sensor Manager for Linux

1 CONTENTS

2 SOFT SENSOR MANAGER (SSM)

This document provides interaction details of SoftSensorManager (SSM) and how it helps in sensing data from various sensors to applications. The purpose of this document is to provide details for developers to understand how to use SDK APIs and how the SSM works to support the APIs.

The first part, SDK API, describes how an application can use the SSM for their purpose. We provide an Ubuntu based sample application which includes the functionality of registering and unregistering query statements to get sensing data.

The next part, SSM Architecture and Components, describes how the main operations of the SDK API are operated in the SSM. In this part, the architecture of SSM will be presented and the components in the architecture will be described in details.

In the third part, SSM Query Statement, a query language called CQL is explained with several examples. The query language is the language used in SSM for applications to get sensing data.

Lastly in the Soft Sensor part, it explains how developers can implement a soft sensor and deploys it in the SSM. An example of SoftSensor, DiscomfortIndexSensor (DISoftSensor), will be presented to help understanding.

3 TERMINOLOGY

3.1 PHYSICAL SENSOR APP

A software application deployed in an open hardware device, such as Arduino board where the device composes hardware sensors such as temperature, humidity, or gyro sensors

This application gets physical sensor data from the hardware board and sends the sensor data to other devices using the Iotivity Base framework.

3.2 SOFT SENSOR (= LOGICAL SENSOR, VIRTUAL SENSOR)

A software module which presents user-defined sensing data

The soft sensor;

1) Collects sensing data from physical and/or other soft sensors,

2) Manipulates the collected sensing data by aggregating and fusing them based on its own composition algorithms, and

3) Provides the manipulated data to applications.

3.3 SOFT SENSOR MANAGER (SSM)

A software service which receives query statements about physical and logical sensors from applications, executes the queries, and returns results to the application through the Iotivity Base.

A more detailed description of Soft Sensor Manager and its relevant components will be provided later in this document.

4 SDK API

SDK API is the facet of SSM to applications and it includes SSMInterface and IQueryEngineEvent as shown in the Figure 1.

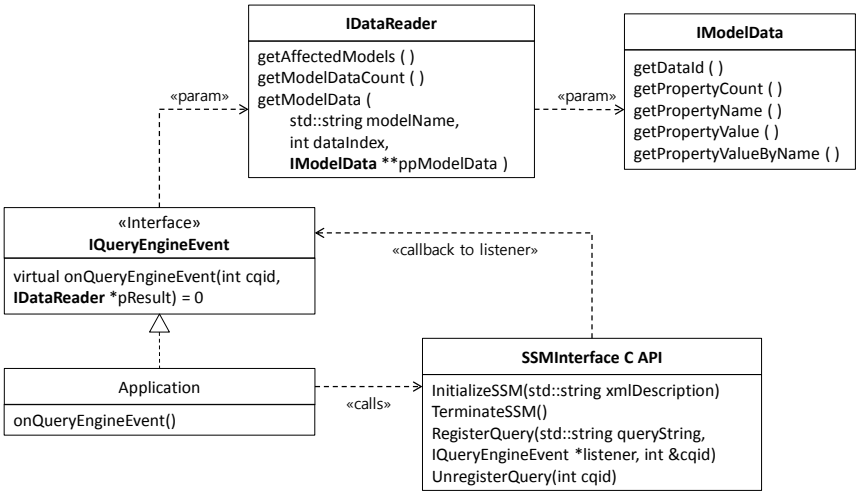


Figure 1. SoftSensorManager SDK APIs and Application

4.1 SSMINTERFACE

This class provides APIs for application to send sensing data requests to SSM, and how to receive the requested sensing data from it. Requests are basically in a query statement.

