Programming Guide

SDK Home (http://developer.anki.com/drive-sdk/)

Bluetooth LE Core Concepts

The Bluetooth 4.0 specification defines a wireless communication protocol for use with low energy devices. The protocol is conceptually similar to a client-server architecture, in which a client device (central) can connect to and communicate with a server device (peripheral). For more information on Bluetooth 4.0, see the Bluetooth 4.0 Core specification (https://www.bluetooth.org/docman/handlers/downloaddoc.ashx?doc_id=229737).

Anki Drive vehicles are Bluetooth low energy peripheral (server) devices. Any device capable of acting as a central (client), such as a smartphone or computer, can discover, connect to and control vehicles. Each vehicle advertises identifying information and a service that consists of characteristics for sending and receiving data. This document describes the format of the vehicle advertisement data, which can be used to decode the vehicle identifier, name and model. Once connected, a central device can communicate with vehicles using the Anki Drive message protocol outlined below.

Discovering Vehicles

Vehicles broadcast identifying information and service definitions in the form of advertising packets. An advertising packet contains binary data in a parsable format defined by a generic attribute profile (GATT). Depending on the bluetooth API available on the central, this information may be exposed in different ways. On iOS, CoreBluetooth (https://developer.apple.com/library/ios/documentation/NetworkingInternetWeb/Conceptual/CoreBluetooth_concepts/AboutCoreBluetooth/Introduction.html) parses this data internally and provides an NSDictionary containing the profile data. On Linux and Android, the raw scan bytes are exposed via the bluetooth API (BlueZ (http://www.bluez.org/), Linux; android.bluetooth (http://developer.android.com/guide/topics/connectivity/bluetooth-le.html), Android) and must be parsed to obtain the GATT profile data members of interest.

Parsing Advertising Packet data

If raw scan data from the Extended Inquiry Response (EIR) is available, it needs to be parsed to obtain the LOCAL_NAME and MANUFACTURER_DATA. These records types are defined as part of the GATT profile specification (https://www.bluetooth.org/en-us/specification/assigned-numbers/generic-access-profile). The Anki Drive SDK provides methods (https://github.com/anki/drive-sdk/blob/master/include/ankidrive/advertisement.h) to parse EIR data into records and to extract vehicle information from specific types of records.

Advertisement Data Format

Vehicle advertisements consist of the Anki Drive service UUID (`service_id`), along with a unique identifier (`mfg_data`), name and state information (`local_name`).

```
/**
 * Vehicle information present in Bluetooth LE advertising packets.
 *
 * flags: EIR flags
 * tx_power: transmission power
 * mfg_data: parsed data from the MANUFACTURER_DATA bytes
 * local_name: parsed data from the LOCAL_NAME string bytes
 * service_id: Anki Vehicle UUID (128-bit)
 */
typedef struct anki_vehicle_adv {
    uint8_t                     flags;
    uint8_t                     tx_power;
    anki_vehicle_adv_mfg_t      mfg_data;
    anki_vehicle_adv_info_t     local_name;
    uuid128_t                   service_id;
} anki_vehicle_adv_t;
```

The `service_id` for an Anki Drive vehicle is defined in the GATT profile for the vehicle, and will always be the same 128-bit UUID. This UUID can be used to identify vehicles during scanning, or as a handle to discover the read and write characteristics after connecting to a vehicle.

```
#define ANKI_STR_SERVICE_UUID       "BE15BEEF-6186-407E-8381-0BD89C4D8DF4"
```

The manufacturer data is a uint64_t value that uniquely identifies each vehicle. This value specifies the 'make/model' of the vehicle (`model_id`) and a unique identifier for each vehicle of the specified model (`identifier`).

```
/**
 * Vehicle hardware information encoded in the MANUFACTURER_DATA
 * record of an advertising packet.
 *
 * - identifier: Unique identifier for a physical vehicle
 * - model_id: The model type of a vehicle
 * - product_id: Value identifying the vehicle as Anki Drive hardware
 */
typedef struct anki_vehicle_adv_mfg {
    uint32_t    identifier;
    uint8_t     model_id;
    uint8_t     _reserved;
    uint16_t    product_id;
} anki_vehicle_adv_mfg_t;
```

