**Description of the communication protocol to the Positive Grid Spark 40 amplifier**

By Paul Hamshere with a great acknowledgement to Yury Tsybizov (ytsibizov) and Justin Nelson (jrnelson). Thanks to Ian McKellar for pointing out the MIDI SysEx similarity and that the packed data format is actually based on msgpack (www.msgpack.org)

**Overview of communication**

The Spark 40 amplifier communicates with the Spark app over Bluetooth. This seems to be 'serial bluetooth' for the Android app and 'BLE' for iOS.

The app sends messages to change preset, change an effect, change the parameter for an effect (eg gain). It can also request the details of each hardware preset, the name of the amp and the serial number.

In return, the amp will send messages when one of the presets is changed or when a knob is moved. This allows the app to mimic the settings on the amp at all points.

When the app starts, it asks the Spark for its name, serial number and all four hardware presets.

Then communication is event driven – either from the app or the amp.

**Overview of message format**

The bluetooth messages are exchanged in a specific data format. The terminology below is one I created to help understand the underlying structure.

Messages are exchanged in blocks. Each block contains one or more chunks. Each chunk contains data – which is all, or part of, the message.

Blocks and chunks appear to have size limits which means: messages span chunks, and chunks span blocks.

The simple messages are from the app to the amp, and are usually just one block, one chunk and the data.

Sending a preset, or receiving a preset, is more complex and involves multiple blocks and chunks.

Figure 1 shows the relationship between the blocks, chunks and data that make up the message.

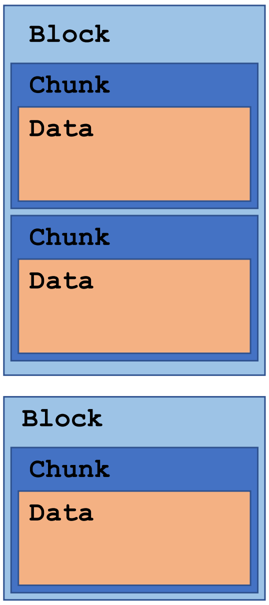


**Figure 1**

When the app sends a message then the Spark (usually) responds with an acknowledgement message.

Blocks sent to the amp seem to have a maximum size of 0xad.

Blocks sent from the amp seem to have a maximum size of 0x6a.

**Block format**

Each block has a header and then contains the chunk / data.

| **Offset** | **Length** | **Description** |
|---|---|---|
| 0 | 4 | 0x01fe0000 |
| 4 | 2 | Direction of the message:<br>0x41ff – from Spark<br>0x53fe – to Spark |
| 6 | 1 | Size of this block (including this header) |
| 7 | 9 | Zeros |
| 10 | | The chunk / data |

Figure 2 shows an example of a block header.

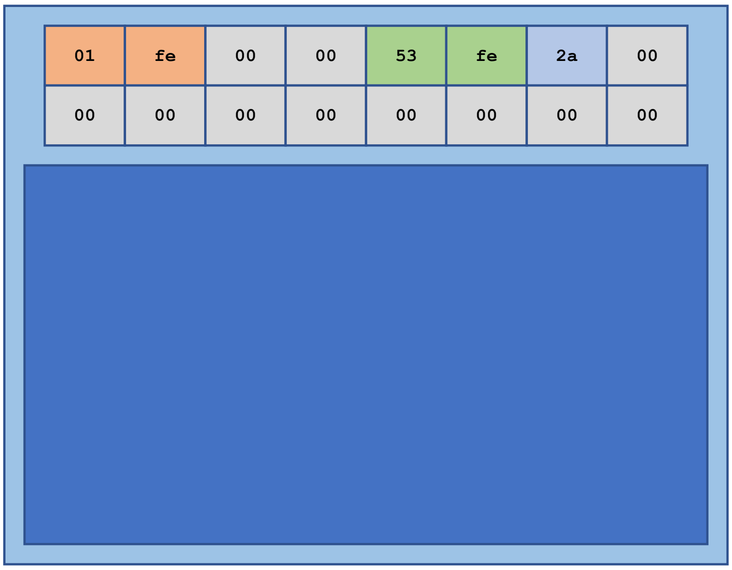


**Figure 2**

**Chunk format**

| **Offset (in block)** | **Length** | **Description** |
|---|---|---|
| 10 | 2 | 0xf001 |
| 12 | 1 | Sequence number |
| 13 | 1 | Checksum (8 bit Xor) |
| 14 | 1 | Command |
| 15 | 1 | Sub-command |
| 16 | | Data |
| | 1 | 0xf7 |

The chunk starts with fixed bytes of 0xf001 and ends with the byte 0xf7. This is very like the MIDI SysEx wrapper of 0xf0 and 0xf7.

The header includes a sequence number which increments with each message (so it remains consistent across chunks and blocks for the same message). When the amp acknowledges a message it contains the sequence number in the acknowledgement message.

The checksum is an 8-bit xor checksum of the data part – it excludes the chunk header and the f7 trailer.

The command and sub-command describe what the change is to the amp or from the amp (eg change gain on the amp model).

Figure 3 shows a block header, chunk header and trailer.

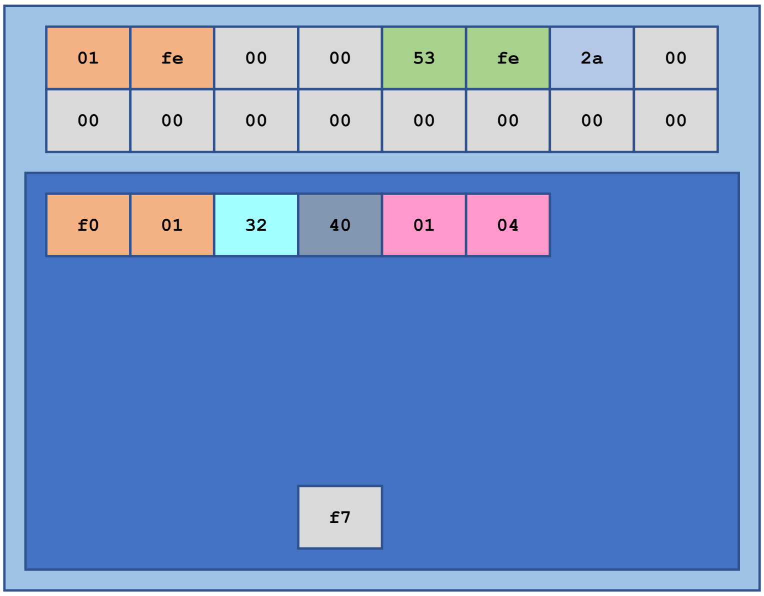


**Figure 3**

**Data format**

The message is a sequence of variables. Each variable has a distinct pattern which identifies it.

