DJ Link Packet Analysis

James Elliott
Deep Symmetry, LLC

February 18, 2020

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.

Contents

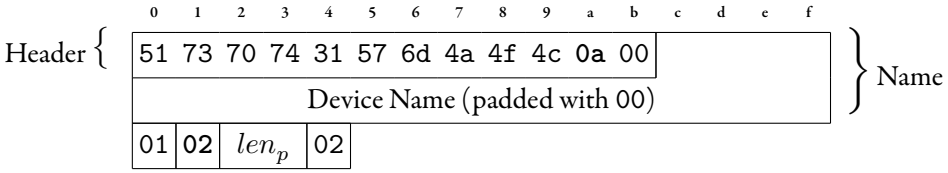

Figure 1: Initial announcement packets from Mixer

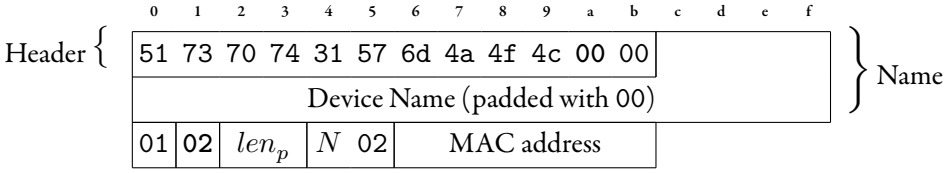

Figure 2: First-stage Mixer device number assignment packets

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.

These have a data length[2] of 25 bytes, appear roughly every 300 milliseconds, and have the content shown in Figure 1.

Byte 0a (inside what is labeled the header) is bolded because its value changes in the different types of packets which follow.

The byte following the device name (at byte 20) seems to always have the value 1 in every kind of packet seen. The next byte is bolded as well because it seems to indicate the structure of the remainder of the packet. The value 02 is followed by a two-byte value len_p that indicates the length of the entire packet (including the preceding header bytes), and followed by the payload. In the case of this kind of packet, the length is 0025, and the payload is the single-byte value 02.

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 2c bytes, reflected in len_p, are again sent to UDP port 50000 on the local network broadcast address, at

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

[2] Values within packets, packet lengths, and byte offsets are all shown in hexadecimal.

roughly 300 millisecond intervals, and have the content shown in Figure 2.

The value N at byte 24 is 01, 02, or 03, 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 and len_p value of 32 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.

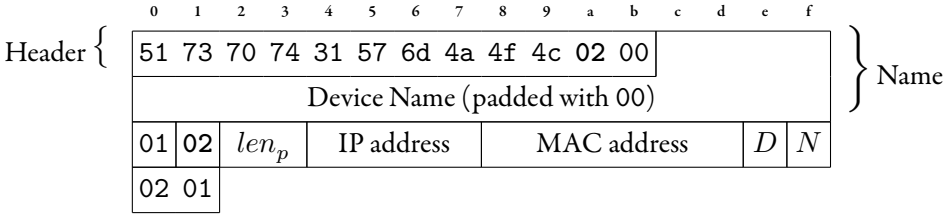

Figure 3: Second-stage Mixer device number assignment packets

I identify these as claiming/identifying the device number because the value D at byte 2e is the same as the device number that the mixer uses to identify itself (21) and the same is true for the corresponding packets seen from the CDJs (they use device numbers 02 and 03, as they are connected to those ports/channels on the mixer).

As with the previous series of three packets, the value N at byte 2f takes on the values 01, 02, and 03 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 26 bytes of data and the content shown in Figure 4.

As before the value D at byte 24 is the same as the device number that the mixer uses to identify itself (21) and N at byte 25 takes on the values 01, 02, and 03 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 36 bytes long, again sent to

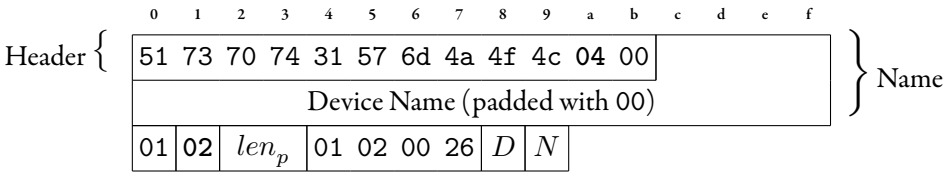

Figure 4: Final-stage Mixer device number assignment packets

port 50000 on the local network broadcast address, roughly every second and a half. They have the content shown in Figure 5.

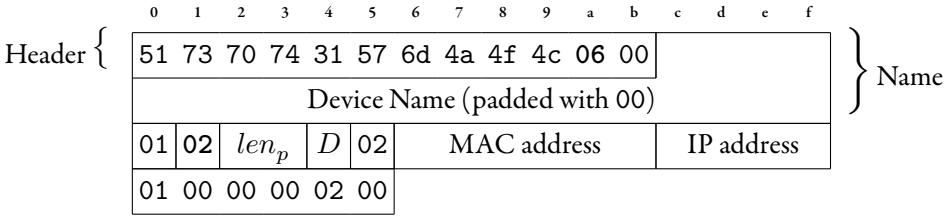

Figure 5: Mixer keep-alive packets

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 01 for the CDJ, and was 02 for the mixer.

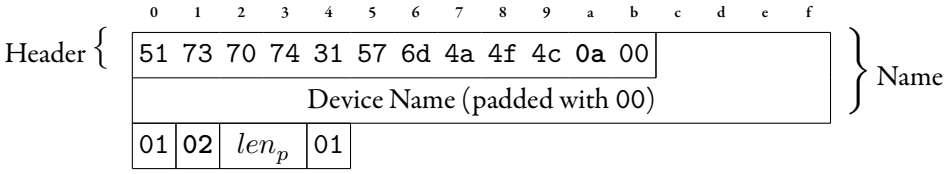

Figure 6: Initial announcement packets from CDJ

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 25 (immediately after the packet counter N), which again is 01 for the CDJ, and was 02 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 26-byte packet with header byte 0a set to 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).

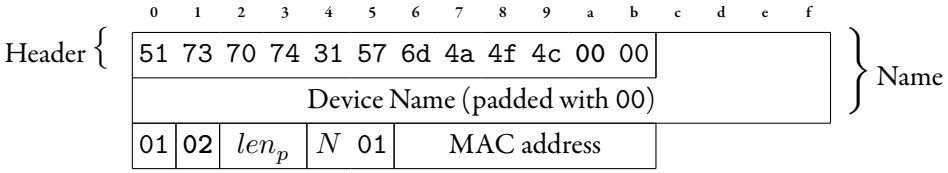

Figure 7: First-stage CDJ device number assignment packets

The CDJ then moves to the keep-alive stage, sending out 36-byte packets with the content shown in Figure 8.

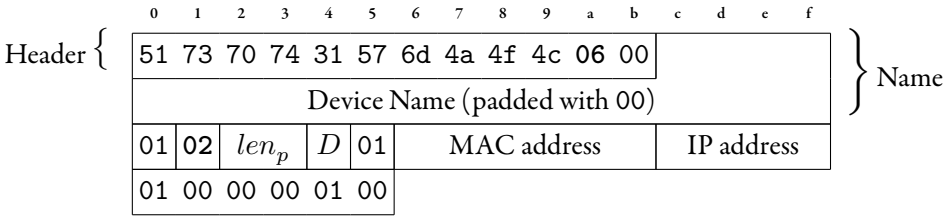

Figure 8: CDJ keep-alive packets

As seems to always be the case when comparing mixer and CDJ packets, the difference between this and Figure 5 is that byte 25 (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 34 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 2d bytes of data, and a longer packet containing 60 bytes. The shorter packets contain information about which channels are on-air, and fader start/stop commands to the players, as described in Section 9.

[3] https://github.com/deep-symmetry/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	len_r		$nextBeat$				$2ndBeat$				$nextBar$			
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 60-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.

This introduces our first packet structure variant following the device name. As always the byte after the name has the value 1, but the subtype value which follows that (at byte 20) has the value 00 here, rather than 02 as we saw in the startup packets. In packets of subtype 00 the subtype indicator is followed by the Device Number D at byte 21; this is the Player Number as displayed on the CDJ itself, or 21 for the mixer, or another value for a computer running rekordbox. And that is followed by a different kind of length indicator: len_r at bytes 22–23 reports the length of the *rest of the packet*, in other words, the number of bytes which come after len_r. In this packet len_r has the value 003c. For some reason, there is a redundant copy of D at the end of the packet, in byte 5f. That seems common in packets with subtype 00, and is one of many inefficiencies in the protocol.

To facilitate synchronization of variable-tempo tracks, the number of milliseconds after which a variety of upcoming beats will occur are reported. *Note that the timing values for all these upcoming beats are always reported as if the track was being played at normal speed, with a pitch adjustment of 0%. If a pitch adjustment is in effect, you will need to perform the calculation to scale the beat timing values yourself.* $nextBeat$ at bytes 24–27 is the number of milliseconds in which the very next beat will arrive. $2ndBeat$ (bytes 28–2b) is the number of milliseconds until the beat after that. $nextBar$ (bytes 2c–2f) reports the number of milliseconds until the next measure of music begins, which may be from 1 to 4 beats away. $4thBeat$ (bytes 30–33) reports how many millisecond will elapse until the fourth upcoming beat; $2ndBar$ (bytes 34–37) 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 38–3b) 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 54–57, 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 by multipling the following (hexadecimal) equation by decimal 100:

$$\frac{(byte[55] \times 10000 + byte[56] \times 100 + byte[57]) - 100000}{100000}$$

The current BPM of the track playing on the device[5] can be found at bytes 5a–5b (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 (in this case only the byte offsets are hexadecimal):

$$\frac{byte[5a] \times 256 + byte[5b]}{100}$$

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 hexadecimal equation that results in the effective BPM to two decimal places, by combinining the BPM and $Pitch$ values:[6]

$$\frac{(byte[5a] \times 100 + byte[5b]) \times (byte[55] \times 10000 + byte[56] \times 100 + byte[57])}{6400000}$$

The counter B_b at byte 5c 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

[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.

[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.

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 05 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.

But note that use of a non-standard player number (outside the range 1–4) will interfere with your ability to perfrom metadata requests using `dbserver` queries as described in Section 6.2. In situations where there are four actual players on the network you can use alternate ways to get the data, as described in Section 12.3.

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 38 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	len_r			D	00	00	F	$Pitch$			80	00	BPM	
30	00	10	00	00	00	09	M_h	B_b								

Figure 10: Mixer status packets

Since these use packet subtype 00, the len_r value reports there are 14 bytes remaining after it.

Packets coming from a DJM-2000 nexus connected as the only mixer on the network contain a value of 21 for their Device Number D (bytes 21 and 24).

It seems that rekordbox sometimes sends "mixer status" packets like this as well, but with yet another packet subtype variant: it sends a value of 01 for byte 20, and for packets with that subtype, the length at bytes 22–23 is a len_p value, reporting

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

the length of the entire packet, rather than the number of bytes remaining in the packet. The packets are otherwise identical.

The value marked F at byte 27 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 f0 when it is the tempo master, and d0 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 28–2b and bytes 30–33, but since they always 100000 (or $+0\%$), we can't be sure. The first value is structurally in the same place with respect to BPM as it is found in all other packets containing pitch information, so that is the one we are assuming is definitive. In any case, since it is always $+0\%$, the current tempo in beats-per-minute identified by the mixer can be obtained as (only the byte offsets are hexadecimal):

$$\frac{byte[\texttt{2e}] \times 256 + byte[\texttt{2f}]}{100}$$

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 $byte[\texttt{37}]$, 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 $byte[\texttt{36}]$, labeled M_h (master handoff), is used to hand off the tempo master role. It starts out with the value 00 when there is no Master player, but as soon as one appears it becomes ff. If the mixer has been the tempo master and it is currently yielding this role to another player, this value will be the player number that is becoming tempo master during that handoff, as described in Section 5.

4.2 CDJ Status Packets

Packets from a CDJ will have a length of d4 bytes and the content shown in Figure 11 for nexus players. Older players send d0-byte packets with slightly less information. Newer firmware and Nexus 2 players send packets that are 11c or 124 bytes long.

These use yet another packet structure variant following the device name. As always the byte after the name has the value 01, but the subtype value which follows that (at byte 20) has the value 03 here, rather than 00 as we saw in the mixer status packets. Packets of subtype 03 seem structurally equivalent to subtype 00 however: the subtype indicator is followed by the Device Number D at byte 21 and a length-remaining value len_r at bytes 22–23. If anyone can think of a reason why these packets don't simply reuse subtype 00, please share it!

	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	len_r		D	00	01	A	D_r	S_r	T_r	00	$rekordbox$			
30	00	00	$Track$		00	00	00	d_l	00	00	a0	00	00	00	00	00
40	00	00	00	00	00	00	d_n		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	$Sync_n$				00	F	ff	P_2	$Pitch_1$			
90	M_v		BPM		7f	ff	ff	ff	$Pitch_2$				00	P_3	M_m	M_h
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	$Pitch_3$				$Pitch_4$				$Packet$				nx	00	00	00
d0	00	00	00	00												

Figure 11: CDJ status packets

The Device Number in D (bytes 21 and 24) is the Player Number as displayed on the CDJ itself. In the case of this capture, the value of len_r was 00b0.

The activity flag A at byte 27 seems to be 00 when the player is idle, and 01 when it is playing, searching, or loading a track.

When a track is loaded, the device from which the track was loaded is reported in D_r at byte 28 (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 29 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 2a indicates the track type. It has the value 00 when no track is loaded, 01 when a rekordbox track is loaded, 02 when an unanalyzed track is loaded (from a media slot without a rekordbox database, including from a data disc), and 05 when an audio CD track is loaded.

