Sep. 5, 2014
Samsung Iotivity Service
– Soft Sensor Manager

SoftSensorManager (SSM)

This document provides interaction details of SoftSensorManager (SSM) which basically run on the Iotivity Base and provides 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 uses the SSM for their purposes. 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 is explained how developers implements a soft sensor and deploys it in the SSM. An example of SoftSensor, DiscomfortIndexSensor (DISoftSensor), will be presented to help understanding.

Terminology

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 based on the Iotivity Base.

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.

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.

SDK API

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

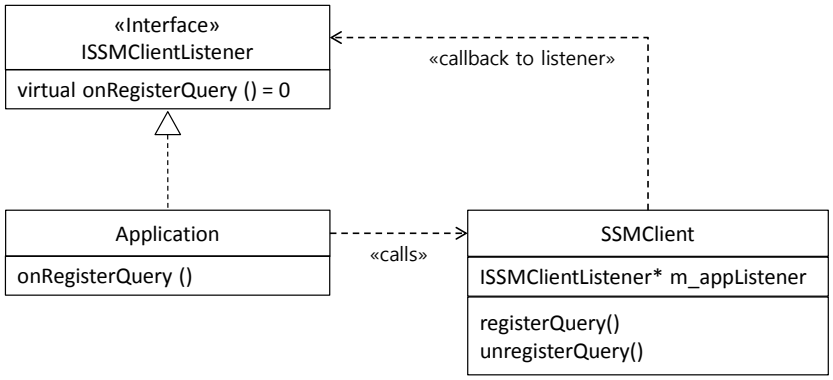


Figure 1. SoftSensorManager SDK APIs and Application

SSMClient

This class provides APIs for application to send sensing data requests to SSM, and receives the requested sensing data where the requests are basically in a query statement.

For an example, 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
```

Which means, the application wants SSM to send DISoftSensor data 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 operations;

- registerQuery()
- unregisterQuery()

The first operation, registerQuery(), is to send the a query statement so that SSM maintains the query statement, check whether conditions in the statement is satisfied, and sends returns if satisfied.

The second operation, unregisterQuery(), is to remove the query SSM maintains so then the conditions would not be checked further.

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

ISSMClientListener and Application

This class is an interface class for the application which has sent 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 ISSMClientListener
{
private:
    SSMClient m_SSMClient;

public:

    SSMTestApp();

    void onRegisterQuery(const AttributeMap& attributeMap,
                         SSMReturn::SSMReturn& eCode);

     . . .
}
```

- The Application should send the pointer of the Application when calls registerQuery(), so that the SSMClient can get the callback pointer.

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

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

Once the SSMClient receives the return from SSM, it calls m_appListener->onRegisterQuery() so that the Application can get the callback from SSMClient.

SSM Architecture and Components

Context Diagram

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

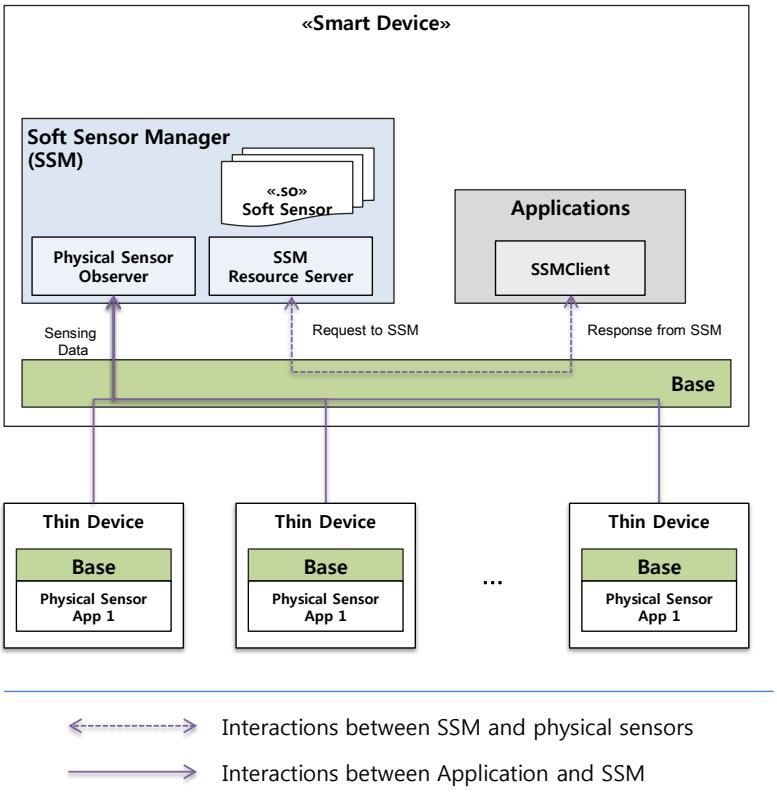


Figure 2. SSM Context Diagram

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 part 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.

