Created in TagSpaces on 2024-06-14.

BLE基本知识

在ble中，服务（service）是一个逻辑实体，用于分组相关的特征（characteristic）。每一个服务都有一个唯一的服务ID，通常是一个UUID（通用唯一标识符，Universally Unique Identifier).

BLE 规范定义了两种类型UUID：

UUID是一个128位的标识符，通常表示为一串32个十六进制字符串，分成5个部分。中间使用连字符分隔，例如：`123e4567-e89b-12d3-a456-426614174000`。

1. 16位UUID，用于标准化服务和特征。
2. 128位UUID，用于自定义服务和特征。

特征（characteristic)

特征值有UUID、属性、值、描述符 四个部分组成

1. **UUID**:
   - 每个特征都有一个唯一的 UUID，用于标识特征的类型。UUID 可以是 16 位或 128 位。
2. **属性（Properties）**：
   - 特征的属性定义了特征的行为和访问权限，例如是否可读、可写、可通知等。常见的属性包括：
     - `Read`：允许客户端读取特征的值。
     - `Write`：允许客户端写入特征的值。
     - `Notify`：当特征的值发生变化时，服务器可以通知客户端。
     - `Indicate`：与通知类似，但需要客户端确认接收到数据。
3. **值（Value）**：
   - 特征的实际数据，可以是任何类型的数据。值的格式和含义由特征的 UUID 和应用程序定义。
4. **描述符（Descriptors）**：
   - 描述符是特征的附加信息，用于描述特征的属性和行为。常见的描述符包括用户描述符（User Description）、客户端特征配置描述符（Client Characteristic Configuration Descriptor, CCCD）等。

     #include <stdio.h> #include <stdlib.h> #include <string.h> #include <glib.h> #include <gio/gio.h> #include <gattlib.h>

     #define HEART_RATE_MEASUREMENT_UUID "00002a37-0000-1000-8000-00805f9b34fb"

     static void on_read_request(const gattlib_request_t *request, void *user_data) { // 心率测量值，例如 75 BPM uint8_t heart_rate_value[2] = {0x00, 75};

     ```
     // 发送响应
     gattlib_send_response(request, GATTLIB_SUCCESS, heart_rate_value,
     sizeof(heart_rate_value));
     ```

     }

     int main() { gattlib_adapter_t *adapter; gattlib_primary_service_t service; gattlib_characteristic_t characteristic;

```
// 初始化 BlueZ 适配器
gattlib_adapter_open(NULL, &adapter);

// 定义服务和特征
gattlib_primary_service_t hr_service = {
    .uuid = GATTLIB_UUID_HEART_RATE_SERVICE,
    .characteristics = {
        {
            .uuid = HEART_RATE_MEASUREMENT_UUID,
            .properties = GATTLIB_CHARACTERISTIC_READ,
            .on_read = on_read_request,
        },
        {0} // 结束标志
    }
};

// 注册服务
gattlib_register_service(adapter, &hr_service);

// 启动适配器
gattlib_adapter_start(adapter);

// 保持程序运行
GMainLoop *loop = g_main_loop_new(NULL, FALSE);
g_main_loop_run(loop);

// 清理
gattlib_adapter_close(adapter);
g_main_loop_unref(loop);

return 0;
```

