Cognionics Raw Data Specification {Main (https://web.archive.org/web/20230314173716/http://cognionics.com/wiki/pmwiki.php/Main)}

This page exists to help you develop software that can work to extract data from a Cognionics EEG headset. It is **strongly recommended** to use the Data Acquisition Software and LSL to stream the pre-formatted EEG data to any platform. However, if extracting information manually is mandatory, this information will provide you with concrete information to design and code your own software for your Cognionics EEG headset.

Cognionics systems all provide a virtual serial port interface after Device Pairing (https://web.archive.org/web/20230314173716/http://cognionics.com/wiki/pmwiki.php/Main/DevicePairing). The raw data can be obtained by opening the appropriate COM port and correctly parsing the byte stream into 24-bit samples. This makes it very simple to build custom applications on most platforms.

It is also possible to directly access the native protocols, in this case a FTDI D2XX USB API. This is somewhat more complex but may offer significant performance improvements. The underlying byte-based data structures (described below), however, is identical in all cases. This makes it easy to separate and abstract the lower level I/O functions (which deal in bytes) from the higher level code which parses the bytes into EEG samples for display/processing/logging.

Contents [hide]

Headset Channels

The complete Quick-20 has 20 EEG channels with reference at A1. It also has an addition 3 accelerometer channels for a total of 23 channels. It operates at 500 samples/sec. The order of channels is:

1. F7 – A1
2. Fp1 – A1
3. Fp2 – A1
4. F8 – A1
5. F3 – A1
6. Fz – A1
7. F4 – A1
8. C3 – A1
9. Cz – A1
10. P8 – A1
11. P7 – A1
12. Pz – A1
13. P4 – A1
14. T3 – A1
15. P3 – A1
16. O1 – A1
17. O2 – A1
18. C4 – A1
19. T4 – A1
20. A2 – A1
21. Accelerometer X
22. Accelerometer Y
23. Accelerometer Z

Note that this list of channels will change depending on what headset you have. The channel order visible in the "Cognionics Device Configuration Page" is the order used in our API \\ If you are not sure what the names of the channels on your headset are, please email us at info@cognionics.com for support.

Additional Optional Channels (AIM, ExG)

Channel order is:

1. Referenced channels
2. USB ExG channels
3. AIM channels
4. Accelerometer channels

Bluetooth/FTDI Serial Port Interface

The headset utilizes a direct Bluetooth connection to a custom dongle with a built-in Bluetooth stack. This bypasses the complex and often unreliable Windows stack. Each dongle is hard-paired to a specific headset and will automatically search and connect to its paired system. The dongle relays the Bluetooth byte stream to the PC using the FTDI USB-serial converter. The serial converter operates at 3,000,000 baud, 8-N-1, with flow control enabled.

It is recommended that you use the FTDI D2XX USB API rather than the virtual COM port interface. The D2XX is more reliable and allows a search by device name rather than arbitrary COM port numbers.

Data Packet Format

Since a serial port is an inherently byte (8-bit) based protocol and our EEG systems produce multi-channel, 24-bit signals, a simple packetization scheme is necessary to define the start of a sample block and provide a means to check for lost packets.

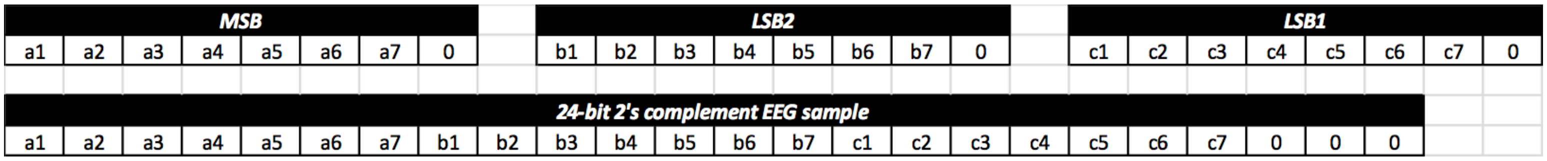
Packet Header

Each packet contains the data for all channels at single sample. The start of the packet is always the byte **0xFF**. By design, no other byte in the data stream can have the value **0xFF** making it an unambiguous start code. The second byte is a packet counter that cycles from **0** to **0x7F**. Lost packets can be detected by checking to see if the packet counters increment properly from sample to sample.

Data Section

The data for each channel is split into 3-bytes. For transmission purposes, EEG/ECG/EMG channels, auxiliary (accessory) channels and accelerometer channels are treated identically. As an example, if you have a 32-channel EEG system with a 3-axis accelerometer and a wireless trigger, your data section will contain 108 bytes ((32+3+1)*3).

