PSoC 4 BLE – Low-power Health Thermometer

Objective

This example demonstrates the use of the BLE Component to design a low-power Health Thermometer BLE application.

Overview

This project demonstrates the use of the BLE Component to implement a Standard BLE Health Thermometer Profile for designing a BLE Health Thermometer application. The device measures the ambient temperature using a thermometer and sends it over the BLE. It also demonstrates how to design a low-power application using te BLE Component.

This project also provides an option to simulate the temperature data, increments by 1 ^{0}C per measurement interval (default value as 1 second).

Requirements

Design Tool: PSoC Creator 3.1 CP1, CySmart 1.0

Programming Language: C (GCC 4.8.4 – included with PSoC Creator)

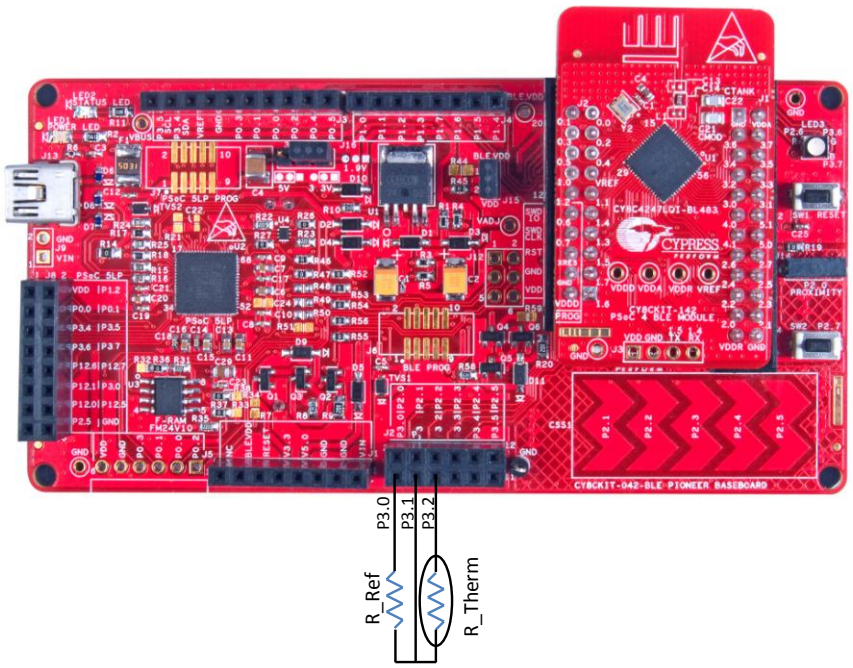
Associated Devices: All PSoC 4 BLE devices

