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)

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.

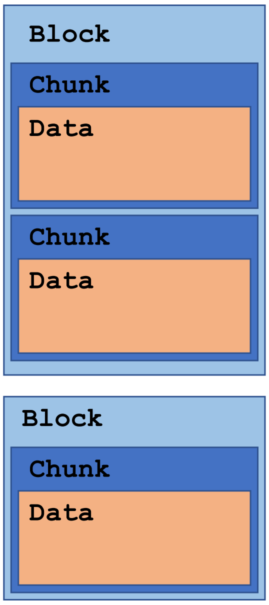

Figure 1

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: 0x41ff – from Spark 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.

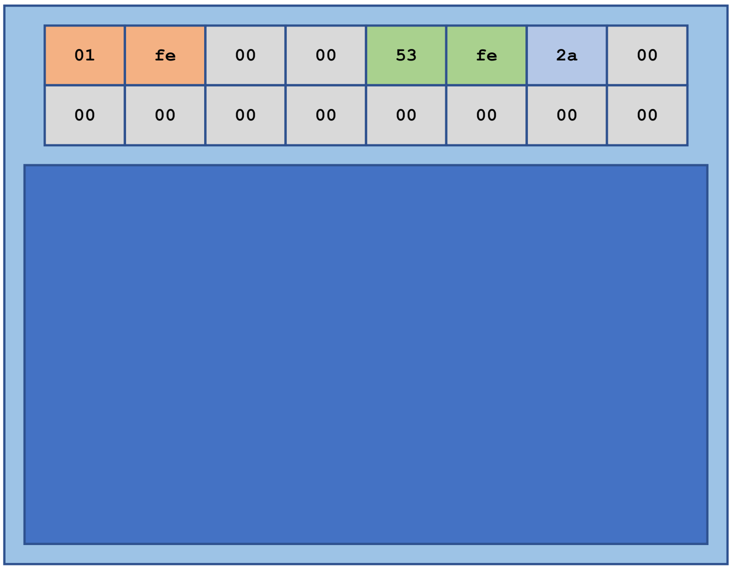

Figure 2

Chunk format

Offset (in block)	Length	Description
10	2	0xf001
12	1	Sequence number
13	1	**Unknown**
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.

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 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.

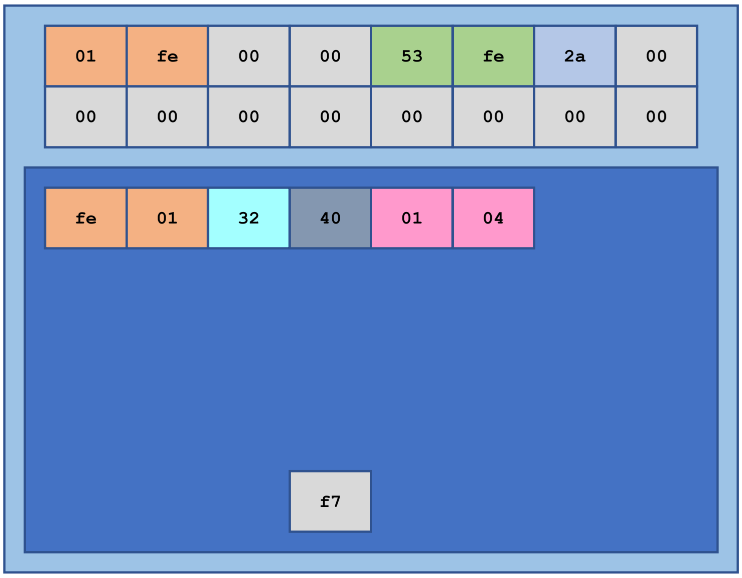

Figure 3

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. The first byte is a format byte and the remaining seven bytes contain the message data.

Figure 4 shows the structure of format byte followed by up to 7 data bytes.

