1 Coding for Metering Service

This document is part of the TIDC-BLE-METER-READING, it details the procedure of adding the Metering Services and its associated application to the SensorTag application example code. Part of the code is listed in blue box in this document for explanation purpose. For complete code, please refer to the TI design TIDC-BLE-METER-READING 's design file download section to download the complete code.

For the sake of simplicity, the metering service is added to the example profile runs on the SensorTag hardware instead of creating everything from scratch. This provides the starting point of the development be on a proven and stable platform.

From this standpoint, creating the metering profile only involves adding the metering service profile into the original SensorTag application. Adding this profile includes two major parts: adding the implementation of the profile and adding the implementation of the service. Some minor modification to the original code is also required to enable the added service.

To add the profile, create two files (*meteringservice.h* and *meteringservice.c) in the same* directory that contains the other profiles' source code files :
ORG_PROJ_DIR\..\..\..\..\..\Profiles\SensorProfile\CC26xx\.

1.1 Metering Service Header File

The *meteringservice.h* file includes the definition of UUID for the metering service and the characteristic of the metering service.

```
#define PROPERITIES_CHAR                0
#define VOLTAGE_CHAR                    4
#define CURRENT_CHAR                    5
#define POWER_CHAR                      6
#define POWER_FACT_CHAR                 7
#define FREQUENCY_CHAR                  8
#define POWER_STATE_CHAR                9
#define PHASE_CHAR                      10
#define NAME_CHAR                       11
#define VERSION_CHAR                    12


// Service UUID
/* The 16 bit UUID listen above is only a part of the
 * full TI random 128 bit UUID:
 * F000EE00-0451-4000-B000-00000000-00000. */
#define METERING_SERV_UUID         0xEE00
#define METERING_PROP_UUID         (0x01 + METERING_SERV_UUID)
#define METERING_ENAB_UUID         (0x02 + METERING_SERV_UUID)
#define METERING_PERI_UUID         (0x03 + METERING_SERV_UUID)
#define METERING_VOLT_UUID         (METERING_SERV_UUID + VOLTAGE_CHAR)
#define METERING_CURR_UUID         (METERING_SERV_UUID + CURRENT_CHAR)
#define METERING_POWR_UUID         (METERING_SERV_UUID + POWER_CHAR)
#define METERING_PWRF_UUID         (METERING_SERV_UUID + POWER_FACT_CHAR)
#define METERING_FREQ_UUID         (METERING_SERV_UUID + FREQUENCY_CHAR)
#define METERING_PSTA_UUID         (METERING_SERV_UUID + POWER_STATE_CHAR)
#define METERING_PHAS_UUID         (METERING_SERV_UUID + PHASE_CHAR)
#define METERING_NAME_UUID         (METERING_SERV_UUID + NAME_CHAR)
#define METERING_VERS_UUID         (METERING_SERV_UUID + VERSION_CHAR)

// Length of characteristic data in bytes
```

```
#define METERING_VOLT_LEN          4
#define METERING_CURR_LEN          4
#define METERING_POWR_LEN          12
#define METERING_PWRF_LEN          2
#define METERING_FREQ_LEN          2
#define METERING_PSTA_LEN          1
#define METERING_PHAS_LEN          1
#define METERING_NAME_LEN          22
#define METERING_VERS_LEN          16
#define METERING_PROP_LEN          18
```

Then four function prototypes, required for every profile, must be declared:

- Metering_addService
- Metering_registerAppCBs
- Metering_setParameter
- Metering_getParameter

```
/*
 * Metering_addService- Initializes the Sensor GATT Profile service by registering
 *          GATT attributes with the GATT server.
 */
extern bStatus_t Metering_addService(void);

/*
 * Metering_registerAppCBs - Registers the application callback function.
 *                    Only call this function once.
 *
 *    appCallbacks - pointer to application callbacks.
 */
extern bStatus_t Metering_registerAppCBs(sensorCBs_t *appCallbacks);

/*
 * Metering_setParameter - Set a Sensor GATT Profile parameter.
 *
 *    param - Profile parameter ID
 *    len   - length of data to write
 *    value - pointer to data to write.  This is dependent on
 *          the parameter ID and WILL be cast to the appropriate
 *          data type (example: data type of uint16_t will be cast to
 *          uint16_t pointer).
 */
extern bStatus_t Metering_setParameter(uint8_t param, uint8_t len, void *value);

/*
 * Metering_getParameter - Get a Sensor GATT Profile parameter.
 *
 *    param - Profile parameter ID
 *    value - pointer to data to read.  This is dependent on
 *          the parameter ID and WILL be cast to the appropriate
 *          data type (example: data type of uint16_t will be cast to
 *          uint16_t pointer).
 */
extern bStatus_t Metering_getParameter(uint8_t param, void *value);
```

1.2 Metering Service C Source File

In *meteringservice.c*, the coding involves putting information into the data structure to represent the metering service, the characteristics, and the implementation of the four functions defined in *meteringserrvice.h*.

1.2.1 Defining Attribute Table

The first part of coding the service data is structure is to create a constant for each of the UUID for the metering service and its associated characteristics.

```
// Service UUID
static CONST uint8_t sensorServiceUUID[TI_UUID_SIZE] =
{
  TI_UUID(SENSOR_SERVICE_UUID),
};

// Characteristic UUID: config (enable)
static CONST uint8_t sensorEnableUUID[TI_UUID_SIZE] =
{
  TI_UUID(SENSOR_ENABLE_UUID),
};

// Characteristic UUID: period
static CONST uint8_t sensorPeriodUUID[TI_UUID_SIZE] =
{
  TI_UUID(SENSOR_PERIOD_UUID),
};

// Characteristic UUID: voltage
static CONST uint8_t meteringvoltUUID[TI_UUID_SIZE] =
{
  TI_UUID(METERING_VOLT_UUID),
};

// Characteristic UUID: current
static CONST uint8_t meteringcurrUUID[TI_UUID_SIZE] =
{
  TI_UUID(METERING_CURR_UUID),
};

// Characteristic UUID: power
static CONST uint8_t meteringpwrUUID[TI_UUID_SIZE] =
{
  TI_UUID(METERING_POWR_UUID),
};

// Characteristic UUID: power facgtor
static CONST uint8_t meteringpwrfUUID[TI_UUID_SIZE] =
{
  TI_UUID(METERING_PWRF_UUID),
};

// Characteristic UUID: frequency
static CONST uint8_t meteringfreqUUID[TI_UUID_SIZE] =
{
  TI_UUID(METERING_FREQ_UUID),
};

// Characteristic UUID: power on state
static CONST uint8_t meteringpstaUUID[TI_UUID_SIZE] =
{
```

```
  TI_UUID(METERING_PSTA_UUID),
};

// Characteristic UUID: phase
static CONST uint8_t meteringphasUUID[TI_UUID_SIZE] =
{
  TI_UUID(METERING_PHAS_UUID),
};

// Characteristic UUID: meter name
static CONST uint8_t meteringnameUUID[TI_UUID_SIZE] =
{
  TI_UUID(METERING_NAME_UUID),
};

// Characteristic UUID: meter version
static CONST uint8_t meteringversUUID[TI_UUID_SIZE] =
{
  TI_UUID(METERING_VERS_UUID),
};

// Characteristic UUID: meter configuration
static CONST uint8_t meteringpropUUID[TI_UUID_SIZE] =
{
  TI_UUID(METERING_PROP_UUID),
};
```