The Bluetooth 4.0 specification requires that the LOCAL_NAME field be a UTF-8 encoded string of up to 248 bytes, with shorter values terminated by a NULL (0x0) byte (Bluetooth 4.0, Part C, 3.2.2.3, 12.1 (https://www.bluetooth.org/docman/handlers/downloaddoc.ashx?doc_id=229737)). However, Bluetooth LE devices may only advertise up to 20 bytes of the LOCAL NAME data (Bluetooth 4.0, Part C, 11.1.2 (https://www.bluetooth.org/docman/handlers/downloaddoc.ashx?doc_id=229737)). To ensure that the entire LOCAL_NAME is available during advertising, Anki Drive vehicles only use up to 20 bytes of LOCAL NAME data.

The LOCAL_NAME advertised by Anki Drive vehicles consists of the vehicle state (`state`), firmware version

(`version`) and a user-defined vehicle name (`name`). The vehicle state and firmware version will always be non-null values in the ASCII range from (0x01 -- 0x7f). However, the remaining data could be NULL, which would still satisfy the requirement for a UTF-8 encoded string. This should be accounted for when parsing the data (https://github.com /anki/drive-sdk/blob/master/include/ankidrive/advertisement.h). This strategy of including additional information in the LOCAL NAME field enhances the user interface experience in the Anki Drive app, but may result in the LOCAL_NAME data changing during repeated advertisements if, for example, a vehicle is removed from a charger.

```
/**
 * Vehicle information packed in the LOCAL_NAME string record
 * of an advertising packet.
 *
 * - state: Current vehicle state.
 *   NOTE: Changes to the vehicle state will cause the LOCAL_NAME value
 *   to change.
 * - version: Firmware version running on the vehicle
 * - name: User-defined name in UTF-8 encoding
 */
typedef struct anki_vehicle_adv_info {
    anki_vehicle_adv_state_t state;
    uint16_t            version;
    uint8_t             _reserved[5];
    unsigned char       name[13]; // UTF8: 12 bytes + NULL.
} anki_vehicle_adv_info_t;

/**
 * The state of a vehicle recorded in the advertising packet.
 *
 * - full_battery: The vehicle battery is fully charged
 * - low_battery: The vehicle battery has a low charge and will die soon
 * - on_charger: The vehicle is currently on the charger
 */
typedef struct anki_vehicle_adv_state {
  uint8_t _reserved:4;
  uint8_t full_battery:1;    // 4: TRUE if Car has full battery
  uint8_t low_battery:1;     // 5: TRUE if Car has low battery
  uint8_t on_charger:1;      // 6: TRUE if Car is on Charger
  uint8_t _unavailable:1;    // 7: UNUSED to avoid NULL string
} anki_vehicle_adv_state_t;
```

Connecting to Vehicles

In order to establish a Bluetooth LE connection with a vehicle, the device address or a unique hardware identifier must be known. On Android, or Linux and other platforms using BlueZ (http://www.bluez.org/), the device address is exposed during the scanning process. On Apple platforms, CoreBluetooth hides the device address and instead provides a unique identifier (UUID) that identifies each peripheral. The identifiers can be associated with vehicle data obtained during scanning and stored for later use in connecting to vehicles without re-scanning peripherals.

Establishing a BLE connection

Three steps are required to establish bi-directional communication with a vehicle:

1. Connect to the vehicle using the hardware identifier.

   This step establishes a connection between the central device and the radio in the vehicle.

2. Discover GATT services associated with the vehicle.

   Once connected, this step provides access to the GATT service available on the vehicle, which provides characteristics for reading and writing data.

