ChatGPT

Guide to Analog Hardware Design for EEG, Audio, and Eye-Tracking Projects

Fundamentals of Electrical Engineering for Analog Design

Voltage, Current, and Ohm's Law

Voltage (V) is the electrical potential difference between two points, and **current (I)** is the flow of electric charge through a circuit. These are related by **Ohm's Law**, which states that the voltage across a resistor equals the current through it times its resistance (R): $V = I \times R$. In other words, the ratio V/I for a given conductor is its resistance [1]. This linear relation (discovered by Georg Ohm) means if you triple the voltage across a resistor, the current also triples (assuming the resistance remains constant). The unit of resistance is the ohm (Ω), and resistors essentially **oppose current flow** by dissipating energy as heat. For example, a 1 MΩ resistor with 10 V across it will allow only 10 µA of current to flow [2] [3].

In addition to resistors, circuits often involve **energy storage components**: **capacitors** and **inductors**. A **capacitor (C)** stores energy in an electric field (it accumulates charge, Q, with voltage V across it, following Q = C·V). A **capacitor's voltage cannot change instantaneously** – it will source or sink current to oppose sudden changes in voltage [4]. In practical terms, capacitors "smooth" voltage changes by charging or discharging; they can block steady DC (after charged) while allowing AC to pass (especially at higher frequencies). Conversely, an **inductor (L)** stores energy in a magnetic field (when current flows through it). An **inductor's current cannot change instantaneously** – it resists rapid changes in current by inducing a voltage [4]. Thus, **capacitors oppose changes in voltage and inductors oppose changes in current**. These properties make capacitors useful for filtering out voltage spikes and inductors for smoothing current fluctuations, among other applications.

Analog vs. Digital Signals; Signal Conditioning and Noise

Analog signals are continuous in time and amplitude: they can take on an infinite range of values within a given range [5]. For example, an analog voltage might smoothly vary anywhere between 0 and 5 V. **Digital signals**, by contrast, represent information in discrete steps (typically binary levels) [6]. A digital signal switches between defined states (e.g. 0 V and 5 V for logic 0/1), with no in-between values used for data. When plotted over time, an analog signal appears as a smooth, continuous wave, whereas a digital signal looks like a stepwise or square waveform jumping between discrete levels [5] [6].

Because analog signals are continuous, they are susceptible to noise and interference. **Signal conditioning** is the practice of processing analog signals to make them suitable for measurement or conversion. This includes **amplifying** small signals, **filtering** out unwanted noise or frequency components, and **level shifting** or scaling the signal to match the input range of an analog-to-digital converter (ADC) or other device. For example, biopotential signals like EEG are microvolts in amplitude and riding on a DC offset; they must be amplified and filtered before digitization. **Noise filtering** is crucial: common noise sources include mains hum (50/60 Hz power-line interference), electromagnetic interference (EMI) picked up by wires, thermal noise, and more [7]. Passive RC filters or more complex active filters are used to attenuate these unwanted components. Proper signal conditioning maximizes

the **signal-to-noise ratio (SNR)** so that the true signal (e.g. a heartbeat waveform or guitar string vibration) can be accurately captured without distortion or loss in the noise floor.

Passive Components: Resistors, Capacitors, Inductors

Passive components are electronic components that do not require external power to operate (beyond the signals they process). The most common are resistors, capacitors, and inductors:

- **Resistors** convert electrical energy to heat, following Ohm's Law as discussed. In design, resistors are used to limit currents, set bias conditions, and create voltage dividers. A **voltage divider** is a simple two-resistor network that divides an input voltage into a lower output voltage proportional to the resistor ratio. For instance, two equal resistors in series will split a voltage in half. Voltage dividers are fundamental for creating reference voltages and bias points in analog circuits.

- **Capacitors** store charge and impede changes in voltage. They are characterized by capacitance (farads, F). In DC steady-state, an ideal capacitor is an open circuit (once charged, no DC current flows). In AC, a capacitor's **reactance** decreases with frequency ($X_c = 1/(2\pi fC)$), meaning it passes high-frequency signals but blocks low-frequency signals. This property is exploited in filtering: a capacitor in series with a signal acts as a **high-pass filter** (allowing AC through but blocking DC), while a capacitor to ground (in parallel with the signal path) forms part of a **low-pass filter** (shunting high-frequency noise to ground). We will discuss filter configurations shortly.

- **Inductors** store energy in a magnetic field when current flows through them. Inductance is measured in henrys (H). An ideal inductor is a short circuit for DC (steady current flows freely through it) but resists changes in current. Its reactance increases with frequency ($X_L = 2\pi fL$), meaning inductors pass low frequencies and block high frequencies. Inductors are common in power supplies (to smooth current in regulators) and in filters like high-pass filters (in series with the load to block low-frequency or DC components) or low-pass LC filters. However, in low-frequency analog front-ends (like biopotential amplifiers), large inductors are less common due to size; designers often use active filters instead.

These passive components can be combined to create frequency-selective networks. For example, an **RC low-pass filter** can be made with a resistor and capacitor: the resistor feeds the output node, and a capacitor from output to ground. This filter passes low-frequency (and DC) signals to the output but attenuates high-frequency signals, which effectively go through the capacitor to ground [8]. Conversely, an **RC high-pass filter** uses a series capacitor followed by a resistor to ground; it passes high-frequency transients but blocks DC (the capacitor prevents steady DC from reaching the output). The cutoff frequency of such filters is given by $f_c = 1/(2\pi RC)$, above which a high-pass passes signals and below which a low-pass passes signals. By selecting R and C values, you can tune filters to target noise (like 50 Hz mains) or specific signal bands.

Active Components: Operational Amplifiers and Transistors

Active components can amplify or control the flow of electricity and usually require an external power source. Key active devices in analog design are **operational amplifiers (op amps)** and **transistors**:

- **Op Amps:** An op amp is a high-gain differential amplifier integrated circuit. It has two high-impedance inputs (inverting "–" and non-inverting "+") and a low-impedance output [9]. The op amp outputs a voltage proportional to the difference between its inputs, multiplied by a huge

open-loop gain (typically 10^5 or more). By using feedback networks of resistors/capacitors, op amps can implement precise gain, filtering, and other operations. For example, in a **non-inverting amplifier** configuration, the op amp outputs whatever voltage is needed (within its supply limits) to keep the "–" input equal to the "+" input. With a resistor divider feeding back from output to "–", and a reference at "+", you get a stable closed-loop gain. Op amps are extremely versatile: they can amplify microvolt signals to measurable levels, serve as active filters (e.g. in multiple feedback filter topologies), and create buffers, summers, differentiators, integrators, etc. Many analog front-ends rely on op amp circuits as building blocks. For instance, an op amp with resistors and capacitors can act as an active band-pass filter or an amplifier that boosts certain frequencies. (In the classic guitar pedal circuits, an op amp is used in a non-inverting configuration with feedback components setting the gain and frequency response [10].) Modern op amp ICs (like the TL081, LM358, etc.) simplify analog design by providing near-ideal amplifier behavior (high input impedance, low output impedance, predictable gain via feedback).

- **Transistors:** Transistors (BJT or MOSFET) are semiconductor devices that can act as amplifiers or switches. In analog design, a single transistor in a certain bias configuration (such as a **common-emitter BJT amplifier** or a **common-source FET amplifier**) can amplify signals, though with more limited gain and higher distortion compared to op amps. Transistors are the building blocks of op amps themselves, but at the discrete design level, one might use transistors for simple amplifiers (e.g. a one-transistor preamp for a guitar pickup), for creating current sources/sinks, or for active loads. BJTs amplify current (collector current ~ β times base current in active region) while MOSFETs amplify voltage (they are voltage-controlled devices where gate voltage modulates drain current). For our purposes, op amp ICs will handle most analog amplification tasks, but understanding transistors is useful especially for interfacing and when you need to implement analog switches or custom amplification where an op amp might not be available or suitable.

Basic Analog Circuit Analysis: Dividers, Filters, and Amplifiers

Some fundamental analog circuit configurations appear again and again in designs:

- **Voltage Dividers:** As mentioned, two resistors can form a divider to produce a fraction of a voltage. If R1 is connected from a source Vin to a node Vout, and R2 from Vout to ground, then Vout = Vin * (R2 / (R1 + R2)). This principle is used for **biasing** (setting DC operating points). For example, biasing an op amp input at mid-supply (e.g. 2.5 V on a 5 V system) can be done with a 2-resistor divider, providing a reference "virtual ground" for AC-coupled signals. (We will see this in practice for single-supply bio-signal and audio circuits, where we need to shift an AC signal to ride on a DC offset.)

