Spark ESP32 Library Description

Introduction

The Spark ESP32 Library (found in the repository https://github.com/paulhamsh/Spark) is a library to control a Positive Grid Spark 40 amp via Bluetooth using an ESP32 controller.

The library can use BLE or classic Bluetooth for communications.

It also communicates with the Spark app on Android or IOS, and allows app commands to reach the amp.

The library can be used to power a Bluetooth pedal to control the amp, and also facilitates multiple other control mechanism including Bluetooth midi, USB midi and serial DIN midi – plus any custom controls and ESP32 will support.

The Spark amp and how it is controlled via Bluetooth

The Spark amp has some manual controls on the top but is mostly controlled by the Spark app (on Android of IOS).

The amp controls allow selection of a predefined set of amps, selection between four presets, tap tempo change, and volume, tone, delay, mod and reverb changes.

The app allows full control of the amp – any effect model and be selected, turned on or off, the parameters altered and all those changes saved to a preset on the amp.

The app also allows these presets to be saved in ToneCloud and retrieved to the app, or to load presets created by others.

The communication between the amp and the app is over Bluetooth. There are two channels – one for audio and one for amp control.

The communication takes the form of messages in a very specific format (see Spark Protocol Description in the same github repository for additional information on that).

The messages carry the commands from the app, and responses from the amp. Also changes made directly on the amp are send to the app.

There is a message for each operation – turning on or off an effect, changing the model, changing a parameter (moving the virtual knob), saving a preset, changing to a different preset, send a preset from the app.

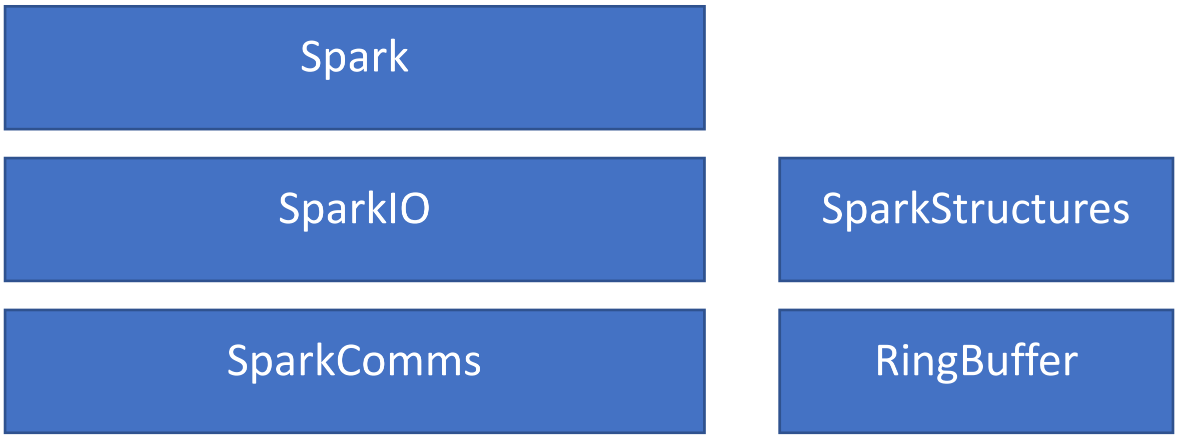
Library structure

The library has the following structure and files.

The main work is done in the SparkIO library, which packs and unpacks messages to and from the amp and the app. This allows a program to send commands to the amp and the app.

The Spark library wraps SparkIO to provide more intuitive functions and also to track the amp state locally. This allows a program to understand the amp state and also make appropriate changes.

The SparkComms library handles all the Bluetooth communication to the amp and the app, also also (optionally) a Bluetooth controller (like the Akai LPD 8 wireless or iRig Blue Board).