Required Hardware: CY8CKIT-042-BLE Bluetooth® Low Energy (BLE) Pioneer Kit

Hardware Setup

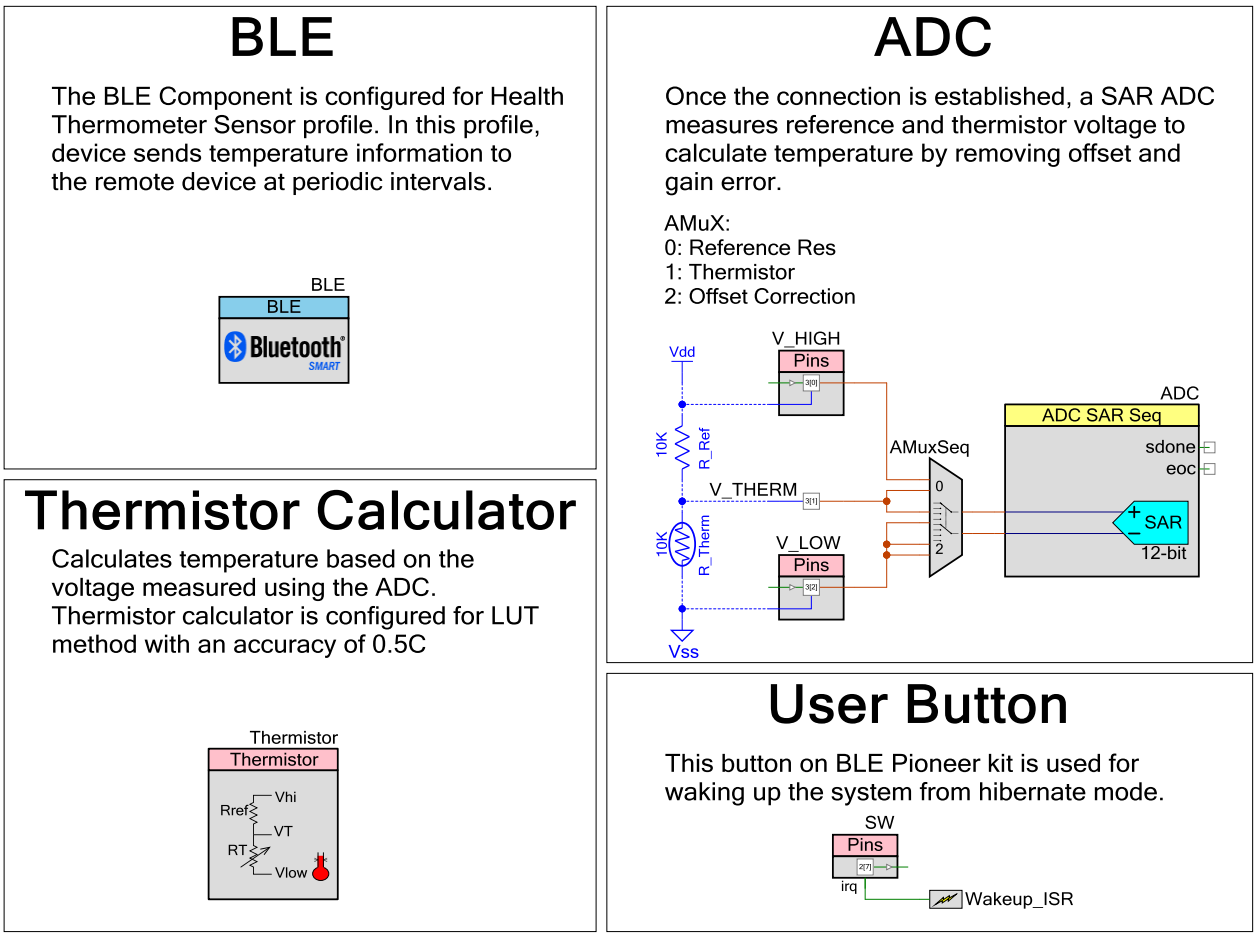
The BLE Pioneer Kit has all of the necessary hardware required for this example. Figure 1 shows the hardware setup for this example. Connect a 10KΩ reference resistor between P3.0 and P3.1 and a 10KΩ thermistor between P3.1 and P3.2.