The second step involves allocating byte memory arrays for storing each characteristic data, definition of the property of each characteristic. There are up to three static values to define for each characteristic:

- Characteristic data –A byte array with a length equal to the number of bytes that the characteristic is supposed to have
- Characteristic property – An unsigned 8-bit value containing the access property of the characteristic
- Characteristic configuration – A pointer to a data structure known as a configuration table (this value must be defined only if the characteristic uses the notify)

```
// Characteristic Value: voltage
static uint8_t voltData[METERING_VOLT_LEN] = { 0, 0, 0, 0};
// Characteristic Properties: voltage
static uint8_t voltDataProps = GATT_PROP_READ | GATT_PROP_NOTIFY;
// Characteristic Configuration: voltage
static gattCharCfg_t *voltDataConfig;

// Characteristic Value: current
static uint8_t currData[METERING_CURR_LEN] = { 0, 0, 0, 0};
// Characteristic Properties: current
static uint8_t currDataProps = GATT_PROP_READ | GATT_PROP_NOTIFY;
// Characteristic Configuration: current
static gattCharCfg_t *currDataConfig;

// Characteristic Value: power (include active, reactive, apparent, 1st 4 bytes active, 2nd
reactive, 3rd apparent)
static uint8_t pwrData[METERING_POWR_LEN] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
// Characteristic Properties:  power
static uint8_t pwrDataProps = GATT_PROP_READ | GATT_PROP_NOTIFY;
// Characteristic Configuration:  power
```

```
static gattCharCfg_t *pwrDataConfig;

// Characteristic Value: power factor
static uint8_t pwrfData[METERING_PWRF_LEN] = { 0, 0};
// Characteristic Properties : power facgtor
static uint8_t pwrfDataProps = GATT_PROP_READ | GATT_PROP_NOTIFY;
// Characteristic Configuration: power factor
static gattCharCfg_t *pwrfDataConfig;

// Characteristic Value: frequency
static uint8_t freqData[METERING_FREQ_LEN] = { 0, 0};
// Characteristic Properties: frequency
static uint8_t freqDataProps = GATT_PROP_READ | GATT_PROP_NOTIFY;
// Characteristic Configuration: frequency
static gattCharCfg_t *freqDataConfig;
```

```
// Characteristic Value: power on state
static uint8_t pstaData[METERING_PSTA_LEN] = {0};
// Characteristic Properties: power on state
static uint8_t pstaDataProps = GATT_PROP_READ | GATT_PROP_WRITE;

// Characteristic Value: phase
static uint8_t phasData[METERING_PHAS_LEN] = {0};
// Characteristic Properties: phase
static uint8_t phasDataProps = GATT_PROP_READ | GATT_PROP_WRITE;

// Characteristic Value: meter name
static uint8_t nameData[METERING_NAME_LEN] = {0};
// Characteristic Properties: meter name
static uint8_t nameDataProps = GATT_PROP_READ;

// Characteristic Value: meter version
static uint8_t versData[METERING_VERS_LEN] = {0};
// Characteristic Properties: meter version
static uint8_t versDataProps = GATT_PROP_READ;

// Characteristic Value: meter properties
static uint8_t propData[METERING_PROP_LEN] = {0};
// Characteristic Properties: meter properties
static uint8_t propDataProps = GATT_PROP_READ;

// Characteristic Properties: enable
static uint8_t sensorEnableProps = GATT_PROP_READ | GATT_PROP_WRITE;

// Characteristic Value: enable
static uint8_t sensorEnable = 0;

// Characteristic Properties: period
static uint8_t sensorPeriodProps = GATT_PROP_READ | GATT_PROP_WRITE;
// Characteristic Value: period
static uint8_t sensorPeriod = SENSOR_MIN_UPDATE_PERIOD / SENSOR_PERIOD_RESOLUTION;
```

The next step is to fill the profile attribute table with the newly-defined data. The profile attribute table is a set of data structures that consists of:

Type – Contains the UUID type size and the UUID value
Permission – This is the GATT permission
Handle – Value assigned internally by an attribute server (usually just put a “0” when initializing)
pValue – Pointer to a byte array containing the corresponding data

This first entry of the profile attribute table is the definition of the metering service.

```
  {
    { ATT_BT_UUID_SIZE, primaryServiceUUID }, /* type */
    GATT_PERMIT_READ,                         /* permissions */
    0,                                        /* handle */
    (uint8_t *)&sensorService                 /* pValue */
  },
```

The “sensorService” is defined earlier as:

```
// Profile Service attribute
static CONST gattAttrType_t sensorService = { TI_UUID_SIZE, sensorServiceUUID };
```

Following the “sensorService” is the list of characteristics entries. The table only lists the characteristic’s declarations and characteristic’s data value for characteristics without the notification capability.

```
      // Characteristic Declaration
      {
        { ATT_BT_UUID_SIZE, characterUUID },
        GATT_PERMIT_READ,
        0,
        &propDataProps
      },

        // Characteristic Value "meter property"
        {
             { TI_UUID_SIZE, meteringpropUUID },
             GATT_PERMIT_READ,
             0,
             propData
        },
```

In situations where the characteristic supports ‘write’; the permission attribute in the characteristic’s definition should be set with GATT_PERMIT_WRITE. See the “meter reading enable” characteristic definition below :

```
  // Characteristic Declaration
  {
    { ATT_BT_UUID_SIZE, characterUUID },
    GATT_PERMIT_READ,
    0,
    &sensorEnableProps
  },

    // Characteristic Value "Enable"
    {
      { TI_UUID_SIZE, sensorEnableUUID },
      GATT_PERMIT_READ | GATT_PERMIT_WRITE,
      0,
      &sensorEnable
    },
```

For characteristics with notification capability, the characteristic’s configuration is also listed in addition to the characteristic’s declaration and data value. See the “voltage” characteristic for example:

```
      // Characteristic Declaration
      {
        { ATT_BT_UUID_SIZE, characterUUID },
        GATT_PERMIT_READ,
        0,
        &voltDataProps
      },

        // Characteristic Value "Voltage"
        {
             { TI_UUID_SIZE, meteringvoltUUID },
             GATT_PERMIT_READ,
             0,
             voltData
        },

        // Characteristic configuration
        {
             { ATT_BT_UUID_SIZE, clientCharCfgUUID },
             GATT_PERMIT_READ | GATT_PERMIT_WRITE,
             0,
             (uint8_t *)&voltDataConfig
        },
```

