**Codie BLE API v1.0**

## GENERAL INFO

Codie communicates with a BLE module, so every device which has a Bluetooth 4.x compatible transciever can send and receive commands to and from the robot. Codie's BLE information:

- Name: Codie
- advertisement flags: undirected, connectable, LE only
- address type: random address
- role: peripheral
- conection mode: "just works", bonding/pairing not supported yet

## COMMUNICATION BLE SERVICE

Codie has a custom BLE service with the following UUID: `52af0001-978a-628d-c845-0a104ca2b8dd` In this service there are two characteristics:

- RX characteristic `{52af0002-978a-628d-c845-0a104ca2b8dd}` : Codie *receives* commands through this characteristic. It is readable and writeable (write without response). Be aware, that only 20 bytes can be written at a time (due to the BLE packet payload limit).
- TX characteristic `{52af0003-978a-628d-c845-0a104ca2b8dd}` : Codie *sends* commands through this characteristic. Readable and can be subscribed for notification (indication not supported).

## CODIE PROTOCOL

Codie communicates with a custom binary protocol. All fields and data is *little-endian*.

The communication takes place among three nodes:

- **App**: The application which can be a mobile app on a mobile platform, a PC program… anything which is in the central BLE role, connecting to Codie.
- **BLE**: The BLE module inside Codie. There are special commands that can be executed only by the BLE module.
- **MCU**: The host microcontroller unit in Codie, executing the majority of the commands.

## PACKET STRUCTURE

The packet structure is as follows:

| INFO[8] | SEQ[16] | CMD[16] | ARGLEN[16] | ARGDAT[n] |
|---|---|---|---|---|
| | | | | |

Field descriptions:

| field | purpose |
|---|---|
| INFO | **Packet information.** (see details later) |
| SEQ | **Sequence number.** A 16-bit packet counter. Used for sorting and acknowledge/reply identification. |
| CMD | **Command ID.** 16-bit command ID. |
| ARGLEN | **Argument length.** For each command, there can be one argument starting with a 16bit length field. |
| ARGDAT | **Argument data.** Data accompanying the command (command specific structure). |

#### INFO

The `INFO` field has two parts: `ROUTE` specifying the sender and recevier of the packet, and `PRIO` holding the priority of the packet.

| INFO | | | | | | | |
|---|---|---|---|---|---|---|---|
| ROUTE | | | | PRIO | | | |
| D1 | D0 | S1 | S0 | P3 | P2 | P1 | P0 |

**ROUTE**

- Destination ( `D0-1` ): destination (receiver) of the packet
- Source ( `S0-1` ): source (sender) of packet

Both two bits parts can have twe following value:

| value (bin) | meaning |
|---|---|
| `00` | App |
| `01` | MCU |
| `10` | BLE |
| `11` | broadcast (can be only a destination) |

For example the ROUTE field of a packet sent by an application for the MCU has a binary value of `0100` , or `0x4` in hex. The broadcast value can be only a destination, meaning that every node (except the sender) will get and parse the package.

**PRIO**

The four `PRIO` bits specify the priority of the packet. Currently only the most significant bit (P3) is used: if P3=0, the packet gets queued in the normal queue, if P3=1, the packet goes to the priority queue. Packets in the priority queue are executed first.

#### SEQ

The packet sequence number is a 16bit unsigned integer. Each node (App,BLE,MCU) starts its own counter from packet 1 and increases with every sent packet. The sequence number gets replied in a command reply as the first 2 bytes of the argument (see the examples). Currently this seq number is not used in any way except in the replies (so a reply can be assigned to its original command).

#### ARG

The argument starts with the 16bit `ARGLEN` argument length in bytes, followed by the argument data. The data structure is command specific.

## COMMANDS AND REPLIES

Each command has a 16bit ID. The commands are grouped to four groups:

- **General commands**, which can be executed by any of the nodes.
- **MCU commands** can be executed only on the MCU.
- **BLE commands** can be executed only on the BLE module.

- **App commands** can be executed by the client application.