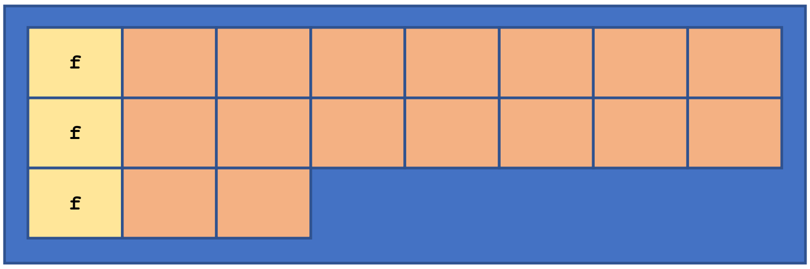

Figure 4

The format byte indicates where each **new** variable starts in the remaining seven bytes. It is a bitmap of the start locations each variable that follows.

For example, if the bits are referenced from 0 to 7, if bit 0 is set then a new variable starts in the second bytes of this sequence. If bit 6 is set then a new variable starts in the eighth byte of this sequence.

Figure 3 shows an example of this mapping. New variables start on bit 3 of the first row represented by 0x08 in the format byte, and on bits 2 and 5 on the second row represented by 0x24 in the format byte.

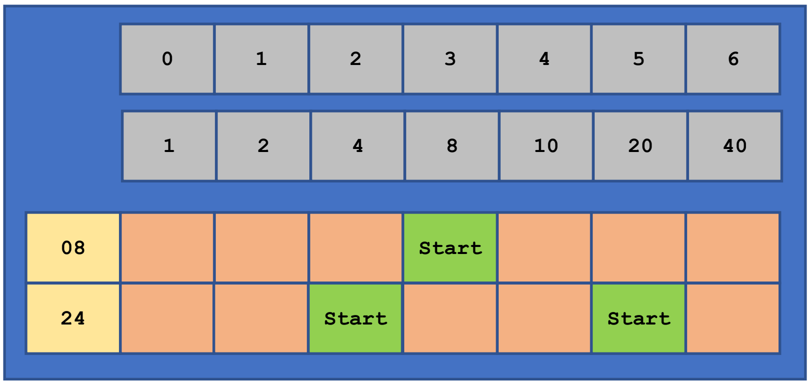

Figure 5

In this manner it is possible to detect where each new variable starts in the structure. (Although it seems that if the bitmap indicates a new variable to start within another variable, this is ignored.)

It is not necessary to use this format information to read the variables. They are always in a predictable order and length in the message data so it is enough to know the overall structure.

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.