SSM Architecture

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

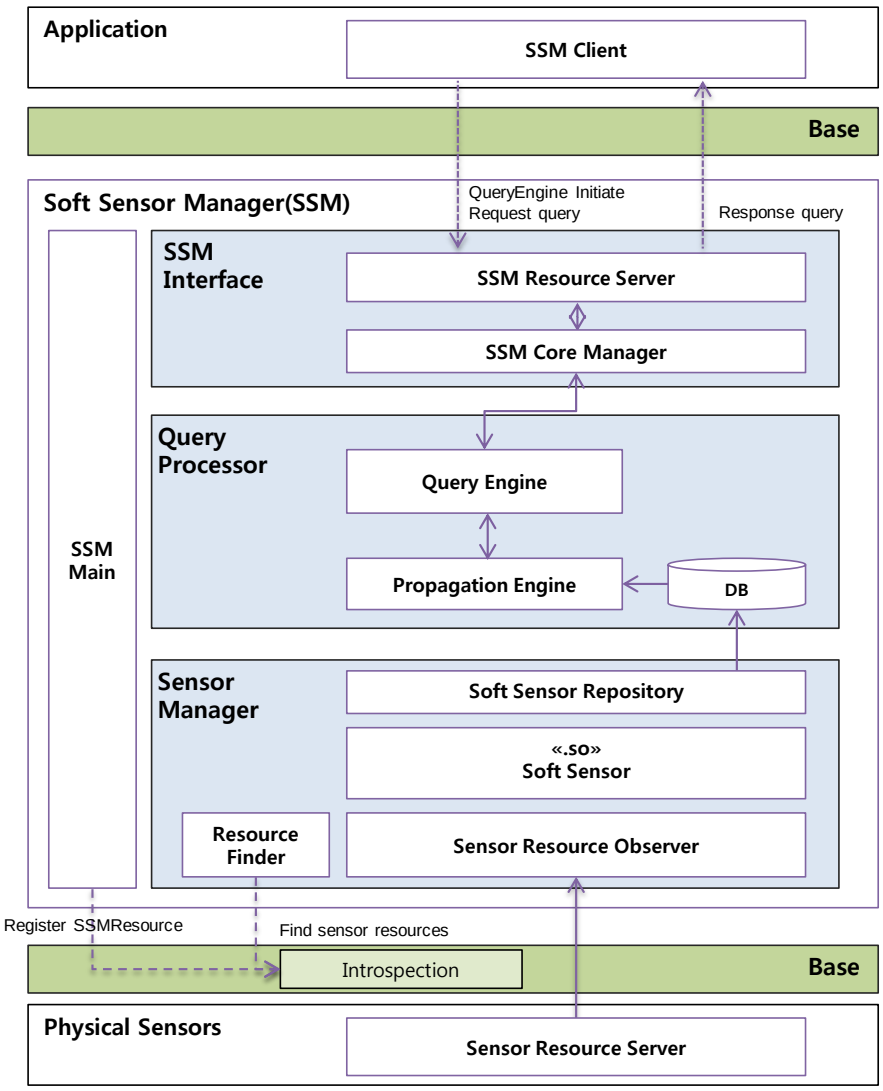


Figure 3. SSM Architecture

SSMInterface is an interface component to get the request from applications, i.e. SSMClient, and to send the callback to the applications. The SSMInterface includes two main components; *SSM Resource Sever,* and *SSM Core Manager*. The SSM Resource Server is a wrapping class to communicate with SSMClient based on the Iotivity Base, i.e. resource model, and the SSM Core Manager is an interface class communicating with the Query Processor component which is not in resource model.

QueryProcessor is a processing engine to get query statements, parse the statements, and extract and register conditions from the statements. It also monitors the registered conditions whether they are satisfied, if it is; it sends the notification to the SSM Core Manager. To do that, the *Query Engine* component is responsible for parsing the query statements and extracting conditions, and the *Propagation Engine* component gets the conditions and registers them into the DB, as well as trigger into the DB so that the DB initiates callback when a condition is satisfied.

Sensor Manager is a component to maintain Soft Sensors registered and collect physical sensor 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. For a soft sensor, it will be described in the fourth part. To collect physical sensor, there is *SensorResourceFinder* class which find specific resources, i.e. physical sensors, and register an Observer into the resources found, so that the physical sensor keep sending its sensing data when the state of data is changed.

SSM Query Statement

In query statements, there 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.