*You may notice that there is some redundancy lurking here… Most of the commands can be executed by only it's respective nodes, but the packets are routed by the `ROUTE` field in `INFO`. So a packet can be addressed for a node which won't be able to execute the command (it will be discarded).*

The MSB of the command ID marks if it's a reply. If MSB=1, that is a reply-command to its respective command pair.

#### Send a move distance

Let's say I would like to construct a packet for Codie to move forward 20cm.

- `INFO`: `0x40`
  - `ROUTE` will be App -> MCU, so `0x4`
  - `PRIO` will suffice as normal, so `0x0`
- `SEQ`: let's say this is our 18th pocket, so `0x12`. But as SEQ is 2bytes and little-endian, the field data will look like `0x12 0x00`.
- `CMD`: Command ID of moving forward a specified distance (*driveDist*) is `0x1061`, so our little-endian field value will be `0x61 0x10`.
- `ARGLEN`: The *driveDist* command has three arguments with a total of 4 bytes, so the 16bit field data is `0x4 0x0`
- `ARGDAT`:
  - the first argument is the distance in mm, which is 200, `0xc8` in hex. But the argument is 2bytes, so the data will be `0xc8 0x0`
  - second is the left track speed in only 1byte in percentage. Let's say 85%: `0x55`.
  - third is the right track speed, similarly. We want to go in a straight line, so give it the same value as the left: `0x55`. See command reference for more info on how the arguments are treated.

To summarize, the packet data is the following (the fields are spearated by "|", the arguments by "," for clarity):

```
0x40 | 0x12 0x00 | 0x61 0x10 | 0x04 0x00 | 0xc8 0x00 , 0x55 , 0x55
```

#### Move distance reply

When Codie gets the command, it will start to execute it. *When the movement is done*, we will get the following packet:

```
0x10 | 0x28 0x00 | 0x61 0x90 | 0x03 0x00 | 0x12 0x00 , 0x00
```

Let's see what this means:

- `INFO`: `0x10`
  - `ROUTE` got swapped to MCU -> App, so with the binary values `01 -> 00`, that's `0x1`
  - `PRIO` remained normal as we sent it: `0x0`
- `SEQ`: MCU sent it's own current SEQ, which is 40 (dec), but this is not important for us right now.
- `CMD`: MCU set the MSB of the command ID, indicating that this is a reply.
- `ARGLEN`: argument length is 3.
- `ARG`:
  - reply-SEQ: as the frist argument we got back the sequence number of the packet we sent the command in, as a 16bit integer.

- nSuccess: the second argument is only 1byte: 0 if command was executed successfully, and non-zero if some error occured.

# COMMAND REFERENCE

The arguments are written in the following form: `argName[i8](%)`. In the square brakets there is the bit count of the argument, prefixed by `i` as integer or `u` as unsigned integer. In the normal brackets you can find the dimension of the value represented by the argument. *ReARG* Depicts the arguments which are sent by Codie in the reply packet argument (after the reply-SEQ).

*Busy call behavior*: At most of the commands, the behavior when Codie still executing a command and there is a new, identical command call, is described.

## General commands

### Null

The Null command has the `0x0` command ID, which is never used in normal communication.

### Echo

ID: **`0x0001`**
ARGS: none

The echo command simply requests an echo. Each node should reply with a packet with:

- swapped `ROUTE`
- same `PRIO`
- MSB set in command ID
- `ARGLEN` =0

## MCU commands

### DriveSpeed

ID: **`0x1060`**
ARG: `speedLeft[i8](%), speedRight[i8](%)`
ReARG: `nSuccessful[u8]`

Set motor speeds to the given value in percents (0-100%). Speed values are signed, negative value means backwards.
**Replies:** *nSuccessful*: 0 means command has been successfully executed, any other value means error.

*Busy call behavior*: execution ends immediately, the command is nacked, and then the new command starts executing.

### DriveDistance

ID: **`0x1061`**
ARG: `distance[u16](mm), speedLeft[i8](%), speedRight[i8](%)`
ReARG: `nSuccessful[u8]`

Move a specified amount of distance (in mm) with given speeds (in percentage 0-100%). Moving backwards can be done with negative speeds.