Figure 1: Kit Setup



PSoC Creator Schematic

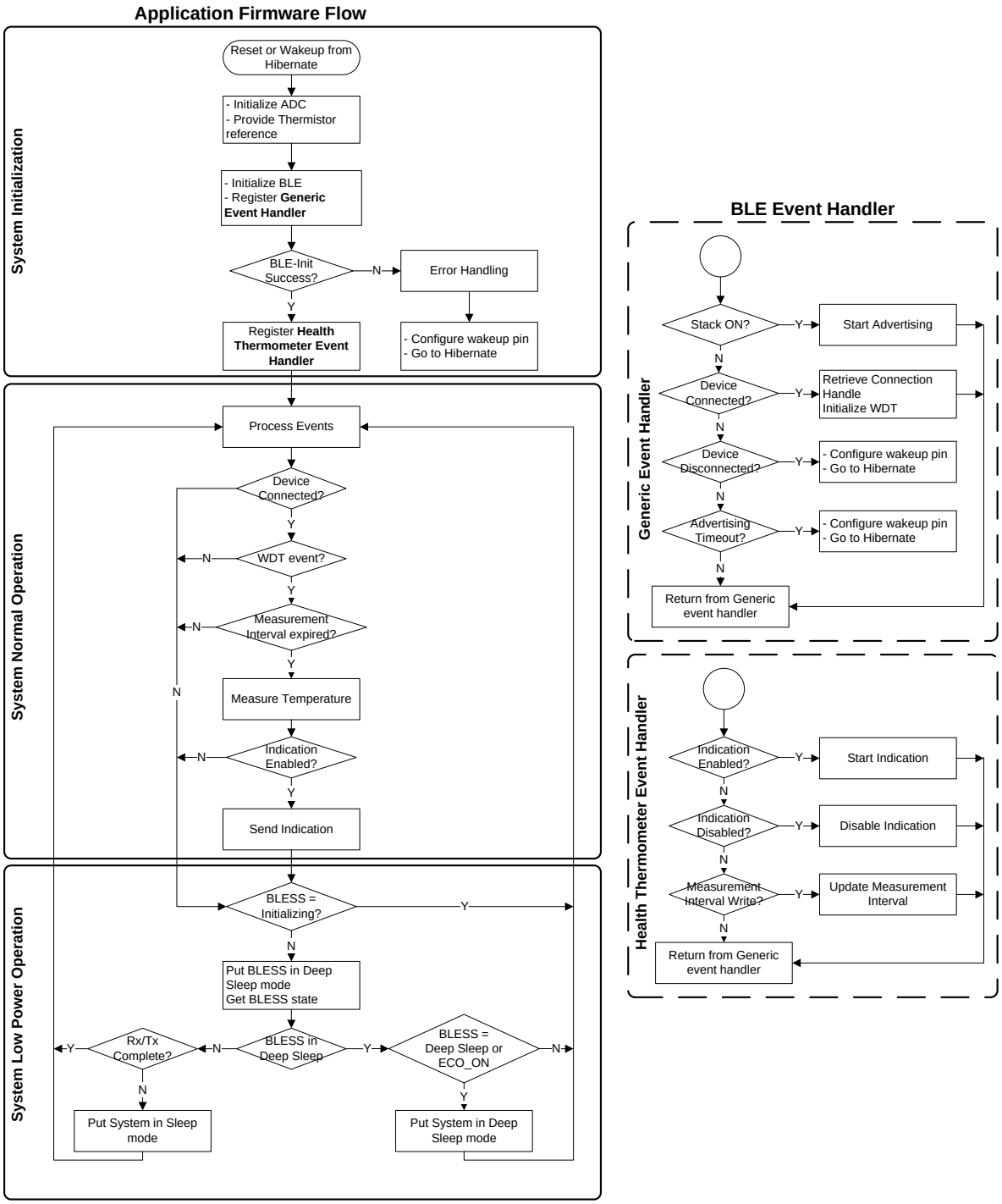
Figure 2. PSoC Creator Schematic



Operation

Figure 3 shows the firmware flow of this example.

Figure 3: Operation Flow



The Health Thermometer application code state machine consists of four states:

- System Initialization
- Event Handler
  - Generic Event Handler
  - Health Thermometer Event Handler
- System Normal operation
- System Low-power operation

These states are discussed in detail in the following sections.

System Initialization

When the device is reset or wakes up from the hibernate mode, the firmware performs the initialization, which includes the starting of the SAR ADC, the enabling global interrupts, starting opamps, and starting the watchdog timer. After the system is initialized, it initializes the BLE Component, which handles the initialization of the complete BLE subsystem.

Note As a part of the BLE Component initialization, the user code must pass a pointer to the event handler function which should be called to receive events from the BLE stack. The Generic Event Handler shown in **Error! Reference source not found.** is registered as a part of the BLE initialization.

Code 1 shows the code to start the BLE Component and register the Generic Event Handler.

Code 1. BLE Initialization

```
apiResult = CyBle_Start(GenericEventHandler);
```

If the BLE Component initializes successfully, the firmware registers the function that is called to receive events for the Health Thermometer Service and switches to the normal operation mode. Code 2 shows the snippet for registering the Health Thermometer Service.

