DJ Link Packet Analysis

James Elliott
Deep Symmetry, LLC

May 25, 2017

Abstract

The protocol used by Pioneer professional DJ equipment to communicate and coordinate performances can be monitored to provide useful information for synchronizing other software, such as light shows and sequencers. By creating a "virtual CDJ" that sends appropriate packets to the network, other devices can be induced to send packets containing even more useful information about their state. This article documents what has been learned so far about the protocol, and how to accomplish these tasks.

1 Mixer Startup

When the mixer starts up, after it obtains an IP address (or gives up on doing that and self-assigns an address), it sends out what look like a series of packets[1] simply announcing its existence to UDP port 50000 on the broadcast address of the local network.
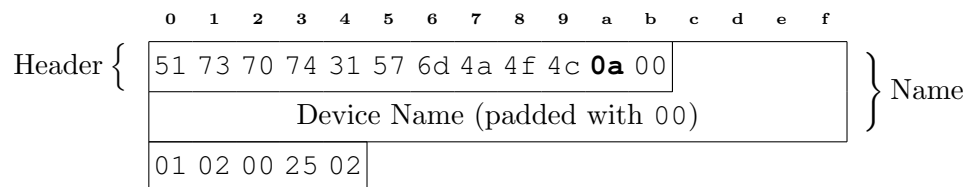

Figure 1: Initial announcement packets from Mixer

---

[1]The packet capture described in this analysis can be found at `https://github.com/brunchboy/dysentery/raw/master/doc/assets/powerup.pcapng`
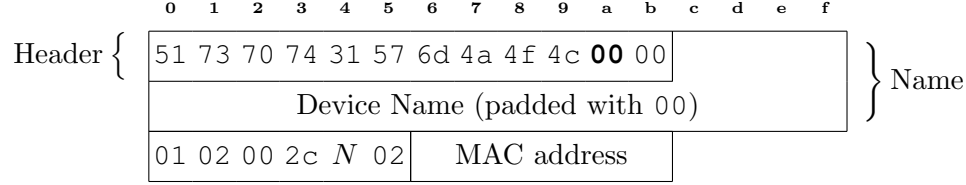
Figure 2: First-stage Mixer device number assignment packets

These have a data length[2] of 37 bytes, appear roughly every 300 milliseconds, and have the content shown in Figure 1.

The tenth byte (inside what is labeled the header) is bolded because its value changes in the different types of packets which follow.

After about three of these packets are sent, another series of three begins. It is not clear what purpose these packets serve, because they are not yet asserting ownership of any device number; perhaps they are used when CDJs are powering up as part of the mechanism the mixer can use to tell them which device number to use based on which network port they are connected to?

In any case, these three packets have a data length of 44 bytes, are again sent to UDP port 50000 on the local network broadcast address, at roughly 300 millisecond intervals, and have the content shown in Figure 2.

The value N at byte 36 is 1, 2, or 3, depending on whether this is the first, second, or third time the packet is sent.

After these comes another series of three numbered packets. These appear to be claiming the device number for a particular device, as well as announcing the IP address at which it can be found. They have a data length of 50 bytes, and are again sent to UDP port 50000 on the local network broadcast address, at roughly 300 millisecond intervals, with the content shown in Figure 3.

I identify these as claiming/identifying the device number because the value D at byte 46 is the same as the device number that the mixer uses to identify itself (`0x21`) and the same is true for the corresponding packets seen from the CDJs (they use device numbers 2 and 3, as they are connected to those ports/channels on the mixer).

As with the previous series of three packets, the value N at byte 47 takes on the values 1, 2, and 3 in the three packets.

These are followed by another three packets, perhaps the last stage of claiming the device number, again at 300 millisecond intervals, to the same port 50000. These shorter packets have 38 bytes of data and

---

[2]Values within packets are shown in hexadecimal, while packet lengths and byte offsets are discussed in decimal.
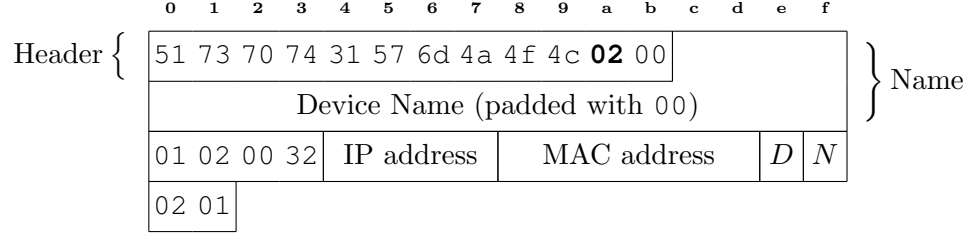
Header {

```
     0   1   2   3   4   5   6   7   8   9   a   b   c   d   e   f
```
51 73 70 74 31 57 6d 4a 4f 4c **02** 00

Device Name (padded with 00)

01 02 00 32 | IP address | MAC address | D | N

02 01

} Name

Figure 3: Second-stage Mixer device number assignment packets

Header {

```
     0   1   2   3   4   5   6   7   8   9   a   b   c   d   e   f
```
51 73 70 74 31 57 6d 4a 4f 4c **04** 00

Device Name (padded with 00)
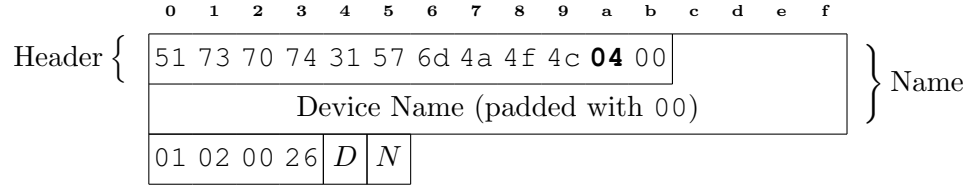
01 02 00 26 | D | N

} Name

Figure 4: Final-stage Mixer device number assignment packets

the content shown in Figure 4.

As before the value D at byte 36 is the same as the device number that the mixer uses to identify itself (0x21) and N at byte 37 takes on the values 1, 2, and 3 in the three packets.

Once those are sent, the mixer seems to settle down and send what looks like a keep-alive packet to retain presence on the network and ownership of its device number, at a less frequent interval. These packets are 54 bytes long, again sent to port 50000 on the local network broadcast address, roughly every second and a half. They have the content shown in Figure 5.

Header {

```
     0   1   2   3   4   5   6   7   8   9   a   b   c   d   e   f
```
51 73 70 74 31 57 6d 4a 4f 4c **06** 00

Device Name (padded with 00)

01 02 00 36 | D | 02 | MAC address | IP address
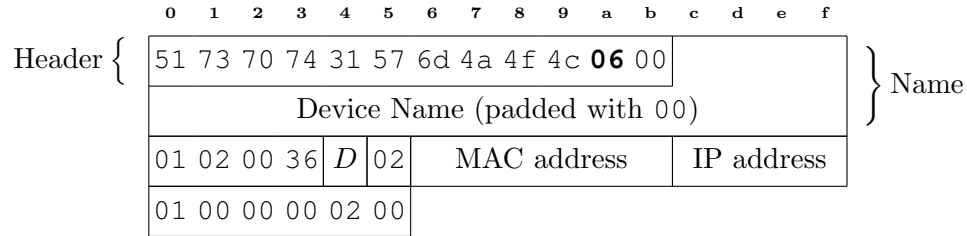
01 00 00 00 02 00

} Name

Figure 5: Mixer keep-alive packets

2   CDJ Startup

When a CDJ starts up the procedure and packets are nearly identical,
with groups of three packets sent at 300 millisecond intervals to port
50000 of the local network broadcast address. The only difference
between Figure 6 and Figure 1 is the final byte, which is `0x01` for the
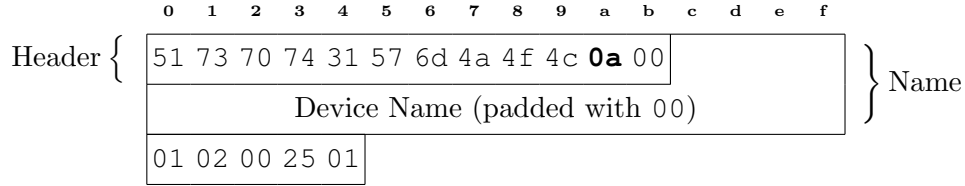CDJ, and was `0x02` for the mixer.