The field $rekordbox$ at bytes 2c–2f contains the rekordbox database ID of the loaded track when a rekordbox track is being played. When a non-rekordbox media slot track is loaded, it is still a unique ID by which the track can be identified for metadata requests, and when an audio CD track is loaded, this is just the track number. In all cases, combined with the player number and slot information, this can be used to request the track metadata as described in Section 6.

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 32 and 33, labeled $Track$. (It may be a 4-byte value and also include bytes 30 and 31, but that would seem an unmanageable number of tracks to search through.)

The field d_l at byte 37 was given this label because we first believed it to indicate when a disc is loaded. It has the value 00 when the disc slot is empty, and seems to have the value 1e when a CD Audio disc is loaded, and 11 when a data disc containing files in MP3, AAC, WAV or AIFF format is loaded. However, we later noticed that it also gets non-zero values when a track is loaded from a playlist (05) or another player menu (different values, depending on the type of menu; if anyone would like to take the time to record all the different values and their meanings, a pull request would be extremely welcome). Relatedly, when a track is loaded from a disc, playlist, or menu, the field d_n at bytes 46–47 changes from 0000 to the number of tracks on the disc or in the playlist or menu (a data disc will generally have one track).

Some of the fields shown as having value 00 in this region will sometimes have other values in them; their meanings are simply not yet known. If you notice any patterns or figure anything out *please* open a pull request and let us know!

Byte 6a, labeled U_a (for "USB activity"), alternates between the values 04 and 06 when there is USB activity—it may even alternate in time with the flashing USB indicator LED on the player, although visual inspection suggests there is not a perfect correlation. Byte 6b, S_a, is the same kind of activity indicator for the SD slot. Byte 6f (U_l for "USB local") has the value 04 when there is no USB media loaded, 00 when USB is loaded, and 02 or 03 when the USB Stop button has been

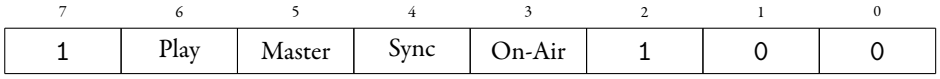

Figure 12: CDJ state flag bits

pressed and the USB media is being unmounted.

Byte 73 (S_l for "SD local") has the value 04 when there is no SD media loaded, 00 when SD is loaded, and 02 or 03 when the SD door has been opened and the SD media is being unmounted.

Byte 75, labeled L (for "Link available"), appears to have the value 01 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 00 otherwise.

Byte 7b, labeled P_1, appears to describe the current play mode. The values seen so far, and their apparent meanings, are shown in Table 1.

Value	Meaning
00	No track is loaded
02	A track is in the process of loading
03	Player is playing normally
04	Player is playing a loop
05	Player is paused anywhere other than the cue point
06	Player is paused at the cue point
07	Cue Play is in progress (playback while the cue button is held down)
08	Cue scratch is in progress
09	Player is searching forwards or backwards
0e	Audio CD has spun down due to lack of use
11	Player reached the end of the track and stopped

Table 1: Known P_1 Values

The $Firmware$ value at bytes 7c–7f is an ASCII represenation of the firmware version running in the player.

The value $Sync_n$ at bytes 84–87 changes whenever a player gives up being the tempo master; at that point it gets set to a value one higher than the highest $Sync_n$ value reported by any other player on the network. This is part of the Baroque master handoff dance described in Section 5.3.

Byte 89, 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 00 for this byte?

[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/deep-symmetry/dysentery/issues

Byte 8b, labeled P_2 seems to be another play state indicator, having the value 7a when playing and 7e 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 03), while P_2 considers it to be stopped (value 7e). Non-nexus players seem to use the value 6a when playing and 6e when stopped, while nxs2 players use the values fa and fe, and the XDJ-XZ uses the values 9a and 9e 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 8c–8f, $Pitch_2$ at bytes 98–9b, $Pitch_3$ at bytes c0–c3, and $Pitch_4$ at bytes c4–c7.

Each of these values represents a four-byte pitch adjustment percentage, where 00100000 represents no adjustment (0%), 00000000 represents slowing all the way to a complete stop (-100%, reachable only in Wide tempo mode), and 00200000 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.

The pitch adjustment percentage represented by $Pitch_1$ would be calculated by multiplying decimal 100 by the following hexadecimal equation:

$$\frac{(byte[\texttt{8d}] \times \texttt{10000} + byte[\texttt{8e}] \times \texttt{100} + byte[\texttt{8f}]) - \texttt{100000}}{\texttt{100000}}$$

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 100000, 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 000000.

When playback stops, either due to the play button being pressed or the jog wheel held down, the value of $Pitch_4$ drops to 000000 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 `92`–`93` (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 (only byte offsets are hexadecimal):

$$\frac{byte[\texttt{92}] \times 256 + byte[\texttt{93}]}{100}$$

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 hexadecimal equation that results in the effective BPM to two decimal places, by combinining the BPM and $Pitch1$ values:

$$\frac{(b[\texttt{92}] \times \texttt{100} + b[\texttt{93}]) \times (b[\texttt{8d}] \times \texttt{10000} + b[\texttt{8e}] \times \texttt{100} + b[\texttt{8f}])}{\texttt{100000}}$$

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 `ffff`.

M_v (bytes `90`–`91`) seems to control whether the BPM value is accepted when this player is the master. It has the value `7fff` when no track is loaded, `8000` when a rekordbox track is loaded, and `0000` when a non-rekordbox track (like from a physical CD) is loaded, and only when the value is `8000` are tempos from this player accepted when it is acting as master (otherwise the mixer shows the master to have a tempo of "`---.-`" and other players do not respond to its tempo changes).

Byte `9d` (labeled P_3) seems to communicate additional information about the current play mode. The meanings that we have found so far are listed in Table 2.

Value	Meaning
`00`	No track is loaded
`01`	Player is paused or playing in Reverse mode
`09`	Player is playing in Forward mode with jog mode set to Vinyl
`0d`	Player is playing in Forward mode with jog mode set to CDJ

Table 2: Known P_3 Values

Byte `9e` (labeled M_m) is another representation of whether this player is currently the tempo master (in addition to bit `5` of F, as shown in Figure 12). It has the value `00` when the player is not the master, the value `01` when it is tempo master and is playing a rekordbox-analyzed track, so that actually has a meaningful effect,

and the value 02 when it is supposed to be the master (and the value of F still indicates that it is), but it is playing a non-rekordbox track, so it is unable to send tempo and beat information to other players.

The following byte, 9f (labeled M_h), is related. As described in Section 5, this normally has the value ff. But when this player is giving up the role of tempo master in response to a request from another player that wants to take over, it holds that player's device number until it sees that device announce itself as the new master, at which point it turns off its own master flags, and this value goes back to ff.

The 4-byte beat counter (which counts each beat from 1 through the end of the track) is found in bytes a0–a3, 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 ffffffff.

The counter B_b at byte a6 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 a4–a5 (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 01ff 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 0100. 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 0001 and the CDJ displays "00.1 bars". On the beat on which the cue point was saved the value is 0000 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 c8-cb 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 cc, labeled nx, seems to have the value 0f for nexus players, 1f for the XDJ-XZ, and 05 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 21 and 24) seems to be 29, 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 c0 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 `100000`, or +0%) are present, as is X. B_b always seems to be zero.

5 Sync and Tempo Master

The DJM has a mode for its touchscreen which allows you to see and control which players are synced and which is the tempo master, and of course individual players can take over being master as well. This section describes the packets used to implement these features.

5.1 Sync Control

To tell a player to turn Sync mode on or off, send a packet like the one shown in Figure 13 to port 50001 of the target device, with the player number that you are pretending to be as the value of D, and set the value of S to `0x10` if you want the player to turn on Sync, and `0x20` if you want it to leave Sync mode.

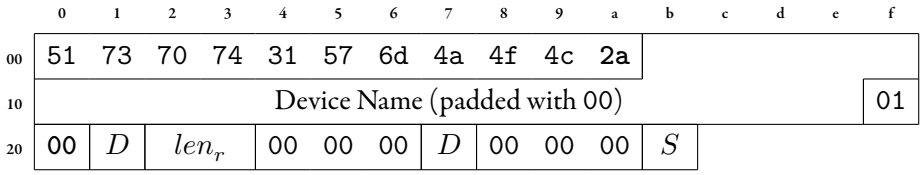

Figure 13: Sync control packet

5.2 Tempo Master Assignment

To tell a player to become tempo master, the same type of packet shown in Figure 13 is used, with a value of `01` for S. This will cause the player to behave as if the DJ had pressed its Master button, following the steps described in the next section. This packet can be sent to a CDJ or DJM mixer. Since this packet uses subtype `00`, the length sent in len_r has the value `0008`, reflecting the eight bytes which follow it.

5.3 Tempo Master Handoff

When a player or mixer is to become tempo master, regardless of whether this was initiated by pressing its Master button or by receipt of the packet described in the preceding section, the same process is followed.

If there is currently no tempo master, the device simply becomes master, and starts sending status packets with appropriate values of F and M_m (mixer status packets only have F in them).

If another player is currently tempo master, however, a coordinated handoff takes place. The device that wants to become tempo master first sends a takeover request packet like the one shown in Figure 14 to port 50001 of the current tempo master, with the player number of the device wanting to become master as the value of D.

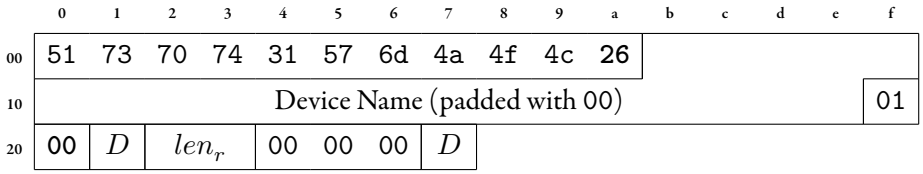

Figure 14: Tempo master takeover request packet

Since this packet uses subtype 00, the length sent in len_r has the value 0004, reflecting the four bytes which follow it.

The current tempo master will agree to the handoff by sending a packet like the one shown in Figure 15 to port 50001 of the device that sent the takeover request, with the its own device number as the value of D.

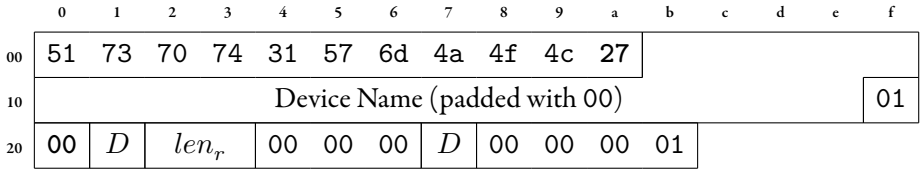

Figure 15: Tempo master takeover response packet

Since this packet uses subtype 00, the length sent in len_r has the value 0008, reflecting the eight bytes which follow it.

Once that is done, the outgoing master will continue to report itself as the master according to its status packets (bit 5 of F, and for CDJs, the value of M_m) but it will announce to the world that the handoff is taking place by sending the device number of the device that is about to become tempo master as the value of M_h. (See Figures 10 and 11 for the locations of these bytes.)

As soon as the device becoming tempo master sees its device number in M_h in the status packets from the outgoing tempo master, it starts reporting itself as the tempo master using F and, for CDJs, M_m in its own status packets.

And as soon as the outgoing tempo master sees the new master has asserted this role in its status packets, it stops reporting itself as tempo master in its own status packets, goes back to sending the value ff in M_h, and sets its $Sync_n$ value to be one greater than the $Sync_n$ value reported by any other player on the network (although mixers do not report this value at all). This concludes the (rather Baroque) handoff.

	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 16: DB Server query packet

5.4 Unsolicited Handoff

While working on synchronizing Pro DJ Link devices with Ableton Link, I accidentally discovered that there is another way the tempo master role can be handed off. If the device that is currently tempo master is stopped (not playing a track), and it sees another device that is both synced and playing, it will set M_h to the device number of synced, playing device, telling it to become the new master. As soon as the device named by M_h sees that status packet, it should take over the role as described in the second-to-last paragraph of Section 5.3 above, even though it did not start the process.

6 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 16.

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. (You can also get metadata for non-rekordbox tracks, even for CD Audio tracks being played in the CD slot, using the variation described in Section 6.6.)

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

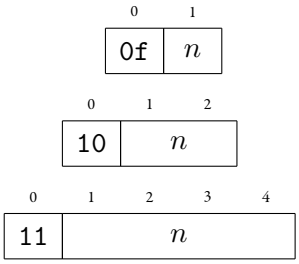

Figure 17: Number Fields of length 1, 2, and 4

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.

6.1 Field Types

The first byte of a field identifies what type of field is coming. The values `0f`, `10`, and `11` are followed by 1, 2, and 4 byte fixed-length integer fields, while `14` and `26` introduce variable-length fields, a binary blob and a UTF-16 big-endian string respectively.

6.1.1 Number Fields

Number fields are indicated by an initial byte `0f`, `10`, or `11` which is followed by big-endian integer value of length 1, 2, or 4 bytes respectively, as shown in Figure 17. 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.

6.1.2 Binary Fields

Variable-length binary (blob) fields are indicated by an initial byte `14`, 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 18.