}

BLE应用类型

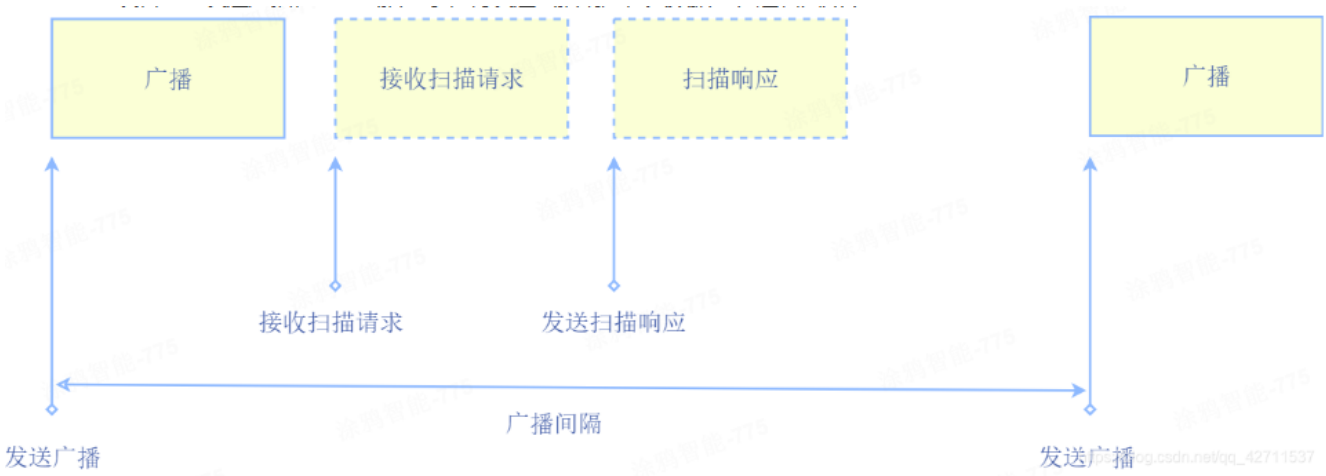
BLE应用可以分成 未连接状态下广播 GAP应用 和 连接状态下的GATT应用

状态	应用类型
未连接状态	GAP 应用
连接状态	GATT 应用

非连接状态

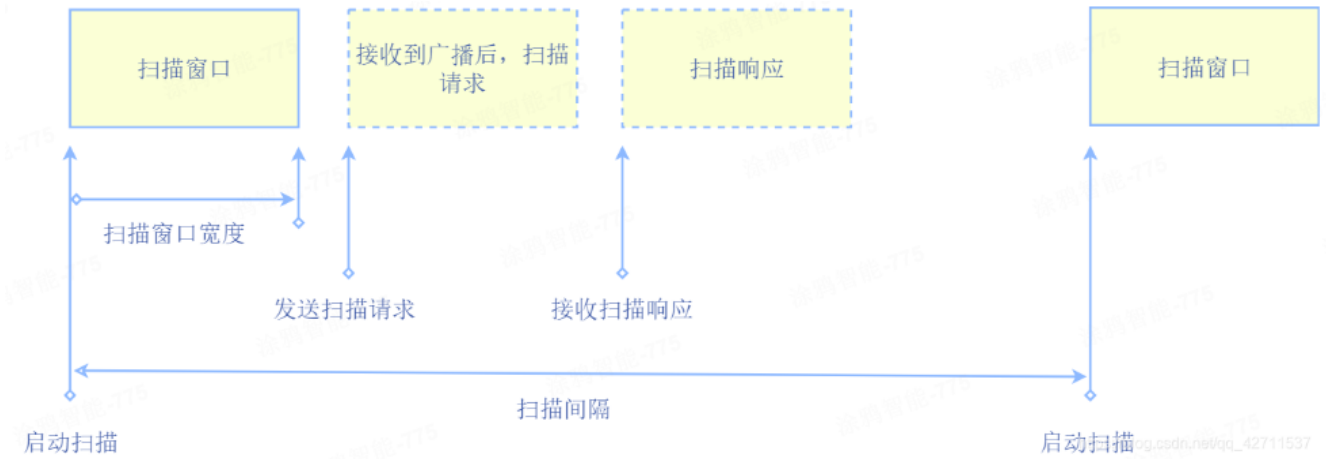
未连接状态下，可以通过广播进行数据交互，手机来scan（扫描），设备进行scan response

