**Assignment 6 Discussion**

* Rename *prox.c* and *prox.h* to *sensor.c* and *sensor.h*
* Change references to proximity to references to sensor, where appropriate.
* Increase the FreeRTOS heap size to 20K
* Peruse the *b-l475e-iot01a1\_hal\_lib* project for the functions to initialize and get readings from the HTS221 humidity and temperature sensor
* Consider the order in which the sensors should be initialized (hint: they are all on the I2C2 bus)
* Pay attention to the description in the assignment of the structure to hold the sensor measurements and the return value type of the sensor measurement functions
* When creating the sensor measurement queue, figure out how to specify the size of each queue object automatically (not by manually adding up how many bytes are in the measurement structure)
  + Hint given: size() can be used for struct (or something like it)
* In the function static void VL53L0X\_PROXIMITY\_MspInit(void) (in *sensor.c*, formerly known as *prox.c*) change the last line to:  
    
  HAL\_Delay(2); // THL was 1000  
    
  This will speed up the initialization of the proximity sensor.

**BLE (Bluetooth Low Energy) Overview**

Download the Bluetooth 5.2 Core Specification (over 3,200 pages!) and the Core Specification Supplement from:

<https://www.bluetooth.com/specifications/bluetooth-core-specification/>

At the highest level, Bluetooth is split between **Bluetooth Classic** (what is used for streaming audio to headphones and speakers) and Bluetooth Low Energy (most commonly referred to as **BLE**, sometimes referred to as **BTLE**), which is used to transfer small amounts of data infrequently where low power consumption is required.

BLE operates at 2.4GHz and its range is typically around 10 meters, but that can be greatly reduced by obstacles, such as human beings (since we consist mostly of water and 2.4GHz is greatly attenuated through water).

A BLE radio consumes around 10mA when transmitting. The very common CR2032 coin cell provides around 200mAh of energy. If the BLE radio is constantly transmitting, that translates to approximately 20 hours of operation. However, because BLE is designed infrequent, small data transfers (very low duty cycle), in many applications a CR2032 coin cell can power a BLE-based device for many months and even many years.

Early BLE radio ICs were just that – radios that needed to be controlled via a host processor (typically a microcontroller, aka MCU). Over time, silicon vendors have integrated the BLE radio with one or two MCUs, to lower the overall system cost and size. Why more than one MCU? To provide separate computational capability for the BLE stack and the application code.

An amazing example of how the cost of an MCU + BLE IC has come down is the EFR32BG22 from Silicon Labs:

<https://www.silabs.com/wireless/bluetooth/efr32bg22-series-2-socs>

At the start of this year, Silicon Labs announced the BG22 (as it’s commonly referred to) with a marketing campaign touting its support for Bluetooth 5.2 and a price of $0.52 if you place an order for 5.2 million pcs, setting a new price point in the electronics industry.

BLE beacons are an example of an application where low cost and low power consumption are essential. One example is where grocery stores place BLE beacons at various locations along the aisles and their customers run the store’s customer loyalty app on their cell phone. As they approach a beacon, the app detects what user is at that location and can look up their purchasing history to then alert them to new products or sales they might be interested in. The store can change what is offered without the customer having to download a new version of the app, by having the app access user data and promotional data from the cloud.

BLE beacons are also being used for location determination in settings where GPS signals cannot be received (inside warehouses, for example) or a GPS receiver would be prohibitively expensive. This can help businesses keep track of their assets, such as forklifts in a warehouse or infusion pumps in a hospital.

The utility of BLE beacons would not be economically feasible if power lines had to be wired to them or if someone had to replace their batteries very often. Getting a year of operation from a $0.10 coin cell turns a great idea into a practical reality.

The three main layers of the software architecture of a BLE device are:

* **Application**
* **Host**
* **Controller**

The application layer refers to the code that makes the device do its thing, such as periodically sending out a beacon or gathering sensor data, analyzing it, and sending it to a mobile app (as is done in a wrist-worn fitness monitor).

The host layer manages connections and contains the following layers:

* **GAP – Generic Access Profile**
* **GATT – Generic Attribute Profile**
* **ATT – Attribute Protocol**
* **SM – Security Manager**
* **L2CAP – Logical Link Control and Adaptation Protocol**
* **HCI – Host Controller Interface (host side)**

The controller layer manages the BLE radio hardware and contains the following layers:

* **PHY – Physical Layer**
* **LL – Link Layer**
* **DTM – Direct Test Mode**
* **HCI – Host Controller Interface (controller side)**

Combined, the host and controller layers are referred to as the **BLE Stack**.

BLE has standardized the protocol between the host layer and the controller layer and it is referred to as the **Host Controller Interface (HCI)**. As mentioned above, early on the application host code ran on a processor that was separate from the controller code, so the protocol was standardized to make it easier to support different BLE radios.

The B-475E-IOT01A board we use in class has the ST Micro SPBTLE-RF module connected to the STM32L475E MCU via the SPI3 bus. The SPBTLE-RF module is based on the ST Micro BlueNRG-MS IC, which contains an ARM Cortex-M0 core running the host and controller layers. ST Micro has defined a superset of HCI called **ACI – Application Command Interface**, which allows the application code to issue HCI commands and higher-level host commands.

The *b-l475e-iot01a1\_hal\_lib* project provides drivers for the MCU peripherals (the HAL) and for most of the board-level peripherals (the BSP) – the accelerometer and gyro, the magnetometer, the proximity sensor, the temperature and humidity sensor, the pressure sensor, and the quad-SPI flash memory. It does not provide drivers for the four radio peripherals – WiFi, BLE, Sub-GHz, and NFC. As a result, we will either have to develop code that talks to the BLE radio at the ACI level or we will have to find suitable example code that we can port for our use.

An embedded (“**Edge**”, as in, being at the edge of the IoT architecture) BLE device is referred to as a **Peripheral**. It communicates with a mobile device such as a phone or a USB BLE dongle plugged into a computer, which is referred to as the **Central**.

The three main states of BLE operation are:

* **Advertising**
* **Discovery**
* **Connected**

The peripheral and central communicate via the following sequence of events:

|  |  |
| --- | --- |
| **Peripheral** | **Central** |
| Advertise | Scan for peripherals (Discover) |
|  | Connect |
|  | Discover Services |
|  | Discover Characteristics |
| Receive commands and data Send data via notifications and upon request | Send commands and data Receive data via notifications |

Just like Ethernet and WiFi devices, a BLE device has a unique 48-bit **MAC address**.

It is not necessary for a connection to be established between a peripheral and a central. Some peripherals, such as BLE beacons, are not designed to connect to a particular central (such as a specific shopper’s phone) but instead broadcast a message that can be received by any central within range. In this case the peripheral is referred to as a **Broadcaster** (which only sends data) and the central is referred to as an **Observer** (which only receives data).

A central can connect to more than one peripheral at a time, such as a phone being connected to a fitness tracker and a heartrate monitor at the same time.

Once a connection is established, the device providing data is referred to as the **Server** and the device sending commands and requests for data and receiving notifications from the server is the **Client**.

If a connection is lost (say, due to the peripheral being out of range of the central), the peripheral immediately begins advertising again. The establishment of a connection is controlled by the central.

**Services** are high-level constructs, such as the condition of the battery in a peripheral.

**Characteristics** are subordinate constructs to services, such as the battery voltage and remaining capacity.

**Class Notes**