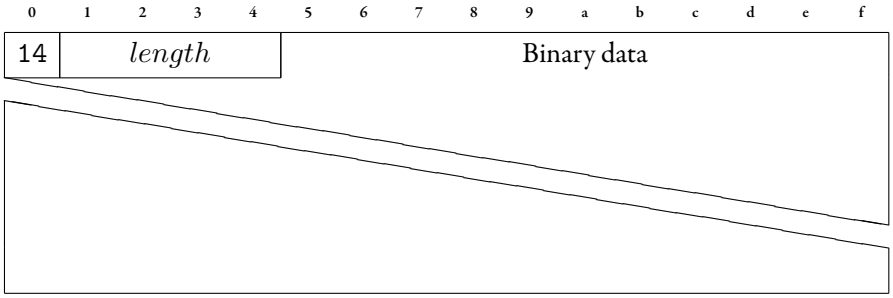

Figure 18: Binary (Blob) Field

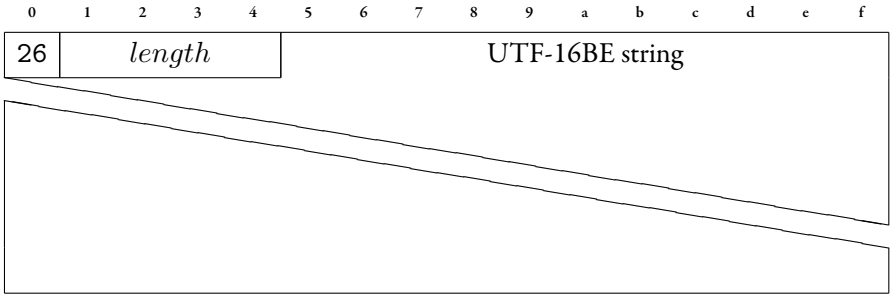

Figure 19: String Field

6.1.3 String Fields

Variable-length string fields are indicated by an initial byte `26`, 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 `0000`. This is illustrated in Figure 19.

6.2 Messages

Messages are introduced by a 4 byte Number field containing a "magic" value (it is always `872349ae`). 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 twelve bytes long, regardless of how few arguments there are (and presumably this means no message ever has more than twelve arguments).

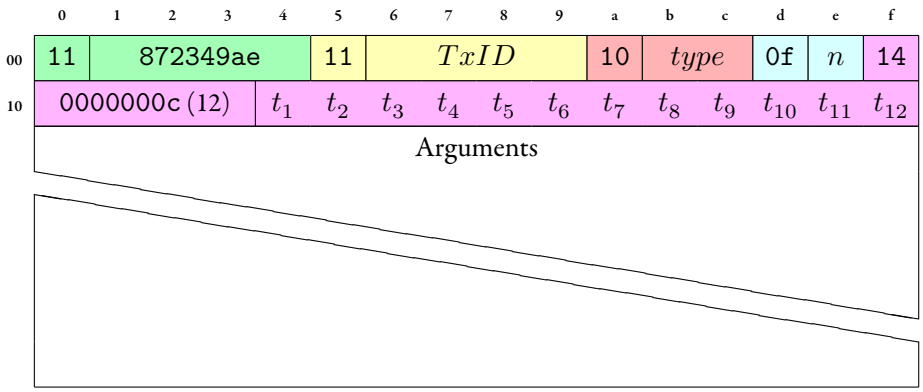

Figure 20: Message Header

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. Table 3 lists the known tag values and their meanings.

I am guessing that if we ever see them, a tag of `04` would represent a 1 byte integer, and `05` 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 20, 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 `00` 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 `0000`, a special $TxID$ value of `fffffffe`, and a single numeric argument, as shown in Figure 21.

Tag	Meaning
02	A string in UTF-16 big-endian encoding, with trailing NUL (zero) character
03	A binary blob
06	A 4 byte big-endian integer

Table 3: Argument Tag Values

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

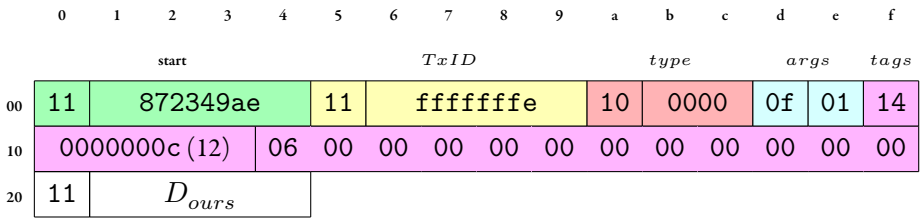

Figure 21: Query context setup message

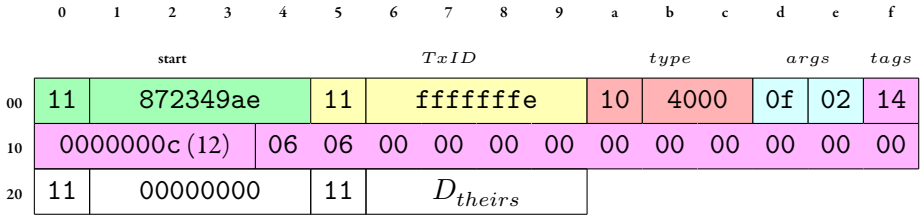

Figure 22: Query context setup response

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 4000, 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 0000), 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 22.

6.3 Rekordbox Track Metadata

To ask for metadata about a particular rekordbox track, send a packet like the one shown in Figure 23.

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, T_r identifies the type of track we want information about; for rekordbox tracks this always has the value 01. And $rekordbox$ identifies the local rekordbox database ID of the track being asked about, as found in the CDJ status packet.

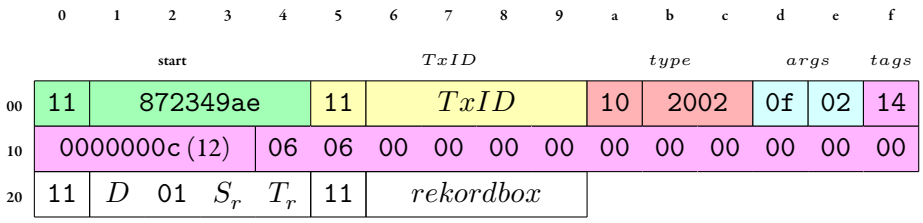

Figure 23: Rekordbox track metadata request message

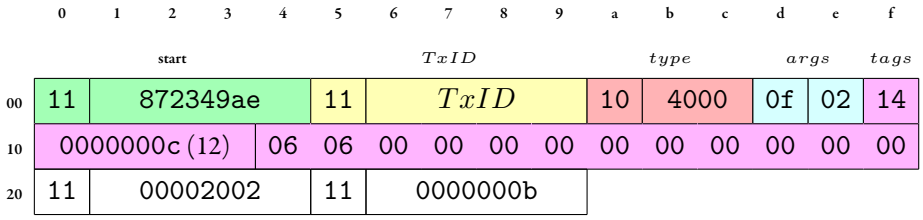

Figure 24: Track metadata available response

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 24.

We've seen this type of "data available" response already in Figure 22, 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 2002, reflects the *type* field of our request. The second argument reports that there are 11 (0b) 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 `ffffffff` to let us know. If we messed up something else about the request, we will get a response with a *type* other than 4000. See Section 6.16 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* 3000 shown in Figure 25.

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

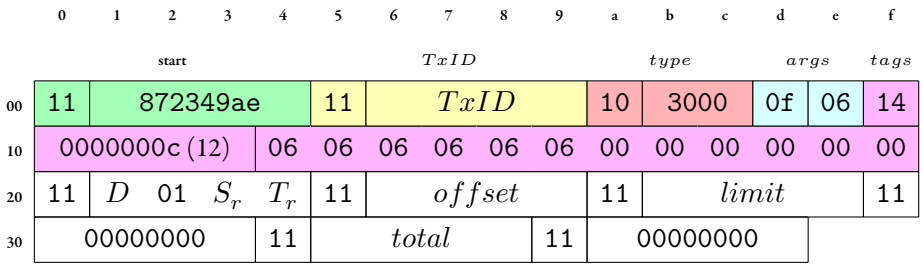

Figure 25: Render Menu request message

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 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.

As described above, T_r has the value 1 for rekordbox tracks.

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, $total$, seems to usually contain the total number of items reported in the intial menu response, but sending a second copy of $limit$ here works too; it may not matter much. 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 26:

This causes the player to send us 13 messages: The 11 metadata items we requested are sent (with $type$ 4101, Figure 28), but they are preceded by a menu header message (with $type$ 4001, Figure 27), and followed by a menu footer message (with $type$ 4201, Figure 29). 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 the slots. They have the general structure shown in in Figure 28, and the arguments are listed in Table 4.

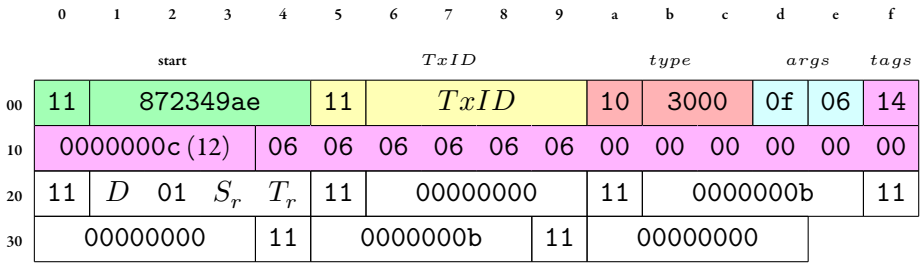

Figure 26: Render track metadata request message

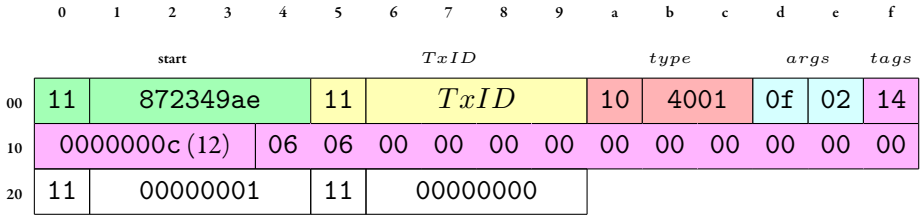

Figure 27: Menu header response

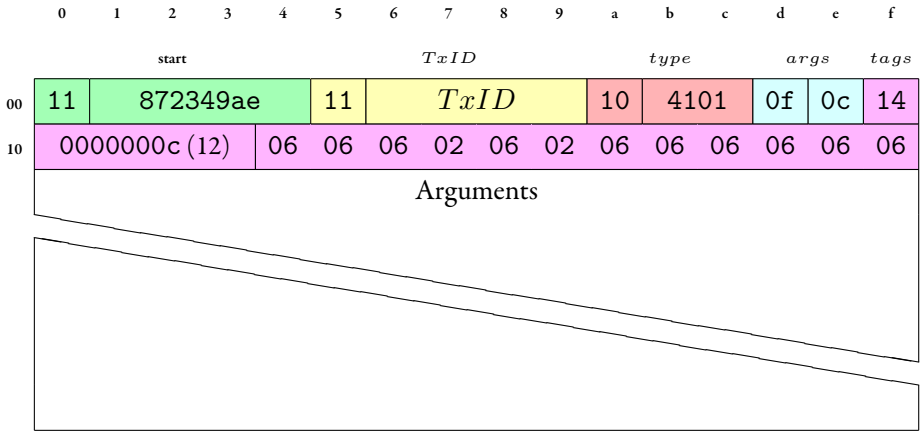

Figure 28: Menu item response

Arg	Type	Meaning
1	number	parent ID, such as an artist for a track item
2	number	main ID, such as *rekordbox* for a track item
3	number	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	number	length in bytes of Label 2
6	string	Label 2 (secondary text, e.g. artist name for playlist entries, where Label 1 holds the title)
7	number	type of this item (see Section 6.5)
8	number	some sort of flags field, details still unclear
9	number	holds *artwork* ID when type is Track Title
10	number	playlist position when relevant, e.g. when listing a playlist
11	number	unknown
12	number	unknown

Table 4: Menu Item Arguments

6.3.1 Track Metadata Item 1: Title

The first item returned after the menu header is the track title, so argument 7 has the value `04`. 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 9 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 6.7.

6.3.2 Track Metadata Item 2: Artist

The second item contains artist information so argument 7 has the value `07`. Argument 2 holds the artist ID, argument 4 contains the text of the artist name.

6.3.3 Track Metadata Item 3: Album Title

The third item contains album title information so argument 7 has the value `02`. Argument 4 contains the text of the album name.

6.3.4 Track Metadata Item 4: Duration

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

6.3.5 Track Metadata Item 5: Tempo

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

6.3.6 Track Metadata Item 6: Comment

The sixth item contains comment information so argument 7 has the value `23`. Argument 4 contains the text of the track comment entered by the DJ in rekordbox.

6.3.7 Track Metadata Item 7: Key

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

6.3.8 Track Metadata Item 8: Rating

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

6.3.9 Track Metadata Item 9: Color

The ninth item contains color information so argument 7 has a value between `13` and `1b` identifying the color, if any, assigned to the track (see Section 6.5 for the color choices), and if the value is anything other than `13`, Argument 4 contains the text that the DJ has assigned for that color meaning in rekordbox.

6.3.10 Track Metadata Item 10: Genre

The tenth item contains genre information so argument 7 has the value `06`. Argument 2 contains the numeric genre ID, and argument 4 contains the text of the genre name.

6.3.11 Track Metadata Item 11: Date Added

The eleventh and final item contains the date added information so argument 7 has the value `2e`. 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.

6.4 Menu Footer Response

The menu footer message has a *type* of `4201` 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 29.

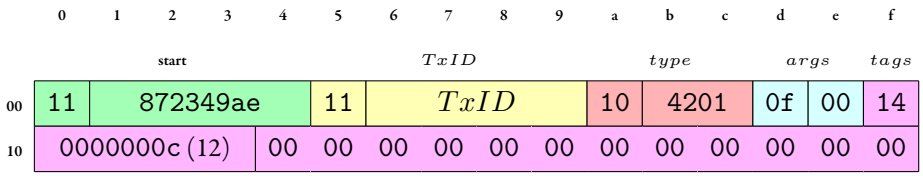

Figure 29: Menu footer response

6.5 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 listed in Table 5.