设备端



那么我们需要设置 1.boardcast的广播数据；2. 设置扫描响应的数据；3. 设置广播参数；4. 启动广播GAP流程；

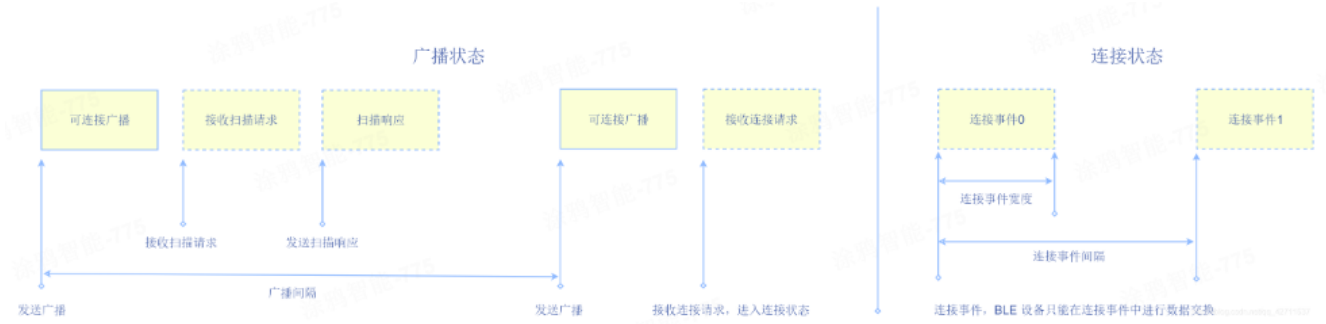
手机端



在手机端，就需要设置 1. 扫描串口等扫描参数；2.启动扫描；3，在广播GAP事件中解析广播参数；

连接状态

在连接状态下手机和BLE进行点对对点通信，BLE设备作为GATT服务器，手机作为GATT客户端。手机使用GATT流程来查找BLE设备定义的服务、特征、并读/写特征值数据和特征描述符属性。



开源协议栈

Bluez、Zephyr、NimBLE、BTstack等开源协议栈到底实现了什么功能？要回答这个问题，从整个ble协议栈结构出发：

1. **物理层（PHY）**：
   - **功能**：负责无线电信号的传输和接收，是 BLE 通信的基础。
   - **组件**：包括发射器和接收器，用于调制和解调信号。
2. **链路层（Link Layer, LL）**：
   - **功能**：管理设备之间的物理连接和数据包的传输。
   - **组件**：包括连接建立、连接参数更新、数据包重传、加密等功能。
3. **直接测试模式（Direct Test Mode, DTM）**：
   - **功能**：用于测试和验证物理层和链路层的性能。
   - **组件**：支持发送和接收测试数据包，测量信号强度和误码率。
4. **控制器接口（Controller Interface, HCI）**：
   - **功能**：定义主机和控制器之间的通信接口。
   - **组件**：包括 HCI 命令、事件和数据传输。
5. **逻辑链路控制和适配协议（Logical Link Control and Adaptation Protocol, L2CAP）**：
   - **功能**：提供逻辑链路和数据分组的传输服务。
   - **组件**：包括数据分段和重组、通道管理和 QoS（服务质量）配置。
6. **属性协议（Attribute Protocol, ATT）**：
   - **功能**：定义了客户端和服务器之间的通信方式，用于发现和访问属性。
   - **组件**：包括属性句柄、属性类型、属性值和权限。
7. **通用属性配置文件（Generic Attribute Profile, GATT）**：
   - **功能**：在 ATT 之上定义了通用的服务和特征的框架。
   - **组件**：包括服务、特征和描述符。
8. **安全管理协议（Security Manager Protocol, SMP）**：
   - **功能**：提供配对、绑定和加密等安全功能。
   - **组件**：包括密钥分发、加密和身份验证。
9. **通用访问配置文件（Generic Access Profile, GAP）**：
   - **功能**：定义了设备的角色和连接过程。
   - **组件**：包括广播、扫描、连接建立和连接参数更新。

将上述描述绘制成层级图如下：

```
+-----------------------------+
|       应用层 (Application)   |
+-----------------------------+
|   GATT (Generic Attribute)   |
+-----------------------------+
|      ATT (Attribute)         |
+-----------------------------+
|       SMP (Security)         |
+-----------------------------+
|       GAP (Generic Access)   |
+-----------------------------+
|       L2CAP (Logic Link)     |
```

```
+----------------------------+
|         HCI (Interface)      |
+----------------------------+
|       链路层 (Link Layer)    |
+----------------------------+
|       物理层 (Physical)      |
+----------------------------+
```