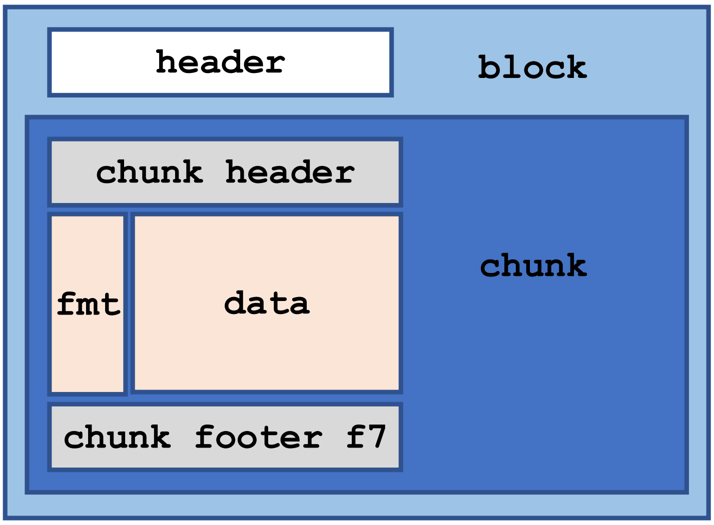

Figure 6

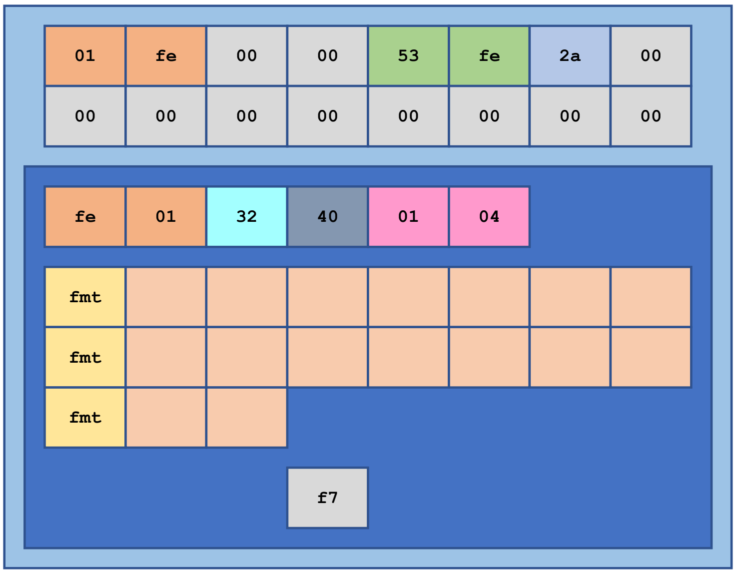

Figure 7

Variable types

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

They have no identifiers to describe what they represent.

In many cases it is possible to determine the data type from the first byte. This appears not to work for short integers, which may have a range 0x00-0xff and is not distinguishable from the first byte of the alternative short string and the first byte of an integer. (This is mostly seen in a simple message to change the value of an effect parameter.)

Type	**Length**	**Description**	**First byte range**
Short integers	1	Data value from 0x00 to 0x0f?	Anything?
Alterative short integer	1	Data value from 0x00 to 0xf, stored as data value + 0x10.	0x10-0x1f
Integer	2	Data value from 0x00 to 0xff Starts with a byte of 0x00	0x00
Short string (1-31 characters)	n+1	First byte is the length + 0x20, then the bytes of the string in ASCII encoding	0x20 – 0x3f
Alternative short string (1-31 characters)	n+2	First byte is the length, next byte is the length + 0x20, then the bytes of the string in ASCII encoding (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 – 0x0f
Long string	n+2	First byte is 0x59, then the length, then the bytes of the string	0x59
Boolean Off	1	A single byte representing effect Off	0x42
Boolean On	1	A single byte representing effect On	0x43
Float	5	A float value – 4 bytes big endian with a preceding byte of 0x4a, stored in a 7 bit format – see description later	0x4a

Figure 8 shows these data types in a visual format

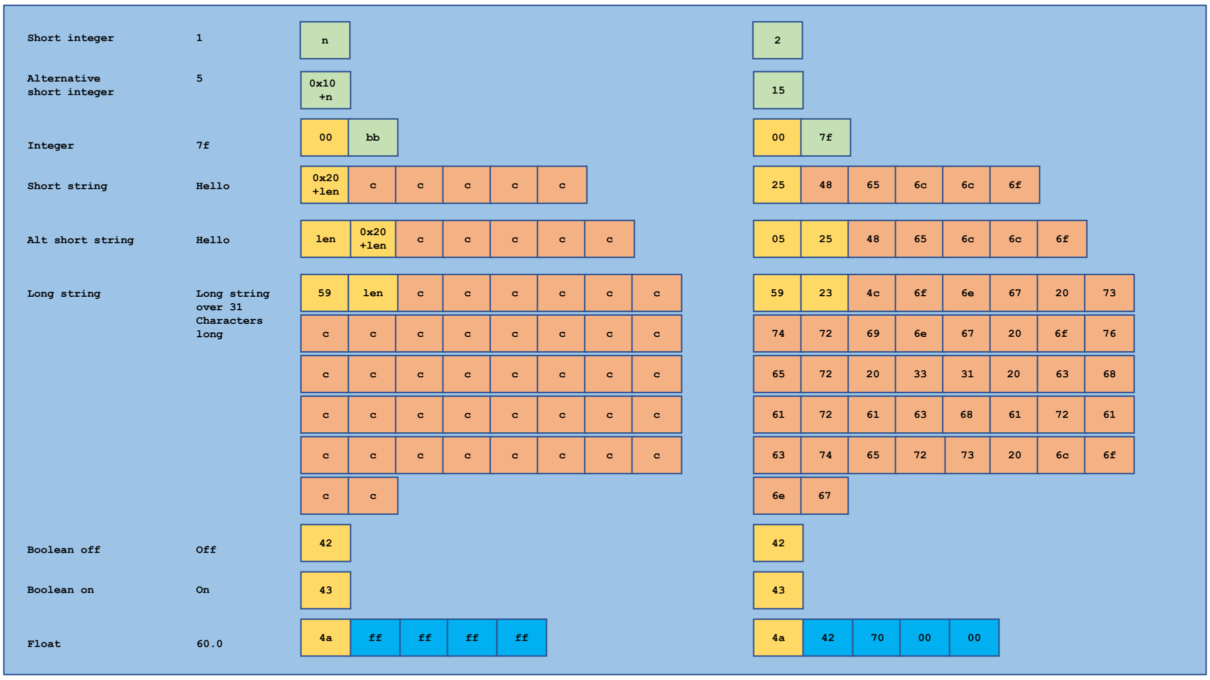

Figure 8

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 0x3f4d4244.