The variables are stored in the data section in sequences of 8 bytes. In the data section bytes have the top bit set to zero, so only carry 7 bits of data. The remaining 8th bit is packed into another byte which only contains the 8th bit of each of the bytes in the sequence.

So the format is the special byte containing the 8th bits, followed by seven data bytes.

Figure 4 shows the structure of the sequence – the '8th bits' byte followed by up to 7 data bytes.

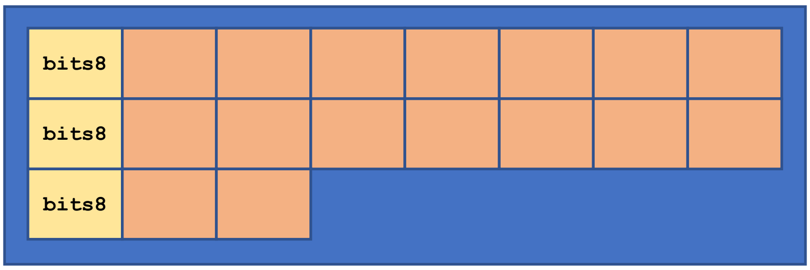


**Figure 4**

Figure 3 shows an example of the mapping for the missing $8^{th}$ bit. In this example, the data in the fourth data byte in the first sequence should have its $8^{th}$ bit added back (represented by bit 3 being set in the '$8^{th}$ bits' byte. And the same for the third and sixth bytes in the second sequence (bits 2 and 5 set in the '$8^{th}$ bits' byte.

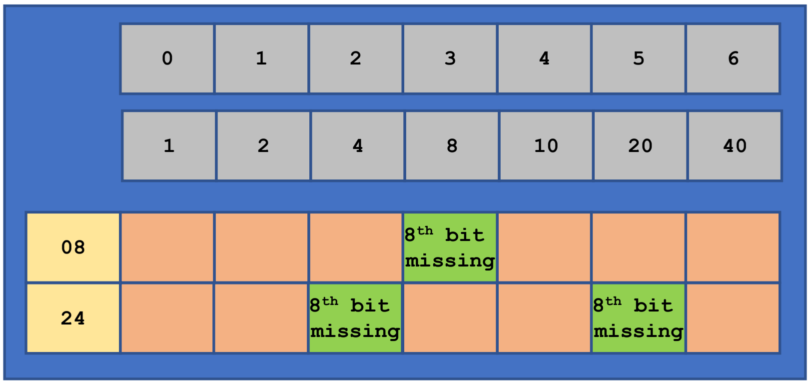


**Figure 5**

To interpret the data it is therefore essential to add back these bits.

**Overall structure**

Figure 6 shows a representation of the overall structure, including headers, trailers and format bytes. Figure 7 shows an example of the headers and footers.

These both show a single block / single chunk message and summarise the description so far.

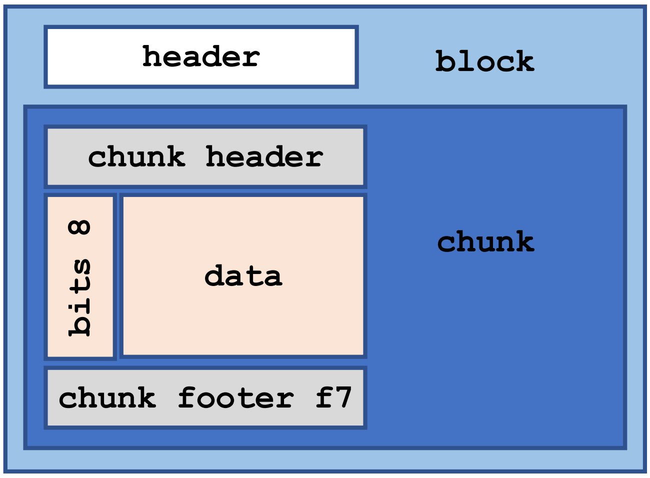


**Figure 6**


**Figure 7**

**Variable types**

The data format is based on msgpack (www.msgpack.org).

The data in the message is a set of variables – integers, strings, Booleans and floating-point values.

| **Type** | **Length** | **Description** | **First byte range** |
|---|---|---|---|
| Integer | 1 | Data value from 0x00 to 0x0f?<br>How is this distinguished from 0x00 as the start of an integer.<br>How is this distinguished from the first byte of the alternative short string? | 0x00 – 0x7f |
| Fixed array size | 1 | Data value from 0x00 to 0x0f, stored as data value + 0x90. | 0x90-0x9f |
| Short string (1-31 characters) | n+1 | First byte is the length + 0xa0, then the bytes of the string in ASCII encoding | 0xa0 – 0xbf |
| Alternative short string (1-31 characters) | n+2 | First byte is the length, next byte is the length + 0xa0, then the bytes of the string in ASCII encoding<br>(Unsure if this is limited to 15 characters but it would be logical given the apparent use of the first byte to describe the data type.) | 0x01 – 0x1f |
| Long string | n+2 | First byte is 0x59, then the length, then the bytes of the string | 0xd9 |
| Boolean Off | 1 | A single byte representing effect Off | 0xc2 |
| Boolean On | 1 | A single byte representing effect On | 0xc3 |
| Float | 5 | A float value – 4 bytes big endian with a preceding byte of 0xca | 0xca |