- **RC Filters:** Simple first-order filters are easily analyzed with the concept of impedance. In an RC low-pass, at high frequencies the capacitor's impedance is low (shunting the signal to ground) so output is small; at low frequencies the capacitor's impedance is high (open circuit), so the output sees the full input through the series resistor [8]. The cutoff frequency f_c is where the reactance of C equals R, yielding about –3 dB output. High-pass behavior is the complement. More complex filters (second-order or higher) can be made by cascading stages or using RLC networks or op amp active filter topologies (like Sallen–Key or multiple feedback filters) to get sharper roll-off. For analog design, it's important to understand Bode plots (frequency response) of filters: e.g. a single-pole RC filter rolls off at –20 dB/decade beyond f_c. Designing a filter involves choosing a topology and component values to get the desired cutoff and attenuation characteristics (for instance, a **notch filter** can be used to sharply attenuate a specific frequency like 50 Hz mains hum [11]).

- **Amplifier Configurations:** With op amps, the two most common configurations are **inverting** and **non-inverting** amplifiers. In an **inverting amplifier**, the input signal is applied via a resistor to the "–" input, and a feedback resistor goes from output back to "–", while "+" is at a reference (often ground). The closed-loop gain is –(Rf/Rin). In a **non-inverting amplifier**, the input is applied to "+" and a feedback network from output to "–" sets the gain as 1 + (Rf/Rg). The non-inverting configuration has high input impedance (great for buffering sensors), whereas the inverting configuration allows summing multiple inputs (useful in audio mixers or multi-sensor circuits). For either case, the op amp must be powered by supply rails that accommodate the output swing needed (for example, to get a 5 Vpp output, the op amp might need ±5 V or a single 10 V supply unless it's rail-to-rail). **Gain-bandwidth** is another consideration: op amps have finite bandwidth, so high gains limit the usable frequency range. In practice, one often uses multi-stage amplification: e.g. two op amp stages at gain 10 each rather than one stage at gain 100, to preserve bandwidth and reduce noise.

Finally, analyzing analog circuits often involves applying **Kirchhoff's laws** (KCL and KVL) and understanding impedance in the frequency domain. For complex circuits, engineers use techniques like nodal analysis or tools like SPICE simulation. But for a hardware hobbyist, mastering the above building blocks – how to divide voltages, filter signals, and amplify with op amps – goes a long way toward creating functional analog systems.

Core Analog Hardware Design Techniques

Analog Front-End Design for Bio-Signals (EEG/ECG)

Bio-potentials like EEG (electroencephalogram) and ECG (electrocardiogram) are very low-amplitude analog signals and require careful front-end design. These signals range on the order of tens of microvolts (EEG) to a few millivolts (ECG) and occur in specific frequency bands (EEG ~0.5–40 Hz of interest, ECG ~0.05–100 Hz) [12]. The challenge is to amplify the tiny differential signal (the potential difference between two electrodes on the body) while rejecting the large common-mode noise that couples into the body and wires (for example, mains interference up to ~1.5 V common-mode [12]).

The solution is to use an **Instrumentation Amplifier (IA)** as the first stage. An **instrumentation amplifier** is a specialized high-performance differential amplifier with extremely high input impedance and high common-mode rejection. It essentially amplifies the voltage difference between its two input terminals while rejecting any voltage common to both inputs (common-mode). This is **crucial for bio-signal acquisition**, because the human body and leads pick up ambient noise (e.g. the 50/60 Hz from power lines) approximately equally on both electrodes – a high CMRR (Common-Mode Rejection Ratio) means the amp greatly attenuates this common noise. In-amps are designed for this task: *"The Instrumentation Amplifier is the hallmark amplifier for bioelectrical measurements"* due to **high input impedance** (it draws virtually no current from the electrodes) and **excellent common-mode rejection** [13]. For example, a popular chip like the AD620 or AD623 can provide gains of ~10–1000 with CMRR well over 80–100 dB, meaning it rejects common-mode voltages by a factor of 10,000 or more [14]. This allows microvolt-level ECG/EEG differences to be extracted from volt-level common-mode interference.

To use an instrumentation amp properly, one typically configures a gain (often via a single resistor on chips like AD620 [15]) and often drives a **reference pin** to set the output baseline. Bio-signals are AC signals centered around a baseline (for ECG, the baseline may wander, and for EEG there's typically a reference electrode). Most IAs have a reference (or "Vref") input that shifts the output. It is common to bias this at mid-supply for single-supply systems so that the amplified signal, which might go positive or negative relative to the body's baseline, is centered in the ADC input range. For example, if using a 5 V

single-supply IA and microcontroller, you might set Vref = 2.5 V. This way, a 0 µV differential (no difference between electrodes) yields an output of 2.5 V; a small positive differential increases the output above 2.5 V, and a negative differential decreases it, without clipping at 0 V or 5 V. In practice, this can be done by a simple voltage divider and buffer to create a 2.5 V rail as the reference. (The instructables ECG project, for instance, uses a Vref at half-supply so that the IA output "will never be negative" with respect to the Arduino's ground [16].)

Another technique in ECG front-ends is **Right-Leg Drive** (for EEG, often called Driven Ground). This involves using a third op amp to actively drive a reference electrode (often attached to the body's leg or ear) with the inverted common-mode voltage. By feeding back the common-mode noise into the body out of phase, the common-mode interference at the amplifier inputs is reduced. This further boosts effective CMRR. Many ECG amplifier designs include a right-leg drive amplifier that takes the common-mode from the IA inputs and drives the body to counteract noise [14]. For DIY EEG/ECG, if using only battery power and proper isolation, a driven reference may be optional, but it's good to be aware of it as a pro technique to reduce noise.

Isolation is a critical consideration as well. For safety, any device connected to a human (especially EEG on the head or ECG on the chest) should be isolated from mains-powered circuits. In practice, hobbyists often power these circuits with batteries or USB (which is isolated through the PC). It is **strongly advised not to directly connect bio-signal circuits to mains-referenced equipment** unless using proper isolation amplifiers or optocouplers. As one guide warns: do *"not connect your ECG circuit to a wall outlet or any instrument powered through the wall outlet for safety reasons"* [17]. Isolation amplifiers or USB isolator devices can provide an extra layer of safety by ensuring there's no direct galvanic path for current from the mains or PC into the subject.

Filtering and Amplification Techniques (Low-Pass, High-Pass, Band-Pass, Notch)

Once the small bio-signal (or any analog signal of interest) is amplified to a workable level, it usually needs filtering to define the signal band and remove remaining noise. In analog front-ends, filters can be **cascaded** in stages:

- **High-Pass Filter (HPF)**: Often, a high-pass filter is used at the input or after the first amp stage to remove DC offsets and slow baseline wander. For example, an ECG signal may drift due to respiration or movement; a HPF with cutoff ~0.5 Hz will keep the ECG waveform centered and stable by blocking ultra-low frequencies (including true DC). This can be as simple as a coupling capacitor with a resistor to ground (forming a high-pass) or an active HPF using an op amp.

- **Low-Pass Filter (LPF)**: To remove high-frequency noise (EMG muscle noise, high-frequency EMI, etc.), a low-pass filter is used. EEG might be low-passed around 30–50 Hz (depending on which brainwaves of interest), and ECG often around 100–150 Hz for diagnostic-quality or ~40 Hz for simple heart-rate monitoring. Active low-pass filters using op amps (like a 2nd-order Butterworth for a flat passband) can provide a sharper cutoff than a single RC. For instance, a simple RC LPF (one pole) attenuates –20 dB/decade beyond the cutoff, whereas a 2-pole active filter can do –40 dB/decade, which is better at rejecting out-of-band noise.

- **Notch Filter (Band-Stop)**: A notch filter is designed to **reject a specific frequency** without affecting others. The classic example in biopotential circuits is a 50 Hz or 60 Hz notch (depending on local mains frequency) to eliminate power line hum. Many EEG/ECG designs include an active notch filter centered exactly at the mains frequency [11]. This greatly reduces the hum present. The width of the notch can be narrow to avoid cutting into signal band (for example, a narrow 50 Hz ±1 Hz notch). Alternatively, some designers avoid notch filters and instead rely on high

CMRR and digital filtering, because notch filters can introduce ringing. But for a simple hardware project, an op amp twin-T notch or state-variable notch is a useful component.