3. Discover characteristics of the vehicle service and register for notifications on the inbound (read) characteristic.

   This step registers the read and write characteristics for the vehicle service, and makes them available for data transfer. To register for notifications when data is written from the vehicle, it is necessary to set a bit on the Client Configuration Characteristic. This step is abstracted by CoreBluetooth (https://developer.apple.com /library/ios/documentation/CoreBluetooth/Reference/CBPeripheral_Class/translated_content /CBPeripheral.html#//apple_ref/occ/instm/CBPeripheral/setNotifyValue:forCharacteristic:) on Apple platforms. When using the BlueZ (http://www.bluez.org/) API, it is necessary to set this value directly. See the vehicle-tool (https://github.com/anki/drive-sdk/blob/master/examples/vehicle-tool/vehicle_cmd.c#L241) example utility for an example.

Once these steps are complete, it is possible to send and receive data from the vehicle by writing to and reading from the characteristics.

Anki Vehicle GATT profile

Anki Drive Vehicles implement a Generic Attribute profile (GATT) that defines how attributes can be transmitted to and from the vehicle. These data are provided in vehicle_gatt_profile.h (https://github.com/anki/drive-sdk/blob /master/include/ankidrive/vehicle_gatt_profile.h), and also summarized below.

```
/** Anki Drive Vehicle Service UUID */
#define ANKI_STR_SERVICE_UUID       "BE15BEEF-6186-407E-8381-0BD89C4D8DF4"

/** Anki Drive Vehicle Service READ Characteristic */
#define ANKI_STR_CHR_READ_UUID      "BE15BEE0-6186-407E-8381-0BD89C4D8DF4"

/** Anki Drive Vehicle Service WRITE Characteristic */
#define ANKI_STR_CHR_WRITE_UUID     "BE15BEE1-6186-407E-8381-0BD89C4D8DF4"
```

Best Practices for Connecting to Vehicles

The Anki Drive app connects to multiple vehicles and iOS devices simultaneously. Furthermore, both vehicles and devices send data at the fastest possible rate. This is in contrast to many other Bluetooth LE devices, which may advertise infrequently or sparingly send data in order to save power.

To minimize problems caused by this particular use case, we found it useful to adopt the following conventions:

Serialize connection attempts

Do not attempt to perform multiple connections simultaneously. Instead, serialize all connection attempts, so that connection, service discovery and characteristic discovery for one peripheral are completed before attempting to establish another connection.

Avoid connections while scanning

It is possible to connect to peripherals while simultaneously scanning. However, our empirical evidence on iOS suggests that the connection procedure is more robust if scanning is not in progress.

Vehicle Message Protocol

Vehicle messages are short, structured sequences of bytes that can represent driving commands, queries about the vehicle state, or convey specific vehicle states or conditions.

Message Format

The current message format is designed for low-bandwidth wireless transmission protocols such as Bluetooth LE. Each message must begin with a 2-byte sequence specifying the message size, and identifier. Certain types of messages may require up to 18 bytes of additional data for a maximum size of 20 bytes.

Vehicle Message Packet Structure

```
+----------------------------+--------+------------+
| size_of(msg_id + payload) | msg_id | payload    |
+----------------------------+--------+------------+
|        1 byte             | 1 byte | 0-18 bytes |
+----------------------------+--------+------------+
```

The message data structure is represented as a C struct that encapsulates data for vehicle communications.

```
/**
 * Basic vehicle message.
 *
 * - size: Size in bytes of the msg_id plus payload
 * - msg_id: Identifier for message
 * - payload: Optional message data for parameters or response info.
 *
 */

#define ANKI_VEHICLE_MSG_MAX_SIZE            20
#define ANKI_VEHICLE_MSG_PAYLOAD_MAX_SIZE    18
#define ANKI_VEHICLE_MSG_BASE_SIZE            1

typedef struct anki_vehicle_msg {
    uint8_t size;
    uint8_t msg_id;
    uint8_t payload[ANKI_VEHICLE_MSG_PAYLOAD_MAX_SIZE];
} anki_vehicle_msg_t;
```

