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.

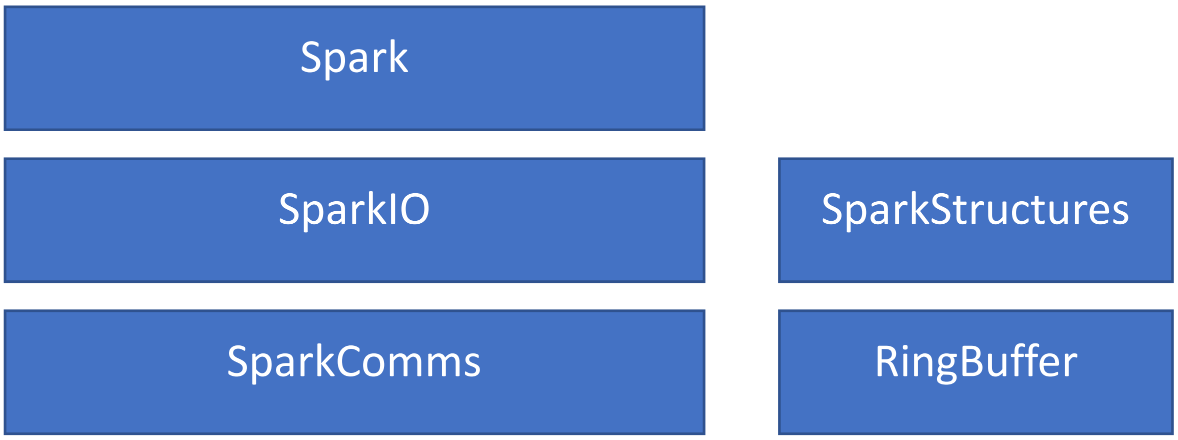
Library structure

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)

`spark_state_tracker_start()` will set up all the bluetooth communication channels needed and prepare the local copy of the amp state.

`update_spark_state()`processes all messages to send to the amp and received from app or amp. It updates the local copy of the amp state for the program to use.

This local copy is a SparkPreset structure in this variable:

```
SparkPreset presets[6]
```

This stores the four hardware presets (0-3) in `presets[0]` to `presets[3]`, preset 0x7f in `presets[4]` and the current amp state in `presets[5]`.

Messages to the amp or app can be created with the change functions in the table below.

The current state of the amp is available as described above. This should be adequate for most programs to send changes to the amp and view the current state.

It is also possible to process messages from the app and amp directly. This is more complex and exposes more of the SparkIO library. This is described in the section on processing amp and app commands.

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"


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(&my_preset, 0x7f);
                 change_hardware_preset(0x7f);                        break;
   }


   update_spark_state();

 }
```

Processing amp and app commands

When `update_spark_state()` is called it both updates the local copy of the amp state (and all the hardware presets) in response to messages from the app and the amp.

With each `loop()` all messages buffered to be sent will be sent, and a maximum of one *received* message will be processed - and the details are kept in variables accessible by the program.

```
unsigned int cmdsub;
SparkMessage msg;
SparkPreset preset;
```

The code of the message is stored in `cmdsub`.

The message details are either stored in `msg` or in `preset` if the message was the full preset details.

See the SparkStructures section for details on these two structures and how they are populated for the messages.

If `update_spark_state()` returns `true` then there is valid message information received from the amp or the app.

In which case, `cmdsub` can be used to act accordingly on that message.

For example, if a program wants to display the serial number of the Spark amp, it can do so by checking `cmdsub` for the value 0x0323 and then reading the serial number from `msg.str1` .

This example code shows this.

```
#include "Spark.h"


void setup() {
  spark_state_tracker_start();

}


void loop() {

  // need to ask for the serial number here but it uses SparkIO
  // so omitted for now
  // and it would also require a trigger for the request
  // which could be a timer or a sequence counter or a GPIO pin change

  if (update_spark_state()) {
    // only gets here if a valid message exists

    if (cmdsub == 0x0323) {
      // now msg.str1 contains the serial number
      Serial.print ("Serial number is ");
      Serial.println (msg.str1);
    }
  }


}
```

This example also demonstrates the asynchronous nature of the library – however we requested the serial number (this will be described later on as it will use the SparkIO library, and so isn't shown in the code above), the response will be handled in the loop code.

This Spark library does not wrap all the messages to the Spark, and requesting the serial number is one such message – to request that requires use of the SparkIO library.

(For the curious, the actual code we would put in setup is