Figure 8 shows these data types in a visual format

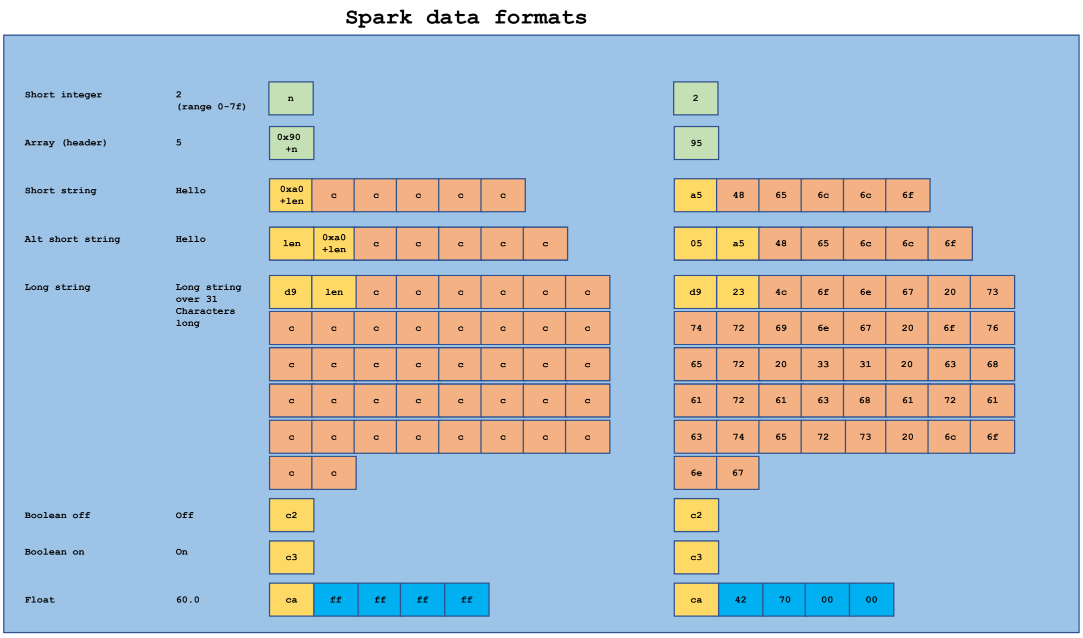


**Figure 8**

Figure 9 shows a completed message, with all headers, footers, data and format bytes.

This is data with a string "LA2AComp", a short integer of 1 and a float represented by 0x3f4d42c4 (with the 8th bit added back to the final 0x44)


**Figure 9**

Figure 10 shows this with explanatory labelling.


**Figure 10**

**Float representation**

Floats are based on the 4-byte IEEE-754 encoding. As with all the other data section formats, the bytes are 7-bit only and the missing 8$^{th}$ bits are in the first byte of any 8 byte sequence.

Figure 11 shows how this works.

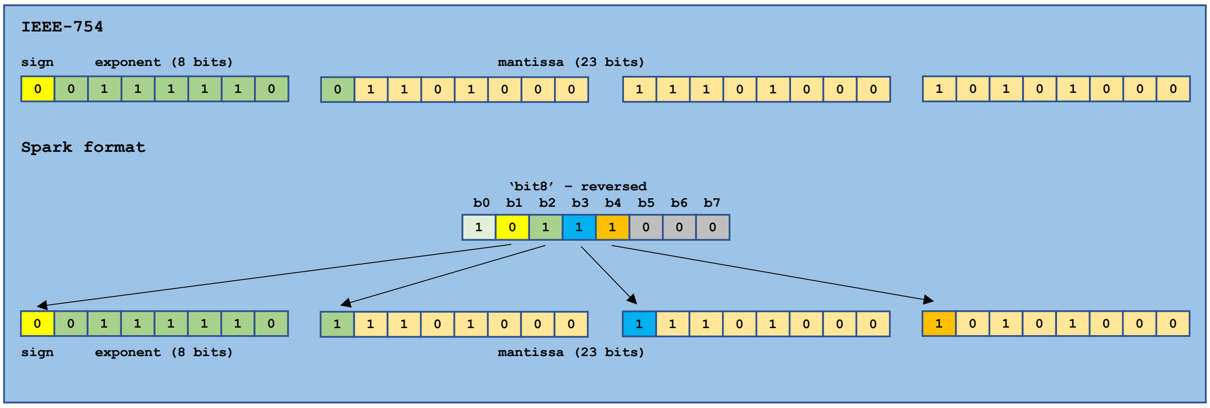


**Figure 11**

Special values are shown below

**Effects with toggle switches (such as Multi Head delay)**

| Hex | Float | Special meaning |
|---|---|---|
| 00 00 00 00 | 0.0 | False (for a toggle switch) |
| 3f 00 00 00 | 0.5 | True (for a toggle switch) |

**Digital delay**

| Hex | Float | Special meaning |
|---|---|---|
| 3f 00 33 33 | | 1s |
| 3f 06 19 19 | | 500ms |
| 3f 07 4c 4c | | 200ms |
| 3f 07 19 19 | | 50ms |

**Multi Head delay**

| Hex | Float | Special meaning |
|---|---|---|
| 3e 4c 4c 01 | | Head 0 + 2 |
| 3e 4c 4c 01 | | Head 0 + 1 (but the same hex as above – missing 8th bit in the data I extracted?) |
| 3f 19 19 01 | | Head 1 + 2 |
| 3f 4c 4c 01 | | Head 0 + 1 + 2 |

**Messages that span chunks and blocks**

The only messages large enough to span multiple chunks and blocks are those sending a complete preset, either to or from the amp. They can be identified by the command and sub-command (see later).

In these cases, the first three bytes of the data (after the format byte) represent which sub-chunk this is.

The size of the chunks and the data in these bytes depends on the direction of the message.

**Multi-chunk messages sent to the amp**

In this case, whilst the message spans multiple chunks, each chunk fills a block. The maximum sending block size is 0xad bytes, so the size of the chunk is 0x9b.