1.2.2 Implementing "Metering_addService" Function

When the profile attribute table is complete with all characteristics filled, then the functions required for the metering service needs to be implemented. The first function be implemented is "Metering_addService". This function is called to tell GATT layer to register all characteristics that has notification capability.

```
/*********************************************************************
 * @fn      Metering_addService
 *
 * @brief   Initializes the metring Profile service by registering
 *          GATT attributes with the GATT server.
 *
 * @return  Success or Failure
 */
bStatus_t Metering_addService(void)
{
  // ---------- Add voltage service
  // Allocate Client Characteristic Configuration table
  voltDataConfig = (gattCharCfg_t *)ICall_malloc(sizeof(gattCharCfg_t) *
                                                 linkDBNumConns);
  if (voltDataConfig == NULL)
  {
    return (bleMemAllocError);
  }

  // Register with Link DB to receive link status change callback
  GATTServApp_InitCharCfg(INVALID_CONNHANDLE, voltDataConfig);


  // ---------- Add current service
  // Allocate Client Characteristic Configuration table
```

```
  currDataConfig = (gattCharCfg_t *)ICall_malloc(sizeof(gattCharCfg_t) *
                                                 linkDBNumConns);
  if (currDataConfig == NULL)
  {
    return (bleMemAllocError);
  }

  // Register with Link DB to receive link status change callback
  GATTServApp_InitCharCfg(INVALID_CONNHANDLE, currDataConfig);


  // ---------- Add power service
  // Allocate Client Characteristic Configuration table
  pwrDataConfig = (gattCharCfg_t *)ICall_malloc(sizeof(gattCharCfg_t) *
                                                 linkDBNumConns);
  if (pwrDataConfig == NULL)
  {
    return (bleMemAllocError);
  }

  // Register with Link DB to receive link status change callback
  GATTServApp_InitCharCfg(INVALID_CONNHANDLE, pwrDataConfig);


  // ---------- Add power factor service
  // Allocate Client Characteristic Configuration table
  pwrfDataConfig = (gattCharCfg_t *)ICall_malloc(sizeof(gattCharCfg_t) *
                                                 linkDBNumConns);
  if (pwrfDataConfig == NULL)
  {
    return (bleMemAllocError);
  }

  // Register with Link DB to receive link status change callback
  GATTServApp_InitCharCfg(INVALID_CONNHANDLE, pwrfDataConfig);


  // ---------- Add frequency service
  // Allocate Client Characteristic Configuration table
  freqDataConfig = (gattCharCfg_t *)ICall_malloc(sizeof(gattCharCfg_t) *
                                                 linkDBNumConns);
  if (freqDataConfig == NULL)
  {
    return (bleMemAllocError);
  }

  // Register with Link DB to receive link status change callback
  GATTServApp_InitCharCfg(INVALID_CONNHANDLE, freqDataConfig);


  // Register GATT attribute list and CBs with GATT Server App
  return GATTServApp_RegisterService( sensorAttrTable,
                                      GATT_NUM_ATTRS (sensorAttrTable),
                                      &sensorCBs);
}
```

1.2.3 Implementing “Metering_registerAppCB” Function

The next function is the “Metering_registerAppCBs”. This function is called to register the application callback function and is only called once when the application is initialized. This function does not vary according to the definition of the characteristics.

```
/*********************************************************************
 * @fn      Metering_registerAppCBs
 *
 * @brief   Registers the application callback function. Only call
 *          this function once.
 *
 * @param   callbacks - pointer to application callbacks.
 *
 * @return  SUCCESS or bleAlreadyInRequestedMode
 */
bStatus_t Metering_registerAppCBs(sensorCBs_t *appCallbacks)
{
  if (sensor_AppCBs == NULL)
  {
    if (appCallbacks != NULL)
    {
      sensor_AppCBs = appCallbacks;
    }

    return (SUCCESS);
  }

  return (bleAlreadyInRequestedMode);
}
```

1.2.4 Implementing “Metering_setParameter” Function

The next function to code is the “Metering_setParameter”. The “Metering_setParameter” function provides an interface for the application to update the value or values of the characteristics. For example, when the meter readings are read from the meter, the readings are parsed and the corresponding characteristic updates. Notification also triggers if the characteristic supports notification.

This function starts by checking the input parameter of the characteristic that the caller wants to update by using the switch case statement.

For a characteristic that does not support notification, this function merely checks if the data is valid, then copies the data to the corresponding characteristic’s data memory, and otherwise sets the return value with an error code. For the “meter properties” characteristic, view an example of what the handling looks like:

```
    case PROPERITIES_CHAR:
              if (len == METERING_PROP_LEN)
              {
                memcpy(propData, value, METERING_PROP_LEN);
              }
              else
              {
                ret = bleInvalidRange;
              }
    break;
```

For characteristics that support the notification capability, this function also triggers the notification. For the “power” characteristic, view an example of what the handling looks like:

```
    case POWER_CHAR:
              if (len == METERING_POWR_LEN)
              {
                memcpy(pwrData, value, METERING_POWR_LEN);
                // See if Notification has been enabled
                ret = GATTServApp_ProcessCharCfg(pwrDataConfig, pwrData, FALSE,
                                                 sensorAttrTable,
                                                 GATT_NUM_ATTRS(sensorAttrTable),
                                                 INVALID_TASK_ID, sensor_ReadAttrCB);
              }
              else
              {
                ret = bleInvalidRange;
              }
    break;
```

Here the “sensor_ReadAttrCB” is a function to implement after performing a series of other or additional steps . When generating a notification, the GATT layer calls to the sensor_ReadAttrCB. The sensor_ReadAttrCB is a function that receives the notification data. The GATT layer then takes the sensor_ReadAttrCB’s return value to begin the notification generation process.
Before implementing the “sensor_ReadAttrCB” function, the user must implement the “Metering_getParameter” function.

1.2.5 Implementing “Metering_getParameter” Function

The “Meter_getParameter” is very simple to implement. This function copies the specified parameter from the characteristic’s data memory to the memory location that the calling routine specifies. Thus the function checks for a specific parameter and then copies that parameter. This function is required when application wants to read the value of a characteristic.

```
/*********************************************************************
 * @fn      Metering_getParameter
 *
 * @brief   Get a Sensor Profile parameter.
 *
 * @param   param - Profile parameter ID
 * @param   value - pointer to data to put.  This is dependent on
 *          the parameter ID and WILL be cast to the appropriate
 *          data type (example: data type of uint16_t will be cast to
 *          uint16_t pointer).
 *
 * @return  bStatus_t
 */
```

```
bStatus_t Metering_getParameter(uint8_t param, void *value)
{
  bStatus_t ret = SUCCESS;

  switch (param)
  {
       case SENSOR_CONF:
              *((uint8_t*)value) = sensorEnable;
       break;

       case SENSOR_PERI:
              *((uint8_t*)value) = sensorPeriod;
       break;
       case  VOLTAGE_CHAR :
              memcpy(value, voltData, METERING_VOLT_LEN);
       break;

       case  CURRENT_CHAR :
              memcpy(value, currData, METERING_CURR_LEN);
       break;

       case  POWER_CHAR :
              memcpy(value, pwrData, METERING_POWR_LEN);
       break;

       case  POWER_FACT_CHAR :
              memcpy(value, pwrfData, METERING_PWRF_LEN);
       break;

       case  FREQUENCY_CHAR :
              memcpy(value, freqData, METERING_FREQ_LEN);
       break;

       case  POWER_STATE_CHAR :
              memcpy(value, pstaData, METERING_PSTA_LEN);
       break;

       case  PHASE_CHAR :
              memcpy(value, phasData, METERING_PHAS_LEN);
       break;

       case  NAME_CHAR :
              memcpy(value, nameData, METERING_NAME_LEN);
       break;

       case  VERSION_CHAR :
              memcpy(value, versData, METERING_VERS_LEN);
       break;

       case  PROPERITIES_CHAR :
              memcpy(value, propData, METERING_PROP_LEN);
       break;


    default:
      ret = INVALIDPARAMETER;
      break;
  }

  return (ret);
}
```