- **Band-Pass Filter**: In some cases, you want to explicitly define a band of frequencies to pass. For example, for EEG you might build a band-pass from 0.5 Hz to 40 Hz; for EMG (muscle signals) maybe 20–500 Hz. A band-pass can be made by cascading a high-pass and low-pass back-to-back. The result passes frequencies between the two cutoff points. Op amp band-pass designs can also achieve this in one stage. In a music context (like a guitar effect), band-pass filters can isolate a certain tone range. An example is a wah pedal which is basically a resonant band-pass filter sweeping through the audio spectrum.

When designing filters, consider the **order** (higher order gives sharper cutoff but more components), the **response shape** (Butterworth = maximally flat, Chebyshev = steeper roll-off but with ripples, Bessel = linear phase, etc.), and whether analog filtering alone is sufficient or if you plan to do additional digital filtering after ADC. In low-frequency applications like EEG, a lot of filtering can be done digitally once the signal is sampled, but you still need analog filters to prevent aliasing (see below on ADC sampling) and protect the amplifier from out-of-band overload.

Additionally, **stability** matters when cascading multiple op amp filters – ensure your op amps are stable with the given configuration (sometimes driving capacitive loads or high gain at certain frequencies can cause oscillations). It's often wise to buffer stages and avoid interactions between filter sections by using op amps as isolators.

Amplification strategy: It's generally best to amplify the signal *early* to overcome noise, but not so much that you saturate (clip) the amplifier with noise or offsets. For example, an EEG amp might use an IA at gain 1000 as the first stage – if the input is 10 µV, output becomes 10 mV, which is easier to digitize. But if there's a 100 µV DC offset difference between electrodes, that becomes 100 mV offset at output; not a big issue. We often distribute gain across stages: a moderate gain IA followed by active filter stages each with some gain. **Instrumentation amps often have limitations on how high a gain is practical before bandwidth suffers**, so you might do gain 100 at the IA and another 10× in a low-pass filter op amp, for overall 1000×. Always ensure the amplified signal will stay within the supply rails of each stage plus some headroom for noise. Using dual power supplies (e.g. ±5 V) helps center the signal around 0, but for portable Arduino projects a single-supply design is more common, hence the need for biasing at mid-supply.

Shielding and Grounding to Reduce Noise

When dealing with low-voltage, high-impedance signals (like EEG from scalp electrodes or even a high-impedance guitar pickup), **electromagnetic interference** can easily couple into your circuit. Good **shielding and grounding practices** are critical to reduce noise:

- **Shielded Cables:** Use coaxial or shielded cables for analog signal leads, especially for sensors that are a distance away. The cable's shield (typically a braided or foil conductor surrounding the signal wire) should be tied to a reference ground. This creates a Faraday cage around the inner conductor, intercepting interference. The rule of thumb is to **ground the shield at one end only (usually at the analog front-end ground)** to avoid ground loops [18]. If you ground both ends of a shield to two different ground points, you might create a loop that can pick up hum like an antenna. By grounding at one end, the shield is held at reference potential and shunts interference to that ground.

- **Star Ground and Common Reference:** In the circuit, have a single common ground point where sensitive analog grounds join, ideally separate from high-current or digital return paths. Ground loops (multiple return paths for current that form a loop) can induce noise. All high-impedance input returns (e.g. the reference electrode in an ECG) should meet at a common node. If you have to connect to other grounds (like the digital ground of an Arduino), do so at one point (a star topology) rather than daisy-chaining sensitive grounds through noisy environments.

- **Guard Traces:** In PCB layouts for high impedance nodes, sometimes a guard ring driven at the same potential as the high-impedance input is routed around it. This "driven guard" prevents leakage currents on the board and also capacitively buffers interference. This is an advanced technique often used in ECG amps around the input terminals (driving the cable shield with the common-mode voltage for example).

- **Cable Placement:** Keep analog signal wires short and away from noisy lines. Twisting differential pair wires (for instance, two electrode leads) can help cancel out magnetically induced noise, since each twists sees the interference in opposite phase.

- **Proper Grounding of Shields and Equipment:** Ensure that any chassis or cable shields are tied to the reference node. As noted, *"an ungrounded shield will induce a voltage in the inner conductor"* from external EMI [19], which is the opposite of what you want. By connecting the shield to ground, interference is diverted to ground instead of into your signal line. Also be mindful that the **power supply ground** for your analog circuit (e.g. the Arduino ground) becomes the reference for signals and shields. If your laptop powering the Arduino is also connected to mains ground, there could be a ground differential—generally not a big issue at these low currents, but good to be aware of in case of noise.

- **Filtering the Power Rails:** Noise can also enter through the power supply. Always decouple the analog supply rails with capacitors (e.g. 0.1 µF ceramic + 10 µF electrolytic) close to the op amps or IA. For extremely sensitive circuits, voltage regulators with low noise or additional filtering (RC or LC filters, or even ferrite beads) might be used to isolate the analog supply from digital noise generated by microcontrollers, etc.

In summary, **minimize interference pickup and provide a clean return path for any noise currents.** Shielding and grounding is something to "get right" by design: a poorly grounded shield can make noise worse by injecting interference into your circuit [20]. On a breadboard prototype, you obviously can't lay out ground planes, but you can still practice good wiring (short leads, twisted pairs for differential signals, a single ground point connecting to Arduino ground). In a finished PCB, using a solid ground plane for analog sections and partitioning analog vs digital areas will help a lot. Enclosing sensitive analog circuits in a metal enclosure tied to ground can also act as a shield against external electric fields.

Interfacing Analog Hardware with Microcontrollers and SBCs

ADCs and DACs: Analog-Digital Conversion Basics

Microcontrollers (like Arduino Uno's ATmega328P or the ESP32) include **Analog-to-Digital Converters (ADCs)** to translate analog voltages into digital values. An ADC measures a voltage and outputs a number (in binary) proportional to that voltage relative to a reference. Key specs of an ADC are its **resolution** (in bits) and **sampling rate**. For example, the Arduino Uno uses a 10-bit ADC with a 0–5 V range by default, giving values 0–1023 (2^10–1) [21]. Each count represents ~4.9 mV in this range [21].

The ESP32 has a 12-bit ADC (0–4095 counts) on a 0–3.3 V range (approximately 0.8 mV per count). Higher resolution means the ADC can discern smaller changes in voltage. If you need more resolution or different ranges, external ADC ICs (ADS1115 is a 16-bit ADC, ADS1299 is a 24-bit biopotential ADC, etc.) can be used via I^2C or SPI.

The **sampling rate** is how many samples per second the ADC can take. According to the Nyquist Sampling Theorem, the sample rate *f_s* must be greater than **2×** the highest frequency in the signal to accurately reconstruct it [22] [23]. For instance, to capture a 1000 Hz signal, you should sample above 2000 Hz (preferably much higher, such as 5–10 kHz, to have some margin and allow digital filtering). If you sample too slowly, higher-frequency components will **alias** (they will appear distorted as lower frequencies in the data) [24] [25]. Arduino's ADC, for example, has a default sampling rate around 9.6 kHz (approx 9600 samples/sec) [26], which is fine for signals up to ~4.8 kHz (audio voice range). For EEG/ECG (under 100 Hz), even a few hundred Hz sampling is sufficient to capture the waveform. However, for audio from a guitar (which can have frequencies up to ~5 kHz for harmonics, and we might want 44.1 kHz for hi-fi audio), the Arduino's internal ADC is too slow and low-resolution. In such cases, one might use an external faster ADC or use an entirely different board (like a Raspberry Pi with an external audio ADC, or a specialized DSP).

On the output side, **Digital-to-Analog Converters (DACs)** do the reverse: convert a digital number to an analog voltage or current. Arduinos typically don't have a true DAC (though the Arduino Due and some others do), but the ESP32 does (it has two 8-bit DAC channels), and the Raspberry Pi can use PWM or external DACs. DAC resolution and update rate determine how smoothly you can generate analog outputs (for example, to produce a waveform or a control voltage for an analog synth). In music projects, you might use a DAC to generate an audio signal from the microcontroller (though doing high-quality audio in real time might be beyond an Arduino's capabilities).

In summary, when selecting ADC/DAC specs, consider: the **voltage range** (does it match your signal after any conditioning?), the **resolution** (do you need fine quantization steps for your application?), and the **speed** (sample rate or settling time). For data logging biosignals, a 10-bit ADC at 250 Hz might be fine. For audio, you might want 16-bit at 44.1 kHz. Also note the **reference voltage** of the ADC: Arduino Uno by default uses 5 V (its supply) as reference; you can use the 1.1 V internal reference or an external precision reference if needed to improve accuracy for low-voltage signals. Reducing the reference range effectively increases sensitivity (at the cost of max range): e.g. using 2.5 V reference on a 10-bit ADC gives ~2.44 mV per count instead of ~4.88 mV [27].