This is calculated as block size – block header – chunk header – chunk trailer (0xad – 0x10 – 0x06 – 0x01 = 0x9b)

The first four bytes of the chunk data are as in the table below - representing the format byte and the multi-chunk sub-header.

| Offset (in chunk) | Length | Description |
|---|---|---|
| 6 | 1 | First '8th bits' byte |
| 7 | 1 | Total number of chunks |
| 8 | 1 | Reference number of this chunk<br>(0 to total number of chunks – 1) |
| 9 | 1 | Size of this chunk (in data bytes which therefore excludes counting the '8th bit' bytes, max 0x80) |

The number of data bytes remaining is a count of useful data bytes - total bytes less the '8th bit' bytes.

Figure 12 shows the overall structure of a multi-chunk message sent to the amp.

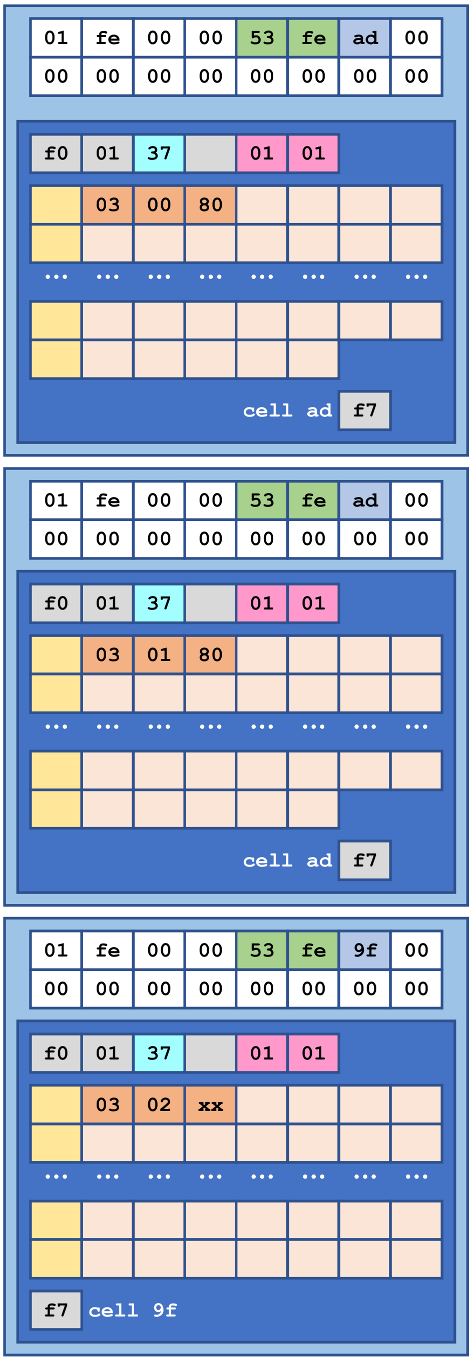


**Figure 12**

**Multi-chunk messages received from the amp**

In this case, whilst the message spans multiple chunks, there are multiple chunks in each block. Each chunk has a maximum size of 0x27 and the block has a maximum size of 0x6a.

The first four bytes of the chunk data are as in the table below – representing the format byte and the multi-chunk sub-header.

| Offset (in chunk) | Length | Description |
|---|---|---|
| 6 | 1 | First format byte |
| 7 | 1 | Total number of chunks |
| 8 | 1 | Reference number of this chunk<br>(0 to total number of chunks – 1) |
| 9 | 1 | For all chunks: Size of this chunk (in useful data bytes, so ignoring the ‘$8^{th}$ bit’ bytes) |

The number of data bytes remaining is a count of bytes excluding the ‘$8^{th}$ bit’ bytes and is present in each chunk. In all full chunks this is 0x19.

Figure 13 shows the overall structure of a multi-chunk message received from the amp.

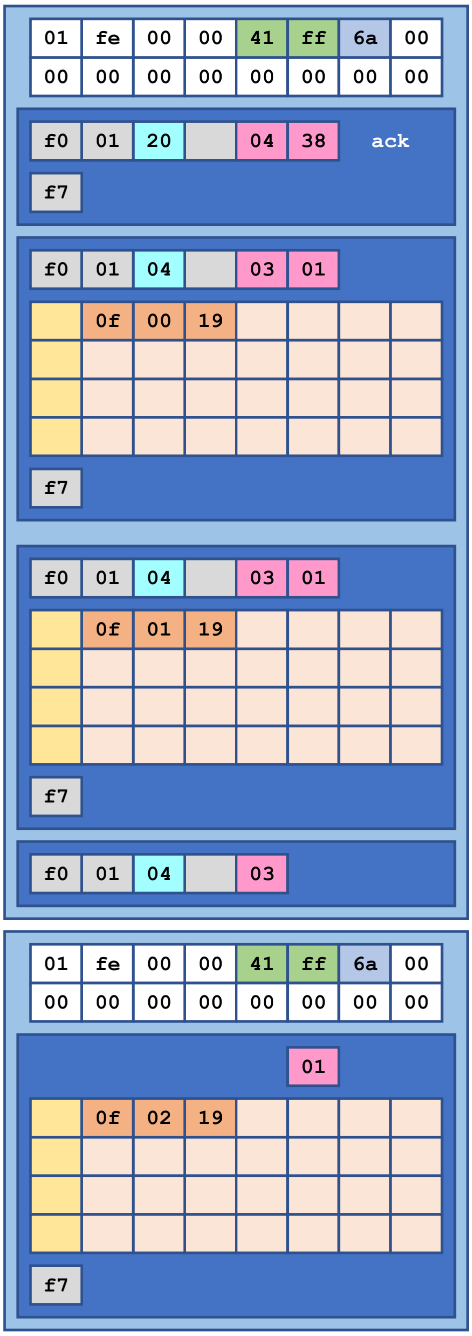


**Figure 13**

**Commands sent to the amp**

These are the commands which can be sent to the amp and the responses expected.

| Command | Sub-command | Meaning | Response |
|---|---|---|---|
| 01 | 01 | Send preset details to the amp | Acknowledge message |
| 01 | 04 | Send new effect parameter | None |
| 01 | 06 | Change effect to new effect | Acknowledge message |
| 01 | 15 | Enable / disable an effect | Acknowledge message |
| 01 | 38 | Change to a different preset | Acknowledge message |
| 02 | 01 | Get preset details from amp | Acknowledge message followed by preset information |
| 02 | 10 | Get current hardware preset number | Acknowledge message followed by preset number information |
| 02 | 11 | Get amp name (“Spark 40”) | Acknowledge message followed by amp name |
| 02 | 23 | Get amp serial number | |
| 02 | 24 | **Unknown** | |
| 02 | 2F | **Unknown** | |