1.2.6 Implementing "sensor_ReadAttrCB" Function

As previously mentioned, the function of "sensor_ReadAttrCB" is for transferring data from the characteristic's data memory to the GATT layer through this callback. Before performing a copy, this callback function checks if the particular characteristic has the permission to read. If the particular characteristic has read permission, it proceeds to check the UUID of the characteristic; and if valid, the callback function performs the copy action.

```
/*********************************************************************
 * @fn          sensor_ReadAttrCB
 *
 * @brief       Read an attribute.
 *
 * @param       connHandle - connection message was received on
 * @param       pAttr - pointer to attribute
 * @param       pValue - pointer to data to be read
 * @param       pLen - length of data to be read
 * @param       offset - offset of the first octet to be read
 * @param       maxLen - maximum length of data to be read
 * @param       method - type of read message
 *
 * @return      SUCCESS, blePending or Failure
 */
static bStatus_t sensor_ReadAttrCB(uint16_t connHandle, gattAttribute_t *pAttr,
                                   uint8_t *pValue, uint8_t *pLen,
                                   uint16_t offset,
                                   uint8_t maxLen, uint8_t method)
{
  uint16_t uuid;
  bStatus_t status = SUCCESS;

  // If attribute permissions require authorization to read, return error
  if (gattPermitAuthorRead(pAttr->permissions))
  {
    // Insufficient authorization
    return (ATT_ERR_INSUFFICIENT_AUTHOR);
  }

  // Make sure it's not a blob operation (no attributes in the profile are long)
  if (offset > 0)
  {
    return (ATT_ERR_ATTR_NOT_LONG);
  }

  if (utilExtractUuid16(pAttr,&uuid) == FAILURE) {
    // Invalid handle
    *pLen = 0;
    return ATT_ERR_INVALID_HANDLE;
  }

  switch (uuid)
  {
    // No need for "GATT_SERVICE_UUID" or "GATT_CLIENT_CHAR_CFG_UUID" cases;
    // gattserverapp handles those reads
       case METERING_VOLT_UUID:
    *pLen = METERING_VOLT_LEN;
```

```
    memcpy(pValue, pAttr->pValue, METERING_VOLT_LEN);
    break;

       case METERING_CURR_UUID:
    *pLen = METERING_CURR_LEN;
    memcpy(pValue, pAttr->pValue, METERING_CURR_LEN);
    break;

       case METERING_POWR_UUID:
    *pLen = METERING_POWR_LEN;
    memcpy(pValue, pAttr->pValue, METERING_POWR_LEN);
    break;

       case METERING_PWRF_UUID:
    *pLen = METERING_PWRF_LEN;
    memcpy(pValue, pAttr->pValue, METERING_PWRF_LEN);
    break;

       case METERING_FREQ_UUID:
    *pLen = METERING_FREQ_LEN;
    memcpy(pValue, pAttr->pValue, METERING_FREQ_LEN);
    break;

       case METERING_PSTA_UUID:
    *pLen = METERING_PSTA_LEN;
    memcpy(pValue, pAttr->pValue, METERING_PSTA_LEN);
    break;

       case METERING_PHAS_UUID:
    *pLen = METERING_PHAS_LEN;
    pValue[0] = *pAttr->pValue;
    memcpy(pValue, pAttr->pValue, METERING_PHAS_LEN);
    break;

       case METERING_NAME_UUID:
    *pLen = METERING_NAME_LEN;
    pValue[0] = *pAttr->pValue;
    memcpy(pValue, pAttr->pValue, METERING_NAME_LEN);
    break;

       case METERING_VERS_UUID:
    *pLen = METERING_VERS_LEN;
    pValue[0] = *pAttr->pValue;
    memcpy(pValue, pAttr->pValue, METERING_VERS_LEN);
    break;

       case METERING_PROP_UUID:
    *pLen = METERING_PROP_LEN;
    pValue[0] = *pAttr->pValue;
    memcpy(pValue, pAttr->pValue, METERING_PROP_LEN);
    break;

    case SENSOR_ENABLE_UUID:
    case SENSOR_PERIOD_UUID:
      *pLen = 1;
      pValue[0] = *pAttr->pValue;
      break;

    default:
      *pLen = 0;
      status = ATT_ERR_ATTR_NOT_FOUND;
```

```
      break;
    }

  return (status);
}
```

1.2.7 Implementing "sensor_WriteAttrCB" Function

The last thing to implement in the *meteringservice.c* file is the sensor_WriteAttrCB function. Unlike its "Read" counterpart, the "sensor_WriteAttrCB" function is for copying the incoming data from the GATT layer to the specified characteristic's data memory. In addition, this function also indicates the application from which the data originates for the application to perform appropriate action. Similar to its "Read" counterpart, this callback function begins by checking if the specified characteristic allows write operation, then checks if the UUID is valid. Then the "sensor_WriteAttrCB" function performs the copy and notifies the application with a value change event.

```
/*********************************************************************
 * @fn      sensor_WriteAttrCB
 *
 * @brief   Validate attribute data prior to a write operation
 *
 * @param   connHandle - connection message was received on
 * @param   pAttr - pointer to attribute
 * @param   pValue - pointer to data to be written
 * @param   len - length of data
 * @param   offset - offset of the first octet to be written
 * @param   method - type of write message
 *
 * @return  SUCCESS, blePending or Failure
 */
static bStatus_t sensor_WriteAttrCB(uint16_t connHandle, gattAttribute_t *pAttr,
                                    uint8_t *pValue, uint8_t len,
                                    uint16_t offset,
                                    uint8_t method)
{
  bStatus_t status = SUCCESS;
  uint8_t notifyApp = 0xFF;
  uint16_t uuid;

  // If attribute permissions require authorization to write, return error
  if (gattPermitAuthorWrite(pAttr->permissions))
  {
    // Insufficient authorization
    return (ATT_ERR_INSUFFICIENT_AUTHOR);
  }

  if (utilExtractUuid16(pAttr,&uuid) == FAILURE) {
    // Invalid handle
    return ATT_ERR_INVALID_HANDLE;
  }

  switch (uuid)
  {
    case METERING_VOLT_UUID:
    case METERING_CURR_UUID:
    case METERING_POWR_UUID:
    case METERING_PWRF_UUID:
    case METERING_FREQ_UUID:
```

```
    case METERING_NAME_UUID:
    case METERING_VERS_UUID:
    case METERING_PROP_UUID:
      // Should not get here
      break;

    case SENSOR_ENABLE_UUID:
      // Validate the value
      // Make sure it's not a blob oper
      if (offset == 0)
      {
        if (len != 1)
        {
          status = ATT_ERR_INVALID_VALUE_SIZE;
        }
      }
      else
      {
        status = ATT_ERR_ATTR_NOT_LONG;
      }

      // Write the value
      if (status == SUCCESS)
      {
        uint8_t *pCurValue = (uint8_t *)pAttr->pValue;

        *pCurValue = pValue[0];

        if (pAttr->pValue == &sensorEnable)
        {
          notifyApp = SENSOR_CONF;
        }
      }
      break;

    case SENSOR_PERIOD_UUID:
      // Validate the value
      // Make sure it's not a blob oper
      if (offset == 0)
      {
        if (len != 1)
        {
          status = ATT_ERR_INVALID_VALUE_SIZE;
        }
      }
      else
      {
        status = ATT_ERR_ATTR_NOT_LONG;
      }
      // Write the value
      if (status == SUCCESS)
      {
        if (pValue[0]>=(SENSOR_MIN_UPDATE_PERIOD/SENSOR_PERIOD_RESOLUTION))
        {

          uint8_t *pCurValue = (uint8_t *)pAttr->pValue;
          *pCurValue = pValue[0];

          if (pAttr->pValue == &sensorPeriod)
          {
            notifyApp = SENSOR_PERI;
```

```
        }
      }
      else
      {
         status = ATT_ERR_INVALID_VALUE;
      }
    }
    break;

  case METERING_PSTA_UUID:
      // Validate the value
      // Make sure it's not a blob oper
      if (offset == 0)
      {
        if (len != 1)
        {
          status = ATT_ERR_INVALID_VALUE_SIZE;
        }
      }
      else
      {
        status = ATT_ERR_ATTR_NOT_LONG;
      }
      // Write the value
      if (status == SUCCESS)
      {
        if (pValue[0] < 2)
        {

          uint8_t *pCurValue = (uint8_t *)pAttr->pValue;
          *pCurValue = pValue[0];

          if (pAttr->pValue == pstaData)
          {
            notifyApp = POWER_STATE_CHAR;
          }
        }
        else
        {
           status = ATT_ERR_INVALID_VALUE;
        }
      }
     break;

  case METERING_PHAS_UUID:
      // Validate the value
      // Make sure it's not a blob oper
      if (offset == 0)
      {
        if (len != 1)
        {
          status = ATT_ERR_INVALID_VALUE_SIZE;
        }
      }
      else
      {
        status = ATT_ERR_ATTR_NOT_LONG;
      }
      // Write the value
      if (status == SUCCESS)
      {
```

```
          if (pValue[0] <= MaxPhase)
          {

            uint8_t *pCurValue = (uint8_t *)pAttr->pValue;
            *pCurValue = pValue[0];

            if (pAttr->pValue == pstaData)
            {
              notifyApp = PHASE_CHAR;
            }
          }
          else
          {
             status = ATT_ERR_INVALID_VALUE;
          }
        }
       break;

    case GATT_CLIENT_CHAR_CFG_UUID:
      status = GATTServApp_ProcessCCCWriteReq(connHandle, pAttr, pValue, len,
                                              offset, GATT_CLIENT_CFG_NOTIFY);
      break;

    default:
      // Should never get here!
      status = ATT_ERR_ATTR_NOT_FOUND;
      break;
  }

  // If a charactersitic value changed then callback function
  // to notify application of change
  if ((notifyApp != 0xFF ) && sensor_AppCBs && sensor_AppCBs->pfnSensorChange)
  {
    sensor_AppCBs->pfnSensorChange(notifyApp);
  }

  return (status);
}
```