Signal Conditioning Before Digitization

We touched on this in fundamentals, but to reiterate: **before feeding an analog signal to a microcontroller's ADC, it must be conditioned to meet the ADC's requirements**. Key steps include:

- **Level Shifting and Range Scaling:** The signal must be within the ADC's input range (often 0 to V_ref, or ±V_ref for bipolar ADCs). Many sensor signals or analog circuits output a **bipolar** signal (positive and negative swings around 0). But most microcontroller ADCs (Arduino, ESP32) can only read **unipolar** voltages (typically 0 to 3.3 V or 5 V). Thus, you often need to shift the signal's baseline. For instance, an audio AC signal centered at 0 V with ±1 V swings should be biased to 1.65 V (midpoint of 0–3.3 V) to be read by ESP32's ADC. This is done by adding a DC offset. A common approach is a coupling capacitor in series (to block the original DC) and a resistor divider to add a DC bias. The output of the coupling capacitor connects to the midpoint of two resistors (forming a 1.65 V divider), and that point goes to the ADC input. This way the AC swings above and below 1.65 V instead of 0. By choosing large resistor values and a suitably sized capacitor, you ensure minimal distortion of the audio band (high-pass cutoff should be well

below the lowest frequency of interest). **Example:** In the MXR Distortion+ guitar pedal circuit analysis, they bias the op amp input at 4.5 V (half of 9 V) through a 1 MΩ resistor, allowing the amplification of the bipolar guitar signal around this midpoint [28]. We do the same concept but then feed an ADC instead of another amp stage.

- **Impedance Matching:** The source impedance feeding an ADC should typically be low (a few kΩ or less) so the ADC's sample-and-hold capacitor can charge quickly and accurately. If you have a high impedance sensor (like >100kΩ), you should buffer it with an op amp buffer (voltage follower) before the ADC. Otherwise, the ADC reading may be slow to settle or noisy. For Arduino's ADC, sources under ~10 kΩ are recommended for best accuracy. In EEG front-ends, the IA outputs can be fed to the ADC directly or via a simple RC anti-alias filter; since the IA has low output impedance, it can drive the ADC input easily.

- **Anti-Alias Filtering:** As mentioned, you should filter out frequencies above half the sampling rate before sampling, to prevent aliasing. This is typically a low-pass filter at the ADC input, sometimes called an anti-alias filter. In many audio ADCs, this is built-in or accompanied by an analog filter. In slower systems (like sampling EEG at 250 Hz), a simple RC filter with cutoff around 125 Hz can serve this purpose. If aliasing is not controlled, high-frequency noise could fold into your band of interest as false low-frequency signals [24] [25].

- **Over-Voltage Protection:** Ensure the input to the ADC never exceeds the allowed range (including slight voltage spikes). Often one uses a series resistor and clamping diodes to the supply rails (some microcontrollers have these diodes internally) to protect against an out-of-range input. For example, if you accidentally induce a 12 V spike on an analog pin expecting 5 V max, a 10 kΩ series resistor and the internal ESD diodes will likely clamp it to 5 V (plus a diode drop) and save the ADC from damage. In critical applications, use external Schottky diodes to V_ref and GND for faster clamping.

In short, **condition your analog signal** such that by the time it hits that `analogRead()` (on Arduino) or ADC pin, it's a nice, within-range voltage with the bandwidth limited to what you care about. A properly conditioned signal will yield more accurate and stable digital readings.

Communication Protocols: UART, SPI, I²C, USB, BLE

After you have the digitized data or if you are using external converters/peripherals, you'll encounter various **communication protocols** to interface analog hardware with microcontrollers and single-board computers (SBCs):

- **UART (Serial):** Universal Asynchronous Receiver/Transmitter is a simple two-wire protocol (TX and RX plus a common ground) for serial communication. UART is used for the Arduino's serial monitor and for many sensor modules that send ASCII data. It's asynchronous (no clock line; both sides agree on a baud rate, e.g. 9600 bps). UART is great for streaming data to a PC (via USB-serial) or between microcontrollers. For instance, an Arduino can send ADC readings over UART to an ESP32 or PC. UART is full-duplex (TX and RX simultaneous) but typically only point-to-point. It's human-readable if sending ASCII text, which makes debugging easy. Many BLE (Bluetooth Low Energy) modules present a UART interface to the microcontroller (e.g. the popular HC-05 Bluetooth module communicates via UART). **Use case:** Sending an ECG data stream from an Arduino to a PC at 115200 baud for logging.

- **SPI (Serial Peripheral Interface):** SPI is a synchronous, high-speed interface with a master-slave architecture. It uses four lines typically: SCLK (clock from master), MOSI (master-out, slave-in), MISO (master-in, slave-out), and CS (chip select, one per slave device). SPI is often used for high-sample-rate ADCs, DACs, and display drivers. It's faster than I^2C generally and full-duplex. For example, a 12-bit ADC MCP3208 can be read by Raspberry Pi or Arduino over SPI, sending clock pulses and receiving data bits rapidly. SPI requires more wires but shines in speed (tens of MHz clock possible). It's commonly used for **audio ADC/DAC chips** (often under a variant called I^2S for stereo audio data) and for high-throughput sensors.

- **I^2C (Inter-Integrated Circuit):** I^2C is a two-wire *clocked* protocol (SCL clock and SDA data) that supports addressing multiple devices on one bus. It's slower than SPI (often 100 kHz or 400 kHz standard, up to a few MHz in fast modes) and half-duplex (master and slaves share the SDA line). Many integrated sensors, ADCs, and DACs use I^2C because it uses minimal pins and supports multiple devices easily. For instance, the ADS1115 16-bit ADC module communicates via I^2C (addressed by 7-bit address). The Arduino can query it for a conversion result and get the data in a few bytes. I^2C is very convenient for **slow-to-moderate speed** data like temperature sensors, light sensors, or configuration of analog front-end chips (like setting gains on an instrumentation amp IC). It's also used on Raspberry Pi for attaching ADC expanders since Pi lacks analog inputs. Just note that I^2C is not ideal for high bandwidth waveforms (like high-fidelity audio) due to limited speed.

- **USB:** While not typically a direct sensor interface, USB is the bridge to PCs. Arduino Leonardo and newer boards can act as native USB devices (e.g. USB MIDI or HID). For most Arduinos, the USB port is actually a USB-serial converter tied to UART. USB allows fast data transfer to a computer (12 Mbit/s for USB 1.1 full-speed, 480 Mbit/s for USB 2.0 high-speed). If you're streaming real-time data (like an EEG or high-frequency audio) to a PC for processing, you'll utilize USB either via serial or a custom driver. USB is complex (in terms of protocol stack), but many microcontrollers have libraries to send data easily (e.g. as a virtual COM port). Single-board computers like Raspberry Pi have USB host ports where you can connect peripherals (like an Arduino or a USB audio interface).

- **BLE (Bluetooth Low Energy):** BLE is a wireless protocol often used to send sensor data to smartphones or PCs without cables. Many IoT projects use BLE modules. Typically, a BLE module will interface with your microcontroller via UART or SPI (abstracting the BLE communication). For example, an Arduino can send sensor readings over UART to a BLE module (like HM-10 or an Adafruit BLE friend), and the module takes care of radio communication, so your data can be received by a phone app. BLE is suitable for relatively low data rates (tens of kB/s) which is fine for bio-signals or control signals, but not for high-quality audio streaming (classic Bluetooth or BLE Audio standard would be needed there). BLE's advantage is low power and direct compatibility with mobile devices (e.g. streaming heart rate or EEG data to a phone app for visualization).

In summary, **choose the protocol based on speed and complexity needs**: Use UART for simple point-to-point links and debugging, I^2C for multiple slow sensors, SPI for fast ADC/DAC or high-rate data, USB for connecting to computers with high throughput, and BLE (or Wi-Fi) if you need wireless freedom.