**Commands sent from the amp**

These are the commands / responses sent from the amp. Response to the amp are unknown.

| Command | Sub-command | Meaning | Response |
|---|---|---|---|
| 03 | 06 | Change of effect (amp model) on the amp | None |
| 03 | 15 | Enable / disable an effect | None |
| 03 | 27 | Store current preset in hardware preset | None |
| 03 | 37 | Change of effect parameter on amp | None |
| 03 | 38 | Change of preset selected on the amp | None |
| 03 | 63 | Tap tempo | None |
| 03 | 01 | Response to a preset information query command | None |
| 04 | As per command received by amp | Acknowledgement from the amp that it received a message. | |

**Detail of commands**

**0x0101 – see later**

**0x0104 – change effect parameter**

| Type | Length | Content | Example |
|---|---|---|---|
| Alternative short string | n+2 | Effect name | 0x04 0xa4 Twin |
| Integer | 1 | Number of the parameter starting at 0 | 0x00 (Gain) |
| Float | 5 | Value for the parameter (0-1.0, with 1.0 representing 10 in the user interface) | 0xca<br>0x3f 0x21 0x72 0x13 |

**0x0106 – swap effects**

| Type | Length | Content | Example |
|---|---|---|---|
| Alternative short string | n+2 | Old effect name | 0x08 0xa8 LA2AComp |
| Alternative short string | n+2 | New effect name | 0x08 0xa8 BlueComp |

**0x0115 – enable / disable effect**

| Type | Length | Content | Example |
|---|---|---|---|
| Alternative short string | n+2 | New effect name | 0x08 0xa8 BlueComp |
| Boolean | 1 | New status<br>0xc2 off<br>0xc3 on | 0xc3 |

**0x0138 – change to a new hardware preset**

| Type | Length | Content | Example |
|---|---|---|---|
| Integer | 1 | 0 | 0x00 |
| Integer | 1 | New preset number 0-3, 0x7f | 0x03 |

**0x0201 – get preset information**

| Type | Length | Content | Example |
|---|---|---|---|
| Integer | 2 | factory hardware preset number 0x00-0x03<br><br>current preset 0x7f<br><br>current live preset 0x0100<br><br>current hardware stored preset 0x80-0x83 | 0x00 0x03 |
| Short integer | 1 x 30 | 30 bytes of 0x00 (which, when adding the 8th bit in a separate byte, looks like 34 bytes of 00 in the raw data) | 0x00 |

**0x0211 – get amp name**

No data in message for this command

**0x0223 – get amp serial number**

No data in message for this command

**0x0224 – unknown command**

Not known what this command does

| Type | Length | Content | Example |
|---|---|---|---|
| Fixed array | 1 | 0x94 | 0x94<br>(4) |
| Integer | 1 | 0x00 | 0x00 |
| Integer | 1 | 0x01 | 0x01 |
| Integer | 1 | 0x02 | 0x02 |
| Integer | 1 | 0x03 | 0x03 |

**0x0306 – change of effect (amp model) on the amp**

| Type | Length | Content | Example |
|---|---|---|---|
| Alternative short string | n+2 | Old amp name | 0x0d 0xad<br>GK800 |
| Alternative short string | n+2 | New amp name | 0x05 0xa5<br>Twin |

When this is sent to the app, the app responds by sending five parameter messages back to the amp (0x0104). These cover the five parameters (gain, bass, middle, treble and volume). It is not clear how these parameter values are derived.

**0x0315 – enable / disable effect**

| Type | Length | Content | Example |
|---|---|---|---|
| Alternative short string | n+2 | New effect name | 0x08 0xa8 BlueComp |
| Boolean | 1 | New status<br>0xc2 off<br>0xc3 on | 0xc3 |

Only seems to come from the amp when using the Mod or Delay knobs.

**0x0327 – current preset stored to hardware on the amp**

| Type | Length | Content | Example |
|---|---|---|---|
| Integer | 1 | 0 | 0x00 |
| Integer | 1 | Number of the preset where settings are stored | x02<br>(2) |

**0x0337 – change of parameter for effect on the amp**

| Type | Length | Content | Example |
|---|---|---|---|
| Alternative short string | n+2 | Effect name | 0x04 0x24<br>Twin |
| Integer | 1 | Parameter number | 0x00<br>(0) (Gain)<br>**OR**<br>0x03<br>(3) (Bass)<br>**OR**<br>0x04<br>(4) (Master) |
| Float | 5 | New value | 0xca<br>0x3e 0x6d 0x5b 0x37 |

**0x0338 – change of preset selected on the amp**

| Type | Length | Content | Example |
|---|---|---|---|
| Integer | 1 | 0 | 0x00 |
| Integer | 2 | Number of the new preset | 0x02<br>(2) |

**0x0363 – tap tempo**

| Type | Length | Content | Example |
|---|---|---|---|
| Float | 5 | New tempo value | 0xca<br>0x3e 0x6d 0x5b 0x37 |

**0x04nn – acknowledgement**

This has command 0x04 and the same sub-command as was issued to the amp. It has the same sequence number as the command issued to the app.

There is no body to this message – just the chunk header and the trailer (0xf7)

**0x0101 – send preset**

A new preset is a multi-chunk message, so the first three bytes of each new chunk are the chunk sub-header.

The preset format contains data for the preset, and then information for each effect – 7 in total.

Each effect contains data for the effect, and then a value for each parameter in the effect.