2 Metering Application

After creating the metering service in the profile, the next step is to create the metering application that actually interacts with the meter. To create this metering application, create a *SensorTag_Metering.h* file and a *SensorTag_Metering.c* file in this relative directory path:
"ORG_PROJ_DIR\..\..\..\Source\Application\"

The content of the *SensorTag_Metering.h* file is simply a few definitions of function prototypes that the following code example lists:

```
#ifndef SENSORTAGMETERING_H
#define SENSORTAGMETERING_H

#ifdef __cplusplus
extern "C"
{
#endif
```

```
/*********************************************************************
 * INCLUDES
 */
#include "sensortag.h"

/*********************************************************************
 * CONSTANTS
 */

/*********************************************************************
 * MACROS
 */

/*********************************************************************
 * FUNCTIONS
 */

/*
 * Create the IR temperature sensor task
 */
void SensorTagMetering_createTask(void);

/*
 * Task Event Processor for characteristic changes
 */
extern void SensorTagMetering_ProcessCharChangeEvt(void);

/*
 * Task Event Processor for the BLE Application
 */
extern void SensorTagMetering_reset( void);

/*********************************************************************
*********************************************************************/

#ifdef __cplusplus
}
#endif

#endif /* SENSORTAGMETERING_H */
```

2.1 Implementing SensorTag_Metering.c

2.1.1 "SensorTagMetering_createTask" Function

This function is the interface to the initialization part of the main() function, for which the metering application is considered a standalone, running task. This function sets up and creates the task for the underlying operating system:

```
/*********************************************************************
 * @fn      SensorTagMetering_createTask
 *
 * @brief   Task creation function for the SensorTag
 *
 * @param   none
 *
 * @return  none
 */
```

```
void SensorTagMetering_createTask(void)
{
  Task_Params taskParames;

  // Create the task for the state machine
  Task_Params_init(&taskParames);
  taskParames.stack = sensorTaskStack;
  taskParames.stackSize = SENSOR_TASK_STACK_SIZE;
  taskParames.priority = SENSOR_TASK_PRIORITY;

  Task_construct(&sensorTask, sensorTaskFxn, &taskParames, NULL);
}
```

Note that there are parameters that require defining in advance, such as the "SENSOR_TASK_STACK_SIZE" and "SENSOR_TASK_PRIORITT" parameters. To allow proper operation of the application, the SENSOR_TASK_PRIORITY must be set to "1". The SENSOR_STACK_SIZE is selected as "1024". There is also a parameter for creating the task: "sensorTaskFxn" – the loop that runs the metering application.

```
// Task configuration
#define SENSOR_TASK_PRIORITY    1
#define SENSOR_TASK_STACK_SIZE  1024
```

2.1.2 Implementing "SenosrTagMetering_ProcessCharChangeEvt" Function

Section 3.3.2.1.7 explains how the sensor_WriteAttrCB function notifies the application whenever writable characteristic are written from the *Bluetooth* LE client. The "SensorTagMetering_ProcessCharChangeEvt" function receives the notification and reacts. In this example, the application takes action on three of the writable characteristics. For example, if the value to write is "SENSOR_CONF" (which means enable the metering function), the enable status is read and then stored into the sensorConfig variable. The data value for the meter characteristics are cleared if the value written is to disable the sensor.

```
/*********************************************************************
 * @fn      SensorTagMetering_ProcessCharChangeEvt
 *
 * @brief   SensorTag Metering event handling
 *
 */
void SensorTagMetering_ProcessCharChangeEvt(void)
{
       uint8_t newValue;

  if (charChangePending)
  {

    switch (paramID)
    {
    case SENSOR_CONF:
      if ((sensorTestResult() & ST_METER) == 0)
      {
        sensorConfig = ST_CFG_ERROR;
      }

      if (sensorConfig != ST_CFG_ERROR)
```

```
      {
        Metering_getParameter(SENSOR_CONF, &newValue);

        if (newValue == ST_CFG_SENSOR_DISABLE)
        {
          // Reset characteristics
          initCharacteristicValue(VOLTAGE_CHAR, 0, METERING_VOLT_LEN);
          initCharacteristicValue(CURRENT_CHAR, 0, METERING_CURR_LEN);
          initCharacteristicValue(POWER_CHAR, 0, METERING_POWR_LEN);
          initCharacteristicValue(POWER_FACT_CHAR, 0, METERING_PWRF_LEN);
          initCharacteristicValue(FREQUENCY_CHAR, 0, METERING_FREQ_LEN);
          initCharacteristicValue(POWER_STATE_CHAR, 0, METERING_PSTA_LEN);
          initCharacteristicValue(PHASE_CHAR, 0, METERING_PHAS_LEN);
        }

        sensorConfig = newValue;
      }
      else
      {
        // Make sure the previous characteristics value is restored
        initCharacteristicValue(SENSOR_CONF, sensorConfig, sizeof ( uint8_t ));
      }
      break;

    case SENSOR_PERI:
      Metering_getParameter(SENSOR_PERI, &newValue);
      sensorPeriod = newValue * SENSOR_PERIOD_RESOLUTION;
      break;

    case POWER_STATE_CHAR:
        Metering_getParameter (PHASE_CHAR, &PendingSwitchPhase);
        Metering_getParameter (POWER_STATE_CHAR, &PendingSwitchValue);
        PendingSwitch = 1;
        break;

    default:
      // Should not get here
      break;
    }

    charChangePending = false;
  }
}
```