Type	Meaning
0001	Folder (such as in the playlists menu)[12]
0002	Album title
0003	Disc
0004	Track Title
0006	Genre
0007	Artist
0008	Playlist
000a	Rating
000b	Duration (in seconds)
000d	Tempo
000e	Label
000f	Key
0010	Bit Rate
0011	Year
0013	Color None
0014	Color Pink
0015	Color Red
0016	Color Orange
0017	Color Yellow
0018	Color Green
0019	Color Aqua
001a	Color Blue
001b	Color Purple
0023	Comment
0024	History Playlist
0028	Original Artist
0029	Remixer

Table 5: Known Menu Item Types

Type	Meaning
002e	Date Added
0080	Genre menu
0081	Artist menu
0082	Album menu
0083	Track menu
0084	Playlist menu
0085	BPM menu
0086	Rating menu
0087	Year menu
0088	Remixer menu
0089	Label menu
008a	Original Artist menu
008b	Key menu
008e	Color menu
0090	Folder menu
0091	Search "menu"
0092	Time menu
0093	Bitrate menu
0094	Filename menu
0095	History menu
00a0	All
0204	Track title and album
0604	Track Title and Genre
0704	Track Title and Artist
0a04	Track Title and Rating
0b04	Track Title and Time
0d04	Track Title and BPM
0e04	Track Title and Label
0f04	Track Title and Key
1004	Track Title and Bit Rate
1a04	Track Title and Color
2304	Track Title and Comment
2804	Track Title and Original Artist
2904	Track Title and Remixer
2a04	Track Title and DJ Play Count
2e04	Track Title and Date Added

Table 5: Known Menu Item Types

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

[12] A nested list of playlists rather than an individual playlist.

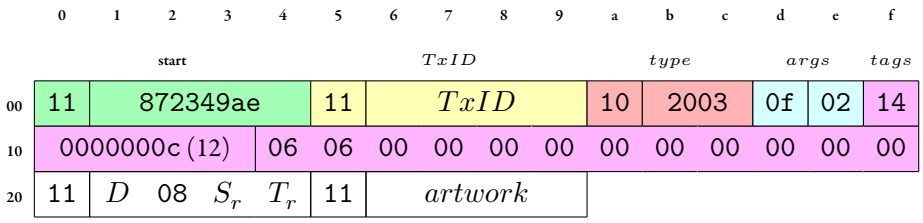

Figure 30: Track artwork request message

6.6 Non-Rekordbox Track Metadata

As noted in the introduction to this section, you can get metadata for non-rekordbox tracks as well (although they don't have beat grids, waveforms, or album art available). All you need to do is use a slight variant of the metadata request message shown in Figure 23, using the value 2202 (instead of 2002) for the message type, and a value of T_r that is appropriate for the kind of track you are asking about (02 for non-rekordbox tracks loaded from media slots, and 05 for CD audio tracks playing in the CD slot, which has a S_r value of 01). After the initial query setup message, the other message types are the same as in the above discussion, but you will continue using the S_r and T_r values appropriate for the slot and media type you are asking about.

Since these tracks don't have rekordbox IDs, you will need to use the *recordbox* value reported in CDJ status packets in order to find out the values used to request the metadata for tracks loaded from solid state media; CD tracks are requested using the simple track number as the *rekordbox* value.

It seems that to reliably get data back when requesting metadata for non-rekordbox tracks, your virtual CDJ needs to be sending properly-formatted status packets, not just device announcement packets.

6.7 Album Art

To request the artwork image associated with a track, send a message with *type* 2003 containing the *artwork* ID that was specified in the track title item (as described in Section 6.3.1), like the one shown in Figure 30.

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 and T_r are the slot in which the track being asked about can be found and, its track type; each has the same values used in CDJ status packets, as shown in

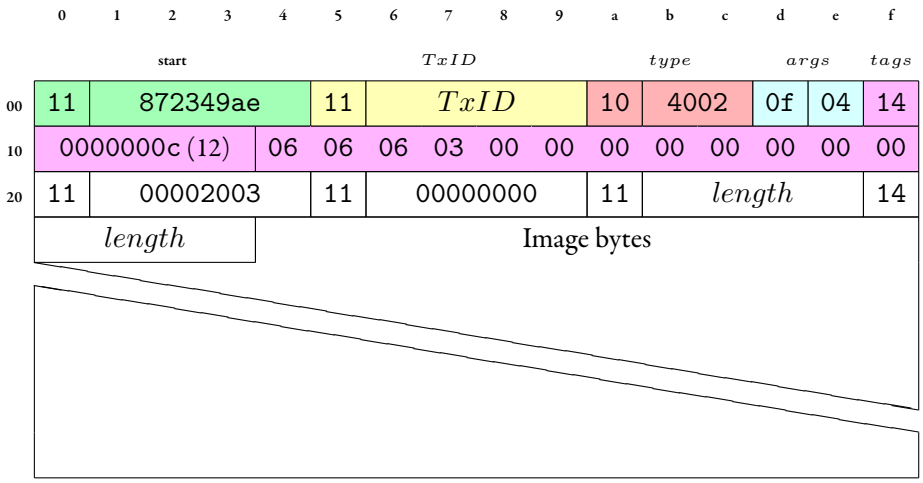

Figure 31: Track artwork response message

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* 4002, containing four arguments. The first argument echoes back our request type, which was 2003. 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 31. However, if there is no image data, this value will be 0, and the blob field will be completely omitted from the response, so you *must not* try to read it!

To experiment with this, start up dysentery in a Clojure REPL and connect to a player as described in Section 6.16, then evaluate an expression like:

```
(def art (db/request-album-art p2 3 3))
```

Replace the arguments with the `var` holding your player connection, the proper S_r number for the media slot the art is found in, and the *artwork* ID of the album art, and dysentery will open a window like Figure 32 showing the image. To load artwork for a non-rekordbox track, add an additional argument with the value 2 at the end.

6.8 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 answer to the problem. The beat grid for a track is a list of every beat which occurs in the track, along with the

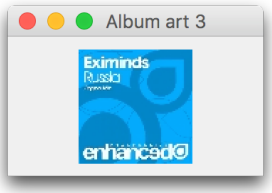

Figure 32: Example album art window

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
	start						$TxID$					$type$		$args$		$tags$
00	11	87	23	49	ae	11	$TxID$				10	22	04	0f	02	14
10	00	00	00	0c	(12)	06	06	00	00	00	00	00	00	00	00	00
20	11	D	08	S_r	01	11	$rekordbox$									

Figure 33: Track beat grid request message

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 as video resources) to sync tightly with DJ playback.

To request the beat grid of a track, send a message with $type$ 2204 containing the $rekordbox$ ID of the track, like the one shown in Figure 33.

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 beat grid 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 for which data is being loaded, is 8.

The response will be a message with $type$ 4602, containing four arguments. The first argument echoes back our request type, which was 2204. 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 34. However, if there is no beat grid available, this value will be 0, and the blob field will be completely omitted from the response, so you *must not* try to read it!

The beat grid itself is spread through the value returned as argument 4, con-

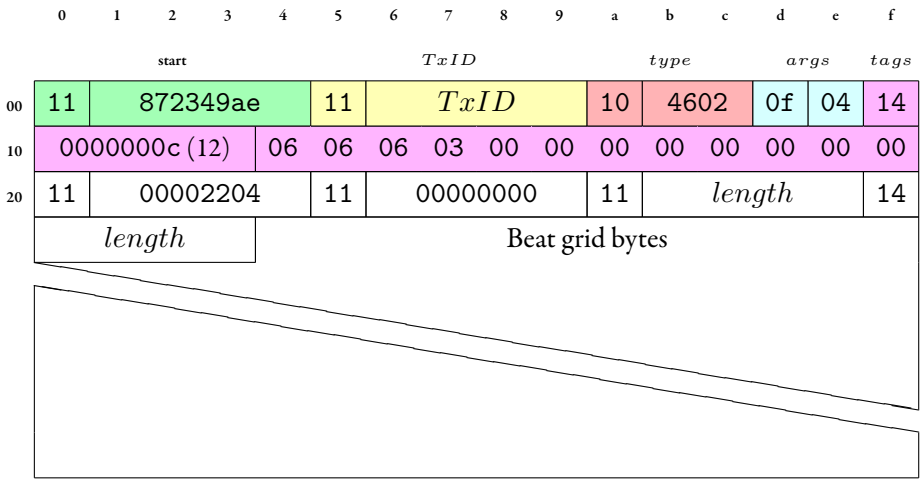

Figure 34: Track beat grid response message

sisting 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 `0x14` of argument 4, and the time at which that beat occurs, in milliseconds, is encoded as a 4-byte little-endian[13] integer at bytes `15`–`18`. Subsequent beats are found at `0x10`-byte intervals, so the second B_b value is found at byte `24`, and the second beat's time, in milliseconds from the start of the track, is the big-endian integer at bytes `25`–`28`. The B_b value for the third beat is at byte `34`, and its millisecond time is at bytes `35`–`38`, 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.

6.9 Requesting Track Waveforms

Waveform data for tracks can be requested, both the preview, which is 400 pixels long, and the detailed waveform, which uses 150 pixels per second of track content. There is also an even tinier 100 pixel preview, which is used by older players such as the pre-Nexus CDJ 900, returned as the final 100 bytes of the preview response.

We also know how to request Nxs2-style higher resolution and color waveforms, see Section 6.10 for details.

[13] Yes, unlike almost all numbers in the protocol, beat grid and cue point times are little-endian.

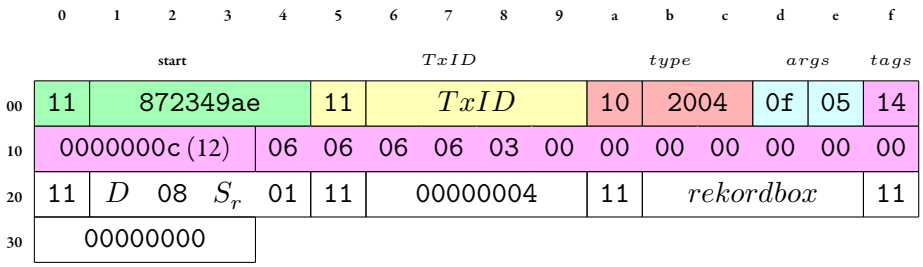

Figure 35: Waveform preview request message

To request the waveform preview of a track, send a message with *type* 2004 containing the *rekordbox* ID of the track, like the one shown in Figure 35.

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 track whose waveform preview 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 for which data is being loaded, is 8.

You may have noticed that the argument list of the message in Figure 35 specifies that there are five arguments, but in fact the message contains only the first four, numeric, arguments. The fifth, blob, argument is missing. This seems to imply the blob is empty, and this very strange feature of the protocol is, in fact, the way the track metadata is requested. The fifth argument must be specified in the message header but not actually present. When reading messages from a player, the same rules apply: There is always a numeric field right before a blob field, and it always contains a seemingly-redundant copy of the blob length, and if that numeric field has the value 0, you *must not* try to read the blob field. Instead, expect the next field or message to follow the numeric field.

The second argument has an unknown purpose, but we have seen values of 3 or 4 for it. The fourth argument is the size of the blob argument we are supposedly going to send; since we are not sending a blob, we always send a 0 here.

The response will be a message with *type* 4402, containing four arguments. The first argument echoes back our request type, which was 2004. The second

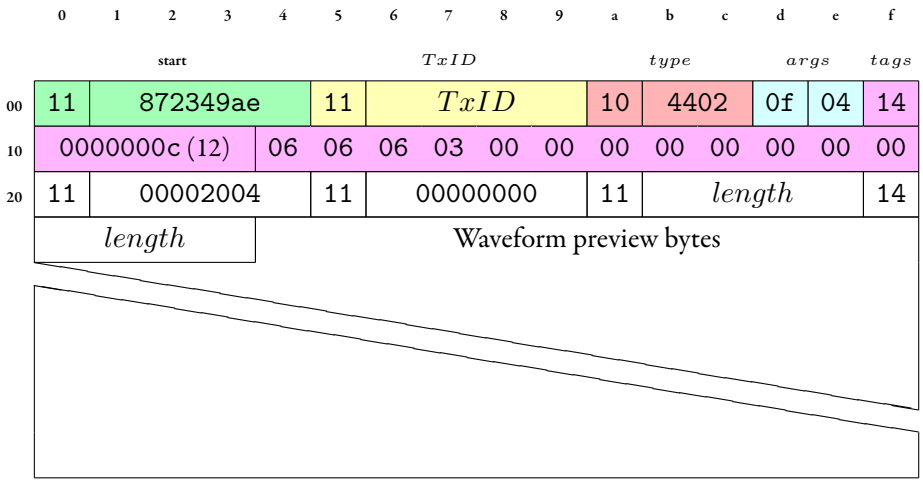

Figure 36: Waveform preview response message

always seems to be 0. The third contains the length of the waveform preview in bytes. If this value is 0, the fourth argument will be omitted from the response. When present, the fourth argument is a blob containing the actual bytes of the waveform preview, as shown in Figure 36.

For this kind of waveform preview request, there are 900 (decimal) bytes of waveform data returned. The first 800 of them contain 400 columns of waveform data, in the form of two-byte pairs, where the first byte is the pixel height of the waveform at that column (a value ranging from 0 to 31), and the second byte is the whiteness, as before, where 0 is blue and 7 is fully white. My players seem to only pay attention to the highest bit of whiteness, drawing the waveform as either very dark or light blue accordingly.