The data section starts with the MSB (highest bits) for the first channel, followed by the LSB2 (middle bits) and the LSB1 (lowest bits). This repeats for channel 2 to the last channel in your system. To convert the three bytes for each channel back into a 24-bit sample, use this chart:



MSB									LSB2									LSB1							
a1	a2	a3	a4	a5	a6	a7	0		b1	b2	b3	b4	b5	b6	b7	0		c1	c2	c3	c4	c5	c6	c7	0
24-bit 2's complement EEG sample																									
a1	a2	a3	a4	a5	a6	a7	b1	b2	b3	b4	b5	b6	b7	c1	c2	c3	c4	c5	c6	c7	0	0	0		

The arrangement is designed such that no combination of bits can possibly produce the **0xFF** packet start code.

Packet Tail

The end of a packet embeds information about the status of the impedance check, battery and trigger. The first tail byte indicates whether or not the impedance check is on (**0x11**) or off (**0x12**). The second byte contains the battery voltage. The last two bytes form a 16-bit value that was received from the wireless trigger.

Summary Structure of Single Packet

- Packet Start - 0xFF
- Packet Counter - increments from 0x0 to 0x7F
- Channel 1 - MSB

- Channel 1 - LSB2
- Channel 1 - LSB1
- ...
- Channel N - MSB
- Channel N - LSB2
- Channel N - LSB1
- Impedance Check Status - 0x11 for on, 0x12 for off
- Battery Voltage
- Trigger MSB
- Trigger LSB

In general, you will see 2+3*CHANNELS+4 number of bytes per packet.

Connecting to the headset using the FTDI Driver

The FTDI D2XX programmer’s manual can be found here (https://web.archive.org/web/20230314173716/http://www.ftdichip.com/Support/Documents/ProgramGuides/D2XX_Programmer%27s_GuideFT_000071.pdf).

The example code below is written in C#. The native language for the drivers is C/C++. Hence the example code may be slightly more complex due to the translation but the essential function calls are the same.

Searching for a list of devices connected to a computer:

```
UInt32 ftDevCount = 0;
ftStatus = ftDev.GetNumberOfDevices(ref ftDevCount);

ftdiDeviceList = new FTDI.FT_DEVICE_INFO_NODE[ftDevCount];
ftStatus = ftDev.GetDeviceList(ftdiDeviceList);
String[] deviceNames = new String[ftDevCount];

for (int c = 0; c < ftDevCount; c++)
{
deviceNames[c] = ftdiDeviceList[c].Description.ToString();
}

Connecting to a device and configuring the serial port settings

if (ftDev.OpenBySerialNumber(ftdiDeviceList[devID].SerialNumber) == FTDI.FT_STATUS.FT_OK)
{    ftDev.SetFlowControl(FTDI.FT_FLOW_CONTROL.FT_FLOW_RTS_CTS, 0x11, 0x13);
    ftDev.SetDataCharacteristics(FTDI.FT_DATA_BITS.FT_BITS_8, FTDI.FT_STOP_BITS.FT_STOP_BITS_1, FTDI.FT_PARITY.FT_PARITY_NONE);
    ftDev.SetLatency(2);
    ftDev.SetBaudRate((uint)3000000);

    connectedName = ftdiDeviceList[devID].Description.ToString();

    return true;
}
else
{
    return false; //failed to open!

}
```

[$[Get Code]] (https://web.archive.org/web/20230314173716/http://cognionics.com/wiki/pmwiki.php/Main/CognionicsRawDataSpec?action=sourceblock&num=1)

Reading a single byte:

```
public byte ReadByte()
{
    UInt32 bytesRead = 0;
    byte[] t_data = new byte[1];
    ftDev.Read(t_data, 1, ref bytesRead);
    return t_data[0];
}
```

[$[Get Code]] (https://web.archive.org/web/20230314173716/http://cognionics.com/wiki/pmwiki.php/Main/CognionicsRawDataSpec?action=sourceblock&num=2)

Writing a single byte:

```
public void WriteByte(byte dat)
{
    UInt32 bytesWritten = 0;
    byte[] data = new byte[1];
    data[0] = dat;
    ftDev.Write(data, 1, ref bytesWritten);
}
```

[$[Get Code]] (https://web.archive.org/web/20230314173716/http://cognionics.com/wiki/pmwiki.php/Main/CognionicsRawDataSpec?action=sourceblock&num=3)

Converting data to volts

The conversion factor for EEG channels to volts is:

$$EEG = data \times \frac{5}{3} \times \frac{1}{2^{32}}$$

The conversion factor for accelerometer channels to volts is:

While the conversion factor for voltage of accelerometer channels to g force is:

The conversion factor for the battery byte to volts is:

$$BATTERY = batteryByte \times \frac{5}{128}$$

The conversion factor for the AIM channels to volts is:

Below is the bare minimum code to read and assemble a single packet of samples. Note: the function byteInterface.ReadByte() is the same as the FTDI function from the previous section to grab a single byte from the serial interface.

```
//wait for sync byte 0xFF
while (byteInterface.ReadByte() != 255) {};

//read packet counter
int packetCount = byteInterface.ReadByte();

//read the 20 EEG channels
int NumEEG = 20;
for (int c = 0; c < NumEEG; c++)
{
    msb =  byteInterface.ReadByte();
    lsb2 = byteInterface.ReadByte();
    lsb1 = byteInterface.ReadByte();

    int tempEEG = (msb << 24) | (lsb2 << 17) | (lsb1 << 10);
}

int NumACC = 3;
//read the 3 ACC channels
for (int c = 0; c < NumACC; c++)
{
    msb =  byteInterface.ReadByte();
    lsb2 = byteInterface.ReadByte();
    lsb1 = byteInterface.ReadByte();

    int tempACC = (msb << 24) | (lsb2 << 17) | (lsb1 << 10);
}


//read packet tail
int impStatus = byteInterface.ReadByte();

//read battery voltage
int batteryByte = byteInterface.ReadByte();

//read trigger
int trigger = (byteInterface.ReadByte()<<8) + byteInterface.ReadByte();
```

[$[Get Code]] (https://web.archive.org/web/20230314173716/http://cognionics.com/wiki/pmwiki.php/Main/CognionicsRawDataSpec?action=sourceblock&num=4)

Impedance Check

Cognionics systems all integrate a real-time impedance check on all EEG channels to assess contact quality. The impedance signal is a carrier wave, at ¼ the sample rate where the amplitude of the carrier wave is proportional to the contact impedance. Some filtering is required to separate the impedance check carrier wave from the raw EEG signal.

USB ExG channels also carry the same carrier wave as all of the standard (reference) channels on a system.

The AIM does not perform an impedance check on any of its channels.

Recovering the EEG

We suggest a series of two IIR notch filters to remove the impedance check carrier wave at 1/4 the sample rate and its harmonic at 1/2 the sample rate. The filter coefficients are:

$$\frac{0.85z^{-2}+0z^{-2}+0.85z^{-0}}{0.7z^{-2}+0z^{-1}+1}$$

for the filter at ¼ the sample rate and

$$\frac{0.8z^{-1}+0.8z^{-0}}{0.6z^{-1}+1}$$

for ½ the sample rate.

This filter has a -3dB point at 0.22 of the sample rate or 110 Hz for a system that operates at 500 samples/sec.

Alternatively, a simpler method is to use a 4-point moving average filter. The output is simply the sum of the current sample, plus the 3 previous samples divided by 4. The drawback is that the moving average filter has significant passband attenuation. The -3dB point would be at about 1/10 the sample rate. Hence for a system that is running at 500 Hz, the effective bandwidth is only 50 Hz which may be acceptable for certain applications but not others.

Extracting the Impedance Carrier Wave

The impedance carrier wave is locked to ¼ the sample rate. The easiest way is to correlate the raw signal against template of a sine and cosine wave also at ¼ the sample rate. Usually a sequence length of 500 points is enough. The two sequences are:

1/250, 0, -1/250, 0, 1/250, 0 …

0, 1/250, 0, -1/250, 0, 1/250 …

Note the two sequences are identical except shifted by 1 sample (90 degrees).

Multiply the raw signal against each template and sum each result separately. Then square each of the two sums, add them and take the square root. The result is the magnitude (MAG) of the impedance check carrier wave at 1/4 the sample rate.

Alternatively, a simpler filter is to simply keep a running buffer of the latest 4 samples. Perform the following subtractions:

Abs(sample[0] – sample[2])/2 Abs(sample[1] – sample[3])/2

Take the greater of the two differences, which is an approximate of the magnitude of the impedance check carrier wave. For a reliable measure, the results should be further filtered by a moving average or other low-pass filter with a history of around 1 second.

In either case, the conversion factor to impedance (Z) is: Z = MAG * 265,000,000

For our active electrode dry electrode systems, impedances of under 4000 kOhms generally produces acceptable signals for resting applications. For ideal signal quality, the impedance should be under 2500 kOhms.

Enabling and Disabling the Impedance Check

The impedance check can be disabled by writing 0x12 to the serial port interface. It can also be turned back on by writing 0x11.

The current status of the impedance check is shown in the impedance check byte in the packet tail (see above). If the impedance check is not enabled (0x12), then all filtering should be disabled to allow for the full bandwidth of the ADC converters (-3dB point at 1/4 the sample rate).

Page last modified on November 06, 2017, at 08:01 PM