Code 2. Health Thermometer Service Event Handler

```
CyBle_HtsRegisterAttrCallback(HealthThermometetEventHandler);
```

Event Handler

In the BLE Component, results of any operation performed on the BLE stack are relayed to the application firmware via a list of events. These events provide the BLE interface status and data information. Events can be categorized as follows:

- Common events

  Operations performed at the GAP layer, the GATT layer, and the L2CAP layer of the stack generate these events. For example, a CYBLE_EVT_STACK_ON event is received when the BLE stack is initialized and turned ON; a CYBLE_EVT_GAP_DEVICE_CONNECTED event is received when a connection with a remote device is established, and a CYBLE_EVT_GATTS_WRITE_CMD_REQ event is generated when a “Write Command” is received from the client. For more details on common events, see the API documentation of the BLE Component (right-click the BLE Component in PSoC Creator and select the **Open API Documentation** option).

  The application firmware must include an event handler function to handle these events and to be able to successfully establish and maintain the BLE link.

  Code 3 shows the implementation of the GenericEventHandler, where events generated on initialization of the BLE stack, device connection and disconnection, and timeout are handled.

- Service-specific events

  Service-specific events are generated because of operations performed on the standard services defined by the Bluetooth SIG. For example, a CYBLE_EVT_HTSS_INDICATION_ENABLED event is received by the server when the client writes the client configuration characteristic descriptor to enable the indication for the Temperature Measurement Characteristic. For more details on the service specific events, see the API documentation of the BLE Component.

The BLE Component can route these events to a service-specific event handler. It is recommended that the application firmware should include a service-specific event handler function to handle these events. If a service-specific event handler is not supported, then these events must be handled by the common event handler (GenericEventHandler).

Code 4 shows the implementation of HtssEventHandler.

Code 3. Generic Event Handler

```
void GenericEventHandler(uint32 event, void *eventParam)
{
  switch(event)
  {
    /* This event is received when component is Started */
    case CYBLE_EVT_STACK_ON:
      /* Stop watchdog to reduce power consumption during advertising */
      WatchdogTimer_Stop();
      /* Start Advertisement and enter Discoverable mode*/
      CyBle_GappStartAdvertisement(CYBLE_ADVERTISING_FAST);
    break;

    /* This event is received when device is disconnected or advetising times out*/
    case CYBLE_EVT_GAP_DEVICE_DISCONNECTED:
    case CYBLE_EVT_TIMEOUT:
      /* Sets the ENABLE_HIBERNATE flag to put system in Hibernate mode */
      SystemFlag |= ENABLE_HIBERNATE;
    break;

    /* This event is received when connection is established */
    case CYBLE_EVT_GATT_CONNECT_IND:
        /* Start watchdog timer with 1s refresh interval */
        /* Note: For this application, wakeup should be 1s because htssInterval
        *  resolution is configured as 1s */
        WatchdogTimer_Start(REFRESH_INTERVAL);
        /* Retrieve BLE connection handle */
        connectionHandle = *(CYBLE_CONN_HANDLE_T *) eventParam;
    break;

    default:
      /* Error handling */
    break;
  }
}
```

Code 4. Health Thermometer Service Event Handler

```
void HtssEventHandler(uint32 event, void* eventParam)
{
  CYBLE_HTS_CHAR_VALUE_T *interval;
  switch(event)
  {
    /* This event is received when indication are enabled by the central */
    case CYBLE_EVT_HTSS_INDICATION_ENABLED:
      /* Set the htssIndication flag */
      htssIndication = true;
    break;

    /* This event is received when indication are enabled by the central */
    case CYBLE_EVT_HTSS_INDICATION_DISABLED:
      /* Reset the htssIndiciation flag */
      htssIndication = false;
    break;

    /* This event is received when measurement interval is updated by
    *  the central */
    case CYBLE_EVT_HTSS_CHAR_WRITE:
      /* Retrive interval value */
      interval = ((CYBLE_HTS_CHAR_VALUE_T *)eventParam);
      htssInterval = interval->value->val[1];
      /* Update htssInterval with the updated value */
      htssInterval = (htssInterval << 8) | interval->value->val[0];
    break;

    default:
      /* Error handling */
    break;
  }
}
```

System Normal Operation

In the system normal operation state, firmware periodically calls `CyBle_ProcessEvents()` to process the BLE stack-related operations and checks if the connection is established.