上述各个层级实现的功能如下：

1. **物理层（PHY）**：
   - **调制和解调**：使用 GFSK（Gaussian Frequency Shift Keying）进行信号调制和解调。
   - **信道**：在 2.4 GHz ISM 频段工作，分为 40 个信道，每个信道宽度为 2 MHz。
2. **链路层（LL）**：
   - **广播**：设备可以广播数据包，其他设备可以扫描和接收这些数据包。
   - **连接**：支持设备之间的连接建立、维护和断开。
   - **功率控制**：管理发射功率以优化功耗和通信范围。
3. **直接测试模式（DTM）**：
   - **测试模式**：用于测试物理层和链路层的性能，测量信号强度和误码率。
4. **控制器接口（HCI）**：
   - **HCI 命令**：主机可以发送命令给控制器，例如启动扫描、建立连接等。
   - **HCI 事件**：控制器可以发送事件给主机，例如连接完成、数据接收等。
   - **HCI 数据传输**：主机和控制器之间的数据包传输。
5. **逻辑链路控制和适配协议（L2CAP）**：
   - **数据分段和重组**：将大数据包分段传输，并在接收端重组。
   - **通道管理**：管理逻辑通道，提供多路复用。
   - **QoS 配置**：配置服务质量参数。
6. **属性协议（ATT）**：
   - **属性发现**：客户端可以发现服务器上的属性。
   - **属性读写**：客户端可以读写服务器上的属性。
   - **属性通知**：服务器可以通知客户端属性的变化。
7. **通用属性配置文件（GATT）**：
   - **服务**：定义一组相关的特征，例如心率服务。
   - **特征**：定义具体的数据点，例如心率测量值。
   - **描述符**：提供附加信息，例如特征的格式和范围。
8. **安全管理协议（SMP）**：
   - **配对**：设备之间交换密钥以建立安全连接。
   - **加密**：对数据传输进行加密，确保数据安全。
   - **身份验证**：验证设备的身份，防止中间人攻击。
9. **通用访问配置文件（GAP）**：
   - **角色**：定义设备的角色，例如中央设备（Central）、外设设备（Peripheral）。
   - **广播和扫描**：外设设备广播数据，中央设备扫描和接收数据。
   - **连接建立**：中央设备和外设设备建立连接。

所有BlueZ等开源协议栈实现了包括HCI及以上层级的工作。

开源协议栈的移植

将 BlueZ 移植到你的系统上涉及几个步骤，包括下载源代码、编译、安装以及配置。

步骤 1：准备环境

在开始之前，请确保你的系统满足以下依赖项：

- `glib2.0`
- `dbus`
- `libical`
- `libreadline`

你可以使用包管理器来安装这些依赖项。例如，在基于 Debian 的系统上：

sh

```
sudo apt-get update
sudo apt-get install libglib2.0-dev libdbus-1-dev libical-dev libreadline-dev
```

步骤 2：下载 BlueZ 源代码

你可以从 BlueZ 的官方源码仓库下载最新版本的源码：

sh

```
wget https://www.kernel.org/pub/linux/bluetooth/bluez-5.68.tar.xz
tar -xf bluez-5.68.tar.xz
cd bluez-5.68
```

步骤 3：编译和安装 BlueZ

在编译和安装 BlueZ 之前，确保你拥有必要的编译工具，例如 `gcc` 和 `make`。你可以使用以下命令安装这些工具：

sh

```
sudo apt-get install build-essential
```

然后，按照以下步骤编译和安装 BlueZ：

sh

```
./configure --prefix=/usr --mandir=/usr/share/man --sysconfdir=/etc --localstatedir=/var
make
sudo make install
```

步骤 4：配置 BlueZ

安装完成后，你需要配置系统以使用 BlueZ。以下是一些常见的配置步骤：

1. 启动 Bluetooth 服务

确保 Bluetooth 服务已启动并在系统启动时自动启动：

sh

```
sudo systemctl enable bluetooth
sudo systemctl start bluetooth
```

2. 配置 D-Bus

BlueZ 使用 D-Bus 进行进程间通信。确保你的系统上运行了 D-Bus 服务：

sh

```
sudo systemctl enable dbus
sudo systemctl start dbus
```

3. 配置 Bluetooth 设备

你可以使用 `bluetoothctl` 工具来管理和配置 Bluetooth 设备：

```
sudo bluetoothctl
```

在 `bluetoothctl` 交互式命令行中，你可以输入以下命令来配置设备：

```
power on
agent on
default-agent
scan on
```