Here the user must #define ST_METER and add it to the *sensor.h* file. This file is located in SensorTag→Board→Interfaces→*sensor.h*

```
/* Bit values for power on self-test */
#define ST_IRTEMP                       0x01
#define ST_HUMIDITY                     0x02
#define ST_LIGHT                        0x04
#define ST_PRESSURE                     0x08
#define ST_MPU                          0x10
#define ST_METER                        0x20
```

2.1.3 "SenosrTagMetering_reset" Function

This function provides an access point to reset the characteristic's data memory.

```
/*********************************************************************
 * @fn      SensorTagMetering_reset
 *
 * @brief   Reset characteristics
 *
 * @param   none
 *
 * @return  none
 */
void SensorTagMetering_reset(void)
{
  sensorConfig = ST_CFG_SENSOR_DISABLE;
  sensorPeriod = SENSOR_DEFAULT_PERIOD;
  initCharacteristicValue(VOLTAGE_CHAR, 0, METERING_VOLT_LEN);
  initCharacteristicValue(CURRENT_CHAR, 0, METERING_CURR_LEN);
  initCharacteristicValue(POWER_CHAR, 0, METERING_POWR_LEN);
  initCharacteristicValue(POWER_FACT_CHAR, 0, METERING_PWRF_LEN);
  initCharacteristicValue(FREQUENCY_CHAR, 0, METERING_FREQ_LEN);
  initCharacteristicValue(POWER_STATE_CHAR, 0, METERING_PSTA_LEN);
  initCharacteristicValue(PHASE_CHAR, 0, METERING_PHAS_LEN);
  initCharacteristicValue(SENSOR_CONF, sensorConfig, sizeof(uint8_t));

}
```

2.1.4 "sensorTaskInit" Function

The "sensorTaskInit" is a private function that is called once only when the task is initialized. This function registers the metering service to the *Bluetooth* LE stack and registers the callbacks to the service. The sensorTaskInit then performs initialization of variables, initializing the metering protocol stack and UART (*dlt645_init*).

```
/*********************************************************************
 * @fn      sensorTaskInit
 *
 * @brief   Initialization function for the SensorTag Meter
 *
 */
static void sensorTaskInit(void)
{
  // Add service
  Metering_addService();

  // Register callbacks with profile
  Metering_registerAppCBs(&sensorCallbacks);

  // Initilialize the module state variables
  sensorPeriod = SENSOR_DEFAULT_PERIOD;
  paramID = 0;
  charChangePending = false;

  // Initialise characteristics and sensor driver
  SensorTagMetering_reset();
  initCharacteristicValue(SENSOR_PERI,
                          SENSOR_DEFAULT_PERIOD / SENSOR_PERIOD_RESOLUTION,
                          sizeof ( uint8_t ));
```

```
  // Initialise the driver
    dlt645_init ();
}
```

2.1.5 "sensorTaskFxn" Function

At this point in the coding process, the essential interface has been set up. The next step is coding the 'main loop' of the metering application. This is the loop that interacts with the metering module through the UART.
The task function code starts with the declaration of variables to hold the meter readings, the password to pass to the meter, and the operating state of the meter.

```
static void sensorTaskFxn(UArg a0, UArg a1)
{
  volatile char MeterName[32];
  volatile Reading MeterReading;
  volatile VersionData MeterVersion;
  volatile ConfigurationData MeterProperty;
  volatile int32_t PowerTemp[3];
  volatile uint8_t PowerState;

  uint8_t Password[8] = {0x34, 0x12, 0x78, 0x56, 0xBC, 0x9A, 0xF0, 0xDE};
  int16_t MeterReady = 0;
  int16_t UsePhase = 0;
```

Then sensorTaskFxn calls to the operating system to register this application and initialize the task itself.

```
  // Register task with BLE stack
  ICall_registerApp(&sensorSelfEntity, &sensorSem);

  // Initialize the task
  sensorTaskInit();
```

After this initialization, the sensorTaskFxn attempts to connect to the meter by sending the password to the meter. Then the sensorTaskFxn attempts to read the meter property, meter name, and meter version:

```
  if (CmdSetPassword(Password) == 0)
  {
        MeterReady = 1;
        MeterProperty.Configuration.MeterReady = MeterReady;
        if (CmdGetMeterName(MeterName) == 0)
        {
              Metering_setParameter(NAME_CHAR, METERING_NAME_LEN, MeterName);
        }
        else
        {
              Metering_setParameter(NAME_CHAR, METERING_NAME_LEN,
                                    "Unknown Meter                    ");
        }

        if (CmdGetMeterVersion(&MeterVersion) == 0)
        {
              Metering_setParameter(VERSION_CHAR, METERING_VERS_LEN, &MeterVersion);
        }
```

```
        else
        {
               Metering_setParameter(VERSION_CHAR, METERING_VERS_LEN, "0000000000000000");
        }

        if (CmdGetMeterConfiguration(&MeterProperty) == 0)
        {
               Metering_setParameter(PROPERITIES_CHAR, METERING_PROP_LEN, &MeterProperty);
               MaxPhase = MeterProperty.Configuration.NumberOfPhases;
        }
        else
        {
               Metering_setParameter(PROPERITIES_CHAR, METERING_PROP_LEN,
                              "Unknown Properties              ");
        }
  }
```