For an example, if we have a soft sensor providing an indoor discomfort index, called DiscomfortIndexSensor (DISoftSensor). The DISoftSensor provides different index levels as follows;

- ALL_FEEL_COMFORT
- BEGIN_TO_FEEL_UNCOMFORTABLE
- HALF_OF_YOU_FEEL_UNCOMFORTABLE
- ALL_FEEL_UNCOMFORT

So, we can make a query statement for a request to SSM;

```
subscribe DISoftSensor if DISoftSensor.level = BEGIN_TO_FEEL_UNCOMFORTABLE
```

In this scenario, application receives notification of DISoftSensor data from SSM when the condition, DISoftSensor.level = BEGIN_TO_FEEL_UNCOMFORTABLE, is satisfied. Therefore, there can be a time difference between the time the application sends the request and the time the application receives the return because the SSM sends back the return to application only if the condition is satisfied.

In the client,

There are two main functions and two operations.

Main functions;

- InitializeSSM(std::string xmlDescription)
- TerminateSSM()

The first function is for initializing framework with given option which should follow below format.

```
<SSMCore>
  <Device>
    <UDN>abcde123-31f8-11b4-a222-08002b34c003</UDN>
    <Name>MyPC</Name>
    <Type>PC</Type>
  </Device>
  <Config>
    <SoftSensorDescription>/home/iotivity/SoftSensorDescription.xml</SoftSensorDescription>
    <SoftSensorRepository>/home/iotivity/lib/</SoftSensorRepository>
  </Config>
</SSMCore>
```

<SoftSensorDescription> tag is for specifying Soft Sensor description xml file path.
<SoftSensorRepository> tag is for specifying location of Soft Sensors to search.

The second function is for terminating framework.

Main operations;

- RegisterQuery()
- UnregisterQuery()

The first operation, registerQuery(), is to register the query statement to the SSM. After successful response message for the registration, SSM will send the event to the client when query is satisfied as explained above.

The second operation, unregisterQuery (), is to un-register the previously registered query to SSM. The SSM will not send any further message after successful un-registration.

For the query statement, it will be described further in details.

4.2 IQUERYENGINEEVENT AND APPLICATION

This class is an interface class for the application which sends a query statement to get the return from SSM asynchronously.

For the callback listening, precondition is as followings;

- The application should implement the pure virtual function.

```
class SSMTestApp: public IQueryEngineEvent
{
private:
    SSMInterface m_SSMClient;
```

```
public:

    SSMTestApp();

    void onQueryEngineEvent(int cqid, IDataReader *pResult);

    . . .
}
```

- The application should send the application pointer when it calls registerQuery(), so that the SSMInterface can get the callback pointer.

```
void SSMTestApp::registerQuery()
{
   .
   rtn = m_SSMClient.registerQuery(std::string(l_queryString), this, l_qid);
   .
}

SSMReturn::SSMReturn SSMInterface::registerQuery(std::string queryString,
IQueryEngineEvent *listener, std::string &cqid)
{
  .
  .
   m_appListener = listener;
  .
  .
}
```

Once the SSMInterface receives the return from SSM, it calls m_appListener-> onQueryEngineEvent () so that the application can get the callback from the SSMInterface.

The onQueryEngineEvent() has two parameters, *std::string modelName* and *IModelData *ppModelData*. The model in this context means the soft sensor, so the model data means the output data of the soft sensor which was requested. Since the ouput data of the softe sensor can be more than one, it is encapsulated with classes called *IDataReader* and *IModelData.* That is, the onQueryEngineEvent() provides a reference of *IModelData,* which provides getModelData() with which soft sensor output data (i.e. ModelData) can be accessed as property name and property value.