Figure 6: Initial announcement packets from CDJ

Similarly, the next series of three packets from the CDJ are nearly
identical to those from the mixer. The only difference between Figure 7
and Figure 2 is byte 37 (immediately after the packet counter N), which
again is `0x01` for the CDJ, and was `0x02` for the mixer.

However it appears that in this capture the CDJ skips the second
stage of claiming a device number, probably because it is configured
to be automatically assigned a device number based on the port of the
mixer to which it is connected, and we cannot see a packet that the
mixer sent it assigning it that device number. Instead, it jumps right
to the end of the third and final stage, sending a single 38-byte packet
with header byte 10 set to tt 04 (which identified the three packets of
the third stage when the mixer was starting up), with content identical
to Figure 4.

Even though the value of N is `01`, this is the only packet in this
series that the CDJ sends. It would probably behave differently if
configured to assign its own device number (behaving like we saw the
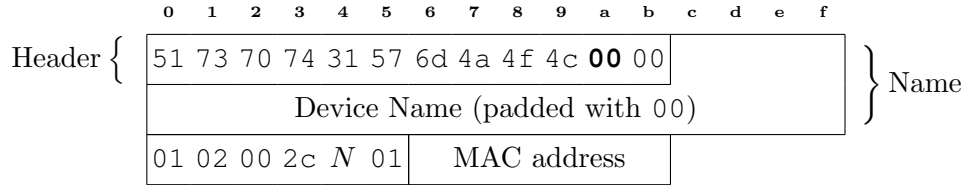mixer behave in claiming its device number).

Figure 7: First-stage CDJ device number assignment packets

The CDJ then moves to the keep-alive stage, sending out 54-byte packets with the content shown in Figure 8.

Figure 8: CDJ keep-alive packets

As seems to always be the case when comparing mixer and CDJ packets, the difference between this and Figure 5 is that byte 37 (following the device number D) has the value 01 rather than 02, and the same is true of the second-to-last byte in each of the packets. (Byte 52 is 01 in Figure 8 and 02 in Figure 5.

3   Tracking BPM and Beats

For some time now, Afterglow[3] has been able to synchronize its light shows with music being played on Pioneer equipment by observing packets broadcast by the mixer to port 50001. Until recently, however, it was not possible to tell which player was the master, so there was no way to determine the down beat (the start of each measure). Now that it is possible to to determine which CDJ is the master player using the packets described in Section 4, these beat packets have become far more useful, and Afterglow will soon be using them to track the down beat based on the beat number reported by the master player.

To track beats, open a socket and bind it to port 50001. The devices seem to broadcast two different kinds of packets to this port, a shorter packet containing 45 bytes of data, and a longer packet containing 96 bytes. The shorter packets seem to all have identical content, and do not seem to convey useful information, so we currently simply ignore them.

The 96-byte packets are sent on each beat, so even the arrival of the packet is interesting information, it means that the player is starting a new beat. (CDJs send these packets only when they are playing *and only for rekordbox-analyzed tracks.* The mixer sends them all the time, acting as a backup metronome when no other device is counting beats.) The content of these packets is shown in Figure 9.

---

[3]https://github.com/brunchboy/afterglow#afterglow

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00	51	73	70	74	31	57	6d	4a	4f	4c	**28**					
10	Device Name (padded with 00)															01
20	00	D	00	3c	$nextBeat$				$2ndBeat$				$nextBar$			
30	$4thBeat$				$2ndBar$				$8thBeat$				ff	ff	ff	ff
40	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff
50	ff	ff	ff	ff	$Pitch$				00	00	BPM		B_b	00	00	D

Figure 9: Beat packets

The Device Number in D (bytes 33 and 95) is the Player Number as displayed on the CDJ itself, or 33 for the mixer, or another value for a computer running rekordbox.

To facilitate synchronization of variable-tempo tracks, the number of milliseconds after which a variety of upcoming beats will occur are reported. $nextBeat$ at bytes 36–39 is the number of milliseconds in which the very next beat will arrive. $2ndBeat$ (bytes 40–43) is the number of milliseconds until the beat after that. $nextBar$ (bytes 44–47) reports the number of milliseconds until the next measure of music begins, which may be from 1 to 4 beats away. $4thBeat$ (bytes 48–51) reports how many millisecond will elapse until the fourth upcoming beat; $2ndBar$ (bytes 52–55) the interval until the second measure after the current one begins (which will occur in 5 to 8 beats, depending how far into the current measure we have reached); and $8thBeat$ (bytes 56–59) tells how many millieconds we have to wait until the eighth upcoming beat will arrive.

The player's current pitch adjustment[4] can be found in bytes 84–87, labeled $Pitch$. It represents a three-byte pitch adjustment percentage, where `0x00100000` represents no adjustment (0%), `0x00000000` represents slowing all the way to a complete stop (−100%, reachable only in Wide tempo mode), and `0x00200000` represents playing at double speed (+100%).

The pitch adjustment percentage represented by $Pitch$ is calculated as follows:

$$100 \times \frac{(byte[85] \times 65536 + byte[86] \times 256 + byte[87]) - 1048576}{1048576}$$

The current BPM of the track playing on the device[5] can be found

---

[4]The mixer always reports a pitch of +0%.

[5]The mixer passes along the BPM of the master player.

at bytes 90–91 (labeled BPM). It is a two-byte integer representing one hundred times the current track BPM. So, the current track BPM value to two decimal places can be calculated as:

$$\frac{byte[90] \times 256 + byte[91]}{100}$$

In order to obtain the actual playing BPM (the value shown in the BPM display), this value must be multiplied by the current pitch adjustment. Since calculating the effective BPM reported by a CDJ is a common operation, here a simplified equation that results in the effective BPM to two decimal places, by combinining the BPM and $Pitch$ values:[6]

$$\frac{(byte[90] \times 256 + byte[91]) \times (byte[85] \times 65536 + byte[86] \times 256 + byte[87])}{104857600}$$

The counter B_b at byte 92 counts out the beat within each bar, cycling $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$ repeatedly, and can be used to identify the down beat if it is coming from the master player.

4   Creating a Virtual CDJ

Although some useful information can be obtained simply by watching broadcast traffic on a network containing Pioneer gear, in order to get important details it is necessary to cause the gear to send you information directly. This can be done by simulating a "Virtual CDJ".[7]

To do this, bind a UDP server socket to port 50002 on the network interface on which you are receiving DJ-Link traffic, and start sending keep-alive packets to port 50000 on the broadcast address as if you were a CDJ. Follow the structure shown in Figure 8, but use the actual MAC and IP addresses of the network interface on which you are receiving DJ-Link traffic, so the devices can see how to reach you.

You can use a value like 5 for D (the device/player number) so as not to conflict with any actual players you have on the network, and any name you would like. As long as you are sending these packets roughly every 1.5 seconds, the other players and mixers will begin sending packets directly to the socket you have opened on port 50002.

Each device seems to send status packets roughly every 200 milliseconds.

---

[6]Since the mixer always reports a pitch adjustment of +0%, its BPM value can be used directly without this additional step.

[7]Thanks are due to Diogo Santos for discovering the trick of creating a virtual CDJ in order to receive detailed status information from other devices.

We are just beginning to analyze all the information which can be gleaned from these packets, but here is what we know so far.[8]

4.1   Mixer Status Packets

Packets from the mixer will have a length of 56 bytes and the content shown in Figure 10.

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00	51	73	70	74	31	57	6d	4a	4f	4c	**29**					
10	Device Name (padded with 00)															01
20	00	D	00	14	D	00	00	F	00	10	00	00	80	00	BPM	
30	00	10	00	00	00	09	X	B_b								

Figure 10: Mixer status packets

Packets coming from a DJM-2000 nexus connected as the only mixer on the network contain a value of 33 (0x21) for their Device Number D (bytes 33 and 36).

The value marked F at byte 39 is evidently a status flag equivalent to the one shown in Figure 12, although on a mixer the only two values seen so far are 0xf0 when it is the tempo master, and 0xd0 when it is not. So evidently the mixer always considers itself to be playing and synced, but never on-air.