The final byte of the preset is currently unknown. It could be a checksum but doesn't impact the data sent as the preset.

| **Type** | **Length** | **Content** | **Example** |
|---|---|---|---|
| Integer | 1 | 0 | 0x00 |
| Integer | 1 | If saving to a hardware preset location this is 0-3 otherwise 0x7f | 0x7f |
| Long string | 36 | UUID of preset | |
| Short string | n+1 | Name | 0xad<br>Spooky Melody |
| Short string | n+1 | Version | 0xa3<br>0.7 |
| Short string / Long string | n+1 / n+2 | Description | 0xb7<br>Description for Alternative Preset 1 |
| Short string | n+1 | Icon name | 0xa8<br>icon.png |
| Float | 5 | BPM | 0xca<br>0x42 0x70 0x00 0x00<br>(60.0) |
| Fixed array | 1 | Number of effects – always 7. | 0x97<br>(7) |
| *Effect 0* | | *See below* | |
| *Parameter 0* | | *See below* | |
| *Parameter 1* | | *See below* | |
| *Effect 1* | | *See below* | |
| *Parameter 0* | | *See below* | |
| *Effect 2* | | *See below* | |
| *Effect 3* | | *See below* | |
| *Effect 4* | | *See below* | |
| *Effect 5* | | *See below* | |
| *Effect 6* | | *See below* | |
| Integer | 1 | **Unknown** | |

Each effect then has a section describing the effect (7 effects in total)

| **Type** | **Length** | **Content** | **Example** |
|---|---|---|---|
| Short string | n+1 | Effect name | 0x08 0xa8 BlueComp |
| Boolean | 1 | Status<br>0xc2 off<br>0xc3 on | 0xc3<br>(On) |
| Fixed array | 1 | Number of parameters for this effect | 0x94<br>(4) |

And then each parameter has a section describing the value for the parameter

| Type | Length | Content | Example |
|---|---|---|---|
| Integer | 1 | Parameter reference | 0x01 (1) |
| Fixed array | 1 | Number of values (always 1) | 0x91 |
| Float | 5 | Value for this parameter | 0xca 0x3e 0x35 0x55 0x3f |

It seems this is a broken implementation of msgpack, because the fixed array for each parameter looks more like it should be key:value pairs, and the fixed array wrapping the float seems redundant.

The current interpretation breaks msgpack because what should be:

fixmap(3) = {{1, float}, {2, float}, {3, float}}

is actually:

fixarray(3)=[0, fixarray(1)[float], 1], fixarray(1)[float], 2, fixarray(1)[float]

Parsing can be achieved by ignoring any integer within a fixarray and by extracting the float from within a fixarray of length 1 – but this is a workaround for a broken implementation.

It seems the amp and app do not pack or unpack msgpack properly – they must be looking for the specific data rather than unpacking and indexing the unpacked data

Figure 14 shows the overall structure.


Hardware preset number 00 7f
UUID 0 7 0 7 9 0 6 3 - 9 4 A 9 - 4 1 B 1 - A B 1 D - 0 2 C B 5 D 0 0 7 9 0
Name D a r k S o u l
Version 0 . 7
Description 1 - C l e a n
Icon i c o n . p n g
BPM 120
Number of effects (fixarray) 97
Effect name b i a s . n o i s e g a t e
On / off C3
Number of parameters (fixarray) 93
Parameter 0 00 91 0.52
Parameter 1 01 91 0.88
Parameter 2 02 91 0.13
.........
Effect name b i a s . r e v e r b
On / off C3
Number of parameters (fixarray) 97
Parameter 0 00 91 0.52
… … … … … … …
Parameter 6 06 91 0.13
Unknown filler 17

**Figure 14**

**Appendix 1 – Preset locations**

The amp has 4 presets which can be selected from the top panel. These are represented by presets 0-3 in this document.

There is also preset 7f which is the current 'in memory' preset – this is the one which holds whatever the app sends over.

It is possible to save this to any of the hardware preset locations – just use the 'send new preset' command with the preset location between 0 and 3.

Normally the preset commend sends preset 7f and then selects preset 7f as the active one, but it doesn't have to be that way – it can send it to any preset location.

## Appendix 2 - Effect and amp names

### Noisegate

| Name | Spark name |
|---|---|
| Noisegate | bias.noisegate |

### Compressors

| Name | Spark name |
|---|---|
| LA Comp | LA2AComp |
| Sustain Comp | BlueComp |
| Red Comp | Compressor |
| Bass Comp | BassComp |
| Optical Comp | BBEOpticalComp |

### Drive

| Name | Spark name |
|---|---|
| Booster | Booster |
| Tube Drive | DistortionTS9 |
| Over Drive | Overdrive |
| Fuzz Face | Fuzz |
| Black Op | ProCoRat |
| Bass Muff | BassBigMuff |
| Guitar Muff | GuitarMuff |
| Bassmaster | MaestroBassmaster |
| SAB Driver | SABdriver |

### Amps

| Name | Spark name |
|---|---|
| Silver 120 | RolandJC120 |
| Black Duo | Twin |
| AD Clean | ADClean |
| Match DC | 94MatchDCV2 |
| Tweed Bass | Bassman |
| AC Boost | AC Boost |
| Checkmate | Checkmate |
| Two Stone SP50 | TwoStoneSP50 |
| American Deluxe | Deluxe65 |
| Plexiglass | Plexi |
| JM45 | OverDrivenJM45 |
| Lux Verb | OverDrivenLuxVerb |
| RB 101 | Bogner |
| British 30 | OrangeAD30 |
| American High Gain | AmericanHighGain |
| SLO 100 | SLO100 |
| YJM100 | YJM100 |
| Treadplate | Rectifier |
| Insane | EVH |
| Switch Axe | SwitchAxeLead |
| Rocker V | Invader |
| BE 101 | BE101 |
| Pure Acoustic | Acoustic |
| Fishboy | AcousticAmpV2 |
| Jumbo | FatAcousticV2 |
| Flat Acoustic | FlatAcoustic |
| RB-800 | GK800 |
| Sunny 3000 | Sunny3000 |
| W600 | W600 |
| Hammer 500 | Hammer500 |

**Modulation**

| Name | Spark name |
| --- | --- |
| Tremolo | Tremolo |
| Chorus | ChorusAnalog |
| Flanger | Flanger |
| Phaser | Phaser |
| Vibrato | Vibrato01 |
| UniVibe | UniVibe |
| Cloner Chorus | Cloner |
| Classic Vibe | MiniVibe |
| Tremolator | Tremolator |
| Tremolo Square | TremoloSquare |