Figure 9

Figure 10 shows this with explanatory labelling.

Figure 10

Float representation

Floats are not a simple IEEE-754 encoding but I can’t actually work out how they work.

Bit 7 of each byte is set to 0. This is ok for the exponent byte because that is a sign bit.

For the next byte, this is the last digit of the exponent which is important.

It could be that the bits are bit shifted to the right to allow for the zero bit 7. Exploring this seems to work although the mapping between the UI and the float is no longer clear – if you treat it as a normal float then it seems to be that the float value is 1/10th that of the UI (5.0 in the UI maps to 0.5 in the float).

So is the float representation is the IEEE-754 encoding but will the mantissa bit-shifted so that bit 7 of each byte is empty (a 0)?

Figure 11 shows the comparison of the two formats.

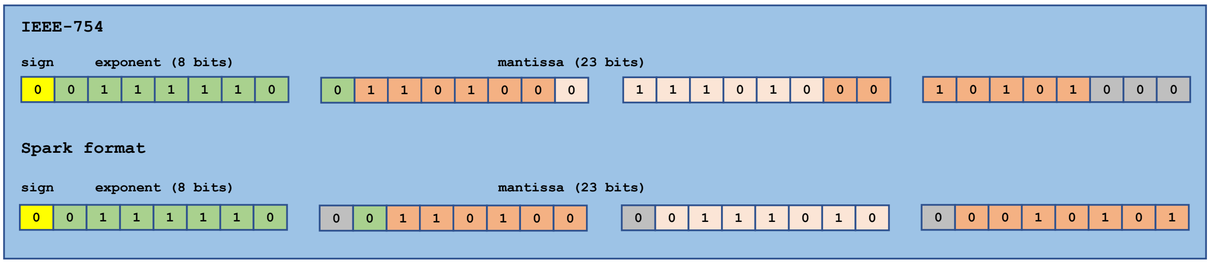

Figure 11

If so, then that the floats used are in the range 0.0 – 2.0, mapping to 0 – 10 on the app UI, and to 0 – 1.0 in the json format found in the Android APK.

Special values are shown below

Hex	Float	Special meaning
00 00 00 00	0.0	
3f 00 00 00	0.5	

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 (excluding the format byte) represent which 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 format byte For the first chunk this has to have bit 2 set (logical or with 0x04) For the last chunk this has to have bit 2 clear (logical and with 0xfb)
7	1	Total number of chunks
8	1	Reference number of this chunk (0 to total number of chunks – 1)
9	1	For the last chunk: Size of this chunk (in useful data bytes, so ignoring the format bytes) For other chunks: 0x00

The number of data bytes remaining is a count of bytes excluding the format bytes, and whilst always in the final chunk seems not to be in prior chunks when sending to the amp.

The Format byte seems to have a special difference in this scenario, and has bit 2 (| 0x04) set for the all the chunks except the last, where is seems to have bit 2 empty (& 0xfb) but may have bit 4 set (| 0x08).

This implies this bit can be used to determine the final chunk although it is not required for that. Having bit 3 set is inconsistent with the use of the format byte as it indicates data starting in byte 4, which is still part of this inner header.