```
spark_msg_out.get_serial();

spark_process();
```

and this will be explained in the section on SparkIO).

SparkStructures

Preset format

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;`	This is 1 if this represents the current state of the amp
`uint8_t  preset_num;`	The preset number – 0-3 or 0x7f
`char UUID[STR_LEN];`	The UUID of the preset
`char Name[STR_LEN];`	The name of the preset
`char Version[STR_LEN];`	The version of the preset
`char Description[STR_LEN];`	A description of the preset
`char Icon[STR_LEN];`	A reference to the icon (always “icon.png”)
`float BPM;`	The BPM setting for time based effects
`struct SparkEffects {`	
`  char EffectName[STR_LEN];`	The effect name
`  bool OnOff;`	Whether the effect is on (true) or off (false)
`  uint8_t  NumParameters;`	How many parameters the effect has
`  float Parameters[10];`	Each parameter (0.0 – 0.99)
`  } effects[7];`	
`uint8_t chksum;`	The checksum (only used by the app, not the amp)

Non-preset messages

For other messages (except the Hendrix license key) the SparkMessage structure is used for any received messages.

The actual content of each field is determined by the message being processd.

Field	Description
`uint8_t param1;`	Used for a byte value
`uint8_t param2;`	Used for a byte value
`uint8_t param3;`	Used for a byte value
`uint8_t param4;`	Used for a byte value
`uint32_t param5;`	Used for a 32 bit value
`float val;`	Used for parameter values (0-0.99)
`char str1[STR_LEN];`	Used for a string value
`char str2[STR_LEN];`	Used for a string value
`bool onoff;`	Used for on or off values

The table below shows the use of these fields.

Messages from the app

Code	Message description	param1	param2	param3	param4	param5	val	str1	str2	onoff
0201	Query preset details	current	preset number							
0210	Query current preset number									
0211	Query amp name									
0223	Query serial number									
022a	Query hardware preset checksums	01	02	03	04					
022f	Query firmware version									
0101	Send preset details	**SparkPreset structure used**								
0104	Send effect parameter	param number					value	effect		
0106	Change effect type							old	new	
0115	Enable / disable effect							effect		on / off
0138	Change hardware preset	preset								
0170	Send license key	**license_key[64] used**								

Messages from the amp

Code	Message description	param1	param2	param3	param4	param5	val	str1	str2	onoff
0301	Send preset details	**SparkPreset structure used**								
0310	Send current preset number	current	preset number							
0311	Send amp name							name		
0323	Send serial number							Serial		
032a	Semd hardware preset checksums	chk0	chk1	chk2	chk3					
032f	Query firmware version							version		
0304	Send effect parameter	param number					value	effect		
0306	Change effect type							old	new	
0315	Enable / disable effect							effect		on / off
0327	Store in hardware preset	preset								
0338	Change hardware preset	preset								
0363	Send tap tempo						tempo			
0370	Send license key									

SparkIO

Introduction

SparkIO packs and unpacks messages for the Spark amp and app.

It uses SparkComms to communicate with both app and amp.

All messages are buffered, and this library processes everything as a state machine as it interprets each message.

It must be called in `loop()` each and every loop.

All functions to send messages to the amp or app actually just queue the message in a buffer, and the messages are only sent and received once full buffers are present.

A by-product of this is that no message is necessarily sent when a function is called, so all code must allow for this by processing received messages asynchronously.

For example, a program cannot request a the amp serial number and expect the next response to be the serial number – it must send the request, and in `loop()` be prepared to receive the serial number response at any time.

If linking the two messages is absolutely necessary then some type of semaphore must be used (just a Boolean will probably be ok).

This is critical to understand and may cause bugs in your program if not observed.

Important to remember
All messages are buffered and processed sequentially.
The next message received may not be a response to a message you just buffered.
Never 'send' a message and block whilst waiting for the response.
Keep calling the library in `loop()`

SparkIO core functions

Description	Description
`void spark_start(bool passthru);`	Set up the SparkIO library. Passthru determines whether the library blocks app to amp and amp to app communication or allows it. If blocked then the program is controlling all messages processed and sent.
`void spark_process();`	Process the queues of messages to and from the amp
`void app_process()`	Process the queues of messages to and from the app

SparkIO change functions

Each of these functions will create a message and queue it on the queue for the class instance

Description	Description
`void create_preset(SparkPreset *preset);`	Create a preset details message
`oid turn_effect_onoff(char *pedal, bool onoff);`	Create a message to turn an effect on or off
`void change_hardware_preset(uint8_t curr_preset, uint8_t preset_num);`	Create a message to change the hardware preset
`void change_effect(char *pedal1, char *pedal2);`	Create a message to change an effect type
`void change_effect_parameter(char *pedal, int param, float val);`	Create a message to change the parameter for an effect
`void get_serial();`	Create a message to request the serial number
`void get_name();`	Create a message to request the amp name
`void get_hardware_preset_number();`	Create a message to request current hardware preset number
`void get_preset_details(unsigned int preset);`	Create a message to request the details of a specific preset
`void save_hardware_preset(uint8_t curr_preset, uint8_t preset_num);`	Create a message to save the current preset into a hardware preset
`void send_firmware_version(uint32_t firmware);`	Create a message to send a firmware version
`void send_0x022a_info(byte v1, byte v2, byte v3, byte v4);`	Create a message to send a set of preset checksums
`void send_preset_number(uint8_t preset_h, uint8_t preset_l);`	Create a message to send a preset number
`void send_key_ack();`	Create a message to send an acknowledgement for a license key
`void send_serial_number(char *serial);`	Create a message to send a serial number version
`void send_ack(unsigned int cmdsub);`	Create a message to send an acknowledgment for a cmdsub

Using SparkIO

If not using the Spark library then the connections to the Spark and app need to be created by calling the SparkComms function `connect_to_all()`. The SparkIO classes also need to be set up by calling `spark_start(true)` – where `true` sets up a bypass link so that all app to amp communication is sent directly and not blocked by the library.

The messages are processed in `loop()` with the following functions