**Delay**

| Name | Spark name |
| --- | --- |
| Digital Delay | DelayMono |
| Echo Filt | DelayEchoFilt |
| Vintage Delay | VintageDelay |
| Reverse Delay | DelayReverse |
| Multi Head | DelayMultiHead |
| Echo Tape | DelayRe201 |

**Reverb**

| Name | Spark name |
| --- | --- |
| All Reverbs | bias.reverb |

**Appendix 3 – app startup messages**

These are the messages sent when the app connects to the Spark amp.

| **Direction** | **Command / subcommand** | **Description** | **Example** |
|---|---|---|---|
| To amp | 0x0211 | Get amp name | |
| From amp | 0x0311 | Amp name | 0x08 Spark 40 |
| To amp | 0x0224 | **Unknown** | 0x14 0x00 0x01 0x02 0x03 |
| From amp | 0x0223 | Get serial number | Serial number<br>0x77 |
| To amp | 0x0201 | Get preset 0 | |
| From amp | 0x0301 | Preset 0 | |
| To amp | 0x0201 | Get preset 1 | |
| From amp | 0x0301 | Preset 1 | |
| To amp | 0x0201 | Get preset 2 | |
| From amp | 0x0301 | Preset 2 | |
| To amp | 0x0201 | Get preset 3 | |
| From amp | 0x0301 | Preset 3 | |
| To amp | 0x0210 | | |
| From amp | 0x0310 | | |
| To amp | 0x022f | | |

**Appendix 4 – Calculating effective data bytes from total number of bytes including format byte**

This visualises how to calculate the number of data bytes to go into the multi-chunk sub-header:

```
total_bytes – int ( (total_bytes+2) / 8 )
```

<table>
<tr><th></th><th>bytes</th><th>bytes+2</th><th>int( (bytes+2) / 8)</th><th>bytes – int( (bytes+2) / 8)</th></tr>
<tr><td><table><tr><td></td><td></td><td></td><td></td><td>n</td><td>f7</td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table></td><td>1</td><td>3</td><td>0</td><td>1</td></tr>
<tr><td><table><tr><td></td><td></td><td></td><td></td><td>n</td><td>n</td><td>f7</td><td></td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table></td><td>2</td><td>4</td><td>0</td><td>2</td></tr>
<tr><td><table><tr><td></td><td></td><td></td><td></td><td>n</td><td>n</td><td>n</td><td>f7</td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table></td><td>3</td><td>5</td><td>0</td><td>3</td></tr>
<tr><td><table><tr><td></td><td></td><td></td><td></td><td>n</td><td>n</td><td>n</td><td>n</td></tr><tr><td>f7</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table></td><td>4</td><td>6</td><td>0</td><td>4</td></tr>
<tr><td><table><tr><td></td><td></td><td></td><td></td><td>n</td><td>n</td><td>n</td><td>n</td></tr><tr><td>x</td><td>n</td><td>f7</td><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table></td><td>6</td><td>8</td><td>1</td><td>5</td></tr>
<tr><td><table><tr><td></td><td></td><td></td><td></td><td>n</td><td>n</td><td>n</td><td>n</td></tr><tr><td>x</td><td>n</td><td>n</td><td>f7</td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table></td><td>7</td><td>9</td><td>1</td><td>6</td></tr>
<tr><td><table><tr><td></td><td></td><td></td><td></td><td>n</td><td>n</td><td>n</td><td>n</td></tr><tr><td>x</td><td>n</td><td>n</td><td>n</td><td>f7</td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table></td><td>8</td><td>10</td><td>1</td><td>7</td></tr>
<tr><td><table><tr><td></td><td></td><td></td><td></td><td>n</td><td>n</td><td>n</td><td>n</td></tr><tr><td>x</td><td>n</td><td>n</td><td>n</td><td>n</td><td>f7</td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table></td><td>9</td><td>11</td><td>1</td><td>8</td></tr>
</table>

<table>
<tr><td><table><tr><td></td><td></td><td></td><td></td><td>n</td><td>n</td><td>n</td><td>n</td></tr><tr><td>x</td><td>n</td><td>n</td><td>n</td><td>n</td><td>n</td><td>f7</td><td></td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table></td><td>10</td><td>12</td><td>1</td><td>9</td></tr>
<tr><td><table><tr><td></td><td></td><td></td><td></td><td>n</td><td>n</td><td>n</td><td>n</td></tr><tr><td>x</td><td>n</td><td>n</td><td>n</td><td>n</td><td>n</td><td>n</td><td>f7</td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table></td><td>11</td><td>13</td><td>1</td><td>10</td></tr>
<tr><td><table><tr><td></td><td></td><td></td><td></td><td>n</td><td>n</td><td>n</td><td>n</td></tr><tr><td>x</td><td>n</td><td>n</td><td>n</td><td>n</td><td>n</td><td>n</td><td>n</td></tr><tr><td>f7</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table></td><td>12</td><td>14</td><td>1</td><td>11</td></tr>
<tr><td><table><tr><td></td><td></td><td></td><td></td><td>n</td><td>n</td><td>n</td><td>n</td></tr><tr><td>x</td><td>n</td><td>n</td><td>n</td><td>n</td><td>n</td><td>n</td><td>n</td></tr><tr><td>x</td><td>n</td><td>f7</td><td></td><td></td><td></td><td></td><td></td></tr></table></td><td>14</td><td>16</td><td>2</td><td>12</td></tr>
</table>

**Appendix 5 - TODO**

What I still don't understand:

* What the byte 0x11 is for in each pedal preset.

It is msgpack for an array of size 1! Actually 0x91.

* Whether the nibble or byte data type really exists

Not in msgpack – it is a range 0x00 to 0x7f.

* Why the chunk header sometimes contains the count of data bytes remaining (excluding format bytes) and sometimes doesn't

It always does – but sometimes that is 0x80 which is shown as 0x00 in the 7bit / 8bit encodig scheme

* What the format bytes are really used for, especially the one in front of the chunk header - it seems to have a special meaning

It is the 7bit/8 bit encoding!