Interfacing Analog Sensors and Signal Sources (Arduino, ESP32, Raspberry Pi)

Arduino (AVR-based) boards have multiple analog input pins that you can read with `analogRead()`. For example, on the Arduino Uno you get 6 analog inputs (ADC channels) multiplexed into one ADC. The process is straightforward: after setting up any necessary reference (default 5 V or using

`analogReference()` for 1.1 V or external), you call `analogRead(A0)` and get a number 0–1023. Reading analog sensors like potentiometers, light-dependent resistors (with a resistor forming a divider), or analog accelerometers involves connecting them in the appropriate circuit (often as part of a voltage divider or supplying them with the Arduino's 5 V and reading the output). Because the ADC is relative to the board's ground and V_ref, you must ensure the sensor's output also shares that ground and stays within 0–V_ref. Most simple analog sensors (thermistors, photocells, etc.) output 0–5 V if wired correctly with resistors, so they can connect directly to an analog pin.

For **biopotential or other low-level signals**, as we discussed, you will have an analog front-end circuit (with op amp amplification and filtering) between the electrodes/sensor and the Arduino's analog input. That circuit likely needs its own power rails (which could be the Arduino's 5 V and 3.3 V, etc.) and must output within 0–5 V. One must also consider the Arduino's ADC input impedance and sampling time – if your source impedance is high, you might do an `analogRead()` twice and discard the first reading (allowing the sample/hold capacitor to charge). Or better, buffer the output with an op amp.

ESP32 boards similarly provide analog inputs (often labeled VP, VN, or ADC1_CH0, etc. depending on the pin). The ESP32 ADC is 12-bit, but note it's not as linear and noise-free as one might expect; some calibration is needed for accuracy. The input range by default is 0–1.1 V (the ESP has an internal attenuator you can configure to read higher voltages up to 3.3 V). This means if you connect a sensor directly expecting 0–3.3 V, you should set the ADC attenuation appropriately in software. The ESP32 also has some analog quirks (different attenuation settings and non-linearity at edges), but generally it can handle typical sensors. It even has **two DAC outputs**, which can be used to generate analog voltages (e.g. for audio output or biasing).

Interfacing sensors with the **Raspberry Pi** (which has no built-in ADC on models like Pi 4, Zero, etc.) requires external hardware. A common beginner approach is to use an **MCP3008** (10-bit, 8-channel ADC over SPI) or an **ADS1115** (16-bit, 4-channel ADC over I^2C) with the Pi. These chips connect to Pi's SPI or I^2C pins, and you use a library to read analog values. For example, you might wire a potentiometer to ADS1115 and use an Adafruit Python library on the Pi to get the reading. Raspberry Pi Pico (RP2040 microcontroller) *does* have ADC inputs, but the question context seems more about using a full Pi running an OS, which lacks direct analog input. So for Pi + analog sensor, plan on an ADC chip or using an intermediary like an Arduino to collect analog data and then send to Pi digitally (sometimes done via serial or I^2C bridging).

Interfacing a guitar to microcontroller illustrates a lot of the above concepts: A guitar pickup output can be hundreds of millivolts AC. To feed it to an Arduino: - Use a resistor divider or bias network to shift it to mid-range, - Possibly amplify it with an op amp if needed (though Arduino's 5 V range and 10-bit resolution can cover a few hundred mV – but resolution will be poor if the signal only uses a small portion of the ADC range), - Ensure the impedance is low (a guitar pickup is high impedance, ~100kΩ source, so definitely buffer with a unity-gain op amp or a JFET transistor as an emitter/source follower to avoid loading the pickup), - Then sample at a sufficient rate. The Arduino at 9.6 kHz can just about capture up to ~4.8 kHz (which covers fundamental guitar tones but may miss higher harmonics). An alternative is to use the **ADC in free-running mode** at a higher rate or use an ESP32 (which can achieve higher sampling, albeit with noise) or an external ADC.

Data Logging: Sometimes your microcontroller will store analog data or send it out. For Arduino, you might add an SD card module (using the SPI interface) and write sampled data to a CSV file or similar. There are Arduino libraries for SD card writing. Keep in mind writing to SD is relatively slow (and can pause your sampling if not careful), so often a better approach is streaming to a PC for logging (if continuous high-rate data is needed). For lower rates or intermittent logging (e.g. log temperature

every second), writing to SD is fine. On an SBC like Raspberry Pi, you have the full file system, so you can directly write sensor readings to a file or database.

Analog Signal Acquisition and Data Streaming

Once analog data is in your microcontroller, you often want to **get it to a computer or other system for storage, analysis, or real-time use**. There are several methods to do this, which blend into the next section (integration with laptop software):

- **Serial over USB:** The simplest is using the microcontroller's serial port to send data to a PC via the USB cable. Arduino, for example, can `Serial.print()` values (perhaps comma-separated) which you can read on the PC with a terminal or a program (via a COM port). At moderate baud rates (9600–115200 bps), this is a convenient way to log data in a CSV format that can be copied into Excel or MATLAB later. For higher throughput (say streaming audio), you might push the baud to 1M baud or use native USB CDC (some Arduinos can act as USB serial directly). On the PC side, software like the Arduino IDE serial monitor, PuTTY, or a custom script (Python with PySerial) can read and save the data.

- **Wireless Streaming:** If wired connection is not ideal (say you want to move freely with a wearable sensor), you can use Bluetooth or Wi-Fi. An ESP32 can directly send data over Wi-Fi (e.g. run a small web server or UDP stream sending sensor readings). You could also publish data to MQTT or similar so a PC can subscribe. Bluetooth Classic could emulate a serial port (SPP) for continuous data; BLE can use a custom service characteristic to notify new data to a PC/phone app.

- **Real-time considerations:** If continuous high-rate data is being streamed, ensure your microcontroller loop can keep up and that you don't overflow any buffers. Sometimes a small microcontroller cannot handle acquiring data and simultaneously formatting/streaming at very high rates. Tactics like using circular buffers, DMA (if available, as on some ADCs or UARTs), or simply lowering the data rate to what's necessary (do you really need every sample or can you downsample?) will help. For example, an ECG at 250 Hz sampling could be streamed easily (250 samples/sec, maybe 500 bytes/sec) which is trivial. But an audio signal at 44,100 Hz 16-bit stereo is ~176 kB/sec, which is too much for Arduino serial; that scenario needs a specialized approach (like an audio codec and storing to SD or using a faster microcontroller). Always tailor the approach to the bandwidth needed.

By understanding your ADC and using proper conditioning, you ensure the analog world is translated into the digital domain accurately. With robust interfacing via the appropriate protocol, your microcontroller or SBC can then convey that data to wherever it needs to go next – whether that's immediate display on a laptop, storage on flash, or input to an algorithm for further processing or control.

Integrating with Laptop Software

Many projects require the microcontroller or analog device to communicate with a laptop for data visualization, processing, or to interface with multimedia software. Here's how you can integrate analog hardware data with higher-level software environments:

Serial Communication with Python, MATLAB, and Max/MSP

One of the most common integration methods is through serial communication (over a USB virtual COM port). **Python** is a popular choice for reading serial data because of the `pySerial` library, which makes it easy to open the COM port and read/write data. For example, if your Arduino is sending comma-separated values of sensor readings, a Python script using PySerial can continuously read from the serial port and parse those values for real-time plotting or analysis. *"PySerial is a Python API module used to read and write serial data to Arduino or any other microcontroller"* [29]. You would configure the port (e.g. `COM3` on Windows or `/dev/ttyACM0` on Linux) at the same baud rate. Python can then feed this data into libraries like Matplotlib for plotting or save to a file.

MATLAB has built-in support for serial ports as well. One can use the `serialport` function (in recent versions) to create a serial port object and set a callback or loop to read incoming data [30]. MATLAB is very powerful for analyzing and visualizing data once imported. For instance, you might stream EEG data into MATLAB and perform an FFT or filter in real time. MathWorks also provides an Arduino Support Package that can automatically handle reading analog pins from Arduino and bringing the data into MATLAB as a variable, without you writing the Arduino code (the package handles communication under the hood). This can be convenient for quick setups. However, using the straightforward serial print + fscanf in MATLAB approach gives you more control. MATLAB, being a numerical computing environment, is suited for processing blocks of data (like computing heart rate from ECG peaks, etc.) once the data is acquired. It's not the best at low-latency interactive response (something like Python or dedicated programs might be better for immediate real-time feedback), but for analysis and visualization it works well. MathWorks documentation has examples, e.g., *"Read streaming data from Arduino using serial port"* which demonstrates configuring a serialport object and reading ASCII-terminated data continuously [31].