To experiment with this, start up dysentery in a Clojure REPL and connect to a player as described in Section 6.16, then evaluate an expression like:

```
(db/request-waveform-preview p2 3 1060)
```

Replace the arguments with the `var` holding your player connection, the proper S_r number for the media slot the track is found in, and the *rekordbox* ID of the track, and dysentery will open a window like Figure 37 showing the waveform preview.

The remaining hundred bytes appear to contain an even more compact 100-column preview representation of the waveform which is shown on pre-Nexus CDJ 900 players as shown in Figure 38. Our best guess is that for each byte, the four low-order bits encode the height of the waveform in that column, and the high-order bits may encode saturation again?

Requesting the detailed waveform is very similar to requesting the preview, but the request type and arguments are slightly different. To request the detailed waveform of a track, send a message with *type* 2904 containing the *rekordbox* ID of

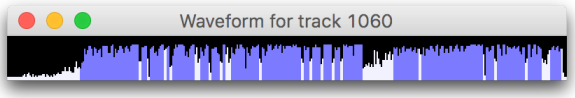

Figure 37: Example waveform preview window

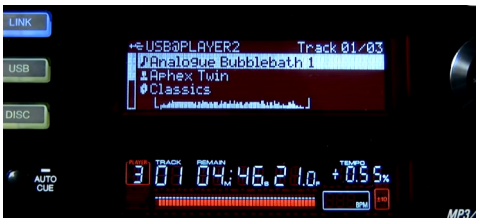

Figure 38: CDJ 900 waveform preview

the track, like the one shown in Figure 39.

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 track whose waveform preview you are requesting, as found in a CDJ status packet or playlist response. Since this is a graphical request, I would expect the value after *D*, which identifies the location of the menu for which data is being loaded, to be 8 like it is for others, but for some reason it is 1, which usually means the main menu... maybe because the scrolling waveform appears in the same location on the display as the main menu? In many ways this protocol is a mystery wrapped in an enigma.

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
	start					*TxID*					type			args		tags
00	11	87	23	49	ae	11	*TxID*				10	29	04	0f	03	14
10	0000000c (12)					06	06	06	00	00	00	00	00	00	00	00	00
20	11	*D*	01	S_r	01	11	*rekordbox*				11	00000000				

Figure 39: Waveform detail request message

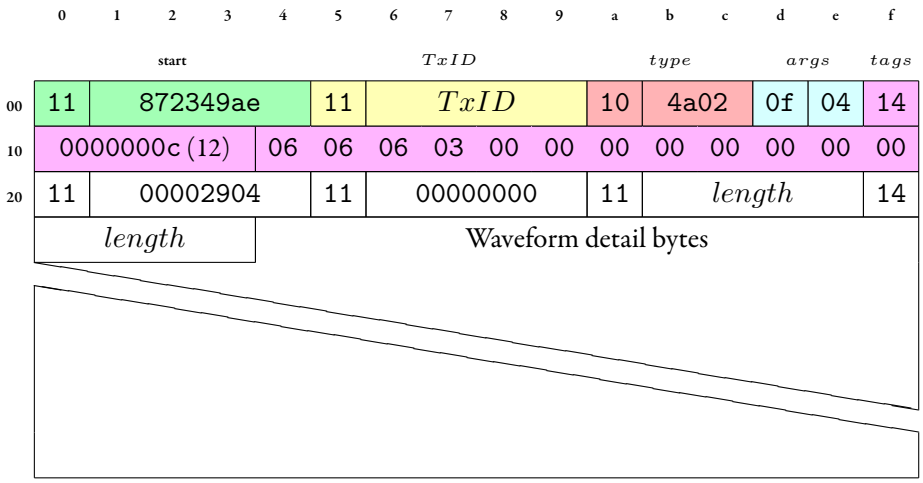

Figure 40: Waveform detail response message

The waveform detail response is essentially identical to the waveform preview response, with just the type numbers changed. It will be a message with *type* `4a02`, containing four arguments. The first argument echoes back our request type, which was `2904`. The second always seems to be 0. The third contains the length of the waveform detail in bytes. If this value is 0, the fourth argument will be omitted from the response. When present, the fourth argument is a blob containing the actual bytes of the waveform detail, as shown in Figure 40.

The content of the waveform detail is simpler and more compact than the waveform preview. Every byte reperesents one segment of the waveform, and there are 150 segments per second of audio. (These seem to correspond to "half frames" counted as `03.5f` following the seconds in the player display.) Each byte encodes both a color and height. The three high-order bits encode the color, ranging from darkest blue at 0 to near-white at 7. The five low-order bits encode the height of the waveform at that point, from 0 to 31 pixels.

6.10 Requesting Nxs2 Track Waveforms

Thanks to some wonderful analysis[14] by @jan2000[15] we now know how to request and interpret the higher resolution and color waveforms supported by the nxs2 series of players.

Both the enhanced waveform preview and enhanced detail are requested using variations on the same request type, which asks for specific content from the

[14] `https://github.com/Deep-Symmetry/dysentery/issues/9`
[15] `https://github.com/jan2000`

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
	start					$TxID$					$type$			$args$		$tags$
00	11	87	23	49	ae	11	$TxID$				10	2c	04	0f	04	14
10	00	00	00	0c (12)		06	06	06	06	00	00	00	00	00	00	00
20	11	D	08	S_r	01	11	$rekordbox$				11	34	56	57	50	11
30	00	54	58	45												

Figure 41: Nxs2 waveform preview request message

`ANLZ0000.EXT` file created by rekordbox. See the Crate Digger project[16], which has its own analysis document, if you would like to learn more about the structure and content of these files.

To request the enhanced waveform preview of a track, send a message with *type* `2c04` containing the *rekordbox* ID of the track, the tag type identifier `PWV4` and the file extension identifier `EXT` encoded as four-byte numbers, holding the ASCII in big-endian (backwards) order, as shown in Figure 41.

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. For some reason the value after D, which identifies the location of the menu for which data is being loaded, is 1 in this case. *rekordbox* identifies the specific track whose analysis you are requesting, as found in a CDJ status packet or playlist response, and the final two numeric arguments specify that you are interested in the `PWV4` tag (which holds the enhanced waveform preview) from the track's `EXT` extended analysis file.

The response will be a message with *type* `4f02`, containing five arguments. The first argument echoes back our request type, which was `2c04`. The second always seems to be 0. The third contains the length of the requested tag (holding the enhanced waveform preview) in bytes. If this value is 0, the fourth argument will be omitted from the response. When present, the fourth argument is a blob containing the actual bytes of the enhanced waveform preview, as shown in Figure 42. The fifth argument has an unknown purpose and so far seems to always have the value 0.

The extended preview data begins at byte `34` and is 7,200 (decimal) bytes long, representing 1,200 columns of waveform preview information.

The color waveform preview entries are the most complex of any of the waveform tags. See the discussion on Github[17] for how the analysis was performed.

[16] `https://github.com/Deep-Symmetry/crate-digger`

[17] `https://github.com/Deep-Symmetry/dysentery/issues/9`

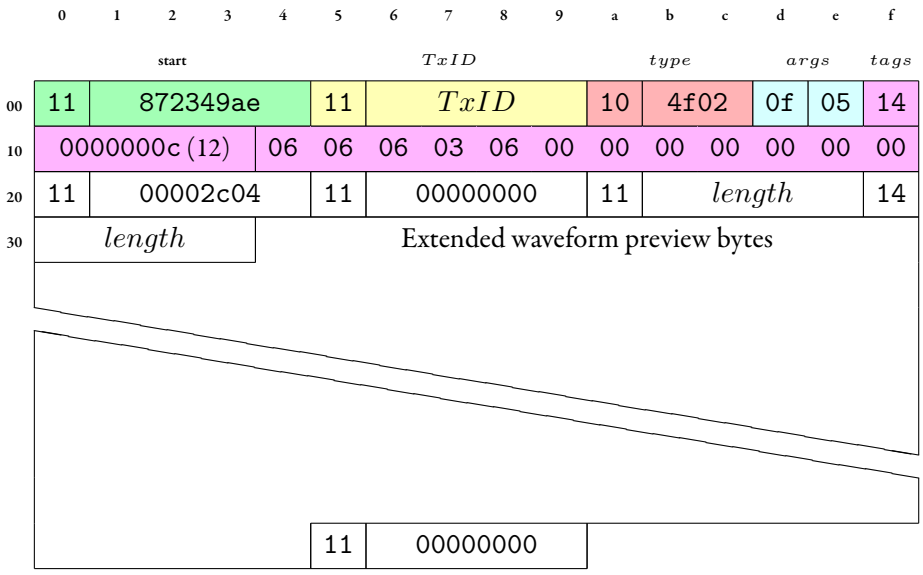

Figure 42: Nxs2 waveform preview response message

`@jan2000` created an audio file containing a 10 second sine wave sweep from 20 Hz to 20 kHz, and analyzed that in rekordbox. The results are represented in Figure 43.

As a summary, the top six stripes plot the values of each six channels of waveform preview information. The first byte of data is the first column of the top stripe, the next byte is the first column of the second stripe, and so on, until we reach the seventh byte, which is the second column of the first stripe.

We are not sure what the top two stripes represent, but they do seem to have an effect on the blue version of the waveform preview, so they somehow encode "whiteness". The next stripe, corresponding to byte `2` of each column, indicates how much sound energy is present in the bottom half of the frequency range (it drops around 10 KHz). The stripe corresponding to byte `3` reflects how much sound energy is present in the bottom third of the frequency range, byte `4` reflects how much sound energy is in the middle of the frequency range, and byte `5` tracks the sound energy in the top of the frequency range.

The stripe labeled "color" reflect's `@jan2000`'s algorithm for combining bytes 3, 4, and 5 into a color preview, and the bottom stripe is his approach for deriving the blue preview from that and the other two stripes.

The calculations used by Beat Link to build its own color previews can be found in the `segmentColor` and `segmentHeight` methods of the `WaveformPreview` class[18], and the way they are used to draw the actual graphical representation can be

[18] `https://deepsymmetry.org/beatlink/apidocs/org/deepsymmetry/beatlink/`

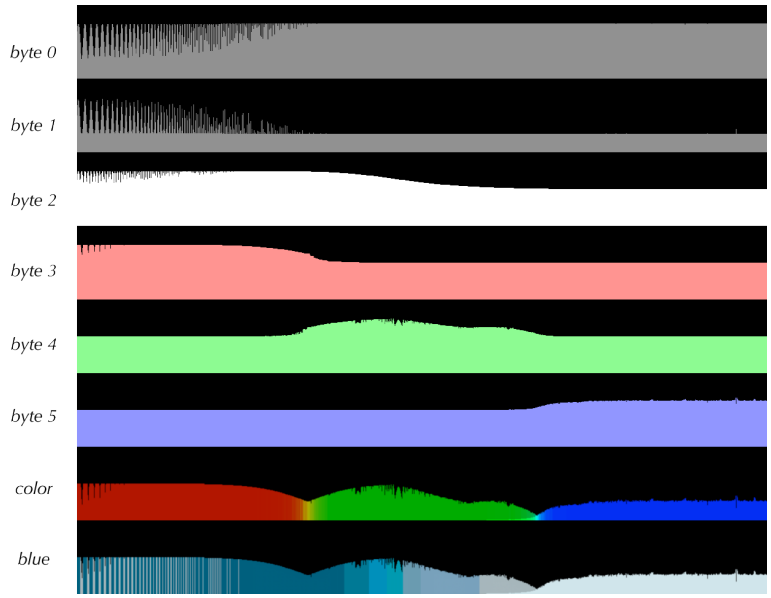

Figure 43: Sine sweep analysis

found in the `updateWaveform` method of the `WaveformPreviewComponent` class[19]. These produce attractive results, but it is certainly possible that refinements can be found in the future.

As mentioned, requesting the detailed color waveform is a simple variant of the request used to obtain the preview. The same request type is used, and the only difference is that the tag type requested to obtain the scrollable detail view is `PWV5`. The full request is shown in Figure 44.

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. For some reason the value after D, which identifies the location of the menu for which data is being loaded, is 1 in this case. $rekordbox$ identifies the specific track whose analysis you are requesting, as found in a CDJ status packet or playlist response, and the final two numeric arguments specify that you are interested in the `PWV5` tag (which holds the enhanced waveform detail) from the track's `EXT` extended analysis file.

`data/WaveformPreview.html`

[19] `https://deepsymmetry.org/beatlink/apidocs/org/deepsymmetry/beatlink/data/WaveformPreviewComponent.html`

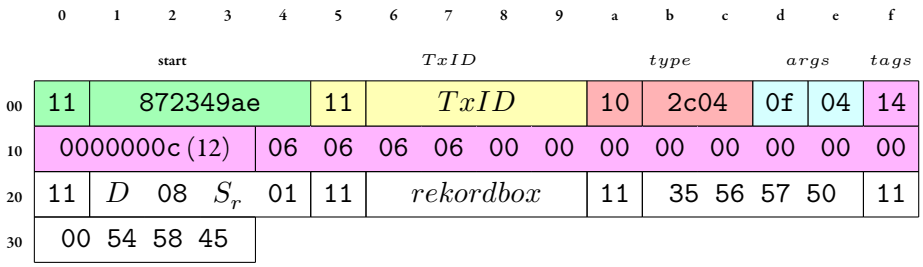

Figure 44: Nxs2 waveform detail request message

The response will be a message with *type* `4f02`, containing five arguments. The first argument echoes back our request type, which was `2c04`. The second always seems to be 0. The third contains the length of the requested tag (holding the enhanced waveform detail) in bytes. If this value is 0, the fourth argument will be omitted from the response. When present, the fourth argument is a blob containing the actual bytes of the enhanced waveform segments, as shown in Figure 45. The fifth argument has an unknown purpose and so far seems to always have the value 0.