The next step is to enter into the 'main loop'. Here the meter continuously loops through read and waits for a delay of METERING_MEAS_DELAY milliseconds. If the meter is disconnected due to any reason, the main loop attempts to reconnect to the meter.

```
// Task loop
  while (true)
  {
    if (sensorConfig == ST_CFG_SENSOR_ENABLE)
    {
      // Read data
      delay_ms(METERING_MEAS_DELAY);
      if (MeterReady)
      {
         if (Metering_getParameter (PHASE_CHAR, &UsePhase) == SUCCESS)
         {
               if (CmdGetReadings(UsePhase, &MeterReading) == 0)
               {
                     // Update GATT
                     Metering_setParameter(VOLTAGE_CHAR, METERING_VOLT_LEN,
                                           &MeterReading.Reading.Voltage);
                     Metering_setParameter(CURRENT_CHAR, METERING_CURR_LEN,
                                           &MeterReading.Reading.Current);
                       PowerTemp[0] = MeterReading.Reading.ActivePower;
                       PowerTemp[1] = MeterReading.Reading.ReactivePower;
                       PowerTemp[2] = MeterReading.Reading.ApparentPower;
                       Metering_setParameter(POWER_CHAR, METERING_POWR_LEN, PowerTemp);
                       Metering_setParameter(POWER_FACT_CHAR, METERING_PWRF_LEN,
                                           &MeterReading.Reading.PowerFacotr);
                       Metering_setParameter(FREQUENCY_CHAR, METERING_FREQ_LEN,
                                           &MeterReading.Reading.Frequency);
                       if (PendingSwitch == 1)
                       {
                            CmdSetSocketStatus(PendingSwitchPhase-1, PendingSwitchValue);
                            PendingSwitch = 0;
                       }

                       if (CmdGetSocketStatus (&PowerState) == 0)
                       {
                              if ((PowerState & (0x01 << (UsePhase-1))) != 0)
                              {
                                     PowerState = 1;
                              }
                              else
```

```
                        {
                                PowerState = 0;
                        }
                        Metering_setParameter(POWER_STATE_CHAR,
                                              METERING_PSTA_LEN,
                                              &PowerState);
                }
            }
            else
            {
                MeterReady = 0;
                MeterProperty.Configuration.MeterReady = MeterReady;
            }
        }
    }
    else
    {
        if (CmdSetPassword(Password) == 0)
        {
            MeterReady = 1;
            MeterProperty.Configuration.MeterReady = MeterReady;
            if (CmdGetMeterName(MeterName) == 0)
            {
                Metering_setParameter(NAME_CHAR, METERING_NAME_LEN, MeterName);
            }
            else
            {
                Metering_setParameter(NAME_CHAR, METERING_NAME_LEN,
                                      "Unknown Meter                    ");
            }

            if (CmdGetMeterVersion(&MeterVersion) == 0)
            {
                Metering_setParameter(VERSION_CHAR, METERING_VERS_LEN,
                                      &MeterVersion);
            }
            else
            {
                Metering_setParameter(VERSION_CHAR, METERING_VERS_LEN,
                                      "0000000000000000");
            }

            if (CmdGetMeterConfiguration(&MeterProperty) == 0)
            {
                Metering_setParameter(PROPERITIES_CHAR, METERING_PROP_LEN,
                                      &MeterProperty);
                  MaxPhase = MeterProperty.Configuration.NumberOfPhases;
            }
            else
            {
                Metering_setParameter(PROPERITIES_CHAR, METERING_PROP_LEN,
                                      "Unknown Properties               ");
            }
        }
    }

    // Next cycle
    delay_ms(sensorPeriod - METERING_MEAS_DELAY);
}
else
{
```

```
      delay_ms(SENSOR_DEFAULT_PERIOD);
    }
  }
}
```

2.1.6 "sensorChangeCB" and "initCharacteristicValue" Functions

In addition to the task function loop, there are two more small functions to insert into the *SensorTag_Metering.c* file to make the structure of the metering application code consistent with application code in the project for other sensors.

The "sensorChangeCB" is a callback function that is called when there is a value change to the writable values (see the code in the *meteringservice.c* file). This call causes further calls to the "SensorTagMetering_ProcessCharChangeEvt" function.

```
/*********************************************************************
 * @fn      sensorChangeCB
 *
 * @brief   Callback from metering Service indicating a value change
 *
 * @param   paramID - parameter ID of the value that was changed.
 *
 * @return  none
 */
static void sensorConfigChangeCB(uint8_t paramIDChanged)
{
  paramID = paramIDChanged;
  charChangePending = true;

  // Wake up the application thread
  SensorTag_charValueChangeCB(SERVICE_ID_METERING);
}
```

The last function to implement is the "initCharacteristicValue" function, which initializes the characteristic value memory to an initial value.

```
/*********************************************************************
 * @fn      initCharacteristicValue
 *
 * @brief   Initialize a characteristic value
 *
 * @param   paramID - parameter ID of the value is to be cleared
 *
 * @param   value - value to initialise with
 *
 * @param   paramLen - length of the parameter
 *
 * @return  none
 */
static void initCharacteristicValue(uint8_t paramID, uint8_t value,
                                    uint8_t paramLen)
{
       char data[12] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
       Metering_setParameter( paramID, paramLen, data);
}
```

3 Changes to SensorTag Application Code

In addition to the above, the SensorTag application code requires a few changes to make the added metering service functional.

3.1 deviceinfoservice-st.c File

Change the highlighted lines

```
/*********************************************************************
 * Profile Attributes - variables
 */

// Device Information Service attribute
static CONST gattAttrType_t devInfoService = {ATT_BT_UUID_SIZE, devInfoServUUID};

// System ID characteristic
static uint8_t devInfoSystemIdProps = GATT_PROP_READ;
static uint8_t devInfoSystemId[DEVINFO_SYSTEM_ID_LEN] = {0, 0, 0, 0, 0, 0, 0, 0};

// Model Number String characteristic
static uint8_t devInfoModelNumberProps = GATT_PROP_READ;
static const uint8_t devInfoModelNumber[] = "CC2650 SensorTagPlusMetering";

// Serial Number String characteristic
static uint8_t devInfoSerialNumberProps = GATT_PROP_READ;
static const uint8_t devInfoSerialNumber[] = "N.A.";

// Software Revision String characteristic
static uint8_t devInfoSoftwareRevProps = GATT_PROP_READ;
static const uint8_t devInfoSoftwareRev[] = "N.A.";

// Firmware Revision String characteristic
static uint8_t devInfoFirmwareRevProps = GATT_PROP_READ;

// Firmware revision string
#define STRINGIFY(x) #x
#define TOSTRING(x) STRINGIFY(x)
//static const uint8_t devInfoFirmwareRev[] = TOSTRING(VERSION)" ("__DATE__")";
static const uint8_t devInfoFirmwareRev[] = "0504";
```

3.2 In *main.c* File

Add: #include "SensorTag_Metering.h" in the include list

```
// BLE
#include "bcomdef.h"

// Modules with their own tasks
#include "SensorTag.h"
#include "SensorTag_Tmp.h"
#include "SensorTag_Hum.h"
#include "SensorTag_Bar.h"
#include "SensorTag_Metering.h"
```

Add “SensorTagMetering_createTask();” in the task creation list

```
  /* Initialize ICall module */
  ICall_init();

  /* Start tasks of external images - Priority 5 */
  ICall_createRemoteTasks();

  /* Kick off profile - Priority 3 */
  GAPRole_createTask();

  /* Kick off application - Priority 1 */
  SensorTag_createTask();
  SensorTagTmp_createTask();
  SensorTagHum_createTask();
  SensorTagBar_createTask();
  SensorTagMetering_createTask();
```

3.3 SensorTag.c File

Add: #include "sensortag_metering.h" to the include list

```
// Sensor devices
#include "st_util.h"
#include "sensortag_tmp.h"
#include "sensortag_hum.h"
#include "sensortag_bar.h"
#include "sensortag_mov.h"
#include "sensortag_opt.h"
#include "sensortag_keys.h"
#include "sensortag_io.h"
#include "sensortag_metering.h"
// Other devices
#include "ext_flash.h"
#include "ext_flash_layout.h"
```