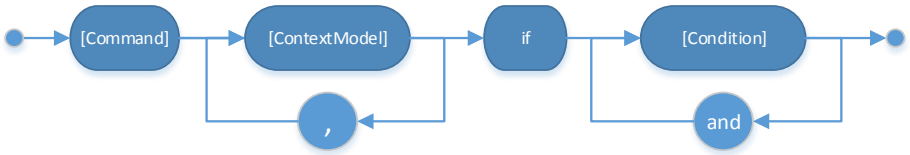
The Device Context Model corresponds to the device the SSM deployed and it provides three properties, UDN, Name, and Type. The UDN contains unique key value which length is 128bit, the Name represents device's own name like 'My Phone', and the Type is device's attribute like Mobile, TV, and PC.

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 access 4^{th} device information like Device.dataId = 4

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.

Context Query Language (CQL)

The grammar of the CQL is as shown below



[Command]

There are two commands provides; *Subscribe* and *Get.*

The Subscribe keyword is used for asynchronous query request that is affective till cancel the registered query statements. 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]

It is the part that application developer describes what Context Model data to retrieve for running result of CQL. The application developer can use comma (‘,’) to retrieve multiple Context Models.

The Context Model can be defined refer to CQL’s [Condition] part description. For example, if [Condition] part is described like ‘Device.EPG.CurrentInfo.value != “null”’ then [ContextModel] can hold 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 can hold 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, Boolean and text, Boolean must be capsulated using double quot.

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

Examples of CQL Statements

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

When Mobile Device on the network appears, inform us of Device information to satisfy this condition.

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

When Conditions are that type of Device is "TV" and the value of number of NumberOfPeopleWatchingTV is greater than 2, if Appeared to satisfy this condition. inform us of Device information to satisfy this condition.

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

Of peripheral devices, when Conditions are that The Device's the value of LiftUpSmartPhone is "true" and Device's number of NumberOfPeopleWatchingTV is greater than 0, if Device to satisfy conditions appears inform us of Device information. (Device is common [ContextModel] of [Condition])

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

If Battery's percentage is greater than 50, inform us of Device information.

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

Of peripheral devices, If Device's the title of PhoneTodaySchedule is "study" and Device's the value of UserAtHome is "true", inform us of Device information to satisfy this condition.

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

Inform us of PhoneGPS information that dataId is 2. (Always the lowest [ContextModel] only have Index. "Get" Query containing index doesn't use to if-clause(a conditional sentence))

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

When callername is "lee" and type is "TV", inform us of Device's information

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

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

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

inform us of BatteryStatus information that dataId is 1. (If the appropriate data don't exist, not doing anything.)

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

If UserAtHome's value is "true", inform us of UserAtHome's information (If the appropriate data don't exist, not doing anything.)

```
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, inform us of Device's 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", inform us of Device's information.

Soft Sensor

Soft sensor, also called Virtual sensor or Logical sensor, is a software component which gets physical sensor data and generates a higher abstraction level of sensing 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.

Development

SSM loads a share library, (*.so) as an soft sensor unit, that is, a soft sensor should be developed and deployed as an 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 the example of DiscomfortIndexSensor, it requires temperature sensors and humidity sensors as inputs. For the input data, the Physical Sensor Apps should be developed as a prerequisite.

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. ***HighContextDictionary.xml***) and this is an example of DiscomfortIndexSensor shown;

For the root_name, it is referred in the query statement from applications. It is also used for SSM to load share library (*.so). That is, it should be the same as the name of the share library file.

```
<high_context_dictionary>

  <high_context>
    <root_name>DiscomfortIndexSensor</root_name>
    <outputs>
      <output_property_count>4</output_property_count>
      <output_property>
        <name>timestamp</name>
        <type>string</type>
        <value>&amp;quot;&amp;quot;</value>
      </output_property>
      <output_property>
        <name>temperature</name>
        <type>int</type>
        <value>0</value>
      </output_property>
      <output_property>
        <name>humidity</name>
        <type>int</type>
        <value>0</value>
      </output_property>
      <output_property>
        <name>discomfortIndex</name>
        <type>int</type>
        <value>0</value>
      </output_property>
    </outputs>
    <inputs>
      <input_count>1</input_count>
      <input>Thing_TempHumSensor</input>
    </inputs>
  </high_context>

</high_context_dictionary>
```

For inputs, the physical sensors required by the target soft sensor can be specified in this tag. Moreover, it can be not only physical sensors but soft sensors.

Execution Logic: With the input data, Soft sensor generates the output, based on its own algorithm. It should be developed as a software code, in SSM, a class as shown in the DiscomfortIndexSensor example.

Basically the soft sensor calls 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 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);

    .
    .
    .
};
```

Deployment

Once a soft sensor is developed as an *.so library, it should be deployed in the SSM, with the manifest file, and it can be simply done by copying the *.so file and appending the xml definition in the HighContextDictionary.xml.

You should copy the *.so in the ‘Outputs’ directory which is made in the SoftSensorManager directory when you run the Makefile.

You should also find the HighContextDictionary.xml in the same directory, ‘Outputs’.