File	Function
`Spark.ino`	Wrapper for SparkIO to make commands clearer and also keep a local copy of the current amp settings
`Spark.h`	Header for the wrapper library
`SparkIO.ino`	Handles all the messages to and from the Spark amp and app. Packs and unpacks the message format into the structures defined in SparkStructures.h
`SparkIO.h`	Header file for the messaging library
`SparkComms.ino`	Handles all Bluetooth communications to and from the Spark amp, Spark app and a Bluetooth controller
`SparkComms.h`	Header file for the comms library
`RingBuffer.ino`	Implementation of a ring buffer
`RingBuffer.h`	Header file for ring buffers
`SparkStructures.h`	Definition of the structures used to interpret amp and app messages

Dependencies

The following libraries are required:

Library	Version
NimBLE	

Spark Library commands

Core functions

`void spark_state_tracker_start();`	Initialise the state tracking process
`bool update_spark_state();`	Update the local state by reading all unprocessed messages
`void update_ui();`	Send messages to the app to update the app 'view' of the amp (warning - this is advanced use and requires deepl understanding of the library)

Change functions

`void change_comp_model(char *new_eff);`	Change the effect model for the compressor
`void change_drive_model(char *new_eff);`	Change the effect model for the drive
`void change_amp_model(char *new_eff);`	Change the effect model for the amp
`void change_mod_model(char *new_eff);`	Change the effect model for the mod
`void change_delay_model(char *new_eff);`	Change the effect model for the delay
`void change_noisegate_onoff(bool onoff);`	Turn the noisegate on or off
`void change_comp_onoff(bool onoff);`	Turn the compressor on or off
`void change_drive_onoff(bool onoff);`	Turn the drive on or off
`void change_amp_onoff(bool onoff);`	Turn the amp on or off
`void change_mod_onoff(bool onoff);`	Turn the modulation on or off
`void change_delay_onoff(bool onoff);`	Turn the delay on or off
`void change_reverb_onoff(bool onoff);`	Turn the reverb on or off
`void change_noisegate_toggle();`	Toggle the noisegate (on to off, off to on)
`void change_comp_toggle();`	Toggle the compressor
`void change_drive_toggle();`	Toggle the drive
`void change_amp_toggle();`	Toggle the amp
`void change_mod_toggle();`	Toggle the mod
`void change_delay_toggle();`	Toggle the delay
`void change_reverb_toggle();`	Toggle the reverb
`void change_noisegate_param(int param, float val);`	Change a parameter on the noisegate
`void change_comp_param(int param, float val);`	Change a parameter on the compressor
`void change_drive_param(int param, float val);`	Change a parameter on the drive
`void change_amp_param(int param, float val);`	Change a parameter on the amp
`void change_mod_param(int param, float val);`	Change a parameter on the modulation
`void change_delay_param(int param, float val);`	Change a parameter on the delay
`void change_reverb_param(int param, float val);`	Change a parameter on the reverb
`void change_hardware_preset(int pres_num);`	Select a different hardware preset (0-3, 0x7f)
`void change_custom_preset(SparkPreset *preset, int pres_num);`	Send a custom preset to preset locations 0-3, 0x7f

Basic program to use the library

This program will connect to the Spark and keep a local state in sync with the amp and app.

It is the most basic use of the library but is passive and tracks the state only.

```
#include "Spark.h"

void setup() {
  // whatever setup code the ESP32 requires
  spark_state_tracker_start();  // set up data to track Spark and app state
}

void loop() {
  if (update_spark_state()) {
    // do your own checks and processing here
    // returns true if there is anything to do
  }
}
```

Program to change some parameters

This program is an example of many of the functions, including sending our own preset to the amp as the last action.

