DJ Link Packet Analysis

James Elliott
Deep Symmetry, LLC

May 22, 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.

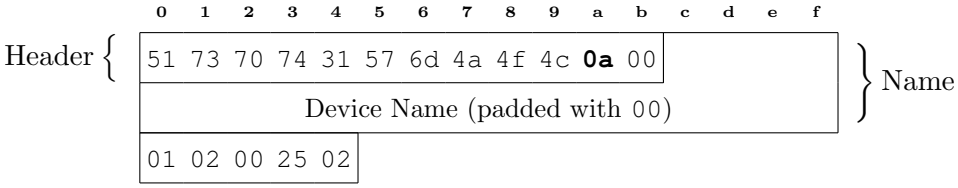

Figure 1: Initial announcement packets from Mixer

[1]The packet capture described in this analysis can be found at https://github.com/brunchboy/dysentery/raw/master/doc/assets/powerup.pcapng

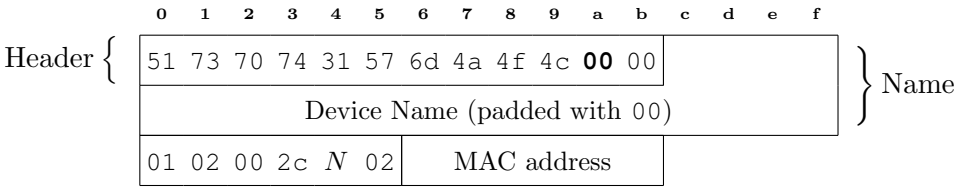

Figure 2: First-stage Mixer device number assignment packets

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.

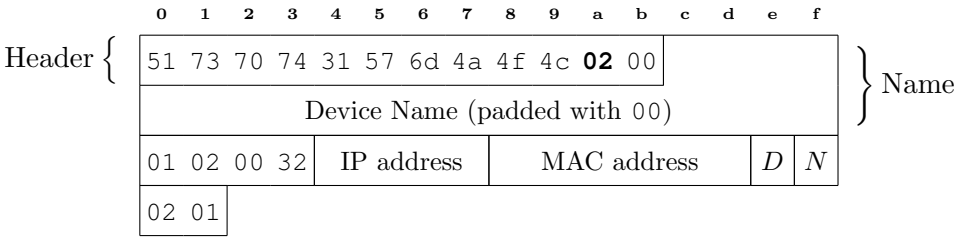

Figure 3: Second-stage Mixer device number assignment packets

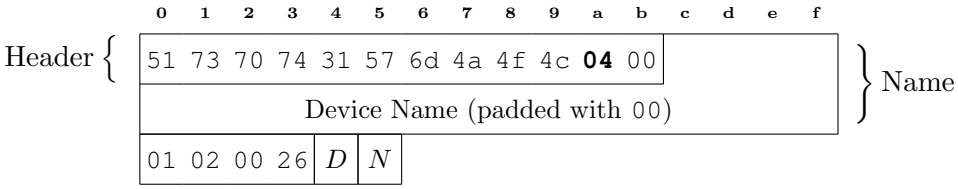

Figure 4: Final-stage Mixer device number assignment packets

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.

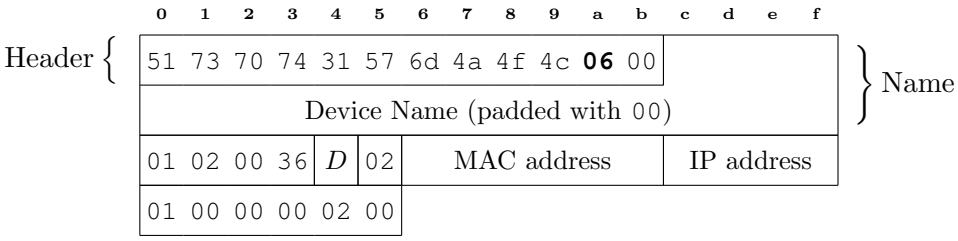

Figure 5: Mixer keep-alive packets

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.