```
connect_spark();              // reconnect if disconnected

spark_process();              // process amp messages

app_process();                // process app messages

if (spark_msg_in.get_message(&cmdsub, &msg, &preset) {

  // act on messages from the amp

}

if (app_msg_in.get_message(&cmdsub, &msg, & preset)) {

  // act on messages from the app

}
```

There are two class instances being used – `spark_msg_in` and `app_msg_in` which process the messages from the amp and app respectively.

There are also class instances for `spark_msg_out` and `app_msg_out` for sending messages.

As in the Spark library, the results of any message are available in `cmdsub`, `msg` and `preset`. The Spark library simply exposes this information from SparkIO.

SparkComms

Introduction

SparkComms handles all the Bluetooth communication with the app and the amp.

It has three function calls described in the table below.

Function	Description
`void connect_to_all();`	Sets up bluetooth and scans for the Spark amp. Also sets up Bluetooth servers for the Spark app to connect to. Optionally will scan for a Bluetooth MIDI controller
`void connect_spark();`	Reconnect to the amp if connection lost
`void connect_pedal();`	Reconnect to the MIDI controller if connection lost

To make a basic connection, `connect_to_all()` should be called in the `setup()` function of the program.

To reconnect, `connect_spark()` is called in the `loop()`.

Bluetooth libraries

There are two main libraries to provide Bluetooth capability for the ESP32 – the BLE library and NimBLE.

NimBLE is smaller but only works with BLE. It seems to reconnect better with BLE if a connection drops.

BLE is larger but also has a Bluetooth classic stack. It doesn't reconnect well with BLE if a connection drops, but handles Bluetooth classic disconnection well.

The Android app only works with Bluetooth classic so this must be defined if wanting to use Android. It also works for both Android and IOS, but doesn't handle disconnections well.

Configuration

There are two `#defines` that can be used to configure the library.

Function	Description
`#define CLASSIC`	If defined, this will force the use of the Arduino BLE library If not defined the NimBLE is used.
`#define BT_CONTROLLER`	If defined this will scan for a Bluetooth MIDI controller The current code works with the Akai LPD8 Wireless and the iRig Blueboard

Communication functions

These are the functions to send and receive data via Bluetooth.

Reading data is done a byte at a time

Writing data is done with a buffer of data.

For Bluetooth MIDI controller data the library does not yet provide a function for this, so the program must access the RingBuffer itself (see below)

Function	Description
`bool sp_available();`	Is there data available from the amp
`bool app_available();`	Is there data available from the app
`uint8_t sp_read();`	Read a byte from the amp
`uint8_t app_read();`	Read a byte from the app
`void sp_write(byte *buf, int len);`	Write a buffer to the amp
`void app_write(byte *buf, int len);`	Write a buffer to the app
`int ble_getRSSI();`	Get the RSSI level of the signal with the app (does not work if BT_CONTROLLER is defined – it crashes the library)

Tracking Bluetooth activity

The library also provides a trace of connections and message being sent and received so that an UI can show this information.

This is stored in two arrays:

```
#define SPK 0
#define APP 1
#define BLE_MIDI 2
#define USB_MIDI 3

#define NUM_CONNS 4

bool conn_status[NUM_CONNS];
unsigned long conn_last_changed[3][NUM_CONNS];
```

`conn_status` is true if that connection is connected, and false if disconnected.

`conn_last_changed` stores the value of `millis()` whenever there is activity. The three columns are:

```
#define TO 0              // for data sent to that connection
#define FROM 1            // for data received from the connection
#define STATUS 2          // for last time the connection status changed
```

So, for example, `conn_last_changed[FROM][APP]` stores the value of `millis()` when the library last received a byte from the app.

If a connection to the amp is lost, then `conn_status[SPK]` will be set to false, and `conn_last_changed[STATUS][SPK]` will store the value of `millis()` when this happened.

This data is exposed to the program so that it can be used to update a display on connection status or send and receive activity.

How the library handles asynchronous Bluetooth messages

To manage the asynchronous nature of Bluetooth messages received are stored in a RingBuffer. The library user does not need to access these are the read functions above manage the RingBuffers.

There are three RingBuffers, one each for the app, the amp and a Bluetooth MIDI controller:

RingBuffer	Description
`RingBuffer ble_in;`	Communications from the Spark amp
`RingBuffer ble_app_in;`	Communications from the Spark app
`RingBuffer midi_in;`	Communications from a Bluetooth MIDI controller

See the section on RingBuffer on how to read from a RingBuffer.

For example, for the Bluetooth MIDI RingBuffer, to see if data is available use:

```
!midi_in.is_empty()
```

and to read the data use:

```
uint8_t b;

midi_in.get(&b);
```