- IDataReader

Operation Name	**P**arameter/**R**eturn		Function
getAffectedModels	P	std::vector<std::string> *pAffectedModels - [OUT] affected ContextModel list	Get affected ContextModels. The CQL can specify multiple ContextModels for retrieving data.
	R	SSMRESULT	
getModelDataCount	P1	std::string modelName - affected ContextModel name	Get affected data count. There are multiple data can exist from given

	P2	int *pDataCount - [OUT] affected dataId count	condition.
	R	SSMRESULT	
getModelData	P1	std::string modelName - affected ContextModel name	Get actual Context Model data
	P2	int dataIndex - affected dataId index	
	P3	IModelData **ppModelData - affected ContextModel data reader	
	R	SSMRESULT	

- IModelData

Operation Name	Parameter/Return		Function
getDataId	P		Get affected DataId. ContextModel has plenty of data so returned data is matched from given condition
	R	Int	
getPropertyCount	P		ContextModel has at least one property that contains data property is described from its specification.
	R	Int	
getPropertyName	P	int propertyIndex -Index of property to read	Retrieve propertyName
	R	std::string	
getPropertyValue	P	int propertyIndex - index of property to read	Retrieve propertyValue
	R	std::string	
getPropertyValueByName	P	std::string propertyName - property name to search value	Retrieve propertyValue using given name
	R	std::string	

5 SSM ARCHITECTURE AND COMPONENTS

5.1 CONTEXT DIAGRAM

The SSM service is basically operated in the Iotivity Base messaging environment as shown in the Figure 2.

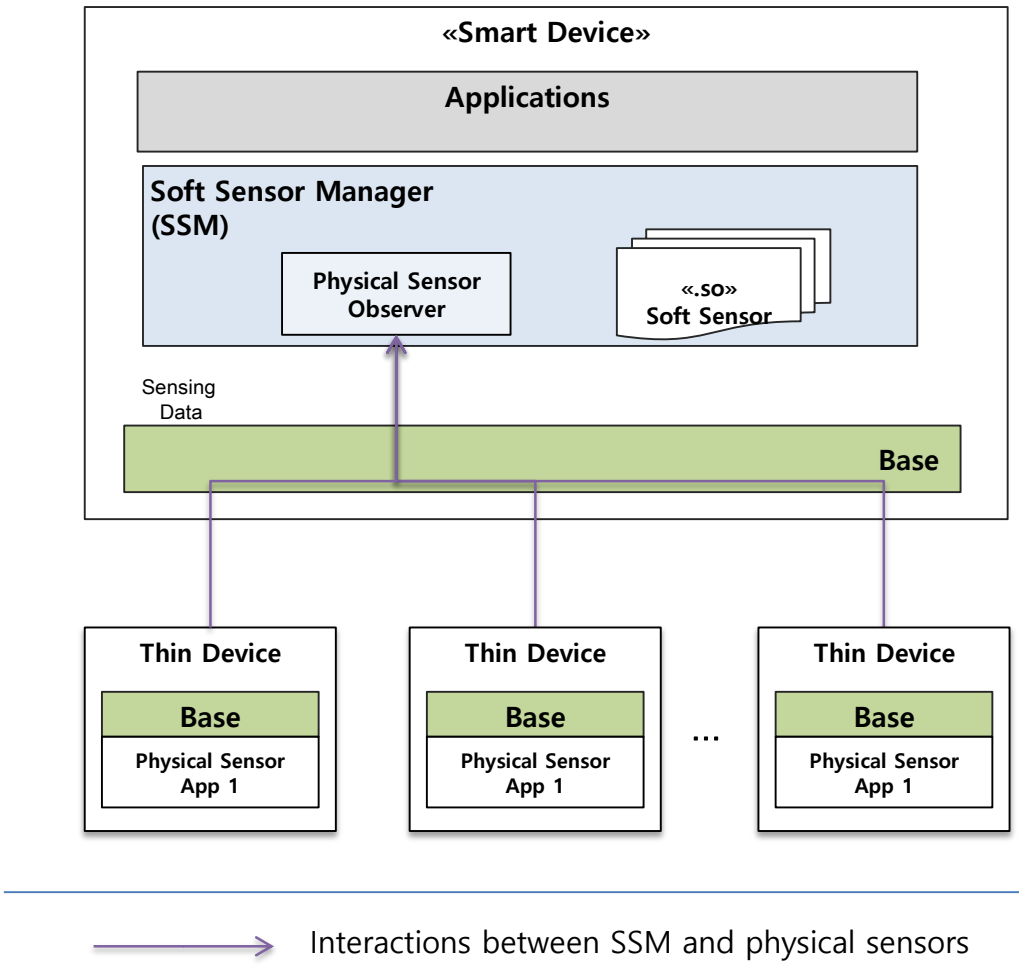


Figure 2. SSM Context Diagram

There are two different types of interactions with SSM;