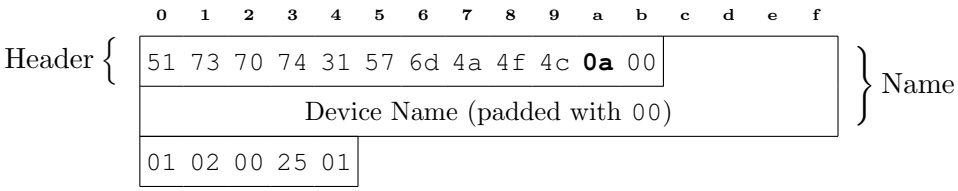

Figure 6: Initial announcement packets from CDJ

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).

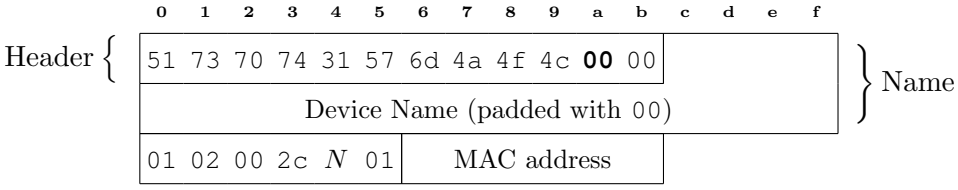

Figure 7: First-stage CDJ device number assignment packets

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

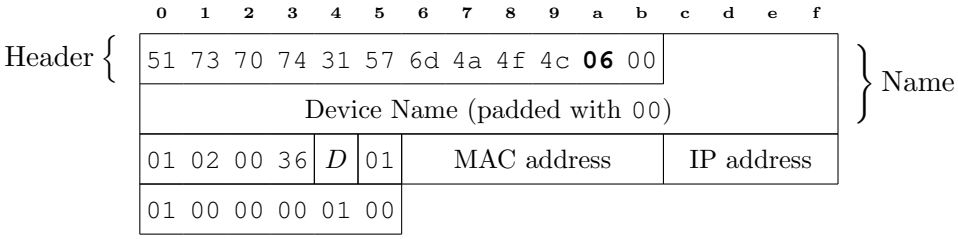

Figure 8: CDJ keep-alive packets

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	00	00	01	00	00	00	00	00	00	00	00	00
30	00	00	00	00	00	00	00	00	00	00	00	00	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	00	*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.

The player's current pitch adjustment[4] can be found in bytes 85–87, labeled *Pitch*. It 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%).

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 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]

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

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

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

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

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.

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.

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

[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.

[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	**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

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 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.

	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	04	00	00	00	U_l
70	00	00	00	04	00	U_g	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

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, `01` when a rekordbox track is loaded, and `05` when a regular 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 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 117, labeled U_g (for "USB global"), appears to have the value 1 whenever USB media is present in any player on the network, whether or not the Link option is chosen in the other players, and 0 otherwise. I don't know if that is true of other linkable SD media as well; this needs more investigation.

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.

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.

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 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.

[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`

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 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.

[10] `https://github.com/EvanPurkhiser`
[11] `https://bitbucket.org/awwright/libpdjl`

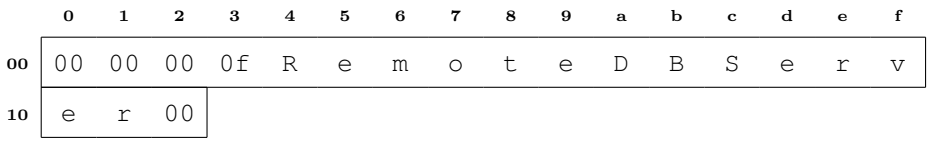

Figure 13: DB Server query packet

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 by at least four value bytes, although in the case of the variable-size types, those 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.

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.

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

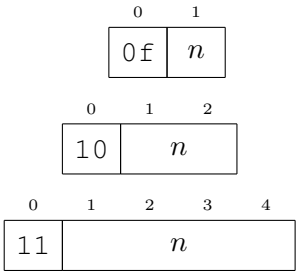

Figure 14: Number Fields of length 1, 2, and 4

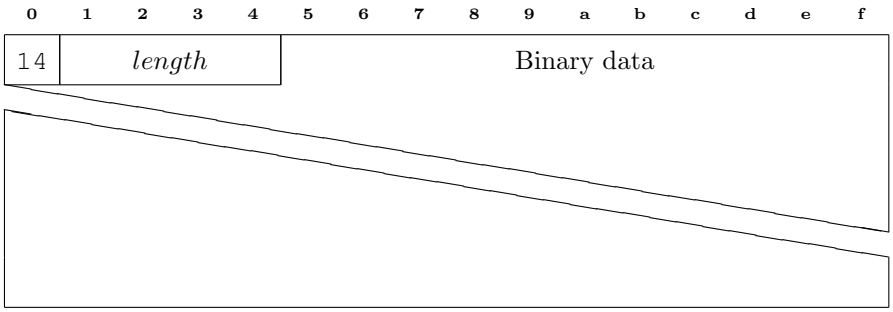

Figure 15: Binary (Blob) Field

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 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.