The extended waveform detail data begins at byte `34` and has a variable length, but a vastly simpler structure than the waveform preview. Every pair of bytes represents one segment of the waveform, and there are 150 segments per second of audio. (These seem to correspond to "half frames" counted as `03.5f` following the seconds in the player display.) Each pair of bytes encodes the height of the waveform at that segment as a five bit value, along with three bits each of red, green, and blue intensity, arranged as shown in Figure 46.

6.11 Requesting Cue Points and Loops

This section discusses how to obtain cue points and loops which are compatible with original nexus players. See Section 6.12 for how you can obtain a newer format which includes hot cue colors, DJ-assigned comment text, and hot cues beyond C.

The locations of the cue points and loops stored in a track can be obtained with a request like the one shown in Figure 47.

As always, $TxID$ 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, and as usual, *rekordbox* is the database

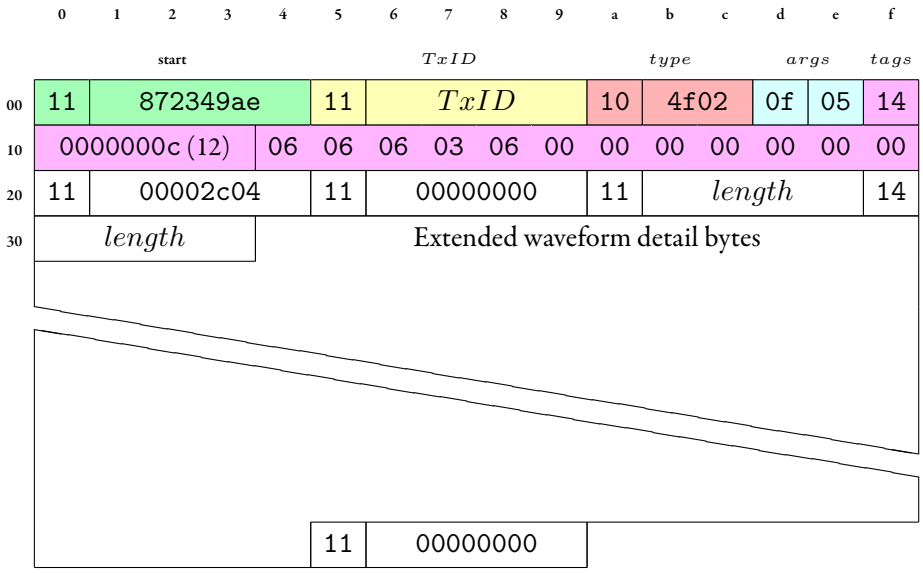

Figure 45: Nxs2 waveform detail response message

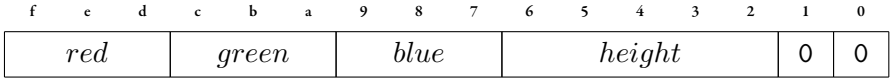

Figure 46: Nxs2 waveform detail segment bits

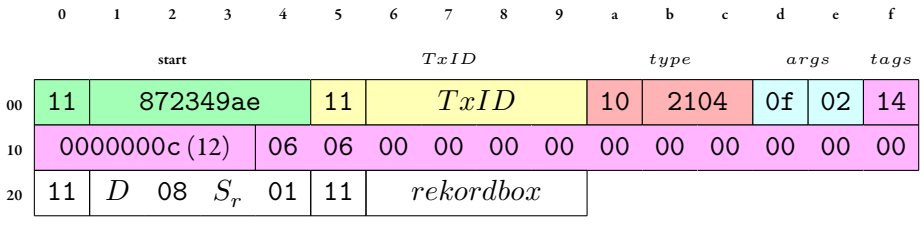

Figure 47: Cue point request message

ID of the track you're interested in.

The response will be a message with *type* 4702, containing nine arguments. The first argument echoes back our request type, which was 2104. The second always seems to be 0. The third contains the length of the blob containing cue and loop points 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 cue and loop points, as shown in Figure 48. However, if there are no cue or loop points, this value will be 0, and the following blob field will be completely omitted from the response, so you ***must not*** try to read it!

The fifth argument is a number with uncertain purpose. It always seems to have the value 0x24, which may be telling us the size of each cue/loop point entry in argument 4 (they do seem to each take up 24 bytes, as shown in Figure 49). The sixth argument, shown as num_{hot}, seems to contain the number of hot cue entries found in argument 4, and the seventh, num_{cue} seems to contain the number of ordinary memory point cues. The eighth argument is a number containing the length of the second binary field which follows it. We don't know the meaning of the final, binary, argument.

As described above, the first binary field in the cue point response is divided up in to 24-byte entries, each of which potentially holds a cue or loop point. They are not in any particular order, with respect to the time at which they occur in the track. They each have the structure shown in Figure 49.

The first byte, F_l, has the value 01 if this entry specifies a loop, or 00 otherwise. The second byte, F_c, has the value 01 if this entry contains a cue point, and 00 otherwise. Entries with loops have the value 01 in both of these bytes, because loops also act as cue points. If both values are 00, the entry is ignored (it is probably a leftover cue point that was deleted by the DJ). The third byte, labeled H, is 00 for ordinary cue points, but has a value if this entry defines a hot cue. Hot cues A through C are represented by the values 01, 02, and 03.

The actual location of the cue and loop points are in the values cue and $loop$. These are both 4-byte integers, and like beat grid positions, but unlike essentially every other number in the protocol, they are sent in little-endian byte order. For non-looping cue points, only cue has a meaning, and it identifies the position of the cue point in the track, in $\frac{1}{150}$ second units. For loops, cue identifies the start of the loop, and $loop$ identifies the end of the loop, again in $\frac{1}{150}$ second units.

6.12 Requesting Nxs2 Cue Points and Loops

For tracks that have been exported since the introduction of the nxs2 series of players, rekordbox includes a richer set of information about cue points and loops. In addition to the information described in Section 6.11, you can learn about any custom colors a DJ has assigned to their hot cues, as well as optional text comments describing hot cues, mamory points, and loops. And while older players only supported hot cues A through C, this new format returns more.

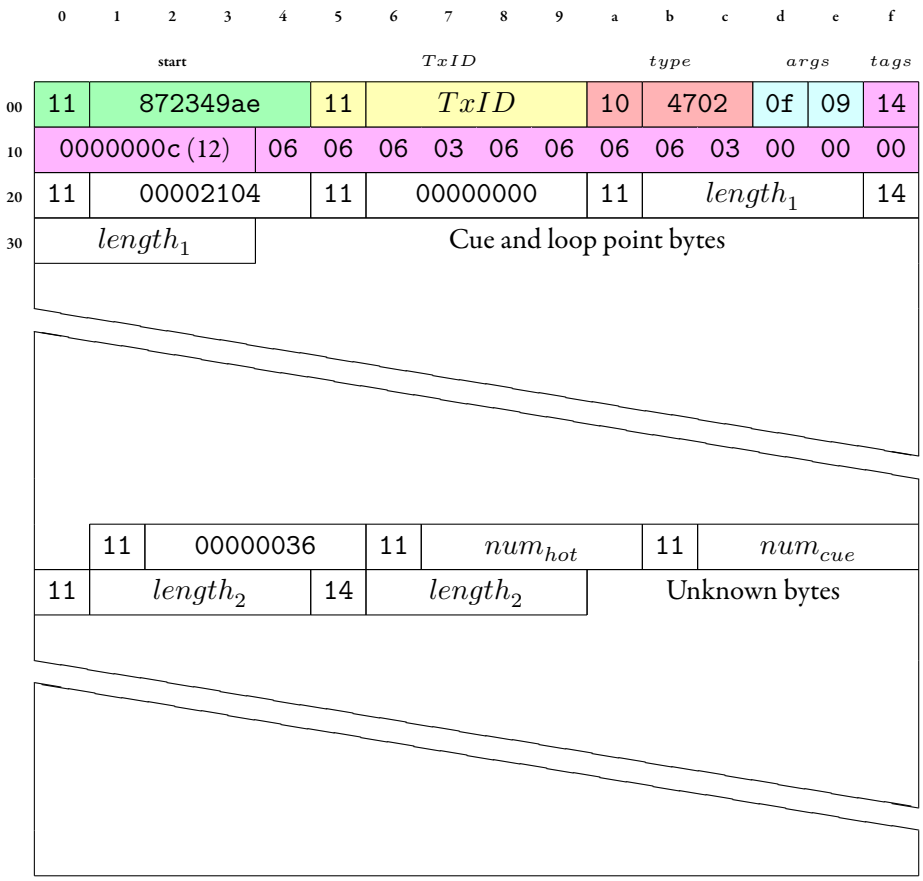

Figure 48: Cue point response message

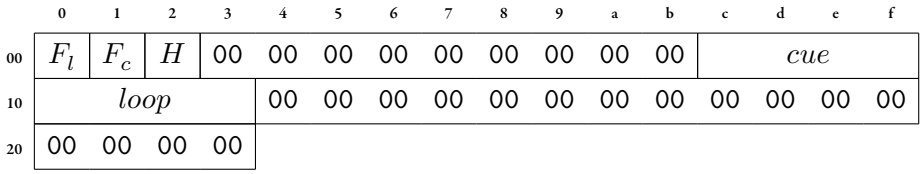

Figure 49: Cue/loop point entry

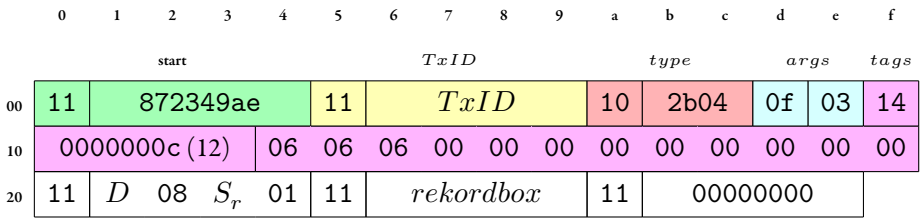

Figure 50: Extended cue point request message

In order to work well in mixed-player environments, the firmware of even older players has been updated to return this information if it is found in the exported data, so it is worth trying to ask for it. If the query described in this section fails, then you can fall back on the one shown in Section 6.11.

Enhanced information about the cue points and loops stored in a track can be obtained with a request like the one shown in Figure 50.[20]

As always, $TxID$ 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, and as usual, $rekordbox$ is the database ID of the track you're interested in.

The response will be a message with $type$ `4e02`, containing five arguments. The first argument echoes back our request type, which was `2b04`. The second always seems to be 0. The third contains the length of the blob containing cue and loop points 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 cue and loop points, as shown in Figure 51. However, if there are no cue or loop points, this value will be 0, and the following blob field will be completely omitted from the response, so you *must not* try to read it!

The fifth argument reports the number of cue point entries found in the blob. Because each extended cue entry can include a comment string, they have variable lengths, so each entry needs to be examined in order to figure out where it ends, and therefore the next one begins. The extended entry structure is shown in Figure 52.

The first four bytes, $length$, hold the length of the current entry, and so can be used to find the start of the next entry. Like other numbers in cue point responses (and unlike most numbers in the protocol), this is sent in little-endian byte order. The fourth byte, labeled H, is `00` for ordinary cue points, but has a value if this entry defines a hot cue. Hot cues A through H are represented by the values `01` through `08`.

[20] Thanks to Matt Positive, `https://soundcloud.com/mattpositive`, for packet captures!

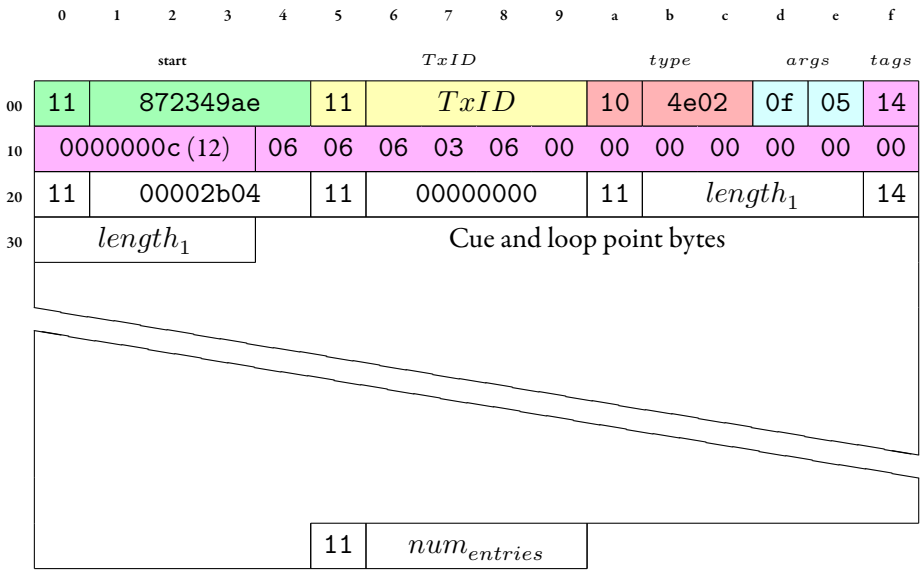

Figure 51: Extended cue point response message

F_l, at byte 06 has the value 01 if this entry specifies a memory point and 02 if it is a loop. If both H and F_l are 00, the entry should be ignored (it is probably a leftover cue point that was deleted by the DJ).