Note Any BLE stack-related operations such as receiving or sending data from or to the link layer and event generation to the application layer are performed as a part of the `CyBle_ProcessEvents()` function call. In this application, Code 1 initializes the stack, but the events related to stack are generated only after the `CyBle_ProcessEvents()` function is called. Similarly, other events related to device connection, disconnection, advertising timeout, and the health thermometer services are generated only after `CyBle_ProcessEvents()` is called.

If the connection is established, firmware measures the temperature at regular intervals (configured by the Measurement Interval Characteristic of the Health Thermometer Service). After measuring the temperature, if Indications are enabled by the Central device, the firmware sends the temperature data to the BLE Central device as indications.

In a BLE application, the device transmits or receives data only at periodic intervals also known as the advertising intervals or the connection intervals, depending on the BLE connection state. Thus, once the system normal operation task is complete, to conserve power, the device enters into the system low power operation mode and wakes up at the next connection/advertisement interval.

System Low-Power Operation

In the system low-power operation state, device operates in one of the three possible power modes:

- Sleep

  This mode is entered when CPU is free but BLE subsystem (BLESS) is active and busy in data transmission or reception. In this scenario, the CPU is put to sleep while the remaining core, such as clocks, regulator, is kept active for the normal BLE operation.

  To be able to conserve power, the internal main oscillator (IMO) frequency is reduced to 3 MHz and on wakeup, it is switched back to 12 MHz.

- Deep Sleep

  Firmware continuously tries to put the BLESS into the deep sleep mode. After the BLESS is successfully put into the deep sleep mode, the remaining system also transitions to the deep sleep mode.

Note Transitioning the device into the deep-sleep mode should happen immediately after the BLESS is put into deep-sleep mode. If this cannot be guaranteed, the firmware should disable the interrupt (to avoid servicing the ISR) and recheck if the BLESS is still in the deep-sleep mode or if in active mode, and whether the ECO has stabilized. If the BLESS is in either of the two modes, then the device can safely enter the deep-sleep mode; if not, the device must wait until the Rx/Tx event is complete.

- Hibernate

  When the device gets disconnected or the advertising interval times out, it enters the ultra-low power mode, i.e.,Hibernate mode. After waking up from this mode, the firmware starts to execute from the beginning of main.c, although the RAM contents are retained.

Sensor Simulation

If you don't have a thermistor and a reference resistor to measure temperature, you may use the temperature simulation mode to test the application. In this mode, the temperature data is simulated and incremented by 1 ^{0}C per measurement interval (default value as 1 second).

This can be done in the application code by changing the value of constant SIMULATE_TEMPERATURE_SENSOR from 1 to 0 in the *Temperature.h* file, as shown in Code 5.