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

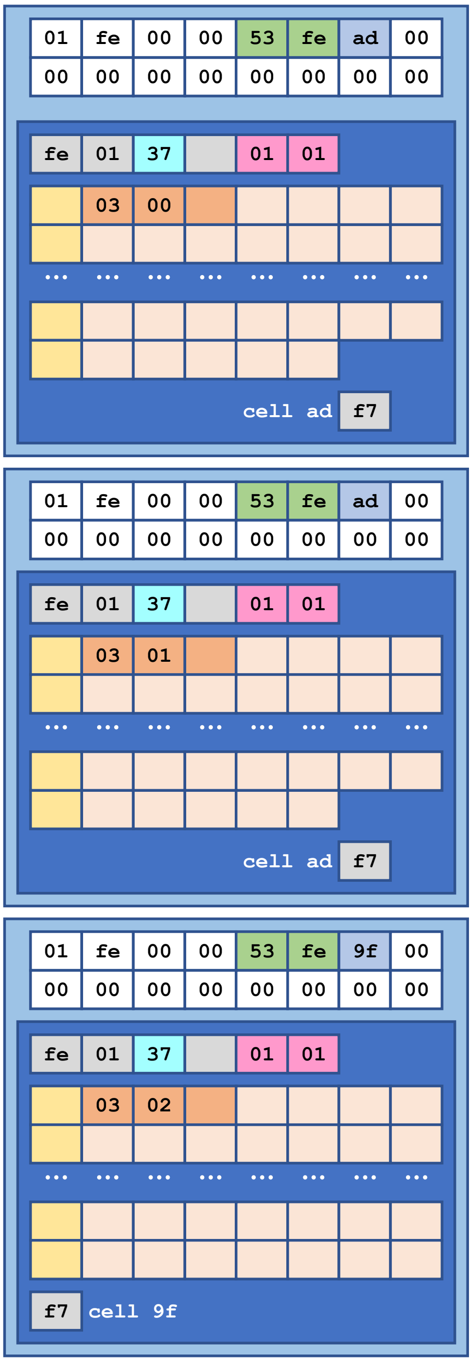
01 fe 00 00 53 fe ad 00
00 00 00 00 00 00 00 00
fe 01 37 01 01
03 00
cell ad f7
01 fe 00 00 53 fe ad 00
00 00 00 00 00 00 00 00
fe 01 37 01 01
03 01
cell ad f7
01 fe 00 00 53 fe 9f 00
00 00 00 00 00 00 00 00
fe 01 37 01 01
03 02
f7 cell 9f

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 (0 to total number of chunks – 1)
9	1	For all chunks: Size of this chunk (in useful data bytes, so ignoring the format bytes)

The number of data bytes remaining is a count of bytes excluding the format bytes and is present in each chunk. In all full chunks this is 0x19.

The format bytes has not been studied to see if it has a special significance in the same way as when sending to the amp.

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

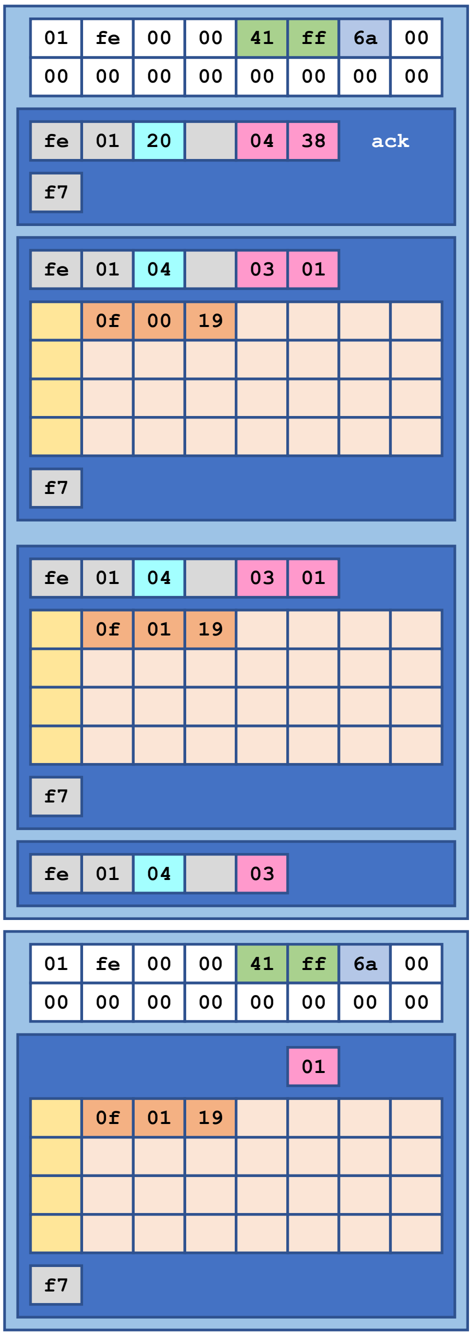
01 fe 00 00 41 ff 6a 00
00 00 00 00 00 00 00 00
fe 01 20 04 38
ack
f7
fe 01 04 03 01
0f 00 19
f7
fe 01 04 03 01
0f 01 19
f7
fe 01 04 03
01 fe 00 00 41 ff 6a 00
00 00 00 00 00 00 00 00
01
0f 01 19
f7