* What that final byte is for in the preset - a checksum? If so it isn't checked

Sometimes MIDI SysEx has a checksum, so probably that

* What the byte after the sequence byte is for

Still a mystery!

**Appendix 6 – SysEx 7bit/8bit encoding functions**

The best reference I could find is in the Arduino MIDI library and refers to Ruin & Wesen's SysEx encoder/decoder - http://ruinwesen.com

Sadly this reference is broken so there is no clarity on whether this encoding has a recognised name.

Ian McKellar

https://git.sr.ht/~ianloic/spark-usb-midi

```
def decode_block(block: bytes)->bytes:
  assert(len(block) > 0)
  top = block[0]
  bottom = block[1:]
  assert(len(bottom) <= 7)
  decoded = []
  for i, b in enumerate(bottom):
    if top & (2**i):
      decoded.append(b | 2**7)
    else:
      decoded.append(b)
  return bytes(decoded)
```

**My code**

https://github.com/paulhamsh/Spark-Parser/blob/main/MidiControl/SparkClass.py

```
chunk_len = len (chunk)
num_seq = int ((chunk_len + 6) / 7)
bytes7 = b''

for this_seq in range (0, num_seq):
    seq_len = min (7, chunk_len - (this_seq * 7))
    bit8 = 0
    seq = b''
    for ind in range (0,seq_len):
        dat = chunk[this_seq * 7 + ind:
        if dat & 0x80 == 0x80:
            bit8 |= (1<<ind)
        dat &= 0x7f
        seq +=  bytes([dat])
    bytes7 += bytes([bit8]) + seq


chunk_len = len (data7bit)
num_seq = int ((chunk_len + 7) / 8)
data8bit = b''
for this_seq in range (0, num_seq):
    seq_len = min (8, chunk_len - (this_seq * 8))
    seq = b''
    bit8 = data7bit[this_seq * 8]
    for ind in range (0,seq_len-1):
        dat = data7bit[this_seq * 8 + ind + 1]
        if bit8 & (1<<ind) == (1<<ind):
            dat |= 0x80
        seq +=  bytes([dat])
    data8bit += seq
```

**Arduino MIDI**

https://github.com/FortySevenEffects/arduino_midi_library/blob/master/src/MIDI.cpp

```
/*! \brief Encode System Exclusive messages.
 SysEx messages are encoded to guarantee transmission of data bytes higher than
 127 without breaking the MIDI protocol. Use this static method to convert the
 data you want to send.
 \param inData The data to encode.
 \param outSysEx The output buffer where to store the encoded message.
 \param inLength The length of the input buffer.
 \param inFlipHeaderBits True for Korg and other who store MSB in reverse order
 \return The length of the encoded output buffer.
 @see decodeSysEx
 Code inspired from Ruin & Wesen's SysEx encoder/decoder - http://ruinwesen.com
 */
unsigned encodeSysEx(const byte* inData,
                     byte* outSysEx,
                     unsigned inLength,
                     bool inFlipHeaderBits)
{
    unsigned outLength  = 0;     // Num bytes in output array.
    byte count          = 0;     // Num 7bytes in a block.
    outSysEx[0]         = 0;

    for (unsigned i = 0; i < inLength; ++i)
    {
        const byte data = inData[i];
        const byte msb  = data >> 7;
        const byte body = data & 0x7f;

        outSysEx[0] |= (msb << (inFlipHeaderBits ? count : (6 - count)));
        outSysEx[1 + count] = body;

        if (count++ == 6)
        {
            outSysEx   += 8;
            outLength  += 8;
            outSysEx[0] = 0;
            count       = 0;
        }
    }
    return outLength + count + (count != 0 ? 1 : 0);
}

/*! \brief Decode System Exclusive messages.
 SysEx messages are encoded to guarantee transmission of data bytes higher than
 127 without breaking the MIDI protocol. Use this static method to reassemble
 your received message.
 \param inSysEx The SysEx data received from MIDI in.
 \param outData The output buffer where to store the decrypted message.
 \param inLength The length of the input buffer.
 \param inFlipHeaderBits True for Korg and other who store MSB in reverse order
 \return The length of the output buffer.
 @see encodeSysEx @see getSysExArrayLength
 Code inspired from Ruin & Wesen's SysEx encoder/decoder - http://ruinwesen.com
 */
unsigned decodeSysEx(const byte* inSysEx,
                     byte* outData,
                     unsigned inLength,
                     bool inFlipHeaderBits)
{
    unsigned count  = 0;
    byte msbStorage = 0;
    byte byteIndex  = 0;

    for (unsigned i = 0; i < inLength; ++i)
    {
```

```
        if ((i % 8) == 0)
        {
            msbStorage = inSysEx[i];
            byteIndex  = 6;
        }
        else
        {
            const byte body     = inSysEx[i];
            const byte shift    = inFlipHeaderBits ? 6 - byteIndex : byteIndex;
            const byte msb      = byte(((msbStorage >> shift) & 1) << 7);
            byteIndex--;
            outData[count++] = msb | body;
        }
    }
    return count;
}
```

**Appendix 7 – msgpack**

This format is described at:

www.msgpack.org

https://github.com/msgpack/msgpack/blob/master/spec.md

There are multiple implementations and the python one is obtained by:

**python -m pip install msgpack**

The Spark data is not an exact msgpack implementation because it does not start as an array, and the effect and effect parameters are malformed as arrays.

The data is like this:

```
\x97
       \xaebias.noisegate
       \xc2
       \x93
              \x00
              \x91
                     \xca>\r\xa1\xec
              \x01
              \x91
                     \xca>f\x08\xd1
              \x02
              \x91
                     \xca\x00\x00\x00\x00
```

This should be an array of 7 elements – one for each effect. Then each effect has three values – name, on/off status, an array of parameters – best as a key/value pair except that msgpack doesn't allow integers as keys.

But the array content is like this:

```
['bias.noisegate',
True,
[0, [ 0.1201], 1],
[0.3314],
2,
[0.0000]
```

Partly because the array is really three entries per pedal, not one, and partly because each parameter is two entries not one.