Message Types

Each message or command is identified by a unique 1-byte identifier. This value identifies the command that the vehicle should perform and is also used to determine whether additional data parameters are required. Because message identifiers are shared with code running on the vehicle, they are inherently linked to the firmware version running on the vehicle. Although we do not expect these to change frequently, any changes that do occur in the future may break compatibility with pre-1.0 releases of the SDK.

```
/** Identifier for a vehicle message */
enum {
    // BLE Connections
    ANKI_VEHICLE_MSG_C2V_DISCONNECT = 0xd,

    // Ping request / response
    ANKI_VEHICLE_MSG_C2V_PING_REQUEST = 0x16,
    ANKI_VEHICLE_MSG_V2C_PING_RESPONSE = 0x17,

    // Messages for checking vehicle version info
    ANKI_VEHICLE_MSG_C2V_VERSION_REQUEST = 0x18,
    ANKI_VEHICLE_MSG_V2C_VERSION_RESPONSE = 0x19,

    // Lights
    ANKI_VEHICLE_MSG_C2V_SET_LIGHTS = 0x1D,

    // Driving Commands
    ANKI_VEHICLE_MSG_C2V_SET_SPEED = 0x24,
    ANKI_VEHICLE_MSG_C2V_CHANGE_LANE = 0x25,
    ANKI_VEHICLE_MSG_C2V_CANCEL_LANE_CHANGE = 0x26,
    ANKI_VEHICLE_MSG_C2V_TURN_180 = 0x32,

    // Light Patterns
    ANKI_VEHICLE_MSG_C2V_LIGHTS_PATTERN = 0x33,

    // SDK Mode
    ANKI_VEHICLE_MSG_C2V_SDK_MODE = 0x90,
};
```

Messages with parameters

Some messages require additional data parameters. The required parameters for each message are defined as packed structs for clarity and convenience. In addition, each type of message has a corresponding function that fills in a generic `anki_vehicle_msg_t` struct parameter. These methods allow callers to control memory allocation, but provide an abstracted way to generate messages with multiple parameters.

SDK Mode

Anki Drive vehicles have a special mode that allows the commands listed above to be used without any additional data or overhead that is required control vehicles in the Anki Drive iOS app. To enable this mode, send the vehicle a message with the `ANKI_VEHICLE_MSG_C2V_SDK_MODE` . This can be easily created using one of the message helper methods.

```
anki_vehicle_msg_t msg;
memset(&msg, 0, sizeof(msg));
uint8_t size = anki_vehicle_msg_set_sdk_mode(&msg, 1, ANKI_VEHICLE_SDK_OPTION_OVERRIDE_
LOCALIZATION);
```

In this example `anki_vehicle_msg_set_sdk_mode` writes `{ 0x03, 0x90, 0x01, 0x01 }` to `msg` and returns 4. Both the buffer and size, should be passed to a function that sends 4 bytes from the buffer to vehicle.

Set vehicle speed

The procedure for creating a message to set the vehicle speed is similar.

```
anki_vehicle_msg_t msg;
memset(&msg, 0, sizeof(msg));
uint8_t size = anki_vehicle_msg_set_speed(&msg, 1000, 25000);
```

All of the available functions are listed in protocol.h (https://github.com/anki/drive-sdk/blob/master/include/ankidrive/protocol.h). For more examples of how to use these functions and send data, see the vehicle-tool (https://github.com/anki/drive-sdk/blob/master/examples/vehicle-tool/vehicle_cmd.c#L471) example utility.

Revision History

2014-01-23 Initial version.

Thanks to GitHub (http://github.com), Jekyll (http://jekyllbootstrap.com/), and Bootstrap (http://getbootstrap.com).