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	11	Get amp name ("Spark 40")	Acknowledge message followed by amp name
02	23	Get amp serial number	
02	24	**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	**Unknown**
03	37	Change of effect parameter on amp	**Unknown**
03	38	Change of preset selected on the amp	**Unknown**
03	01	Response to a preset information query command	**Unknown**
04	As per command received by amp	Acknowledgement from the amp that it received a message.	

Detail of commands

0x0104 – change effect parameter

Type	**Length**	**Content**	**Example**
Alternative short string	n+2	Effect name	0x04 0x24 Twin
Short 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)	0x4a 0x3f 0x21 0x72 0x13

0x0106 – swap effects

Type	**Length**	**Content**	**Example**
Alternative short string	n+2	Old effect name	0x08 0x28 LA2AComp
Alternative short string	n+2	New effect name	0x08 0x28 BlueComp

0x0115 – enable / disable effect

Type	Length	Content	Example
Alternative short string	n+2	New effect name	0x08 0x28 BlueComp
Boolean	1	New status 0x42 off 0x43 on	0x43

0x0138 – change to a new hardware preset

Type	Length	Content	Example
Integer	2	New preset number 0-3, 0x7f	0x00 0x03

0x0201 – get preset information

Type	Length	Content	Example
Integer	2	preset number 0-3 0x7f? 0x100?	0x00 0x03
Short integer	1 x 34	34 bytes of 0x00	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
Alternative short integer	1	0x14	0x14 (4)
Short integer	1	0x00	0x00
Short integer	1	0x01	0x01
Short integer	1	0x02	0x02
Short 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 0x2d GK800
Alternative short string	n+2	New amp name	0x05 0x25 Twin

0x0337 – change of parameter for effect on the amp

Type	Length	Content	Example
Alternative short string	n+2	Effect name	0x04 0x24 Twin
Alternative short integer	1	Parameter number	0x00 (0) (Gain) **OR** 0x03 (3) (Bass) **OR** 0x04 (4) (Master)
Float	5	New value	0x4a 0x3e 0x6d 0x5b 0x37

0x0338 – change of preset selected on the amp

Type	Length	Content	Example
Integer	2	Number of the new preset	0x00 0x02 (2)

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	2	Hardware preset location	0x00 0x07
Long string	36	UUID of preset	
Short string	n+1	Name	0x2d Spooky Melody
Short string	n+1	Version	0x23 0.7
Short string	n+1	Description	0x37 Description for Alternative Preset 1
Short string	n+1	Icon name	0x28 icon.png
Float	5	BPM	0x4a 0x42 0x70 0x00 0x00 (60.0)
Short integer	1	Number of effects – always 7.	0x17 (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*	
Short 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 0x28 BlueComp
Boolean	1	Status 0x42 off 0x43 on	0x43 (On)
Short integer	1	Number of parameters for this effect	0x12 (2)

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

Type	**Length**	**Content**	**Example**
Alternative short integer	n+1	Parameter reference	0x01 (1)
Alternative short integer	1	**Unknown** 0x11	0x11
Float	1	Value for this parameter	0x4a 0x3e 0x35 0x55 0x3f

Figure 14 shows this overall structure.