*Currently both of Codie's tracks will travel the given distance, so covering arcs by specifying different speeds will not yield results as intended… Later we probably will improve this behavior so that distance will mean the length of trajectory of the center of the robot while following an arc.*

**Replies:** *nSuccessful*, 0 means command has been successfully executed, any other value means error.

*Busy call behavior*: execution ends immediately, the command is nacked, and then the new command starts executing.

### DriveTurn

ID: `0x1062`
ARG: `degree[u16](°), speed[i8](%)`
ReARG: `nSuccessful[u8]`

This command makes the robot turn given degrees in one place, by starting the tracks in different directions with given speed. Pozitive speeds turns left, negative speed turns right.
**Replies:** *nSuccessful*: 0 means command has been successfully executed, any other value means error.

*Busy call behavior*: execution ends immediately, the command is nacked, and then the new command starts executing.

### SonarGetRange

ID: `0x1063`
ARG: none
ReARG: `range[u16](mm)`

Request a range measurement with the sonar sensor. The measurement takes time (depends on how far there is a surface in front of Codie), from ~10ms up to 60ms.
**Replies:** *range*: the measured distance by the sonar in mm. 0 means error.

*Busy call behavior*: requests are queued, however you should NOT request a measurement until the reply arrives for the previous request, because the command queue might get jammed.

### SpeakBeep

ID: `0x1064`
ARG: `duration[u16](ms)`
ReARG: `nSuccessful[u8]`

Play a beep on the speaker. The frequency is fixed, you can only specify the duration in milliseconds.
**Replies:** *nSuccessful*: 0 means command has been successfully executed, any other value means error.

*Busy call behavior*: execution ends immediately, the command is nacked, and then the new command starts executing.

### LedSetColor

ID: `0x1065`
ARG: `ledMask[u16], hue[u8], saturation[u8], value[u8]`
ReARG: `nSuccessful[u8]`

Set the color of the LEDs. The first (least significant) 12 bits of `LedMask` is a binary mask of which of the 12 pieces of LEDs to change. Where the bit is set, then the color of the corresponding LED whill change to the given HSV value. Hue, saturation and value values are given on a full 8bit range of 0-255.
**Replies:** *nSuccessful*: 0 means command has been successfully executed, any other value means error.

#### BatteryGetSoc

ID: **`0x1069`**
ARG: none
ReARG: `stateOfCharge[u8](%)`

Get the state of charge of the battery in percentage 0-100%.
**Replies:** *stateOfCharge*: 0-100% the state of charge of the battery.

#### LightSenseGetRaw

ID: **`0x106a`**
ARG: none
ReARG: `lightValue[u16]`

Get a reading from the light sensor.
**Replies:** *lightValue*: A raw reading from the light sensor. The value is measured by ADC on 12 bits, so this can be 0-4096, 0 meaining the brightest and 4095 the darkest light.

#### LineGetRaw

ID: **`0x106b`**
ARG: none
ReARG: `valueLeft[u16], valueRight[u16]`

Get a reading from the line sensor. The line sensors on the bottom of Codie are simple infrared reflective optical sensors.
**Replies:** *valueLeft, valueRight*: Raw 12bit values from the ADC.

#### MicGetRaw

ID: **`0x106c`**
ARG: none
ReARG: `value[u16]`

Get a reading from the microphone.
**Replies:** *value*: Raw value ranging from 0 to about 2048. Several measurements are averaged into one value, the window of averaging is about 50ms.

### BLE commands

None so far.

### App commands

None so far.

## EXAMPLE APPLICATIONS

## QT >5.4 (LINUX / OSX)

Both application is written in Qt. They run only on linux (BlueZ 4.x/5.x) and OSX. For more on compatibility and the BLE stack, see the QT Bluetooth module documention. Qt's page on BLE can be useful as well.

#### Simple Client

This is a simple application, implementing only the essentials to connect to a Codie, execute a goDist command and print the reply. It consists only of a button. Please run it in a console to see debug and other messages.

Download source

#### Complex Client

This is a quite complex example application. It lets you discover nearby BLE devices, and by selecting a Codie robot, you can connect to it. The client implements all supported and usable commands. Commands can be sent with buttons, the command replies are printed in a semi-raw form to a textbox.

Download source