步骤 5：测试 BlueZ

你可以使用 `hciconfig` 和 `hcitool` 工具来测试 BlueZ 的安装和配置。

查看 Bluetooth 设备

```
hciconfig
```

扫描附近的 Bluetooth 设备

```
hcitool lescan
```

示例：编写简单的 BlueZ 应用

以下是一个使用 BlueZ 库的简单 C 语言示例，展示如何扫描附近的 BLE 设备：

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/socket.h>
#include <bluetooth/bluetooth.h>
#include <bluetooth/hci.h>
#include <bluetooth/hci_lib.h>

int main() {
    inquiry_info *ii = NULL;
```

```
    int max_rsp, num_rsp;
    int dev_id, sock, len, flags;
    char addr[19] = { 0 };
    char name[248] = { 0 };

    dev_id = hci_get_route(NULL);
    sock = hci_open_dev(dev_id);
    if (dev_id < 0 || sock < 0) {
        perror("opening socket");
        exit(1);
    }

    len  = 8;
    max_rsp = 255;
    flags = IREQ_CACHE_FLUSH;
    ii = (inquiry_info*)malloc(max_rsp * sizeof(inquiry_info));

    num_rsp = hci_inquiry(dev_id, len, max_rsp, NULL, &ii, flags);
    if (num_rsp < 0) perror("hci_inquiry");

    for (int i = 0; i < num_rsp; i++) {
        ba2str(&(ii+i)->bdaddr, addr);
        memset(name, 0, sizeof(name));
        if (hci_read_remote_name(sock, &(ii+i)->bdaddr, sizeof(name), name, 0) < 0)
            strcpy(name, "[unknown]");
        printf("%s  %s\n", addr, name);
    }

    free(ii);
    close(sock);
    return 0;
}
```

编译和运行示例

你可以使用以下命令编译和运行这个示例：

```
gcc -o ble_scan ble_scan.c -lbluetooth
sudo ./ble_scan
```

##总结

移植 BlueZ 到你的系统上涉及下载、编译、安装和配置多个步骤。确保你的系统满足所有依赖项，并按照上述步骤进行操作。成功安装和配置后，你可以使用 BlueZ 提供的工具和库来开发和测试你的蓝牙应用。对于嵌入式系统来说，就是下载代码进行编译生成lib.a，调用相对应的API接口。

涂鸦 BLE TKL适配

为了实现ble配网，tkl关键的API接口有：

```
OPERATE_RET tkl_ble_stack_init(UCHAR_T role);


OPERATE_RET tkl_ble_gap_callback_register(CONST TKL_BLE_GAP_EVT_FUNC_CB gap_evt);

OPERATE_RET tkl_ble_gatt_callback_register(CONST TKL_BLE_GATT_EVT_FUNC_CB gatt_evt);

OPERATE_RET tkl_ble_gap_adv_start(TKL_BLE_GAP_ADV_PARAMS_T CONST *p_adv_params);

OPERATE_RET tkl_ble_gap_adv_rsp_data_set(TKL_BLE_DATA_T CONST *p_adv, TKL_BLE_DATA_T CONST
*p_scan_rsp);

OPERATE_RET tkl_ble_gap_adv_rsp_data_update(TKL_BLE_DATA_T CONST *p_adv, TKL_BLE_DATA_T
CONST *p_scan_rsp);

OPERATE_RET tkl_ble_gatts_service_add(TKL_BLE_GATTS_PARAMS_T *p_service);
```

ble tkl适配，首先要清楚目前tkl 接口sdk调用的流程：

tkl_ble_gap_callback_register --> tkl_ble_gatt_callback_register --> tkl_ble_gatts_service_add --> tkl_ble_stack_init --> tkl_ble_gap_adv_start --> tkl_ble_gap_adv_rsp_data_set --> tkl_ble_gap_adv_rsp_data_update()

对于这些tkl接口，最特殊的是 tkl_ble_gatts_service_add 这个接口需要将驱动层中新建的service的write characteristic(特征值）给到 tuyaos SDK。对于这个TKL初始化流程，ble协议栈驱动需要保证能给到tuyaos SDK正确的write handle。