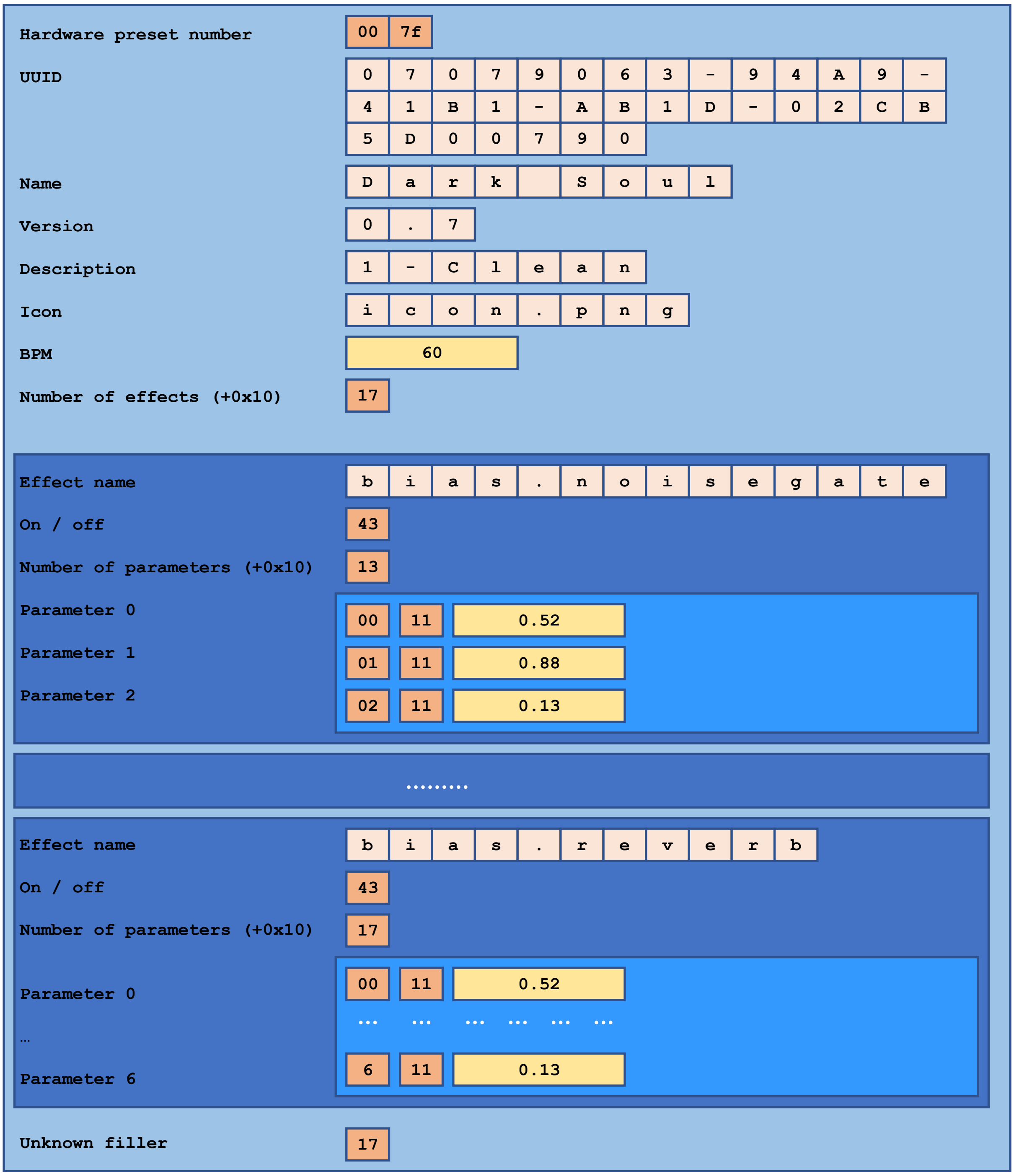
Hardware preset number
00 7f
UUID
07079063-9A9-
41B1-AB1D-02CB
5D00790
Name
Dark Soul
Version
0.7
Description
1-Clean
Icon
icon.png
BPM
60
Number of effects (+0x10)
17
Effect name
bias.noisegate
On / off
43
Number of parameters (+0x10)
13
Parameter 0
00 11 0.52
Parameter 1
01 11 0.88
Parameter 2
02 11 0.13
....
Effect name
bias.reverb
On / off
43
Number of parameters (+0x10)
17
Parameter 0
00 11 0.52
...
Parameter 6
6 11 0.13
Unknown filler
17

Figure 14

Appendix 1 - 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 2 – 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 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	

Appendix 3 – 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 4 - TODO

What I still don't understand:

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

* Whether the nibble or byte data type really exists

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

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

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

* What the byte after the sequence byte is for