- Interactions between Application and SSM
- Interactions between SSM and physical sensors

For the first interaction, SSM provides SSM SDK API described in the previous section which hides the details of Iotivity Base and provides a simple operation set in C++.

The second interaction is implemented within a resource model where a physical sensor is registered as a Resource in the Base and the SSM observes the resource by using the APIs provide by the Base.

5.2 SSM ARCHITECTURE

There is the SSM service between applications and physical sensors, and it consists of the three main components such as SSMInterface, QueryProcessor, and SensorManager, as shown in the Figure 3.

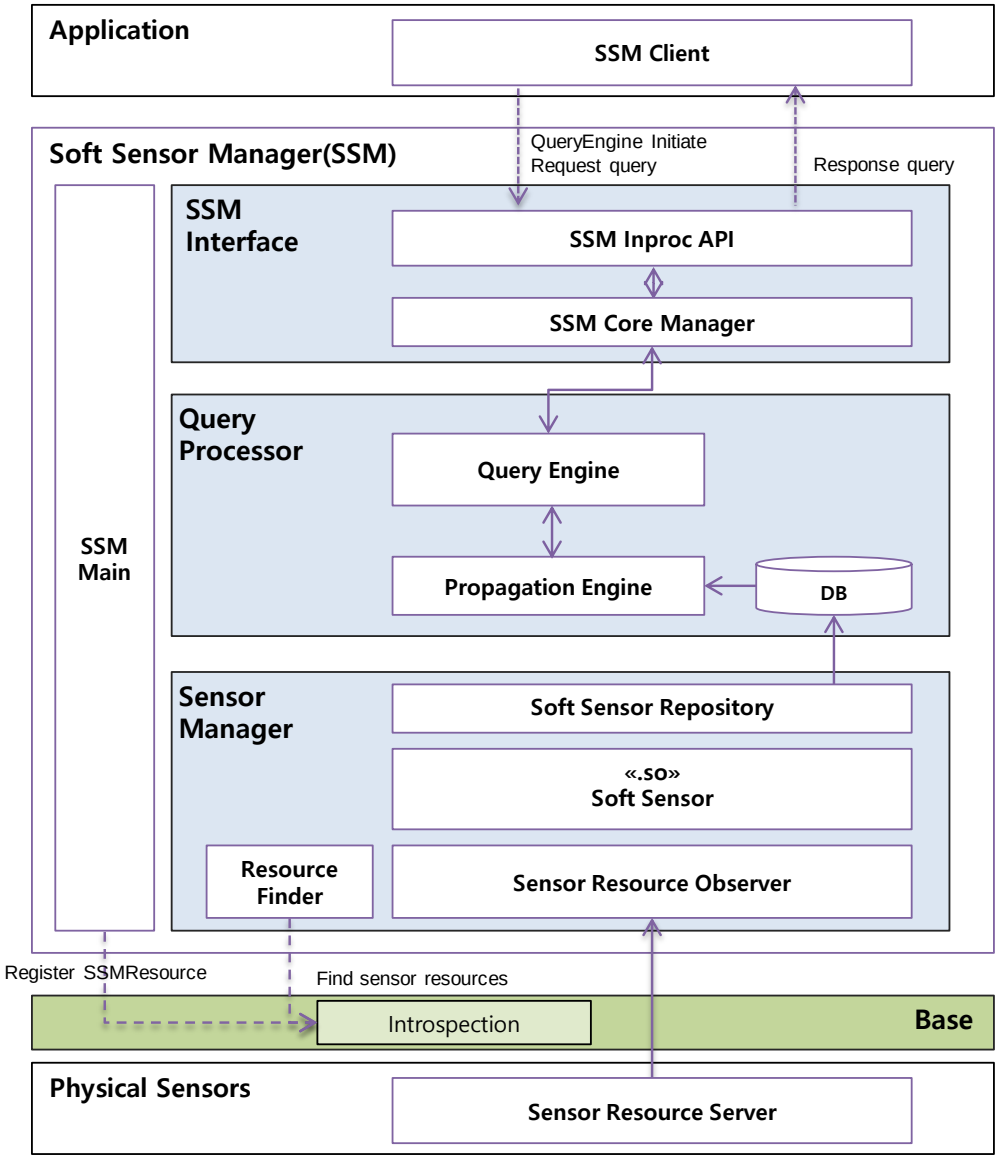


Figure 3. SSM Architecture

SSMInterface is an interface for the application (SSMClient) which sends requests to SSM and get responses from SSM via callback. The SSMInterface includes two main components: SSM Inproc API and SSM Core Manager. The SSM Inproc API is a wrapping class to communicate with SSMClient, and the SSM Core Manager is an interface class communicating with the Query Processor component

QueryProcessor is a processing engine to get query statements, parse the statements, extract conditions and register conditions from the statements. It also monitors the registered conditions whether they are satisfied or not. Once satisfied, it sends the notification to the SSM Core Manager. It includes two main components, Query Engine and Propagation Engine. The Query Engine component is responsible for parsing the query statements and extracting conditions. The Propagation Engine

component gets the conditions and registers them into the database(DB). It also registeres the triggers to the database, so that the DB initiates callback when conditions are satisfied. Once the query has registered, the engine tracks presence of primitive sensors which are described in the query whether it is currently available on the network or local to use. If sensor is not discovered, the engine stores this info to presence list for activating the query in future.