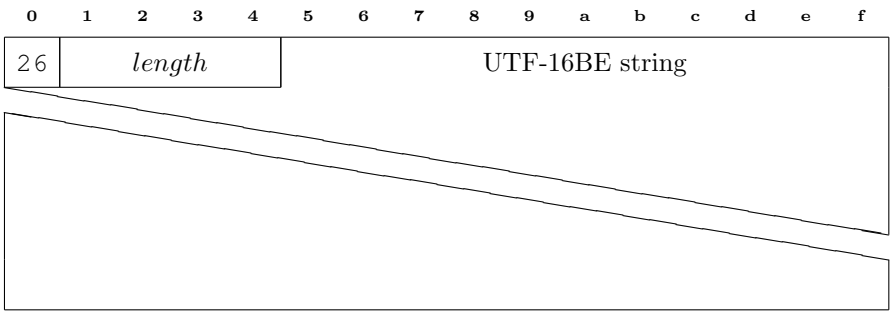

Figure 16: String Field

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 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

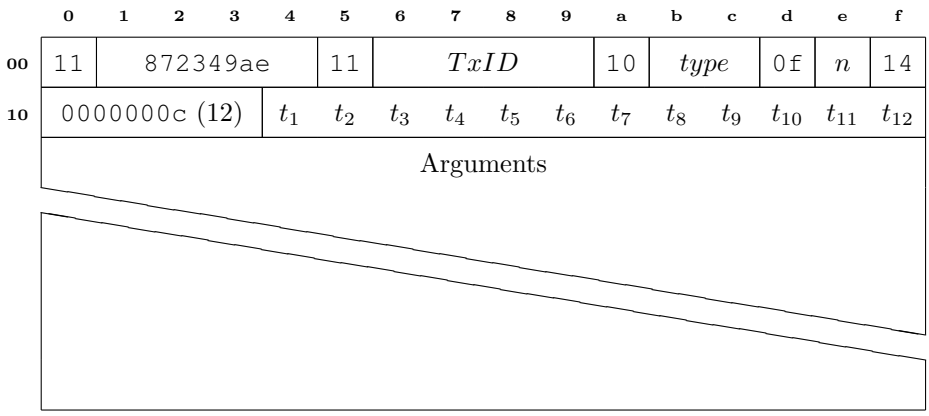

Figure 17: Message Header

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.

TODO: Resume here, showing response packet, now that we understand its structure.

The player responds with a 42 byte packet that starts with the message separator and the message sequence number you sent. We have not yet found any other meaningful information in this response.

At this point, the player is ready to accept queries.

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

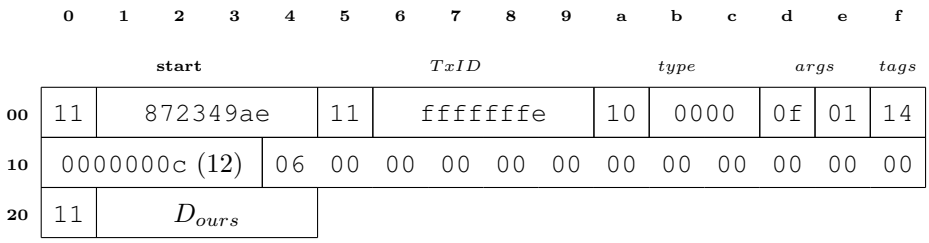

Figure 18: Query Context Setup message

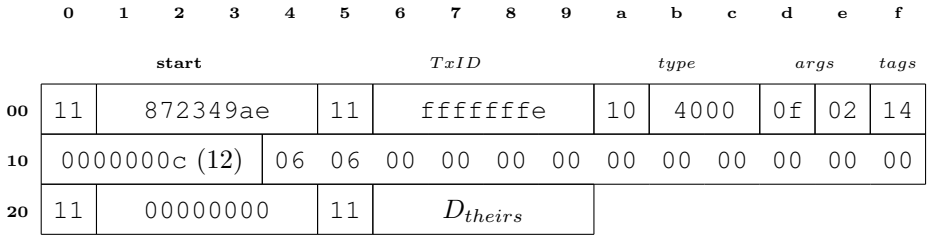

Figure 19: Query Context Setup response

player number. The overall structure is illustrated in Figure 19.