There are two places that might contain pitch values, bytes 41–43 and bytes 50–51, but since they always 0x100000 (or +0%), we can't be sure. The current tempo in beats-per-minute identified by the mixer can be obtained as:

$$\frac{byte[46] \times 256 + byte[47]}{100}$$

This value is labeled BPM in Figure 10. Unfortunately, this BPM seems to only be valid when a rekordbox-analyzed source is playing; when the mixer is doing its own beat detection from unanalyzed audio sources, even though it displays the detected BPM on the mixer itself, and uses that to drive its beat effects, it does not send that value in these packets.

The current beat number within a bar (1, 2, 3 or 4) is sent in $byte[55]$, labeled B_b. However, the beat number is *not* synchronized

---

[8]Examples of packets discussed in this section can be found in the capture at https://github.com/brunchboy/dysentery/raw/master/doc/assets/to-virtual.pcapng

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00	51	73	70	74	31	57	6d	4a	4f	4c	**0a**					
10	Device Name (padded with 00)															01
20	03	D	00	b0	D	00	01	A	D_r	S_r	t_r	00	*rekordbox*			
30	00	00	*Track*		00	00	t_2	t_3	00	00	t_4	t_5	00	00	00	00
40	00	00	00	00	00	00	t_6	t_7	00	00	00	00	00	00	00	00
50	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
60	00	00	00	00	00	00	00	00	01	00	U_a	S_a	00	00	00	U_l
70	00	00	00	S_l	00	L	00	00	01	00	00	P_1	*Firmware*			
80	00	00	00	00	00	00	$Sync_n$		00	F	ff	P_2	00	$Pitch_1$		
90	l_1		BPM		7f	ff	ff	ff	00	$Pitch_2$			00	P_3	00	ff
a0	$Beat$				Cue		B_b	00	00	00	00	00	00	00	00	00
b0	00	00	00	00	00	00	10	00	00	00	00	00	00	00	00	00
c0	00	$Pitch_3$		00	$Pitch_4$		$Packet$				nx		00	00	00	
d0	00	00	00	00												

Figure 11: CDJ status packets

with the master player, and these packets do not arrive at the same time as the beat started anyway, so this value is not useful for much. The beat number should be determined, when needed, from beat packets (described in Section 3) that are sent by the master player.

The value at $byte[54]$, labeled X, has an unknown meaning. It seems to start out with the value 0x00, and then change when a player starts playing to the value 0xff, but it may well do other things as well.

4.2   CDJ Status Packets

Packets from a CDJ will have a length of 212 bytes and the content shown in Figure 11 for nexus players. Older players send 208-byte packets with slightly less information. Newer firmware and Nexus 2 players send packets that are 284 or 292 bytes long.

The Device Number in D (bytes 33 and 36) is the Player Number as displayed on the CDJ itself. In the case of this capture, the CDJs were assigned Player Numbers 2 and 3.

The activity flag A at byte 39 seems to be 0 when the player is idle, and 1 when it is playing, searching, or loading a track.

When a rekordbox track is loaded, the device holding the rekordbox database from which the track was loaded is reported in D_r at byte 40 (if the track was loaded from the local device, this will be the same as D; if it was loaded over the Link, it will be the number of a different device) When no track is loaded, D_r has the value `00`.

Similarly, S_r at byte 41 reports the slot from which the track was loaded: The value `00` means no track is loaded, `01` means the CD drive, `02` means the SD slot, and `03` means the USB slot. When a track is loaded from a rekordbox collection on a laptop, S_r has the value `04`. t_r at byte 42 may be an indicator of whether the track came from rekordbox: It seems to have the value `00` when no track is loaded or an unanalyzed track is loaded, `01` when a rekordbox track is loaded, and `05` when an audio CD is loaded.

The field *rekordbox* at bytes 44–47 contains the rekordbox database ID of the loaded track when a rekordbox track is being played. Combined with the player number and slot information, this can be used to request the track metadata as described in Section 5. When an audio CD is loaded, this just contains the track index on the disc.

The track number being played (its position within a playlist or other scrolling list of tracks, as displayed on the CDJ) can be found at bytes 50 and 51, labeled *Track*. (It may be a 4-byte value and also include bytes 48 and 49, but that would seem an unmanageable number of tracks to search through.)

There are a number of bytes, labeled t_2 through t_7, whose purpose is as yet undetermined. They are all zero when there is no track loaded, but take different values when a track is loaded. (Actually, there are many other bytes than these which behave like this, so they will probably be removed from the chart in a future version of this document.)

Byte 106, labeled U_a (for "USB activity"), alternates between the values 4 and 6 when there is USB activity—it may even alternate in time with the flashing USB indicator LED on the player. Byte 107, S_a, is the same kind of activity indicator for the SD slot. Byte 111 (U_l for "USB local") has the value 4 when there is no USB media loaded, 0 when USB is loaded, and 2 when the USB Stop button has been pressed and the USB media is being unmounted.

Byte 115 (S_l for "SD local") has the value 4 when there is no SD media loaded, 0 when SD is loaded, and 2 when the SD door has been opened and the SD media is being unmounted.

Byte 117, labeled L (for "Link available"), appears to have the value 1 whenever USB, SD, or CD media is present in any player on the network, whether or not the Link option is chosen in the other players, and 0 otherwise.

7	6	5	4	3	2	1	0
1	Play	Master	Sync	On-Air	1	0	0

Figure 12: CDJ state flag bits

Byte 123, labeled P_1, appears to describe the current play mode. The values that have been seen so far, and their apparent meanings, are:

0 No track is loaded.

2 A track is in the process of loading.

3 The player is playing normally.

4 The player is playing a loop.

5 The player is paused anywhere other than the cue point.

6 The player is paused at the cue point.

7 Cue Play is in progress (playback while the cue button is held down).

8 Cue scratch is in progress.

9 The player is searching forwards or backwards.

17 The player reached the end of the track and stopped.

The *Firmware* value at bytes 124–127 is an ASCII represenation of the firmware version running in the player.

The value $Sync_n$ at bytes 134–135 seems to increment whenever the player syncs to a new tempo master (another player or the mixer). I am assuming it is just a 2-byte value, because I tried syncing 256 times and saw the counter expand from byte 135 to include byte 134. It may actually be a 4-byte value and also involve bytes 132 and 133, but I wasn't going to try changing sync 65,536 times or more to find out. Perhaps we could write software to test that someday by forcing tempo master changes.

Byte 137, labeled F, is a bit field containing some very useful state flags, detailed in Figure 12.[9] It seems to only be available on nexus players, and others always send 0 for this byte?

Byte 139, labeled P_2 seems to be another play state indicator, having the value 122 (`0x7a`) when playing and 126 (`0x7e`) when stopped. When the CDJ is trying to play, but is being held in place by the DJ holding down on the jog wheel, P_1 considers it to be playing (value 3), while P_2 considers it to be stopped (value 126). Non-nexus players seem to use the value 106 (`0x6a`) when playing and 110 (`0x6e`) when

---

[9]We have not yet seen any other values for bits 0, 1, 2, or 7 in F, so we're unsure if they also carry meaning. If you ever find different values for them, please let us know by filing an Issue at `https://github.com/brunchboy/dysentery/issues`

stopped, while nxs2 players use the values 250 and 254 (`0xfa` and `0xfe`) so this seems to be another bit field like F.

There are four different places where pitch information appears in these packets: $Pitch_1$ at bytes 141–143, $Pitch_2$ at bytes 153–155, $Pitch_3$ at bytes 193–195, and $Pitch_4$ at bytes 197–199.

Each of these values represents a three-byte pitch adjustment percentage, where `0x100000` represents no adjustment (0%), `0x000000` represents slowing all the way to a complete stop (−100%, reachable only in Wide tempo mode), and `0x200000` represents playing at double speed (+100%).

Note that if playback is stopped by pushing the pitch fader all the way to −100% in Wide mode, both P_1 and P_2 still show it as playing, which is different than when the jog wheel is held down, since P_2 shows a stop in the latter situation.

Here is how the pitch adjustment percentage represented by $Pitch_1$ would be calculated:

$$100 \times \frac{(byte[141] \times 65536 + byte[142] \times 256 + byte[143]) - 1048576}{1048576}$$

We don't know why there are so many copies of the pitch information, or all circumstances under which they might differ from each other, but it seems that $Pitch_1$ and $Pitch_3$ report the current pitch adjustment actually in effect (as reflected on the BPM display), whether it is due to the local pitch fader, or a synced tempo master.