Sensor Manager is a component to maintain Soft Sensors registration and collect physical sensors data required by the Soft Sensors. To register Soft Sensors, a Soft Sensor needs to be deployed in the share library form (*.so) with a manifest file (*.xml) describing the structure of the sensor. Soft sensor and its deployment is described in the fourth section. To collect physical sensors, there is SensorResourceFinder class which for finding specific resources, and registering an Observer to the found resources. It allows the physical sensor to send its sensing data when there is a change in the state or data.

6 SSM QUERY STATEMENT

In query statements, there is a target model called ContextModel, which provides data for applications. In a device, there are three different types of context models; Device, SoftSensor, and PhysicalSensor.

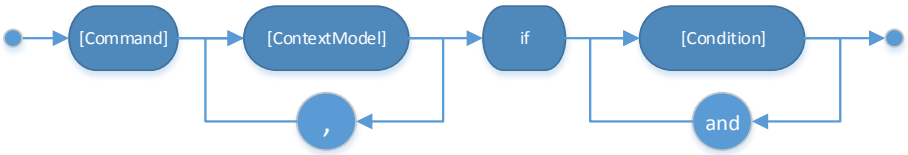
The Device Context Model corresponds to the device provides three properties, UDN, Name, and Type. The UDN contains unique key value with 128bit length, the Name represents device's own name like 'My Phone', and the Type is device's attribute like Mobile, TV, and PC.

Every Context Model includes dataId with which applications can access the Context Model data directly. For example, if Device Context Model contains five data, you can get 4th device information through query engine using *'get Device if Device.dataId = 4'* query statement.

For soft sensors and physical sensors, they generally have their own structures and the Context Models of the sensors are generated with manifest files (*.xml) which are packaged together with the sensors.

6.1 CONTEXT QUERY LANGUAGE (CQL)

The grammar of the CQL is as shown below



[Command]

The query engine provides two commands; Subscribe and Get.

The Subscribe keyword is used for asynchronous query request that is affective till the registered query is removed. That is, the result can be delivered to clients several times whenever conditions meet. The Get keyword is almost same as subscribe, but the result data is delivered only once.