The actual location of the cue and loop points are in the values *cue* and *loop*. These are both 4-byte integers, and as noted above, they are sent in a little-endian byte order. For non-looping cue points, only *cue* has a meaning, and it identifies the position of the cue point in the track, in $\frac{1}{150}$ second units. For loops, *cue* identifies the start of the loop, and *loop* identifies the end of the loop, again in $\frac{1}{150}$ second units.

len_c at byte 48 is a two-byte, little-endian integer that contains the length of the comment. If it is zero, there is no comment. Otherwise, *comment* will follow len_c, taking up that many bytes, as a string (in UTF-16 encoding with a trailing 0000 character). So if len_c isn't zero, it will be an even number with minimum value four, representing a comment that is one character long.

Immediately after *comment* (in other words, starting $len_c + 4a$ past the start of the entry) there are four one-byte values containing color information. c appears to be a code identifying the color with which rekordbox displays the cue, by looking it up in a table. There have been sixteen codes identified, and their corresponding RGB colors can be found by looking at the `findRecordboxColor` static method in the Beat Link library's `CueList` class.[21] The next three bytes, r, g, and b, make up an RGB color specification which is similar, but not identical, to the color that

[21] `https://deepsymmetry.org/beatlink/apidocs/`

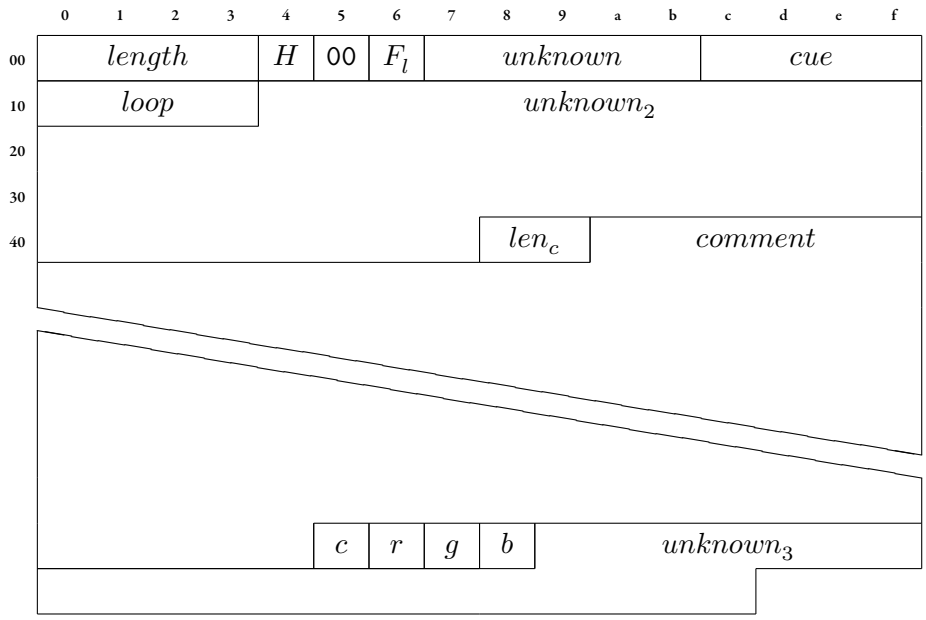

Figure 52: Extended cue/loop point entry

rekordbox displays. We believe these are the values used to illuminate the RGB LEDs in a player that has loaded the cue. When no color is associated with the cue, all four of these bytes have the value 00.

We do not know what, if anything, is sent in the remaining bytes of the the entry.

6.13 Requesting All Tracks

If you want to cache all the metadata associated with a media stick, this query is a good starting point. Send a packet like the one shown in Figure 53.

As always, $TxID$ 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 T_r is the type of the track; these two bytes have the same values used in CDJ status packets, as shown in Figure 11. The new $sort$ parameter determines the order in which the tracks are sorted, and that also affects the item type returned, along with the secondary information (beyond the title) that it contains about the track, as described in Section 6.13.1. We will start out assuming the tracks are being requested in title order, which can be done by sending a $sort$ argument value of 0 or 1, and that the DJ has configured the media device to show artists as the second column.

Track list requests (just like metadata requests) are built on the mechanism that

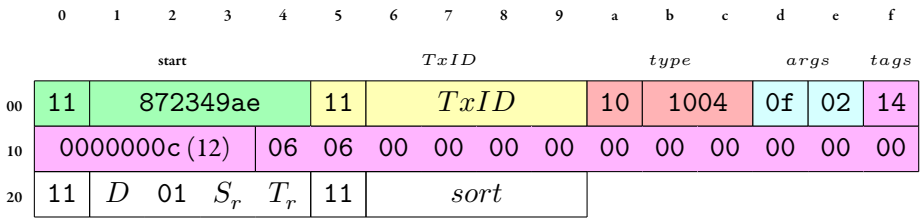

Figure 53: Full track list request message

is used to request and draw scrollable menus on the CDJs, explored in more breadth in Section 7. The player responds with a success indicator, saying it is ready to send these "menu items" and reporting how many of them are available, much like shown in Figure 24, although the first argument will be 1004 to reflect the message type we just sent, rather than 2002 as it was for the metadata request.

As with metadata, the next step is to send a "render menu" request like that in Figure 25 to get the actual results. But the number of results available is likely to be much higher than shown in Figure 24, because we have asked about all tracks in the media slot. That means we will probably need more than one "render menu" request to get them all.

I don't know how many items you can safely ask for at one time. I have had success with values as high as 64 on my CDJ-2000 nexus players, but they failed when asking for numbers in the thousands. So to be safe, you should ask for the results in chunks of 64 or smaller, by setting $limit$ and $limit2$ to the smaller of 64 and the remaining number of results you want, and incrementing $offset$ by 64 in each request until you have retrieved them all.

As with metadata requests, you will get back two more messages than you ask for, because you first get a menu header message (with $type$ 4001, Figure 27), then the requested menu items are sent (with $type$ 4101, Figure 28), and finally these are followed by a menu footer message (with $type$ 4201, Figure 29). 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 details of the menu items are slightly different than in the case of a metadata request. In the example where you are retrieving tracks in the default order, with the second column configured to be artists, they will have the content shown in Table 6.

Arg	Type	Meaning
1	number	artist id
2	number	$rekordbox$ id of track
3	number	length in bytes of Label 1
4	string	Label 1, Track Title

Table 6: Track List Entries with Artists

Arg	Type	Meaning
5	number	length in bytes of Label 2
6	string	Label 2, Artist Name
7	number	type of this item, 0704 for Title and Artist
8	number	some sort of flags field, details still unclear
9	number	unknown
10	number	unknown
11	number	unknown
12	number	unknown

Table 6: Track List Entries with Artists

6.13.1 Alternate Track List Sort Orders

As noted above, you can request the track list in a diferent order by supplying a different value for the *sort* parameter. The value 0 or 1 gives the order just described, with the default second column information configured for the media. The *sort* values discovered so far are shown in the Table 7, and return menu items with the specified item type values in argument 7.

Sort	Type	Description
01	0704	Title and Artist sorted by title
02	0704	Title and Artist sorted by artist
03	0204	Sorted by album, Arg 1 holds album ID, Label 2 holds album name
04	0d04	Sorted by BPM, Arg 1 holds BPM×100, Label 2 empty
05	0a04	Sorted by rating, Arg 1 holds rating, Label 2 empty
06	0604	Sorted by genre, Arg 1 holds genre ID, Label 2 holds genre name
07	2304	Sorted by comment, Arg 1 holds comment ID, Label 2 holds comment
08	0b04	Sorted by time, Arg 1 holds track length in seconds, Label 2 empty
09	2904	Sorted by remixer, Arg 1 holds remixer ID, Label 2 holds remixer
0a	0e04	Sorted by label, Arg 1 holds label ID, Label 2 holds label
0b	2804	Sorted by original artist, Arg 1 holds original artist ID, Label 2 holds original artist
0c	0f04	Sorted by key, Arg 1 holds key ID, Label 2 holds key text
0d	1004	Sorted by bit rate, Arg 1 holds bit rate, Label 2 empty
10	2a04	Sorted by DJ play count, Arg 1 holds play count, Label 2 empty
11	2e04	Sorted by date added, Arg 1 holds date ID, Label 2 holds date text

Table 7: Sort Orders

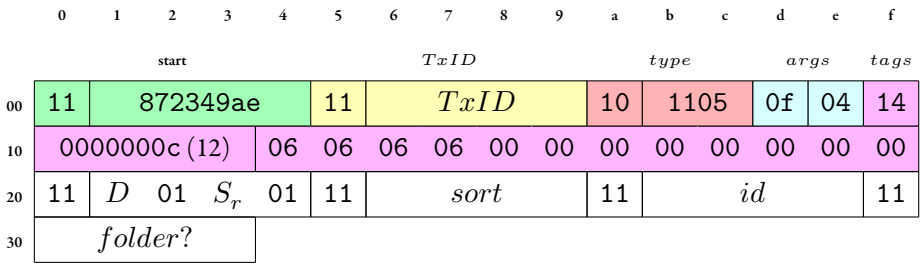

Figure 54: Playlist request message

To experiment with this, start up dysentery in a Clojure REPL and connect to a player as described in Section 6.16, then evaluate an expression like:

```
(db/request-track-list p2 3)
```

Replace the arguments with the `var` holding your player connection and the proper S_r number for the media slot containing the tracks you want to list. You can also add a third argument to specify a sort order, like this to sort all the tracks in the USB slot by BPM:

```
(db/request-track-list p2 3 4)
```

6.14 Playlists

If you want to be more selective about the metadata that you are caching, you can navigate the playlist folder hierarchy and deal with only specific playlists. This process is essentially the same as asking for all tracks, except that in the playlist request you specify the playlist or folder that you want to list. To start at the root of the playlist folder hierarchy, you request folder 0. A playlist request has the struture shown in Figure 54.

As always, $TxID$ 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. You specify the ID of the playlist or folder you want to list in the id argument, and set $folder?$ to 1 if you are asking for a folder, and 0 if you are asking for a playlist. As noted above, to get the top-level list of playlists, ask for folder 0, by passing an id of 0 and passing $folder?$ as 1.

Much as when listing all tracks, the response may tell you there are more entries in the playlist than you can retreive in a single request, so you should follow the procedure outlined in Section 6.13 to request your results in smaller batches. The followup queries that you send are identical for playlists as they are described in that section. The actual menu items returned when you are asking for a folder have the content shown in Table 8.

Arg	Type	Meaning
1	number	parent folder id
2	number	id of playlist or folder
3	number	length in bytes of Label 1
4	string	Label 1, Name of playlist or folder
5	number	length in bytes of Label 2
6	string	Label 2, empty
7	number	type of this item, `01` for folder, `08` for playlist
8	number	unknown
9	number	unknown
10	number	playlist position
11	number	unknown
12	number	unknown

Table 8: Folder List Entries

When you have requested a playlist (by passing its id and a value of 0 for $folder?$) the responses you get are track list entries, just like when you request all tracks as shown in Section 6.13. And just like in that section, you can get the results in a different order by specifying a value for $sort$. The supported values and corresponding item types returned seem to be the same as described there. Additionally, if you pass a $sort$ value of `09`, the playlist entries will come back sorted by track title, Label 2 will be empty, and the item type will be `2904`.

To experiment with this, start up dysentery in a Clojure REPL and connect to a player as described in Section 6.16, then evaluate an expression like:

```
(db/request-playlist p2 3 1)
```

Replace the arguments with the `var` holding your player connection, the proper S_r number for the media slot containing the playlist you want to list, and the playlist ID. You can also add a third argument to specify that you want to list a folder, like this using folder ID 0 to request the root folder:

```
(db/request-playlist p2 3 0 true)
```

Finally, you can add a fourth argument to specify a sort order, like this to sort all the tracks in playlist 12 by genre:

```
(db/request-playlist p2 3 12 false 6)
```

6.15 Disconnecting

If you want to be polite about the fact that you are done talking to the dbserver, you can send it a message like the one shown in Figure 55. This will cause the player to disconnect from its side.

6.16 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

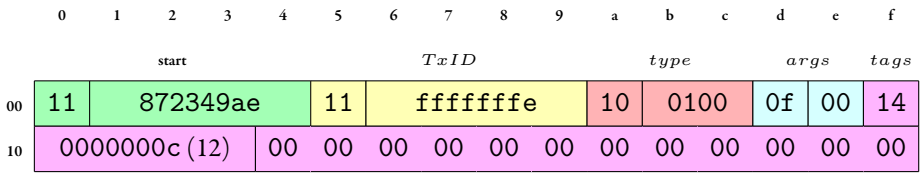

Figure 55: Connection Teardown Message

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. Zoë 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.

7 Menu Requests

We have already seen many examples of the menu mechanism in action, most clearly with metadata requests (Section 6), track list requests (Section 6.13), and playlist

requests (Section 6.14). We'll round out what is known about the other types of menu request here.

The overall flow starts off by sending a menu request message whose type identifies the kind of menu desired (in the message *type* field), along with some parameters that control the specific content to be shown, and perhaps establishing a sort order. See, for example, Figure 54. If all goes well, the player responds with a packet with *type* `4000` like the one in Figure 24, containing two numeric fields that contain the original menu *type* value you requested, followed by the number of menu entries that are available for you to load.

To actually obtain those menu entries, you send one or more "menu render" request messages with *type* `3000` as shown in Figure 25 and described below it, allowing you to paginate through the results. This gets you a number of responses: a menu header message (with *type* `4001`, Figure 27), followed by the number of menu item messages you requested (with *type* `4101`, Figure 28), and finally a menu footer message (with *type* `4201`, Figure 29). The menu item types we have identified so far are listed in Section 6.5.

7.1 Known Menu Request Types

Table 9 shows the menu requests we have figured out so far. Not all menus are available in all rekordbox databaes; the DJ can decide what indices and categories to include, and that will determine which of these requests succeed. To find out what is available, you can request the root menu, which gives you access to all available menus. That is what a player will do when you use the Link button to connect to media mounted on another player. The menus available to you will be returned as entries in the root menu response, with Item Type values in the range `80`–`95`, as shown in Table 5.