$Pitch_2$ and $Pitch_4$ are always tied to the position of the local pitch fader, unless Tempo Reset is active, effectively locking the pitch fader to 0% and $Pitch_2$ and $Pitch_4$ to `0x100000`, or the player is paused or the jog wheel is being held down, freezing playback and locking the local pitch to −100%, in which case they both have the value `0x000000`.

When playback stops, either due to the play button being pressed or the jog wheel held down, the value of $Pitch_4$ drops to `0x000000` instantly, while the value of $Pitch_2$ drops over time, reflecting the gradual slowdown of playback which is controlled by the player's brake speed setting. When playback starts, again either due to the play button being pressed or the jog wheel being released, both $Pitch_2$ and $Pitch_4$ gradually rise to the target pitch, at a speed controlled by the player's release speed setting.

If the player is *not* synced, but the current pitch is different than what the pitch fader would indicate (in other words, the player is in the mode where it tells you to move the pitch fader to the current BPM in order to change the pitch), moving the pitch fader changes the values of $Pitch_2$ and $Pitch_4$ until they match $Pitch_1$ and $Pitch_3$ and begin to affect the actual effective pitch. From that point on, moving the

pitch fader sets the value of all of $Pitch_1$, $Pitch_2$, $Pitch_3$, and $Pitch_4$. This all seems more complicated than it really needs to be...

The current BPM of the track (the BPM at the point that is currently being played, or at the location where the player is currently paused) can be found at bytes 146–147 (labeled BPM). It is a two-byte integer representing one hundred times the current track BPM. So, the current track BPM value to two decimal places can be calculated as:

$$\frac{byte[146] \times 256 + byte[147]}{100}$$

In order to obtain the actual playing BPM (the value shown in the BPM display), this value must be multiplied by the current effective pitch, calculated from $Pitch_1$ as described above. Since calculating the effective BPM reported by a CDJ is a common operation, here a simplified equation that results in the effective BPM to two decimal places, by combinining the BPM and $Pitch1$ values:

$$\frac{(b[146] \times 256 + b[147]) \times (b[141] \times 65536 + b[142] \times 256 + b[143])}{104857600}$$

Because Rekordbox and the CDJs support tracks with variable BPM, this value can and does change over the course of playing such tracks. When no track is loaded, BPM has the value `0xffff`.

The meaning of value l_1 (bytes 144–145) is not currently known. It may simply reflect whether a track is loaded or not: it seems to have the value `0x7fff` when no track is loaded, `0x8000` when a rekordbox track is loaded, and `0x0000` when a non-rekordbox track (like from a physical CD) is loaded.

Byte 157 (labeled P_3) seems to communicate additional information about the current play mode, with the following meanings that we have found so far:

0 No track is loaded.

1 The player is paused or playing in Reverse mode.

9 The player is playing in Forward mode with jog mode set to Vinyl.

13 The player is playing in Forward mode with jog mode set to CDJ.

The 4-byte beat counter (which counts each beat from 1 through the end of the track) is found in bytes 160–163, labeled $Beat$. When the player is paused at the start of the track, this seems to hold the value 0, even though it is beat 1, and when no rekordbox-analyzed track is loaded, *and in packets from non-nexus players*, this holds the value `0xffffffff`.

The counter B_b at byte 166 counts out the beat within each bar, cycling $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$ repeatedly, and can be used to identify the

down beat (as is used in the Master Player display on the CDJs as a mixing aid). Again, when no reckordbox-analyzed track is loaded, this holds the value 0. If you want to synchronize events to the down beat, use the CDJ status packets' F value to identify the master player, but use the beat packets sent by that player (described in Section 3) to determine when the beats are actually happening.

A countdown timer to the next saved cue point is available in bytes 164–165 (labeled *Cue*). If there is no saved cue point after the current play location in the track, or if it is further than 64 bars ahead, these bytes contain the value `0x01ff` and the CDJ displays "--.- bars". As soon as there are just 64 bars (256 beats) to go before the next cue point, this value becomes `0x0100`. This is the point at which the CDJ starts to display a countdown, which it displays as "63.4 bars". As each beat goes by, this value decreases by 1, until the cue point is about to be reached, at which point the value is `0x0001` and the CDJ displays "00.1 bars". On the beat on which the cue point was saved the value is `0x0000` and the CDJ displays "00.0 Bars". On the next beat, the value becomes determined by the next cue point (if any) in the track.

Bytes 200-203 seem to contain a 4-byte packet counter labeled *Packet*, which is incremented for each packet sent by the player. (I am just guessing it is four bytes long, I have not yet watched long enough for the count to need more than the last three bytes).

Byte 204, labeled *nx*, seems to have the value `0x0f` for nexus players, and `0x05` for older players.

4.3 Rekordbox Status Packets

Rekordbox sends status packets which appear to be essentially identical to those sent by a mixer, as shown in Figure 10, sending "rekordbox" as its device name. The device number D (bytes 33 and 36) seems to be 41 (`0x29`), although it will probably use conflict resolution to pick an unused number if multiple copies are running. The F value we have seen remains consistent as a status flag, showing `0xc0` which would indicate that it is always "playing" but not synced, tempo master, nor on the air. The BPM value seems to track that of the master player, and the same potential pitch values (fixed at `0x100000`, or +0%) are present, as is X. B_b always seems to be zero.

5   Track Metadata

Thanks to @EvanPurkhiser[10], we finally started making progress in retrieving metadata from CDJs, and now some shared code from Austin

---

[10]`https://github.com/EvanPurkhiser`

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00	00	00	00	0f	R	e	m	o	t	e	D	B	S	e	r	v
10	e	r	00													

Figure 13: DB Server query packet

Wright[11] is boosting our understanding considerably!

To be polite about it, the first step is to determine the port on which the player is offering its remote database server. That can be determined by opening a TCP connection to port $12,523$ on the player and sending it sending a packet with the content shown in Figure 13.

The player will send back a two-byte response, containing the high byte of the port number followed by the low byte. So far, the response from a CDJ has always indicated a port number of 1051, but using this query to determine the port to use will protect you against any future changes. The same query can also be sent to a laptop running rekordbox to find the rekordbox database server port, which can also be queried for metadata in the exact same way described below.

To find the metadata associated with a particular track, given its rekordbox ID number, as well as the player and slot from which it was loaded (all of which can be determined from a CDJ status packet received by a virtual CDJ as described in Section 4), open a TCP connection to the device from which the track was loaded, using the port that it gave you in response to the DB Server query packet, then send the following four packets.

The first packet sent to the database server contains the five bytes `11 00 00 00 01`, and results in the same five bytes being sent back.

All further packets have a shared structure. They consist of lists of type-tagged fields (a type byte, followed some number of value bytes, although in the case of the variable-size types, the first four bytes are a big-endian integer that specifies the length of the additional value bytes that make up the field). So far, there are four known field types, and it turns out that the packet we just saw is one of them, it represents the number 1 as a 4-byte integer.

5.1 Field Types

The first byte of a field identifies what type of field is coming. The values `0x0f`, `0x10`, and `0x11` are followed by 1, 2, and 4 byte fixed-length integer fields, while `0x14` and `0x26` introduce variable-length fields, a binary blob and a UTF-16 big-endian string respectively.

---

[11]https://bitbucket.org/awwright/libpdjl
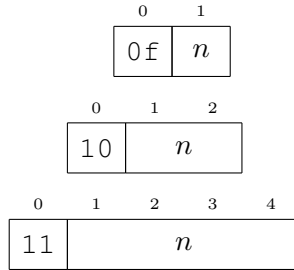
Figure 14: Number Fields of length 1, 2, and 4
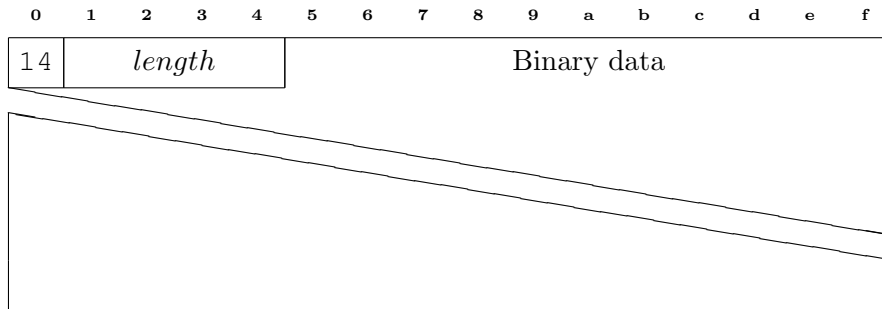
Figure 15: Binary (Blob) Field

5.1.1   Number Fields

Number fields are indicated by an initial byte `0x0f`, `0x10`, or `0x11` which is followed by big-endian integer value of length 1, 2, or 4 bytes respectively, as shown in Figure 14. So, as noted above, the initial greeting packet sent to and received back from the database server is a number field, four bytes long, representing the value 1.