One of main differences between the Subscribe and the Get is number of Callback calls and inclusion of cached data. The result of Subscribe contains cached data at the execution time and new data after execution untill the query statement is unregistered. The Get command returns the most recent data only one time.

[ContextModel]

Application developer should describe what Context Model data to retrieve from running result of CQL. The application developer can use comma (‘,’) or white space (‘ ‘) to retrieve multiple Context Models.

The Context Model can be defined by CQL’s [Condition] part description. For example, if [Condition] part is described like ‘Device.EPG.CurrentInfo.value != “null” then [ContextModel] contains Device, Device.EPG, Device.CurrentInfo Context Models.

If [Condition] part is described to combine conditions like ‘Device.EPG.value != “null” and Device.EPG.CurrentInfo.value != “null” using ‘and’ keyword, the [ContextModel] part contains Device, Device.EPG Context Models because these two things are the only intersect of two conditions.

[Condition]

It is used for application developers to search and trigger data using conditions. The [Condition] grammar is as shown below

The [property] part represents output properties that [ContextModel] has. Basically, every Context Model has ‘dataId’(i.e.property). The [ContextModel] part must be declared with its parent [ContextModel] names. The [comparator] field can hold six operators like = (==), !=, >, >=, <, <=. The [value] field is set value for comparison. Following type of [value] is possible integer, float, double, text and text must be capsulated using double quot.

Ex:) Device.LiftUpSmartPhone.status = “true” or Device.type = “Mobile”

If the operators are combined with text, the query engine checks the condition lexicographically.

Ex:) “Accessory” < “Apple” < “Apple Pie”

6.2 EXAMPLES OF CQL STATEMENTS

```
subscribe Device if Device.type == "Mobile"
```

When Mobile Device on the network appears, notify Device information.

```
subscribe Device if Device.type == "TV" and
Device.NumberOfPeopleWatchingTV.number > 2
```

When type of Device is "TV" and the number of NumberOfPeopleWatchingTV is greater than 2, notify Device information.

```
subscribe if Device.LiftUpSmartPhone.value == "true" and
Device.NumberOfPeopleWatchingTV.number > 0
```

Of peripheral devices, if value LiftUpSmartPhone is "true" and number of NumberOfPeopleWatchingTV is greater than 0, retrieve Device information. (‘Device’ keyword must present every [ContextModel] of [Condition])

```
Get Device if Device.BatteryStatus.percentage > 50
```

If Battery's percentage is greater than 50, notify Device information.

```
Get Device if Device.PhoneTodaySchedule.title = "study” and
Device.UserAtHome.value = "true"
```

Of peripheral devices, if title of PhoneTodaySchedule is "study" and value of UserAtHome is "true", retrieve Device information.

```
Get Device. PhoneGPS[2]
```

Retrieve PhoneGPS information of which dataId is 2. (Always the lowest [ContextModel] only have Index. Only "Get" query can contain index, not “if” statement(a conditional sentence)).

```
Subscribe Device if Device.CallStatus.callername = "lee" and Device.type =
"TV"
```

When callername is "lee" and type is "TV", notify Device information

```
Subscribe Device.TVZipcode if Device.TVZipcode.value != "null" and
Device.dataId = 3
```

If value of TVZipcode is not "null" and dataId of Device is 3, notify TVZipcode's information

```
Get Device.BatteryStatus[1]
```

Notify BatteryStatus information of which dataId is 1. (If the appropriate data doesn't exist, do nothing.)

```
Get Device.UserAtHome if Device.UserAtHome.value = "true"
```

If UserAtHome's value is "true", nofity UserAtHome's information (If the appropriate data doesn't exist, do nothing.)

```
Subscribe Device if Device.LiftUpSmartPhone.value = "true" and
Device.PhoneGPS.latitude != 20
```

If Phone's status is "LiftUp" and PhoneGPS's latitude is not 20, notify Device information.

```
Subscribe Device if Device.BatteryStatus.percentage >= 50 and
Device.LiftUpSmartPhone.value = "true"
```

If BatteryStatus's percentage is greater than or equal to 50 and Phone's status is "LiftUp", notify Device's information.