Max/MSP (by Cycling '74) is a visual programming environment widely used in music and multimedia art for real-time audio/visual processing. Max can interface with hardware using a **serial object** as well [32]. For example, if you have sensors on Arduino and you want them to control sound or graphics in Max, you can send the data via serial USB and use Max's `[serial]` object to receive it. The Max serial object lets you specify the port and baud rate, and it outputs the incoming bytes which you can then parse in the Max patch. Max patches often use `select` or `route` objects to parse structured messages from Arduino (for instance, you might send data in the format "sensor1 523\n" and have Max route the number after the label). An important note: only one program can open a serial port at a time – so if Max is connected to the Arduino, you can't also have the Arduino IDE serial monitor open, etc. [33]. In practice, many artists use the combo of Arduino + Max (sometimes referred to as "Arduino2Max" technique) where the Arduino acts as an I/O interface for sensors, and Max/MSP does the heavy audio/visual work on the computer. The Max `[serial]` object must be polled to get data [34], meaning you bang it periodically to retrieve bytes (though newer versions allow event-driven reads). Once data is in Max, you can map sensor values to MIDI events, audio filter parameters, visuals, etc.

Additionally, there's **Firmata** – a protocol that allows a PC to control/read Arduino I/O pins directly. With **pyFirmata** in Python or Max's `maxuino` externals, you can avoid writing custom Arduino code. Instead, you upload the StandardFirmata firmware to Arduino, and then a Python script or Max patch can send commands like "read analog pin A0" or "set digital pin 13 HIGH". This is handy for quick experiments. For example, PyFirmata allows using Python to do `analog[0].read()` which will give the value on A0 (via continuous polling in the background). Maxuino does similar for Max/MSP. The trade-off is that Firmata can have some latency and overhead, so for time-critical tasks or high data rates, a custom, slim protocol might be better.

Visualization Tools and Real-Time Plotting

For understanding and presenting your analog data, visualization is key. If you just want a quick real-time graph, the Arduino IDE itself has a Serial Plotter which will plot numeric values it receives on the serial port. But for more control and multi-channel plotting:

- **Python + Matplotlib (or Plotly/Dash):** With Python, you can use Matplotlib to update a live graph of incoming data. This could be done with an animation function that periodically updates the plot with the latest points. For example, plotting an ECG waveform scrolling in real time is feasible by continually reading serial data into a buffer and re-drawing. There are higher-level libraries like PyQtGraph (which is optimized for real-time plotting) that can handle faster update rates smoothly. Another modern approach is using Plotly or Dash to create a simple web dashboard that updates with the data – that way you could view sensor data in a web browser interface.

- **Processing (Java):** Processing is a programming environment geared towards visual arts, but it has a Serial library that can read Arduino data easily. Many earlier DIY EEG or sensor projects used a Processing sketch on the PC to visualize data (e.g. drawing waveforms or animations responding to sensors). It's fairly straightforward: in Processing, you open the serial port and then inside `draw()` you parse any new data and draw shapes or lines accordingly. Processing is great for making custom visuals that respond in real-time to analog inputs – like a custom oscilloscope or a graphical representation of muscle activity.

- **Matlab/Simulink:** Matlab not only can plot data after the fact, but also has tools like Simulink with real-time windows, or the Data Acquisition Toolbox where you can see live plots. If using the Arduino support, Matlab even has a GUI called *Serial Plotter* similar to Arduino's but more flexible. For more engineering-oriented tasks (like monitoring an analog sensor in a lab setup), Matlab's plotting with the ability to add annotations, do spectral analysis, etc., can be useful. The example *"Log Temperature Data from Arduino into MATLAB"* [35] shows how one can bring data in and visualize/log it without writing C code on Arduino.

- **Max/MSP and Ableton Live (via Max for Live):** In Max, you can visualize sensor values using GUI objects (numbers, sliders, multislider for plotting history, etc.). If you're turning analog sensor data into music, sometimes you also want to monitor the data. Max can display it or even create audio-reactive visuals. Additionally, **Ableton Live**, a popular music production software, allows integration via **Max for Live** devices. One could create a Max for Live patch that reads Arduino sensor data (through Max's serial) and then use it to control Ableton parameters or MIDI. For example, a musician might use a glove with flex sensors (analog inputs on Arduino) to send data to Ableton to control effects – the data can be smoothed and scaled in Max and then mapped to sound.

- **Oscilloscopes/Analyzers:** For very high-speed analog signals, a PC might not be the primary tool (one would use an oscilloscope or logic analyzer). However, you can treat a sound card as a kind of ADC for certain analog signals (there are PC oscilloscope programs that use the audio input for signals up to ~20 kHz). This is tangential, but worth noting: if you need to visualize an analog waveform that's within audio range, sometimes plugging it into a PC's line-in (with proper attenuation and protection) and using audio spectrum software can substitute for fancier ADC setups.

The main point is **real-time feedback**: seeing analog data live helps tremendously in debugging and analysis. Whether it's seeing your EEG alpha waves on a graph or observing the envelope of a guitar

strum, visualization turns numbers into insight. Plan to include either a simple plotting utility or build one appropriate to your needs (many DIY projects include a small Processing or Python script precisely for this).

MIDI, OSC, and Audio Interfacing for Music Applications

If your projects involve music (like the guitar signal processing or synthesizer control) or interactive media, you'll often interface analog hardware to laptop software not just as data, but as control messages in protocols like MIDI or OSC, or even as audio signals:

- **MIDI (Musical Instrument Digital Interface):** MIDI is a digital protocol designed for music devices, widely used for synthesizers, DAWs, controllers, etc. It encodes events like "Note On, channel 1, note 64, velocity 100" or continuous controller values 0–127. A common desire is to turn analog sensor readings into MIDI messages so they can directly control music software (e.g. turn a flex sensor into a MIDI CC that maps to a filter cutoff in Ableton Live). Traditional MIDI uses a UART at 31,250 bps with a specific byte format, but these days most MIDI goes over USB (USB-MIDI class devices). **Arduino as MIDI controller:** If you use an Arduino Leonardo, Micro, or Teensy (which have USB HID capability), you can have it show up as a MIDI device on the PC, and then you can send MIDI messages when certain analog conditions are met. For example, reading a potentiometer and sending MIDI CC 14 with value 0–127 corresponding to the analog value [36]. If you have an Uno (which doesn't have native USB MIDI), you can still send MIDI data over the serial and use a software "serial-MIDI bridge" on the PC (like the Hairless MIDI serial bridge) to route it to a MIDI port. Alternatively, some people use the Arduino's UART to output the 5 V MIDI signal (via an optocoupler interface as per the MIDI spec) to physical MIDI IN of another device – but with computers, USB MIDI is easier.

The range 0–127 comes from MIDI's 7-bit data for most values. Indeed, *"in MIDI 1.0, all data was in 7-bit values… quantized on a scale of 0 to 127"* [37]. So typically one maps an analog 0–1023 reading to 0–127 by dividing by 8 (or using Arduino's `map()` function). Numerous DIY projects exist for sensors-to-MIDI; for instance, a piezo sensor can be a drum trigger sending a MIDI note, or an analog distance sensor can send CC messages. Arduino code can use libraries like MIDI.h to format messages easily.

- **OSC (Open Sound Control):** OSC is a modern protocol often used in creative coding and multimedia. It's like MIDI in purpose (control messages), but it's built on network protocols (UDP usually) and is far more flexible in data types and naming. For example, you might send an OSC message `/sensor/eyeblink 0.8` to indicate 80% blink strength. OSC is widely used in environments like Max/MSP, PureData, SuperCollider, and can easily interface between devices over Wi-Fi or Ethernet. If you have a Wi-Fi-enabled board (ESP32 or a Raspberry Pi), you can send OSC messages directly to a PC on the same network. For instance, an ESP32 could stream accelerometer data as OSC to a PC running a dance performance visuals program. OSC is considered a *"spiritual successor to MIDI"*, being more flexible and working over wired or wireless networks [38]. There are OSC libraries for Arduino and definitely for Python and other PC languages. A common approach is using a Python script to convert serial data from Arduino into OSC messages that are then picked up in programs like Processing or Max. Some tools like TouchOSC on smartphones allow sending phone sensor data as OSC to PC [39], illustrating how easily sensor data can be routed with OSC.

- **Audio Interfaces:** If your project involves actual audio signal processing (like using the laptop to apply effects to a guitar signal in real-time, or doing synth on the microcontroller), you might treat the microcontroller or circuit as an **audio interface** input. For example, you could connect the output of an analog preamp (say a guitar preamp circuit or EEG amplifier) to the line input of

a USB audio interface on the computer. Then you can use real audio processing software (DAW, Audacity, MATLAB DSP System Toolbox, etc.) on that signal. However, embedding a full USB audio interface into your microcontroller project is complex – usually it's easier to use an existing interface. The exception is something like a Teensy microcontroller which can act as a USB audio class device and stream audio samples to the PC (Teensy has an Audio library that can make it appear as a 16-bit 44kHz stereo sound card). For an Arduino Uno, this is not feasible due to speed.