5.1.2   Binary Fields

Variable-length binary (blob) fields are indicated by an initial byte `0x14`, followed by a 4 byte big-endian integer which specifies the length of the field payload. The length is followed by the specified number of bytes (for example, an album art image, waveform or beat grid). This is illustrated in Figure 15.

5.1.3   String Fields

Variable-length string fields are indicated by an initial byte `0x26`, followed by a 4 byte big-endian integer which specifies the length of the

0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
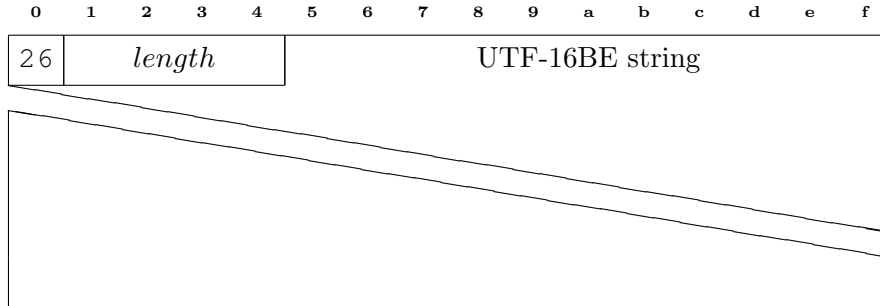
Figure 16: String Field

string, in two-byte UTF-16 big-endian characters. So the length is folowed by $2 \times length$ bytes containing the actual string characters. The last character of the string is always NUL, represented by `0x0000`. This is illustrated in Figure 16.

5.2 Messages

Messages are introduced by a 4 byte Number field containing the magic value `0x872349ae`. This is followed by another 4 byte number field that contains a transaction ID, which starts at 1 and is incremented for each query sent, and all messages sent in response to that query will contain the same transaction ID. This is followed by a 2 byte number field containing the message type, a 1 byte number field containing the number of argument fields present in the message, and a blob field containing a series of bytes which identify the types of each argument field. This blob is always 12 bytes long, regardless of how few arguments there are (and presumably this means no message ever has more than 12 arguments). Tag bytes past the actual argument count of the message are set to 0.

The argument type tags use different values than the field type tags themselves, for some reason, and it is not clear why this redundant information is necessary at all, but that is true a number of places in the protocol as you will see later. Here are the known tag values and their meanings:

`0x02` A string in UTF-16 big-endian encoding, with trailing NUL (zero) character.

`0x03` A binary blob.

`0x06` A 4 byte big-endian integer.

I am guessing that if we ever see them, a tag of 4 would represent

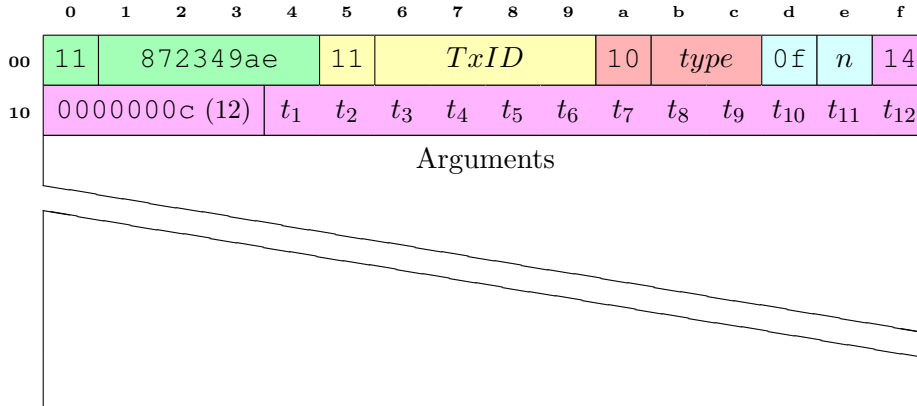|	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00	11		872349ae			11			$TxID$			10		$type$	0f	n	14
10	0000000c (12)					t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}

Arguments

Figure 17: Message Header

a 1 byte integer, and 5 would represent a 2 byte integer. But so far no such messages have been seen.

This header is followed by the fields that make up the message arguments, if any. The header structure is illustrated in Figure 17, where $TxID$ is the transaction ID, n is the number of arguments found in the message, and t_1 through t_{12} are the type tags for each argument, or 0 if there is no argument in that position.

Before you can send your first actual query, you need to send a special message which seems to be necessary for establishing the context for queries. It has a $type$ of 0, a special $TxID$ value of 0xfffffffe, and a single numeric argument, as shown in Figure 18.

The value D is, like in the other packets we have seen, a player device number. In this case it is the device that is asking for metadata information. It must be a valid player number between 1 and 4, and that player must actually be present on the network, must not be the same player that you are contacting to request metadata from, and must not be a player that has connected to that player via Link and loaded a track from it. So the safest device number to use is the device number you are using for your virtual CDJ, but since it must be between 1 and 4, you can only do that if there are fewer than four actual CDJs on the network.

The player will respond with a message of type 0x4000, which is the common "success" response when requested data is available. The response message has two numeric arguments, the first of which is the message type of the request we sent (which was 0), and the second usually tells you the number of items that are available in response to the query you made, but in this special setup query, it returns its own player number. The overall structure is illustrated in Figure 19.

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
			start						TxID			type			args	tags
00	11	872349ae				11	fffffffe				10	0000		0f	01	14
10	0000000c (12)				06	00	00	00	00	00	00	00	00	00	00	00
20	11	D_{ours}														

Figure 18: Query context setup message

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
			start						TxID			type			args	tags
00	11	872349ae				11	fffffffe				10	4000		0f	02	14
10	0000000c (12)				06	06	00	00	00	00	00	00	00	00	00	00
20	11	00000000				11	D_{theirs}									

Figure 19: Query context setup response

5.3   Track Metadata

To ask for metadata about a particular track, send a packet like the one shown in Figure 20.

As described above, $TxID$ should be 1 for the first query packet you send, 2 for the next, and so on. D should have the same value you used in your initial query context setup packet, identifying the device that is asking the question. S_r is the slot in which the track being asked about can be found, and has the same values used in CDJ status packets, as shown in Figure 11. Similarly, $rekordbox$ identifies the local rekordbox database ID of the track being asked about, as

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
			start						TxID			type			args	tags
00	11	872349ae				11	$TxID$				10	2002		0f	02	14
10	0000000c (12)				06	06	00	00	00	00	00	00	00	00	00	00
20	11	D	01	S_r	01	11	$rekordbox$									

Figure 20: Track metadata request message

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
			start					*TxID*				*type*		*args*		*tags*
00	11		872349ae			11		*TxID*			10	4000		0f	02	14
10	0000000c (12)				06	06	00	00	00	00	00	00	00	00	00	00
20	11		00002002			11		0000000b								

Figure 21: Track metadata ready response

found in the CDJ status packet.

Track metadata requests are built on the mechanism that is used to request and draw scrollable menus on the CDJs, so the request is essentially interpreted as setting up to draw the "menu" of information that is known about the track. The player responds with a success indicator, saying it is ready to send these "menu items" and reporting how many of them are available, as shown in Figure 21.

We've seen this type of "data available" response already in Figure 19, but this one is a more typical example. As usual, $TxID$ matches the value we sent in our request, and the first argument, with value 0x2002, reflects the *type* field of our request. The second argument reports that there are 11 (0x0b) entries of track metadata available to be retrieved for the track we asked about, and that the player is ready to send them to us.

If there was no track with ID *rekordbox* in that media slot, the second argument would have the value 0xffffffff to let us know. If we messed up something else about the request, we will get a response with a *type* other than 0x4000. See Section 5.11 for instructions on how to explore these variations on your own.