7 SOFT SENSOR

Soft sensor, also called Virtual sensor or Logical sensor, is a software component which gets physical sensor data and generates new data by data aggregation and fusion. This part shows how to develop a soft sensor, deploy in the SSM, and use the deployed soft sensor.

7.1 DEVELOPMENT

SSM loads a shared library, (*.so) as an soft sensor unit. A soft sensor should be developed and deployed as a share library which includes the entry operation, defined in the Interface, ICtxEvent.

Soft Sensor Definition: a soft sensor consists of three main elements; input data, output data, and execution logic. To be deployed in SSM, the three elements can be implemented as follows;

Input data: the required data by the target soft sensor and it is generally the sensing data from physical sensors. For example, a DiscomfortIndexSensor, requires temperature sensors and humidity sensors as inputs. Output data: the result of data fusion by the target soft sensor. The unit of the output data should be different from the types of soft sensors.

In SSM, the main properties of Soft Sensor, name, input, output, should be described in a manifest file (i.e. **SoftSensorDescription.xml**).

Here is an example of DiscomfortIndexSensor:

The first tag, <name>, is referred in the query statement from applications. It is also used for SSM to load share library (*.so). It should be the same as the name of the share library file.

```
<softsensors>

  <softsensor>
    <name>DiscomfortIndexSensor</name>
    <attributes>
      <attribute>
        <name>version</name>
        <type>string</type>
        <value>1.0</value>
      </attribute>
      <attribute>
        <name>lifetime</name>
        <type>int</type>
        <value>60</value>
      </attribute>
    </attributes>
    <outputs>
      <output>
        <name>timestamp</name>
```

```
        <type>string</type>
      </output>
      <output>
        <name>temperature</name>
        <type>string</type>
      </output>
      <output>
        <name>humidity</name>
        <type>string</type>
      </output>
      <output>
        <name>discomfortIndex</name>
        <type>int</type>
      </output>
    </outputs>
    <inputs>
      <input>Thing_TempHumSensor</input>
      <input>Thing_TempHumSensor1</input>
    </inputs>
  </softsensor>

</softsensors>
```

For inputs, the physical sensors required by the target soft sensor can be specified in this tag. Moreover soft sensors can be used for input sensors.

Execution Logic: With the input data, Soft sensor generates the output, based on its own algorithm.

Soft sensor should implement the ICtxEvent interface which provides the OnCtxEvent() operation, as a pure virtual operation. In SSM, the operation is called by CContextExecutor class right after the class loads the soft sensor's .so file, and when the SSM receives sensing data from physical sensors.

The OnCtxEvent operation requires two input parameters, eventType, contextDataList as follows;

- eventType: It is the time point the onCtxEvent() called by SSM and includes three types, SPF_START, SPF_UPDATE, and SPF_END, where SPF_START is the time when the soft sensor library is loaded, and currently SSM only uses this option.
- contextDataList: it is the input value the soft sensor required and it is provided as an attribute map(key,string) of the sensing data from the physical sensors specified in the input tag in the manifest file. That is, SSM, CContextExecutor generates the attribute map of the input data and delegates to the soft sensor when it calls the OnCtxEvent().

```
class ICtxEvent
{
public:
   virtual void OnCtxEvent( enum CTX_EVENT_TYPE, std::vector<ContextData>) = 0 ;
   virtual ~ICtxEvent(){};
};


class DiscomfortIndexSensor: public ICtxEvent
{
private:

   int RunLogic(std::vector< ContextData > &contextDataList);
```

```
public:
    DiscomfortIndexSensor();

    void OnCtxEvent(enum CTX_EVENT_TYPE eventType,
                    std::vector< ContextData > contextDataList);

   .
   .
};
```