Alternatively, use the microcontroller to **analyze audio and send control** rather than digitizing the whole audio. For instance, a guitar-to-MIDI converter: Instead of sending the raw guitar audio, the Arduino could run a pitch detection algorithm and then send MIDI notes of the detected pitches to the PC (there are projects that attempt this, but doing it reliably is challenging on basic Arduinos). The Arduino MKR Zero and others have been used for simple audio processing (FFT to detect frequency then output nearest MIDI note) [40]. The Arduino example *"Analog to Midi"* demonstrates recognizing an input frequency and outputting the nearest MIDI note [41], essentially a primitive guitar synthesizer approach.

If you want to incorporate laptop-based effects processing *for audio*, an alternative is to feed the analog audio into the laptop's audio input (through an appropriate ADC or interface) and use software like Pure Data or VST plugins to process it, while the Arduino handles other control tasks. For example, an Arduino might control effect parameters via MIDI, while the guitar's actual sound goes through an audio interface to the PC running an amp simulation (like Guitar Rig or VST host).

- **Software Integration:** Environments like **Max/MSP, Pure Data, VCV Rack, or SuperCollider** can mix and match these approaches. E.g., Max can receive analog sensor data via serial, convert it to OSC or MIDI to control other software, and also generate sound itself. If building a custom synthesizer controlled by sensors, you might choose to do the sound generation on the PC for quality and flexibility, using the microcontroller only as a sensor interface.

- **Calibration and Scaling:** Whichever method, some calibration might be needed so the range of sensor values maps nicely to meaningful control changes. E.g., a flex sensor might output 300–700 as ADC readings; you'd map that to MIDI 0–127 but maybe invert it or add an exponential curve for musical responsiveness. This can be done either on the microcontroller before sending (e.g. use `map()` or a lookup table) or on the PC side (in Max or your Python script). In Max, for instance, you could scale and limit ranges easily with objects like `scale` or simple arithmetic, and even smooth data with line or slide objects to avoid jitter.

In summary, **MIDI and OSC allow your analog world to talk the language of digital media software**. MIDI is immediately useful if you want to interface with music production software (almost all of which speak MIDI). OSC is great for custom setups especially across networks (e.g. several devices sending data to one performance computer). Both can be generated by microcontrollers (with the right libraries or firmware). Meanwhile, **audio interfacing** is about getting the actual analog signals (or their digitally sampled form) into the computer for direct processing as sound, which typically involves specialized ADCs or treating the microcontroller as an audio streamer (which only more advanced boards can do effectively). Depending on project goals, you might end up using a combination: for example, an eye-tracking Arduino might send OSC messages of gaze position to control visual software, while a microphone's analog audio goes into the PC's audio input for analysis, and an Arduino output triggers a MIDI note when a blink is detected. The possibilities are endless once you know how to get analog data in and out of the digital realm.

Case Studies and Application Examples

To solidify these concepts, let's look at a few concrete examples that tie together analog design and integration steps, tailored to the user's interests:

DIY EEG/ECG with Arduino

Scenario: You want to build a simple EEG or ECG acquisition system using an Arduino to read signals from the body, then visualize them on a PC and possibly analyze them for patterns (like alpha waves or heart rate).

Analog Front-End: Start with electrodes placed on the body (for ECG, typically two on the arms or chest for differential measurement plus one reference on the leg; for EEG, maybe one on the forehead vs an ear lobe reference). The electrode leads go into a carefully designed analog front-end. Use an **Instrumentation Amplifier** like AD8232 (a chip specifically for heart-rate ECG), or build one from an op amp (though IA chip is easier and safer). The IA will provide high gain (~100x) and high CMRR as discussed. Follow this with a **high-pass filter ~0.5 Hz** to remove DC offset (especially important for EEG where electrode offsets can be several tens of millivolts which, after 1000x gain, would rail the amplifier). Next, add a **second gain stage + low-pass filter**: for EEG, maybe a gain of 10 and low-pass at 40 Hz; for ECG, gain of 5 and low-pass at 100 Hz. You might also incorporate a **50/60 Hz notch filter** here if mains hum is an issue (many hobby ECG circuits include a twin-T notch at 60 Hz, though as mentioned some prefer not to notch EEG to avoid phase distortion, opting for digital notch later).

Make sure to **bias the signals mid-supply** if using single-supply op amps and Arduino. The AD8232, for instance, outputs ECG centered at 1.5 V (assuming 3.3 V supply). If building from scratch with dual supply, you can center around 0, but then you need to shift before feeding Arduino. Often, a **simple resistive divider to 2.5 V and buffering** is used as a virtual ground for all amplifiers in a single-supply design. All signal paths are then referenced to this virtual 2.5 V instead of true ground.

Provide **isolation** by powering the analog front-end from a battery (e.g. a 9 V battery regulated down to ±5 V dual rails, or +5 V and virtual ground). This ensures no direct path from mains to the subject. If using the Arduino connected to a PC, it's good practice to only have optical or wireless connection from the Arduino to PC when a person is wired up, or at least be mindful of the ground path (some use an isolated USB or laptop running on battery). Safety first!

Acquisition with Arduino: The conditioned analog output goes to an Arduino analog pin. Use the Arduino ADC to sample at, say, 200 Hz for EEG (which is plenty for <50 Hz content) or 500 Hz for ECG. You can use a timer interrupt to get consistent sampling intervals. Store the readings (perhaps 128 samples buffer) and send them via serial to the PC.

Data Transmission: The Arduino can stream the data as CSV or binary. A simple approach: `Serial.println(value);` at 250 Hz. That's 250 lines per second, which at even 5 bytes each is 1250 bytes/sec, well within serial at 115200 baud (~11k bytes/sec capacity). For EEG, some projects like OpenBCI use a binary packet format to send multiple channels efficiently. In our simple case, one channel CSV is fine.

Software: On the PC, you could have Processing or Python reading the values and plotting an EEG graph that scrolls. For instance, the **"Brain Grapher"** from OpenBCI (Processing-based) plots EEG in real time. Even the Arduino Serial Plotter can show the waveform. To extract meaningful info, you might implement a band-pass or FFT in Python to detect certain frequency bands (alpha, beta waves). For ECG,

you might do peak detection to measure heart rate (the R-peaks in ECG are prominent). Python with SciPy could detect peaks or you could even do a simple threshold crossing in the Arduino and output inter-beat intervals.

Case Example: A known DIY EEG project by Frontier Nerds (Chris's EEG) used an op amp (TL072) to build a two-stage amplifier with ~G=tot 12,000 and then read via Arduino. They successfully plotted alpha wave (8–13 Hz) increases when the user closed eyes. The key components were: electrodes (they used conductive adhesive electrodes), an instrumentation amp made from three op amps (or could be single IA chip), a 60 Hz notch, and careful shielding (they even built a little foil Faraday cage around the head). They sent data to Processing to visualize [42] [43] . They also used **Electrooculography (EOG)** signals (eye movement potentials) as a side experiment – interestingly, EOG can be picked up on EEG electrodes as slow deflections when eyes move [42] . With a straightforward high-pass, one can separate EEG vs EOG.

Result: Using such a DIY EEG/ECG, you learn about analog filtering (e.g. "why is my signal saturating? oh, increase the high-pass cutoff to remove electrode DC offset" or "50 Hz noise too high? try a notch or better shielding"). You also practice grounding: using a **common reference electrode** tied to bias (which the AD8232, for example, drives to suppress common-mode). You will see that *"small difference in electrode potentials produces a measurable voltage when the eyes move, called an electrooculographic (EOG) signal"* [42] – basically verifying that instrumentation amps can capture even tiny bio-potentials.

Finally, sending to a laptop and plotting closes the loop, showing in real-time the analog phenomena from your body. Extending this, one could use the data for a simple BCI (e.g. blink twice to trigger an action – an EOG blink is a big spike you can detect in code).

Analog Guitar Effects and Audio-to-MIDI Interfaces

Scenario: You want to process a guitar signal, maybe create an effect like distortion or wah, or convert the guitar audio into MIDI notes to drive a synth on your laptop.