Add the highlighted lines at the indicated location

```
/*********************************************************************
 * @fn      SensorTag_processCharValueChangeEvt
 *
 * @brief   Process pending Profile characteristic value change
 *          events. The events are generated by the network task (BLE)
 *
 * @param   serviceID - ID of the affected service
 *
 * @return  none
 */
static void SensorTag_processCharValueChangeEvt(uint8_t serviceID)
{
  switch (serviceID)
  {
        case SERVICE_ID_TMP:
               SensorTagTmp_ProcessCharChangeEvt();
        break;

        case SERVICE_ID_HUM:
               SensorTagHum_ProcessCharChangeEvt();
        break;

        case SERVICE_ID_BAR:
               SensorTagBar_processCharChangeEvt();
        break;
```

```
        case SERVICE_ID_MOV:
                SensorTagMov_ProcessCharChangeEvt();
        break;

        case SERVICE_ID_OPT:
                SensorTagOpt_ProcessCharChangeEvt();
        break;

        case SERVICE_ID_IO:
                SensorTagIO_processCharChangeEvt();
        break;

        case SERVICE_ID_METERING:
                SensorTagMetering_ProcessCharChangeEvt();
        break;
```

Add highlighted lines at the indicated location

```
/*********************************************************************
 * @fn      SensorTag_resetAllSensors
 *
 * @brief   Reset all sensors, typically when a connection is intentionally
 *          terminated.
 *
 * @param   none
 *
 * @return  none
 */
static void SensorTag_resetAllSensors(void)
{
  SensorTagTmp_reset();
  SensorTagHum_reset();
  SensorTagBar_reset();
  SensorTagMov_reset();
  SensorTagOpt_reset();
  SensorTagIO_reset();
  SensorTagMetering_reset();
}
```

3.4 Sensortag.h File

Add the highlighted line

```
/*********************************************************************
 * INCLUDES
 */
#include "ICall.h"
#include "peripheral.h"
#include <ti/sysbios/knl/Clock.h>
#include <ti/drivers/PIN.h>

/*********************************************************************
 * CONSTANTS
 */

// Service ID's for internal application use
#define SERVICE_ID_TMP       0x01
#define SERVICE_ID_OPT       0x02
#define SERVICE_ID_MOV       0x04
```

```
#define SERVICE_ID_HUM        0x05
#define SERVICE_ID_BAR        0x06
#define SERVICE_ID_IO         0x07
#define SERVICE_ID_KEYS       0x08
#define SERVICE_ID_CC         0x09
#define SERVICE_ID_METERING   0x0a
```

4 What's Next ?

After the code is edit follow the procedure in section 3.6 and further in the TIDC-BLE-METER-READING user guide to proceed with compilation, downloading and testing.

IMPORTANT NOTICE FOR TI REFERENCE DESIGNS

Texas Instruments Incorporated ("TI") reference designs are solely intended to assist designers (“Buyers”) who are developing systems that incorporate TI semiconductor products (also referred to herein as “components”). Buyer understands and agrees that Buyer remains responsible for using its independent analysis, evaluation and judgment in designing Buyer’s systems and products.

TI reference designs have been created using standard laboratory conditions and engineering practices. **TI has not conducted any testing other than that specifically described in the published documentation for a particular reference design.** TI may make corrections, enhancements, improvements and other changes to its reference designs.

Buyers are authorized to use TI reference designs with the TI component(s) identified in each particular reference design and to modify the reference design in the development of their end products. HOWEVER, NO OTHER LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE TO ANY OTHER TI INTELLECTUAL PROPERTY RIGHT, AND NO LICENSE TO ANY THIRD PARTY TECHNOLOGY OR INTELLECTUAL PROPERTY RIGHT, IS GRANTED HEREIN, including but not limited to any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI components or services are used. Information published by TI regarding third-party products or services does not constitute a license to use such products or services, or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

TI REFERENCE DESIGNS ARE PROVIDED "AS IS". TI MAKES NO WARRANTIES OR REPRESENTATIONS WITH REGARD TO THE REFERENCE DESIGNS OR USE OF THE REFERENCE DESIGNS, EXPRESS, IMPLIED OR STATUTORY, INCLUDING ACCURACY OR COMPLETENESS. TI DISCLAIMS ANY WARRANTY OF TITLE AND ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, QUIET ENJOYMENT, QUIET POSSESSION, AND NON-INFRINGEMENT OF ANY THIRD PARTY INTELLECTUAL PROPERTY RIGHTS WITH REGARD TO TI REFERENCE DESIGNS OR USE THEREOF. TI SHALL NOT BE LIABLE FOR AND SHALL NOT DEFEND OR INDEMNIFY BUYERS AGAINST ANY THIRD PARTY INFRINGEMENT CLAIM THAT RELATES TO OR IS BASED ON A COMBINATION OF COMPONENTS PROVIDED IN A TI REFERENCE DESIGN. IN NO EVENT SHALL TI BE LIABLE FOR ANY ACTUAL, SPECIAL, INCIDENTAL, CONSEQUENTIAL OR INDIRECT DAMAGES, HOWEVER CAUSED, ON ANY THEORY OF LIABILITY AND WHETHER OR NOT TI HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, ARISING IN ANY WAY OUT OF TI REFERENCE DESIGNS OR BUYER’S USE OF TI REFERENCE DESIGNS.

TI reserves the right to make corrections, enhancements, improvements and other changes to its semiconductor products and services per JESD46, latest issue, and to discontinue any product or service per JESD48, latest issue. Buyers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All semiconductor products are sold subject to TI’s terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its components to the specifications applicable at the time of sale, in accordance with the warranty in TI’s terms and conditions of sale of semiconductor products. Testing and other quality control techniques for TI components are used to the extent TI deems necessary to support this warranty. Except where mandated by applicable law, testing of all parameters of each component is not necessarily performed.

TI assumes no liability for applications assistance or the design of Buyers’ products. Buyers are responsible for their products and applications using TI components. To minimize the risks associated with Buyers’ products and applications, Buyers should provide adequate design and operating safeguards.

Reproduction of significant portions of TI information in TI data books, data sheets or reference designs is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Buyer acknowledges and agrees that it is solely responsible for compliance with all legal, regulatory and safety-related requirements concerning its products, and any use of TI components in its applications, notwithstanding any applications-related information or support that may be provided by TI. Buyer represents and agrees that it has all the necessary expertise to create and implement safeguards that anticipate dangerous failures, monitor failures and their consequences, lessen the likelihood of dangerous failures and take appropriate remedial actions. Buyer will fully indemnify TI and its representatives against any damages arising out of the use of any TI components in Buyer’s safety-critical applications.

In some cases, TI components may be promoted specifically to facilitate safety-related applications. With such components, TI’s goal is to help enable customers to design and create their own end-product solutions that meet applicable functional safety standards and requirements. Nonetheless, such components are subject to these terms.

No TI components are authorized for use in FDA Class III (or similar life-critical medical equipment) unless authorized officers of the parties have executed an agreement specifically governing such use.

Only those TI components that TI has specifically designated as military grade or “enhanced plastic” are designed and intended for use in military/aerospace applications or environments. Buyer acknowledges and agrees that any military or aerospace use of TI components that have ***not*** been so designated is solely at Buyer's risk, and Buyer is solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI has specifically designated certain components as meeting ISO/TS16949 requirements, mainly for automotive use. In any case of use of non-designated products, TI will not be responsible for any failure to meet ISO/TS16949.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265