But assuming everything went well, we can get the player to send us all eleven of those metadata entries by telling it to render all of the current menu, using a "render menu" request with *type* 0x3000 shown in Figure 22.

As always, the value of $TxID$ should be one higher than the one you sent in your setup packet, while the values of D and S_r should be identical to what you sent in it.

The request has six numeric arguments. At this point it is worth talking a bit more about the byte after D in the first argument. This seems to specify the location of the menu being drawn, with the value 1 meaning the main menu on the left-hand half of the CDJ display, while 2 means the submenu (for example the info popup when it is open) which overlays the right-hand half of the display. We don't yet know exactly what, if any, difference there is between the response

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
		start						$TxID$				type			args	tags
00	11	872349ae				11	$TxID$				10	3000		0f	06	14
10	0000000c (12)				06	06	06	06	06	06	00	00	00	00	00	00
20	11	D	01	S_r	01	11	offset				11	limit				11
30	00000000				11	$limit_2$				11	00000000					

Figure 22: Render Menu request message

details when 2 is used instead of 1 here. And special data requests use different values: for example, the track waveform summary, which is drawn in a strip along the entire bottom of the display, is requested with a menu location number of 8 in this second byte.

The byte after S_r seems to always have the value 1. We've seen it described as "source analyzed" but don't know exactly what that means.

The second argument, $offset$, specifies which menu entry is the first one you want to see, and the third argument, $limit$, specifies how many should be sent. In this case, since there are only 11 entries, we can request them all at once, so we will set $offset$ to 0 and $limit$ to 11. But for large playlists, for example, you need to request batches of entries that are smaller than the total available, or the player will be unable to send them to you. We have not found what the exact limit is, but getting 64 at a time seems to work on Nexus 2 players.

We don't know the purpose of the fourth argument, but sending a value of 0 works. The fifth argument, $limit_2$, is also poorly understood, but sending a second copy of $limit$ here works. And the sixth and final argument also has an unknown purpose, but 0 works.

So, for our metadata request, the packet we want to send in order to get all the metadata will have the specific values shown in Figure 23:

This causes the player to send us 13 messages: The 11 metadata items we requested are sent (with *type* 0x4101, Figure 25), but they are preceded by a menu header message (with *type* 0x4001, Figure 24), and followed by a menu footer message (with *type* 0x4201, Figure 26). This wrapping happens with all "render menu" requests, and the menu footer is an easy way to know you are done, although you can also count the messages.

The menu item responses all have the same structure, and use all twelve message argument slots, containing ten numbers and two strings, although they generally don't have meaningful values in all of

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
			start						*TxID*			*type*		*args*		*tags*
00	11		872349ae			11			*TxID*		10		3000	0f	06	14
10	0000000c (12)				06	06	06	06	06	06	00	00	00	00	00	00
20	11	*D*	01	*S_r*	01	11		00000000			11		0000000b			11
30		00000000			11		0000000b		11		00000000					

Figure 23: Render track metadata request message

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
			start						*TxID*			*type*		*args*		*tags*
00	11		872349ae			11			*TxID*		10		4001	0f	02	14
10	0000000c (12)				06	06	00	00	00	00	00	00	00	00	00	00
20	11		00000001		11		00000000									

Figure 24: Track metadata menu header response

the slots. They have the general structure shown in in Figure 25, and the arguments are:

1 Numeric, a parent ID, such as an artist for a track item.

2 Numeric, the main ID, such as *rekordbox* for a track item.

3 Numeric, the length in bytes of Label 1.

4 String, Label 1 (main text, such as the track title or artist name, as appropriate for the item type).

5 Numeric, the length in bytes of Label 2.

6 String, Label 2 (secondary text, e.g. for playlist entries the artist name, and Label 1 holds the title).

7 Numeric, the type of this item (see Section 5.5).

8 Numeric, some sort of flags field, details still unclear.

9 Numeric, holds the album art ID when the type is a Track Title.

10 Numeric, playlist position when relevant, e.g. when listing a playlist.

11 Numeric, purpose unknown.

12 Numeric, purpose unknown.

0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f

| | start | | | | | *TxID* | | | | *type* | | *args* | | *tags* | |

00 | 11 | 872349ae | | | | 11 | *TxID* | | | | 10 | 4101 | | 0f | 0c | 14 |

10 | 0000000c (12) | | | | 06 | 06 | 06 | 02 | 06 | 02 | 06 | 06 | 06 | 06 | 06 | 06 |
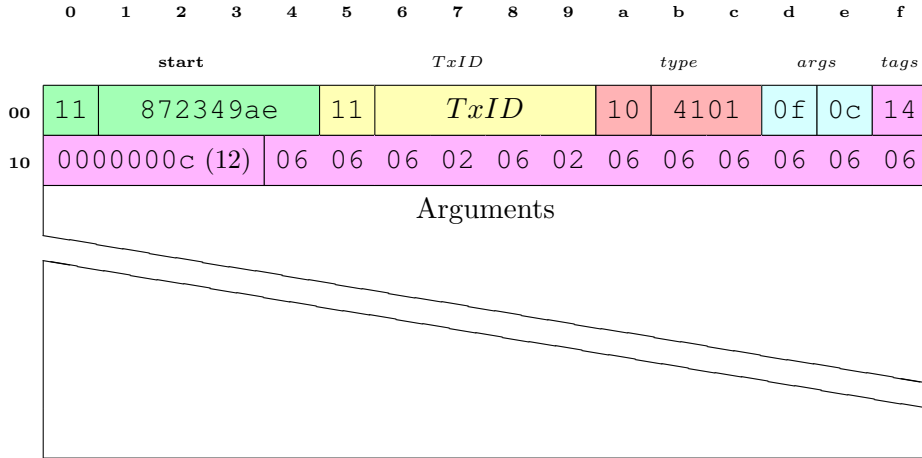
Arguments

Figure 25: Track metadata item response

5.3.1   Track Metadata Item 1: Title

The first item returned after the menu header is the track title, so argument 7 has the value `0x04`. Argument 1, which may always be some kind of parent ID, holds the artist ID associated with the track. The second argument seems to always be the main ID, and for this response it holds the *rekordbox* ID of the track. Argument 4 holds the track title text, and argument 8 holds the album *artwork* ID if any is available for the track. This ID can be used to retrieve the actual album art image as described in Section 5.6.

5.3.2   Track Metadata Item 2: Artist

The second item contains artist information so argument 7 has the value `0x07`. Argument 2 holds the artist ID, argument 4 contains the text of the artist name.

5.3.3   Track Metadata Item 3: Album Title

The third item contains album title information so argument 7 has the value `0x02`. Argument 4 contains the text of the album name.

5.3.4   Track Metadata Item 4: Duration

The fourth item contains track duration information so argument 7 has the value `0x0b`. Argument 2 contains the length, in seconds, of the track when played at normal tempo.

5.3.5   Track Metadata Item 5: Tempo

The fifth item contains tempo information so argument 7 has the value `0x0d`. Argument 2 contains the track's starting tempo, in BPM, times 100, as reported in BPM values in other packets described earlier.

5.3.6   Track Metadata Item 6: Comment

The sixth item contains comment information so argument 7 has the value `0x23`. Argument 4 contains the text of the track comment entered by the DJ in rekordbox.

5.3.7   Track Metadata Item 7: Key

The seventh item contains key information so argument 7 has the value `0x0f`. Argument 4 contains the text of the track's dominant key signature.

5.3.8   Track Metadata Item 8: Rating

The eighth item contains rating information so argument 7 has the value `0x0a`. Argument 2 contains a value from 0 to 5 corresponding to the number of stars the DJ has assigned the track in rekordbox.

5.3.9   Track Metadata Item 9: Color

The ninth item contains color information so argument 7 has the value `0x13`. I assume Argument 2 contains a value corresponding to the color the DJ has assigned the track in rekordbox, but I have not yet figured out the details.

5.3.10   Track Metadata Item 10: Genre

The tenth item contains genre information so argument 7 has the value `0x06`. Argument 2 contains the numeric genre ID, and argument 4 contains the text of the genre name.

5.3.11   Track Metadata Item 11: Date Added

The eleventh and final item contains the date added information so argument 7 has the value `0x2e`. Argument 4 contains the date the track was added to the collection in the format "yyyy-mm-dd". This information seems to propagate into rekordbox from iTunes.