Each second it will send a new command to the amp as defined in the *switch* statement.

```
#include "Spark.h"


unsigned long tim;
void setup() {
  spark_state_tracker_start();

SparkPreset my_preset{
  0x0,0x7f,
  "F00DF00D-FEED-0123-4567-987654321004",
  "Paul Preset Test",
  "0.7",
  "My preset name",
  "icon.png",
  120.000000,{
    {"bias.noisegate", true,  2, {0.316873, 0.304245} },
    {"Compressor",     false, 2, {0.341085, 0.665754} },
    {"Booster",        true,  1, {0.661412} },
    {"Bassman",        true,  5, {0.768152, 0.491509, 0.476547, 0.284314,
0.389779} },
    {"UniVibe",        false, 3, {0.500000, 1.000000, 0.700000} },
    {"VintageDelay",   true,  4, {0.152219, 0.663314, 0.144982, 1.000000} },
    {"bias.reverb",    true,  7, {0.120109, 0.150000, 0.500000, 0.406755,
0.299253, 0.768478, 0.100000} }
  },
  0x00};

unsigned long tim;
int action;

void setup() {
  M5.begin();
  spark_state_tracker_start();

  tim = millis();
  action = 0;
}
```

```
void loop() {
  M5.update();
  //check timer and move to next 'action'
  if (millis() - tim > 1000) {
    tim = millis()
    action++;
    if (action > 20)
      action = 0;
  }

  switch (action) {
    case 1:     change_hardware_preset(0);                           break;
    case 2:     change_hardware_preset(1);                           break;
    case 3:     change_hardware_preset(2);                           break;
    case 4:     change_hardware_preset(3);                           break;
    case 5:     change_drive_toggle();                               break;
    case 6:     change_mod_toggle();                                 break;
    case 7:     change_delay_toggle();                               break;
    case 8:     change_reverb_toggle();                              break;
    case 9:     change_amp_param(AMP_GAIN, 0.5);                     break;
    case 10:    change_amp_param(AMP_BASS, 0.9);                     break;
    case 11:    change_amp_param(AMP_MID, 0.1);                      break;
    case 12:    change_amp_param(AMP_TREBLE, 0.6);                   break;
    case 13:    change_amp_param(AMP_MASTER, 0.3);                   break;
    case 14:    change_amp_model("94MatchDCV2");                     break;
    case 15:    change_drive_model("Booster");                       break;
    case 16:    change_delay_model("DelayMono");                     break;
    case 17:    change_mod_model("GuitarEQ6");                       break;
    case 18:    change_amp_model("Twin");                            break;
    case 19:    change_mod_onoff(false);                             break;
    case 20:    change_custom_preset(SparkPreset &my_preset, 0x7f);
                change_hardware_preset(0x7f);                        break;
  }


  update_spark_state();

}
```

SparkStructures

Preset format

`void spark_state_tracker_start();`	Initialise the state tracking process
`bool update_spark_state();`	Update the local state by reading all unprocessed messages
`void update_ui();`	Send messages to the app to update the app 'view' of the amp (warning - this is advanced use and requires deepl understanding of the library)

The Spark preset has a defined format in the messages sent to and from the amp.

The closest C representation of that is defined in SparkStructures.h:

```
#define STR_LEN 40

typedef struct  {
  uint8_t  curr_preset;
  uint8_t  preset_num;
  char UUID[STR_LEN];
  char Name[STR_LEN];
  char Version[STR_LEN];
  char Description[STR_LEN];
  char Icon[STR_LEN];
  float BPM;
  struct SparkEffects {
    char EffectName[STR_LEN];
    bool OnOff;
    uint8_t  NumParameters;
    float Parameters[10];
  } effects[7];
  uint8_t chksum;
} SparkPreset;
```

In the example program we used these preset details:

```
SparkPreset my_preset{
  0x0,0x7f,
  "F00DF00D-FEED-0123-4567-987654321004",
  "Paul Preset Test",
  "0.7",
  "My preset name",
  "icon.png",
  120.000000,{
    {"bias.noisegate", true,  2, {0.316873, 0.304245} },
    {"Compressor",     false, 2, {0.341085, 0.665754} },
    {"Booster",        true,  1, {0.661412} },
    {"Bassman",        true,  5, {0.768152, 0.491509, 0.476547, 0.284314, 
0.389779} },
    {"UniVibe",        false, 3, {0.500000, 1.000000, 0.700000} },
    {"VintageDelay",   true,  4, {0.152219, 0.663314, 0.144982, 1.000000} },
    {"bias.reverb",    true,  7, {0.120109, 0.150000, 0.500000, 0.406755, 
0.299253, 0.768478, 0.100000} }
  },
  0x00};
```

The preset has:

- preset location (for the amp)
- name
- UUID
- version number
- description
- reference to an icon
- BPM setting
- details for each of the seven effects in the preset chain
- a checksum

And each effect has the following details:

- name
- boolean showing whether on or off
- the number of parameters for the effect
- a list of the value for each parameter

And the description of these fields is in the table below.

Field	**Description**
`uint8_t  curr_preset;`	
`uint8_t  preset_num;`	
`char UUID[STR_LEN];`	
`char Name[STR_LEN];`	
`char Version[STR_LEN];`	
`char Description[STR_LEN];`	
`char Icon[STR_LEN];`	
`float BPM;`	
`struct SparkEffects {`	
`  char EffectName[STR_LEN];`	
`  bool OnOff;`	
`  uint8_t  NumParameters;`	
`  float Parameters[10];`	
`  } effects[7];`	
`uint8_t chksum;`	