— TODO: Heavy revision needed from here to end of section! —

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

As described above, *seq* 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

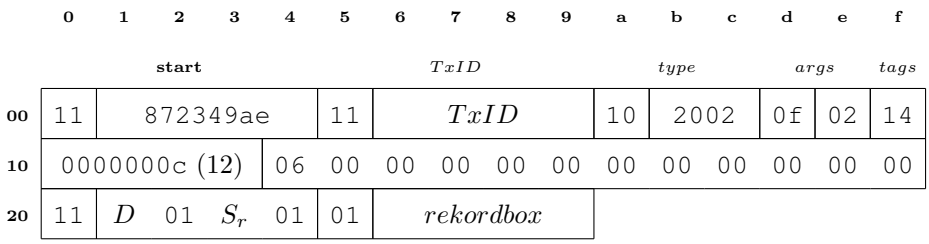

Figure 20: Track metadata request message

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00	11	87	23	49	ae	11	*seq*				10	30	00	0f	06	14
10	00	00	00	0c	06	06	06	06	06	06	00	00	00	00	00	00
20	11	D	01	S_r	01	11	00	00	00	00	11	00	00	00	0b	11
30	00	00	00	00	11	00	00	00	0b	11	00	00	00	00		

Figure 21: Render Menu request message (TODO: redo)

rekordbox database ID of the track being asked about, as found in the CDJ status packet.

Once again the player responds with a 42 byte packet that starts with the message separator and the message sequence number you sent, and which contains no known useful information. But at this point there is only one more message you need to send in order to get the metadata response. That has the content shown in Figure 21.

The value of *seq* should be one higher than the one you sent in your previous packet, while the values of D and S_r should be identical to what you sent in it.

This will cause the player to send back a series of messages containing the metadata fields describing the track you requested. Each field will begin with the six-byte message separator sequence, followed by the value of *seq* you sent in the final metadata query packet. Because this may be split across multiple packets, you should keep reading until you see the final response field, which will have the content shown in Figure 22.

The first two metadata response fields don't contain any values we have yet figured out. Several contain string values. These are stored as UTF-16 big-endian strings whose length is stored at bytes 42-45, and where the string itself begins at byte 46. (The byte indices here are counted from the start of the four-byte message sequence number that follows the message separator, so the sequence number is found at bytes 0-3.)

The string-containing fields are the third (track title), fourth (artist), fifth (album), eighth (comment), ninth (musical key), twelfth (genre), and thirteenth (label), although that has always been blank so far. It may represent the color label assigned to a track, and I just don't have tracks with that set.

The fifth field contains the track length in seconds, stored as a four-byte integer at bytes 32-35.

In addition to containing the track title, the third field also contains the track's artwork ID, as a 4-byte big-endian integer starting 19 bytes

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00	11	87	23	49	ae	11	*seq*				10	42	01	0f	00	14
10	00	00	00	0c	00	00	00	00	00	00	00	00	00	00	00	00

Figure 22: Menu footer message (TODO: redo)

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00	11	87	23	49	ae	11	*seq*				10	20	03	0f	02	14
10	00	00	00	0c	06	06	00	00	00	00	00	00	00	00	00	00
20	11	D	08	S_r	01	11	*artwork*									

Figure 23: Artwork request message (TODO: redo)

before the end of the field.

To request the artwork image corresponding to that ID, send a packet like the one shown in Figure 23.

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.

The response will be a single field, starting with the six-byte message separator sequence followed by the sequence number you sent as *seq*. The length in bytes of the image will be specified by a four-byte big-endian integer starting at byte 48 of the response field, and the image itself will be sent as the indicated number of bytes immediately following the length (so starting at byte 52 of the response).

5.3 Background Research

Prior to Evan's breakthrough, 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

[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/`

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`