0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
		start					TxID				type		args		tags

```
00  11   872349ae    11      TxID     10   4201   0f  00  14
10  0000000c (12)   00  00  00  00  00  00  00  00  00  00  00  00
```

Figure 26: Track metadata menu footer response

5.4   Menu Footer Response

The menu footer message has a *type* of 0x4201 and no arguments, so it has a header only, and is always made up of the exact same sequence of bytes (apart from the *TxID*), as shown in Figure 26.

5.5   Menu Item Types

As noted above, the seventh argument in a menu item response identifies the type of the item. The meanings we have identified so far are:

0x0001 Folder (such as in the playlists menu, a nested list of playlists rather than an individual playlist)

0x0002 Album title

0x0003 Disc

0x0004 Track Title

0x0006 Genre

0x0007 Artist

0x0008 Playlist

0x000a Rating

0x000b Duration (in seconds)

0x000d Tempo

0x000f Key

0x0013 Color

0x0023 Comment

0x002e Date Added

0x0704 Track List Entry

As noted above, track metadata responses use many of these types. Others are used in different kinds of menus and queries.

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
			start						*TxID*				*type*		*args*	*tags*
00	11		872349ae			11			*TxID*			10	2003	0f	02	14
10	0000000c (12)				06	06	00	00	00	00	00	00	00	00	00	00
20	11	D	08	S_r	01	11		*artwork*								

Figure 27: Track artwork request message

5.6   Album Art

To request the artwork image associated with a track, send a message
with *type* 0x2003 containing the *artwork* ID that was specified in the
track title item (as described in Section 5.3.1), like the one shown in
Figure 27.

As usual, *seq* should be incremented each time you send a query,
and will be used to identify the response messages. D should have
the same value you used in your initial query context setup packet,
identifying the device that is asking the question. S_r is the slot in
which the track being asked about can be found, and has the same
values used in CDJ status packets, as shown in Figure 11. Finally,
artwork identifies the specific artwork image you are requesting, as it
was specified in the track medatata response. As with other graphical
requests, the value after D, which identifies the location of the menu
for which data is being loaded, is 8.

The response will be a message with *type* 0x4002, containing four
arguments. The first argument echoes back our request type, which
was 0x2003. The second always seems to be 0. The third contains the
length of the image in bytes, which seems redundant, because that is
also conveyed in the fourth argument itself, which is a blob containing
the actual bytes of the image data, as shown in Figure 28.

5.7   Beat Grids

The CDJs do not send any timing information other than beat numbers
during playback, which has made it difficult to offer absolute timecode
information. The discovery of beat grid requests provides a clean an-
swer to the problem. The beat grid for a track is a list of every beat
which occurs in the track, along with the point in time at which that
beat would occur if the track were played at its standard (100%) tempo.
Armed with this table, it is possible to translate any beat packet into
an absolute position within the track, and, combined with the tempo
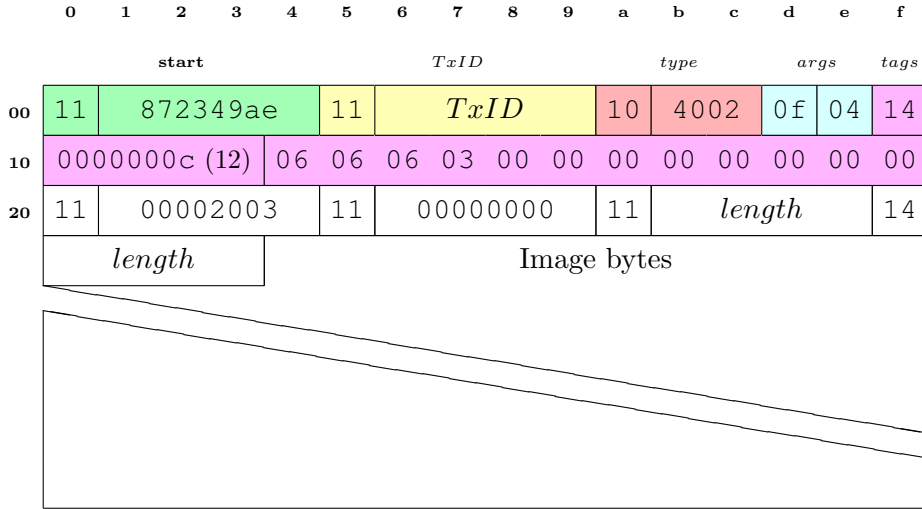information, to generate timecode signals allowing other software (such

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
			start					*TxID*				*type*		*args*		*tags*
00	11	872349ae				11	*TxID*				10	4002		0f	04	14
10	0000000c (12)					06	06	06	03	00	00	00	00	00	00	00
20	11	00002003				11	00000000				11	*length*				14

length	Image bytes

Figure 28: Track artwork response message

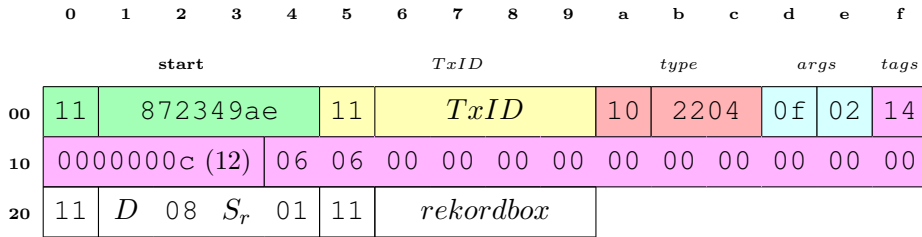|	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
			start					*TxID*				*type*		*args*		*tags*
00	11	872349ae				11	*TxID*				10	2204		0f	02	14
10	0000000c (12)					06	06	00	00	00	00	00	00	00	00	00
20	11	D	08	S_r	01	11	*rekordbox*									

Figure 29: Track beat grid request message

as video resources) to sync tightly with DJ playback.

To request the beat grid of a track, send a message with *type* 0x2204 containing the *rekordbox* ID of the track, like the one shown in Figure 29.

As usual, *seq* should be incremented each time you send a query, and will be used to identify the response messages. D should have the same value you used in your initial query context setup packet, identifying the device that is asking the question. S_r is the slot in which the track being asked about can be found, and has the same values used in CDJ status packets, as shown in Figure 11. Finally, *rekordbox* identifies the specific artwork image you are requesting, as found in a CDJ status packet or playlist response. As with graphical requests, the value after D, which identifies the location of the menu
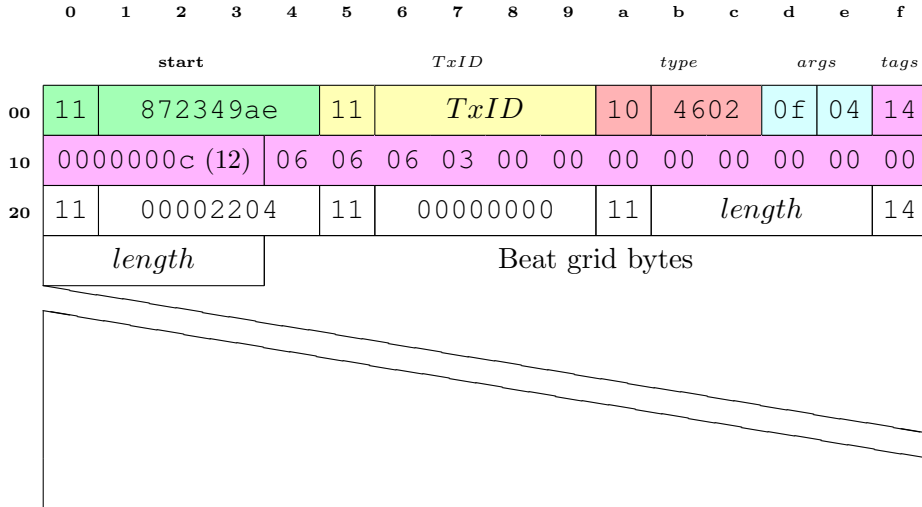
Figure 30: Track beat grid response message

for which data is being loaded, is 8.