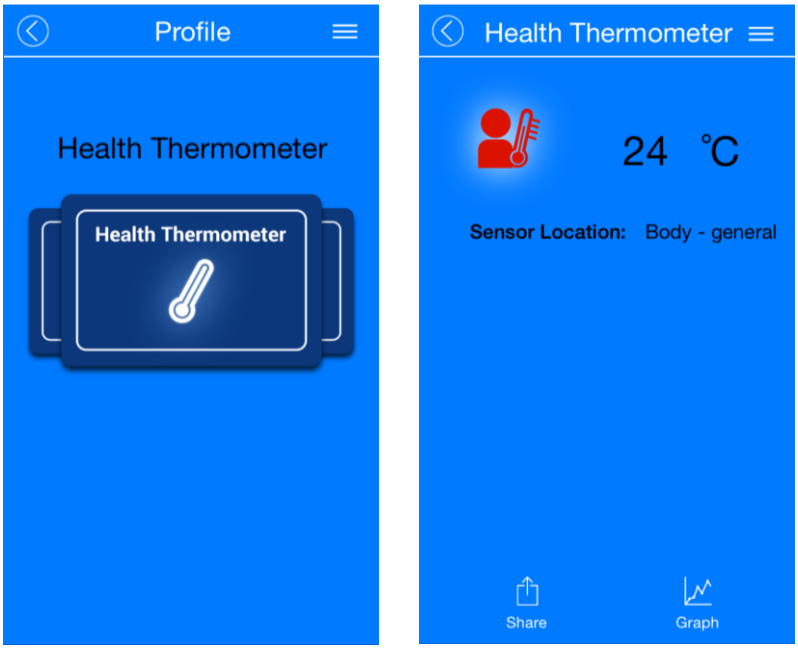
Code 5. Simulate Temperature Sensor

```
#define SIMULATE_TEMPERATURE_SENSOR (0u)
```

Testing with CySmart Mobile App

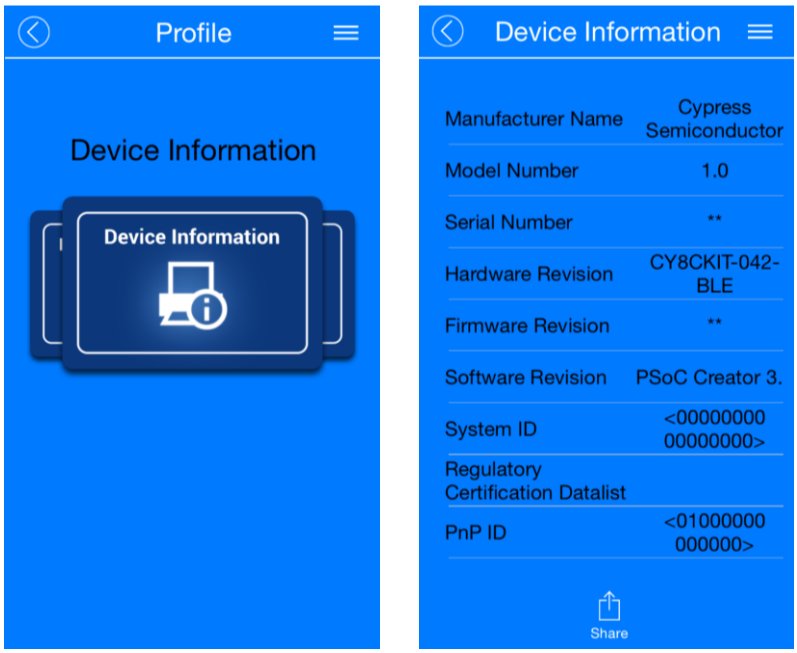
1. Open the **CySmart Mobile App** on your phone. If you do not have Bluetooth switched on already, the app will ask you to do it.
2. Connect to your GATT Server device on the app. Once connected, the app shows you all the Services exposed by the GATT Server. It automatically detects the Health Thermometer and Device Information Services.
3. Select Health Thermometer Service, it'll show you the current temperature. If SIMULATE_TEMPERATURE_SENSOR option is selected then temperature will increase by 1 ^{0}C per measurement interval (default value as 1 second). See Figure 4.

Figure 4: CySmart Android Mobile App – Health Thermometer Service



4. GO back and Select Device Information Service. It shows the device information configured as apart of the project. See Figure 5.

Figure 5. CySmart iOS Mobile App - Device Information Service



Related Documents

Table 1 lists all relevant application notes, code examples, knowledge base articles, device datasheets, and Component datasheets.

Table 1. Related Documents

Document	Title	Comment
AN91267	Getting Started with PSoC 4 BLE	Provides an introduction to PSoC 4 BLE device that integrates a Bluetooth Low Energy radio system along with programmable analog and digital resources.
AN91445	Antenna Design Guide	Provides guidelines on how to design an antenna for BLE applications.