The first argument to every menu request is a four-byte number where each byte means something different. This byte is referred to as *r:m:s:t* in the Beat Link code because of the function of its component bytes:

The first byte is always the device number D of the player making the *request*.

The second byte of the first argument identifies which *menu location* on the player will be used to display the result (for example, the screen is sometimes split with the user scrolling down the left, which is menu location 1, while displaying the contents of the selected item on the right, which is menu location 2), and the response format can be different in these cases. When showing metadata preview for a selected track, the menu "location" value is 3, and when loading non-text data like waveforms, album art or beat grid, the value of this byte is 8.

The third byte of this argument identifies the media *slot*, S_r, that information is being requested from. The values are as described in the discussion of Figure 11.

And the final byte identifies the media *type*, T_r being asked about, with values also describe in the discussion of Figure 11.

To save space in Table 9, this always-present argument will be simply shown as *rmst*.

Type	Meaning	Arguments
1000	Root Menu	*rmst*, sort, 00ffffff
1001	Genre Menu	*rmst*, sort
1002	Artist Menu	*rmst*, sort
1003	Album Menu	*rmst*, sort
1004	Track Menu[22]	*rmst*, sort
1006	BPM Menu	*rmst*, sort
1007	Rating Menu	*rmst*, sort
1008	Year Menu	*rmst*, sort
100a	Label Menu	*rmst*, sort
100d	Color Menu	*rmst*, sort
1010	Time Menu	*rmst*, sort
1011	Bitrate Menu	*rmst*, sort
1012	History Menu	*rmst*, sort?
1013	Filename Menu	*rmst*, sort?
1014	Key Menu	*rmst*, sort?
1302	Original Artist Menu	*rmst*, sort
1602	Remixer Menu	*rmst*, sort
1101	Artists for Genre	*rmst*, sort, genre id
1102	Albums for Artist	*rmst*, sort, artist id
1103	Tracks for Album	*rmst*, sort, genre id
1107	Tracks for Rating	*rmst*, sort, rating id
1108	Years for Decade	*rmst*, sort, decade
110a	Artist for Label	*rmst*, sort, label id
110d	Tracks for Color	*rmst*, sort, color id
1110	Tracks for Time	*rmst*, sort, time
1112	Tracks for History	*rmst*, sort, history id
1114	Distances for Key	*rmst*, sort, key id
1402	Albums for Orig. Artist	*rmst*, sort, artist id
1702	Albums for Remixer	*rmst*, sort, artist id
1201	Albums for Genre and Artist	*rmst*, sort, genre id, artist id*
1202	Tracks for Artist and Album	*rmst*, sort, artist id, album id*
1206	Tracks for BPM +/- %	*rmst*, sort, bpm id, distance (+/- %; can be 0–6)
1208	Tracks for Decade and Year	*rmst*, sort, decade, year*
120a	Albums for Label and Artist	*rmst*, sort, label id, artist id*
1214	Tracks near Key	*rmst*, sort, key id, distance
1502	Tracks for Original Artist and Album	*rmst*, sort, artist id, album id*
1802	Tracks for Remixer and Album	*rmst*, sort, artist id, album id*

Table 9: Menu Request Types

[22] See Figure 53.

Type	Meaning	Arguments *Use -1 for "all".
1301	Tracks for Genre, Artist, and Album	*rmst*, sort, genre id, artist id,* album id*
130a	Tracks for Label, Artist, and Album	*rmst*, sort, label id, artist id,* album id* *Use -1 for "all".
1105	Playlist Menu[23]	*rmst*, sort, playlist or folder id, type (0:playlist; 1:folder)
1300	Search by substring	*rmst*, sort, search string byte size, search string (uppercase), unknown (0)
2006	Folder Menu	*rmst*, sort?, folder id, 0?

Table 9: Menu Request Types

7.2 Search

As noted in Table 9 there is a search "menu". This is how the text-search feature of CDJs with touch-strips is implemented. By passing an uppercase string argument (in UTF-16 with a trailing 0000 character), preceded by its byte length, you can obtain a list of all matching database entries (tracks, albums, artists, etc.)

8 Fader Start

Thanks to @ErikMinekus[24] we know that we can cause players to start or stop playing by sending a packet like the one shown in Figure 56 to port 50001 of the players, with appropriate command values for C_1 through C_4 telling that player what to do. A command value of 00 tells the corresponding player to start playing if it isn't already. The command 01 tells that player to stop playing and return to the cue point, and the value 02 tells the player to stay in its current state. (It also seems to work to broadcast the packet on port 50001, which makes sense, since it can be interpreted individually by each player, so a single packet can be used to affect the states of all four players if desired.)

Since this packet uses subtype 00, the length sent in len_r has the value 0004, reflecting the four bytes which follow it.

[23] See Section 6.14.
[24] https://github.com/ErikMinekus

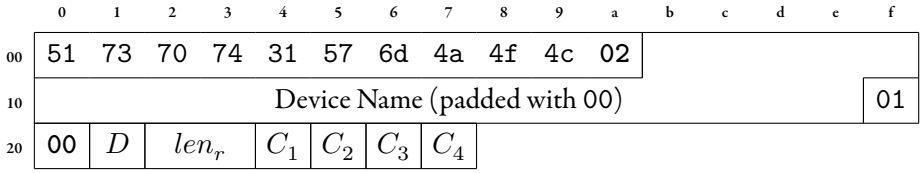

Figure 56: Fader start packet

9 Channels On Air

Thanks to @jan2000[25] we know how the mixer reports which channels are currently on-air, and we can simulate this feature ourselves when there is no DJM on the network. (If there is a DJM present, it will quickly reassert its own on-air state for all the channels.)

The mixer broadcasts a packet like the one shown in Figure 57 to port 50001, with appropriate flag values for F_1 through F_4 telling each player whether its channel is on-air. A flag value of 00 tells the corresponding player it is off the air (silenced, either due to the cross fader, channel fader, or input source switch for that channel), while 01 means the player's channel is on the air.

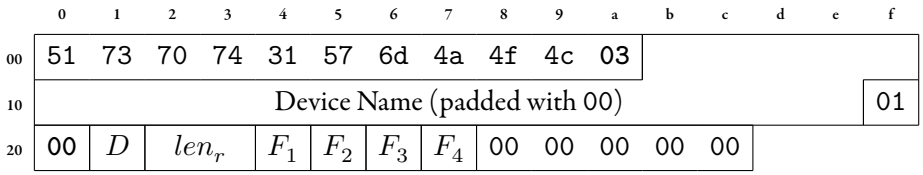

Figure 57: On Air flags packet

Since this packet uses subtype 00, the length sent in len_r has the value 0009, reflecting the nine bytes which follow it.

10 Loading Tracks

When running rekordbox, you can tell a player to load a track from the collection by dragging the track onto the player icon. This is implemented by a command that tells the player to load the track, and that command can be used to cause any player to load a track which is available somewhere on the network (whether in the rekordbox collection, or in a media slot on another player).

To do that, send a packet like the one shown in Figure 58 to port 50002 on the player that you want to cause to load the track, with appropriate values for D (the device number you are posing as), D_r (the device from which the track should

[25] https://github.com/jan2000

be loaded), S_r (the slot from which the track should be loaded), T_r (the type of the track), and $rekordbox$ (the track ID). These are the same values used in CDJ status packets, as shown in Figure 11

	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	19					
10	Device Name (padded with 00)															01
20	00	D	len_r		D	00	00	00	D_r	S_r	T_r	00	$rekordbox$			
30	00	00	00	32	00	00	00	00	00	00	00	00	00	00	00	00
40	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
50	00	00	00	00	00	00	00	00								

Figure 58: Load Track command packet

Since this packet uses subtype 00, the length sent in len_r has the value 0034, reflecting the number of bytes which follow it.

Assuming the track can be loaded, the player will respond with a packet whose type indicator (at byte 0a) has the value 1a to acknowledge the command, and will load the specified track.

11 Media Slot Queries

In order to correctly ask for the root menu for a media slot, you need to know what kind of media (rekordbox or unanalyzed) is present in the slot so you can send the proper T_r value in your menu requests. You might also want to show information about the entire media collection, such as its name, size, date created, number of tracks, number of playlists, and free space. All of these things can be determined by sending a packet like the one shown in Figure 59 to port 50002 on the player that holds the slot, with appropriate values for D (the device number you are posing as), D_r (the device owning the slot), and S_r (the slot you're interested in). These are the same values used in CDJ status packets, as shown in Figure 11

	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	05					
10	Device Name (padded with 00)															01
20	00	D	len_r		IP address				00	00	00	D_r	00	00	00	S_r

Figure 59: Media query packet

Since this packet uses subtype 00, the length sent in len_r has the value 000c, reflecting the twelve bytes which follow it.

The player will respond by sending a packet like the one shown in Figure 60 to port 50002 on the IP address specified in your query packet (so you want to supply your own address to get the response). This contains the information about the media mounted in the slot.

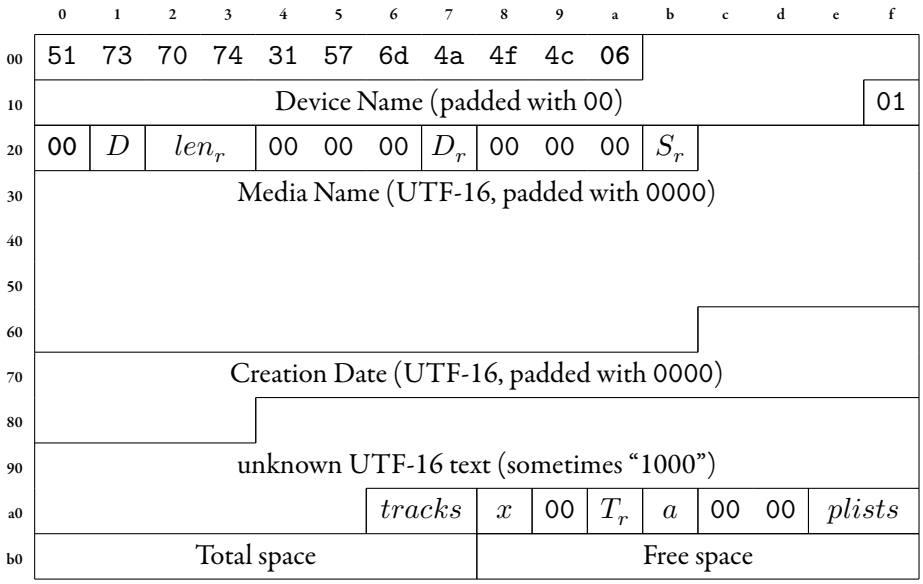

Figure 60: Media response packet

Since this packet uses subtype 00, the length sent in len_r has the value 009c, reflecting the number of bytes which follow it.

The name of the media stick (as assigned in rekordbox) can be found as a UTF-16 encoded string starting at byte 2c, and up to 40 bytes long, padding with trailing null characters. This is followed by a textual representation of the creation date of the media database, encoded in the same way, starting at byte 6c, up to 28 bytes long. (Neither of these fields have meaningful values for "media" served by rekordbox mobile from its collection on a phone.)

The number of rekordbox tracks present in the database is found in bytes a6 and a7, (this will be zero if there is no rekordbox database present). The purpose of value x at byte a8 is uncertain, we have seen values of 06 and 00, but don't know what they mean.

T_r at byte aa reports the type of tracks available, with the same values used in CDJ status packets, as shown in Figure 11. In other words, it will have the value 01 when a rekordbox database is present, and 02 otherwise; this is the same value that

must be sent as T_r in order to make a successful request for the root menu associated with this media. Again, we don't know the purpose of a at byte `ab`, though it seems that it might have the value `01` when there is a rekordbox database present, and `00` otherwise, hence the use of a to suggest "analyzed."

The number of rekordbox playlists present on the media (also zero if there is no rekordbox database) is found at bytes `ae` and `af`.

Finally, there are two eight-byte numbers at the end of the packet. The value at bytes `b0` through `b7` is the total capacity of the media (in bytes), and the value at bytes `b8` through `bf` is the number of unused bytes left on it.

12 What's Missing?

We know this analysis isn't complete. Here are some loose ends to explore.

12.1 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/deep-symmetry/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/deep-symmetry/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).

12.2 Mysterious Values

There are still many values with unknown meanings described above, and undoubtedly menu types that have yet to be explored; I have focused on the ones that will be immediately useful to Beat Link Trigger. Contributions of additional research and insight are eagerly welcomed—I would have not gotten nearly this far without help!

12.3 Reading Data with Four Players

In order to offer metadata, timecode, waveforms, and so on, when there are four actual CDJs on the network, it is necessary get the data using a different mechanism. See the Crate Digger project[26] for the solution we have found.

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,[27] and there used to be a free quick start suite for working with it[28] but that site no longer exists because the original (California) company Encirq[29] was acquired by the Japanese Ubiquitous Corporation in 2008.[30] It seems to still be available,[31] but I'd be surprised if they wanted to help out an open source effort like this one.

12.4 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.

12.5 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/deep-symmetry/dysentery`

List of Figures

[26] `https://github.com/Deep-Symmetry/crate-digger`
[27] `https://www.quora.com/What-database-system-did-Greg-Kemnitz-develop`
[28] `http://java.sys-con.com/node/328557`
[29] `https://www.crunchbase.com/organization/encirq-corporation`
[30] `http://www.ubiquitous.co.jp/en/news/press/pdf/p1730_01.pdf`
[31] `http://www.ubiquitous.co.jp/en/products/db/md/devicesql/`

List of Tables

http://deepsymmetry.org