The response will be a message with *type* 0x4602, containing four arguments. The first argument echoes back our request type, which was 0x2204. The second always seems to be 0. The third contains the length of the beat grid in bytes, which seems redundant, because that is also conveyed in the fourth argument itself, which is a blob containing the actual bytes of the beat grid, as shown in Figure 30.

The beat grid itself is spread through the value returned as argument 4, consisting of one-byte beat-within-bar numbers (labeled B_b in Figure 11), followed by four-byte timing information, specifying the number of milliseconds after the start of the track (when played at its native tempo) at which that beat falls.

The B_b value of the first beat in the track is found at byte 20 of argument 4, and the time at which that beat occurs, in milliseconds, is encoded as a 4-byte big-endian integer at bytes 21–24. Subsequent beats are found at 16-byte intervals, so the second B_b value is found at byte 36, and the second beat's time, in milliseconds from the start of the track, is the big-endian integer at bytes 37–40. The B_b value for the third beat is at byte 52, and its millisecond time is at bytes 53–56, and so on.

The purpose of the other bytes within the beat grid is so far undetermined. It looks like there may be some sort of monotonically increasing value following the beat millisecond value, but what it means, and why it sometimes skips values, is mysterious.

5.8 Requesting the Waveform Overview

TODO: Describe!

5.9 Requesting All Tracks

If you want to cache all the metadata associated with a media stick, this query is a good starting point.

TODO: Fill out!

5.10 Playlists

If you want to be more selective about the metadata that you are caching, you can navigate the playlist folder hierarcy and deal with only speific playlists.

TODO: Fill out!

5.11 Experimenting with Metadata

The best way to get a feel for the details of working with these messages is to load dysentery into a Clojure REPL, as described on the project page, and play with some of the functions in the `dysentery.dbserver` namespace. Have no more than three players connected and active on your network, so you have an unused player number for dysentery to use. In this example, player number 1 is available, so we set dysentery up to pose as player 1:

```
> lein repl
nREPL server started on port 53806 on host 127.0.0.1 –
 nrepl://127.0.0.1:53806
REPL-y 0.3.7, nREPL 0.2.12
Clojure 1.8.0
Java HotSpot(TM) 64-Bit Server VM 1.8.0_77-b03
dysentery loaded.
dysentery.core=> (view/watch-devices :player-number 1)
Looking for DJ Link devices...
Found:
   DJM-2000nexus 33 /172.16.42.3
   CDJ-2000nexus 2 /172.16.42.4

To communicate create a virtual CDJ with address
  /172.16.42.2, MAC address 3c:15:c2:e7:08:6c,
  and use broadcast address /172.16.42.255
:socket #object[java.net.DatagramSocket 0x22b952b1
                "java.net.DatagramSocket@22b952b1"],
 :watcher #future[:status :pending, :val nil 0x3eb8f41b]
dysentery.core> (def p2 (db/connect-to-player 2 1))
Transaction: 4294967294, message type: 0x4000
```

```
   (requested data available), argument count: 2, arguments:
   number:          0 (0x00000000) [request type]
   number:          2 (0x00000002) [# items available]
#'dysentery.core/p2
dysentery.core> (def md (db/request-metadata p2 2 1))
Sending > Transaction: 1, message type: 0x2002
 (request track metadata), argument count: 2, arguments:
  number:   16843265 (0x01010201) [player, menu, media, 1]
  number:          1 (0x00000001) [rekordbox ID]
Received > Transaction: 1, message type: 0x4000
 (requested data available), argument count: 2, arguments:
  number:       8194 (0x00002002) [request type]
  number:         11 (0x0000000b) [# items available]
Sending > Transaction: 2, message type: 0x3000
 (render menu), argument count: 6, arguments:
  number:   16843265 (0x01010201) [player, menu, media, 1]
  number:          0 (0x00000000) [offset]
  number:         11 (0x0000000b) [limit]
  number:          0 (0x00000000) [unknown (0)?]
  number:         11 (0x0000000b) [len_a (= limit)?]
  number:          0 (0x00000000) [unknown (0)?]
Received 1  > Transaction: 2, message type: 0x4001
 (rendered menu header), argument count: 2, arguments:
  number:          1 (0x00000001) [unknown]
  number:          0 (0x00000000) [unknown]
Received 2  > Transaction: 2, message type: 0x4101
 (rendered menu item), argument count: 12, arguments:
  number:          1 (0x00000001) [numeric 1 (parent id)]
  number:          1 (0x00000001) [numeric 2 (this id)]
  number:         80 (0x00000050) [label 1 byte size]
  string: "Escape ft. Zoe Phillips" [label 1]
  number:          2 (0x00000002) [label 2 byte size]
  string: "" [label 2]
  number:          4 (0x00000004) [item type: Track Title]
  number:   16777216 (0x01000000) [column configuration?]
  number:          0 (0x00000000) [album art id]
  number:          0 (0x00000000) [playlist position]
  number:        256 (0x00000100) [unknown]
  number:          0 (0x00000000) [unknown]
...
Received 13  > Transaction: 2, message type: 0x4201
 (rendered menu footer), argument count: 0, arguments:
#'dysentery.core/md
dysentery.core>
```

In this interaction, after setting up the watcher so we can find players on the network, we set the var p2 to be a connection to player 2, in which we are posing as player 1. Then we submit a metadata request to p2, requesting the track in slot 2 (SD card), with

rekordbox id 1. You can see the messages being sent and received to accomplish that. For more functions that you can call, including the very flexible `experiment` function, look at the source for the `dysentery.dbserver` namespace. Most of the response messages containing track metadata were omitted for brevity; you will get more meaningful results trying it with your own tracks, and then you can see all the details.

5.12   Background Research

Prior to Evan and Austin's breakthroughs, here is all we knew:

By setting up a managed switch to mirror traffic sent directly between CDJs, we have been able to see how the Link Info operation is implemented: The players open a direct TCP connection between each other, and send queries to obtain the metadata about tracks with particular rekordbox ID values.

Using an Ethernet switch with port mirroring was, as we hoped, very helpful. As can be seen in the capture at `https://github.com/brunchboy/dysentery/raw/master/doc/assets/LinkInfo.pcapng`, which shows a CDJ with IP address 169.254.192.112 booting, the new CDJ opens two TCP connections to the other CDJ at 169.254.119.181.

The first session (given id 0 by Wireshark), which begins at packet 206, connecting to port 12523, determines the port to use for metadata queries.

The second TCP connection (Wireshark display filter `tcp.stream eq 1`), beginning at packet 212 and connecting to port 1051, shows the track information used by the Link Info display passing between the CDJs. You can see packets reflecting the initial display of a track that was already loaded, then new information as the linked CDJ loaded three other tracks.

There is another capture at `https://github.com/brunchboy/dysentery/raw/master/doc/assets/LinkInfo2.pcapng`, with more Link Info streams to be studied (all of the odd numbered `tcp.stream` values in Wireshark are the relevant ones).

6   What's Missing?

We know this analysis isn't complete. There are some values we know are available and useful (because CDJs and rekordbox itself show them), but we have not figured out how to find them yet.

Before we discovered how to ask players for metadata about particular tracks, we did some research into the underlying rexordbox

database. The database format is called DeviceSQL,[12] and there used to be a free quick start suite for working with it[13] but that site no longer exists because the original (California) company Encirq[14] was acquired by the Japanese Ubiquitous Corporation in 2008.[15] It seems to still be available,[16] but I'd be surprised if they wanted to help out an open source effort like this one.

6.1 CDJ Packets to Rekordbox

Performing a packet capture while rekordbox is running reveals that the CDJs send unicast packets to the rekordbox address on port 50000, in addition to the packets they normally broadcast on that port. Figuring out how to pose as rekordbox might be useful in order to see what additional data these can offer, although that may be much more work than posing as a CDJ.

6.2 Dysentery

If you have access to Pioneer equipment and are willing to help us validate this analysis, and perhaps even figure out more details, you can find the tool that is being used to perform this research at:
https://github.com/brunchboy/dysentery

http://deepsymmetry.org

---

[12]https://www.quora.com/What-database-system-did-Greg-Kemnitz-develop
[13]http://java.sys-con.com/node/328557
[14]https://www.crunchbase.com/organization/encirq-corporation
[15]http://www.ubiquitous.co.jp/en/news/press/pdf/p1730_01.pdf
[16]http://www.ubiquitous.co.jp/en/products/db/md/devicesql/