Guitar Preamp and Buffer: As mentioned, the first step is buffering the guitar's signal. Electric guitar pickups are high impedance (~100kΩ-1MΩ). A classic rule: *"the input impedance of a pedal should be 1 MΩ minimum"* to avoid tone sucking [44] . So you design a **buffer** (could be a simple op amp voltage follower or an emitter follower transistor) with around 1 MΩ input impedance. This ensures the guitar's full frequency spectrum (especially the high frequencies which would be lost with low impedance load) passes through. Many guitar pedals put a FET input op amp or transistor at the very front just for this reason.

Effects (Analog): If doing analog effects on Arduino might be limited (Arduino can't do real-time DSP for high-quality audio), you can implement analog circuits for distortion, tone control, etc., before or after the Arduino stage: - A **distortion/overdrive** can be done by op amp clipping: e.g., the classic MXR Distortion+ circuit uses a non-inverting op amp with diodes in the feedback to clip the waveform, producing rich harmonics. You could recreate that or use a ready-made pedal kit. The analog output of that could then be fed to a microcontroller if needed or directly to an amp. - A **wah filter** is basically a band-pass filter whose center frequency is sweepable (traditionally by a foot pedal moving a pot). You could analog-build it or use the Arduino to control a digital potentiometer that sweeps an analog filter circuit. - If you want to **digitize the guitar** for processing in code (like building your own digital effects on an Arduino or sending to PC), note that standard Arduino's 10-bit at ~9 kHz sampling is not sufficient for good audio quality. It might be okay for a lo-fi effect or for detecting notes, but not for high fidelity sound. If the goal is to do heavy DSP, consider an ESP32 (which can do ~12-bit at higher rate, or even use the I2S interface for an external audio codec IC).

Audio to MIDI (Pitch Detection): Converting a monophonic guitar melody to MIDI involves detecting the pitch of the note being played. One approach: perform an FFT or zero-crossing period measurement on the incoming audio. The Arduino Uno might struggle with FFT on audio in real time. But a simpler method is *zero-crossing detection with a period averaging.* Essentially: - Filter the guitar signal to a single waveform (maybe a band-pass 80 Hz–1 kHz to isolate fundamentals). - Convert it to a pulse (schmitt trigger) and measure the time between zero-crossings or positive peaks. - Compute frequency = 1/ period. Then map that frequency to the nearest MIDI note number (there's a known formula: MIDI note 69 = A4 = 440 Hz; each semitone is 2^(1/12) frequency ratio). - Send a MIDI Note On for that note. Possibly also derive velocity from amplitude (e.g. how strong was the pluck from the signal's peak magnitude).

There are Arduino projects that attempt this; results are mixed because guitar tones have many harmonics and the fundamental might be weak on certain notes, causing octave errors. Smoothing and median filtering the measured period helps. Alternatively, one could use autocorrelation to find the fundamental frequency. The Arduino MKR1000 project *"Analog to Midi"* hints at doing an input frequency recognition and outputting nearest MIDI note [41] .

Practically, if you send these MIDI notes via serial or USB to a synth on the PC, you can effectively have your guitar play synthesized instruments. There's commercial products (e.g. Sonuus G2M) that do exactly this, with some latency. A powerful microcontroller (or offloading to PC via audio interface and doing pitch-detect in software like Ableton or JamOrigin's MIDI Guitar software) yields better results.

Integration: If you built an analog distortion pedal and want to integrate with PC, you might just take the **output audio into the PC** (through line-in or an audio interface) and record it or further process it with VST effects. On the other hand, if you built a sensor-laden guitar (say you put accelerometers or force sensors on it), those analog sensors could feed an Arduino and then into the PC as MIDI/OSC to modulate effects in real-time (e.g. tilt guitar to control wah – this is very feasible: use an accelerometer analog outputs to Arduino, send OSC to Max which controls a wah filter on the guitar's audio stream).

Example use-case: A guitarist wants to control delay time by how hard they press on the guitar body. They attach a piezo disc (acting as pressure sensor) to the guitar, feed that analog signal to Arduino, Arduino measures intensity and sends MIDI CC messages. In the DAW, the delay effect is mapped to that MIDI CC. Now the guitar becomes an expressive controller beyond just playing notes.

In summary, analog design for guitar projects means respecting the audio nature: use proper impedance buffers, filter out noise (guitar cables can pick up hum, so shielding and maybe a notch for 60 Hz if needed), and ensure no extreme levels hit the ADC (clipping an ADC sounds very unpleasant digitally – if you want distortion, do it in analog or with proper digital handling, not by over-ranging the ADC unintentionally). Finally, leverage protocols to interface with music software: *MIDI* for note/control, or even have the Arduino enumerate as a USB MIDI device for plug-and-play use with any DAW.

Eye-Tracking Applications (EOG and IR Sensing)

Scenario: You want to track eye movements for a human-computer interface or for analyzing drowsiness (common in automotive research) – possibly using Arduino or Raspberry Pi and either analog sensor techniques or a camera.

There are two primary ways to do DIY eye tracking without fancy cameras: **Electrooculography (EOG)** and **infrared optical tracking**.

Electrooculography (EOG): The eye maintains a steady electric potential (cornea is positive relative to retina). When the eye moves, this potential shift can be measured with electrodes placed around the eye. Essentially, the eye is like a dipole; moving it changes the relative voltages at the electrodes [42]. For example, an electrode on the left of the eye and one on the right will see a voltage difference that varies with horizontal eye movement. Likewise, electrodes above and below the eye measure vertical movement. These EOG signals are fairly slow and in the tens to hundreds of microvolts.

Design: An **instrumentation amplifier** similar to EEG is used. Often a single AD620 or INA128 can amplify the difference between two electrodes placed around the eye. If measuring horizontal movement, you might place one electrode at the temple beside the left eye, one at the temple beside the right eye (or on the forehead toward the right). When eyes move toward one electrode, that electrode sees the cornea (positive) more directly, the other sees more of the retina (negative), creating a voltage difference [42]. Moving the eyes in the opposite direction flips the polarity of the voltage difference. This differential signal is then amplified and filtered (EOG bandwidth is roughly 0.1 – 10 Hz for movement, blinks add high-frequency spikes).

A typical EOG amplifier might have a gain of 1000, a high-pass around 0.1 Hz (to remove DC drift), and a low-pass around 30 Hz [43] (since we only care about relatively slow eye motion and perhaps blink transients). Using three electrodes (horizontal pair plus ground on forehead) you can get horizontal movement. For 2D tracking, you'd need another pair for vertical or arrange 4 around the eye (left, right, up, down around the orbit) and do some vector math.

The analog output representing, say, horizontal gaze position (after appropriate scaling) can be read by Arduino ADC. One could map this to an onscreen pointer position. However, EOG is not very precise – it tells you general direction of gaze, not exactly where someone is looking on a screen, and it's prone to drift. It is, however, relatively easy to implement with just analog parts and is used in some assistive tech for paralyzed patients to detect eye gestures (like looking left-right to spell out messages).

Example: A project uses EOG to control a computer cursor. Electrodes on temples feed an AD8221 IA, output goes through a 2-pole active filter. Arduino samples it at ~50 Hz and sends the values to a Processing sketch which moves a dot on screen left or right proportionally. With some calibration, the user learns to control the dot by eye movements. The data from EOG is "noisy" and drifts, so the software might implement a baseline calibration and perhaps only detect large deflections (like deliberate eye gestures) rather than try to continuously map angle. Indeed, an EOG signal is roughly linear with eye angle within a certain range (±30°), but can saturate or become non-linear beyond that [45].

Alternatively, EOG can simply detect events: e.g., a blink produces a distinctive rapid upward deflection (as the eye rolls up during blink) – this could be a trigger (like clicking via blink). In one example, blinking in a certain pattern is used as a switch input (Arduinos can easily detect the blink spike in EOG channel and then send a keystroke via a connected PC).

Infrared (IR) Eye Tracker: This approach uses optical reflection. A common DIY method is to use a pair of IR LED-phototransistor or photodiode combos placed near the eye. Typically, an IR LED illuminates the eye (often from near the glasses frame), and a photodiode next to it measures reflected IR. As the eye moves, the amount of white sclera vs dark pupil in front of the sensor changes the reflection. For instance, one instructable used two QTR-1A reflectance sensors (which have IR LED and phototransistor) on either side of an eye on a pair of glasses [46]. When the iris moves towards one sensor, that sensor sees less IR reflection (because the dark iris/pupil absorbs IR) and its reading changes [47]. Conversely, the sensor on the opposite side sees more white sclera (highly reflective) and gets a stronger reflection. By comparing the two, you deduce direction.

Design: Each IR sensor gives an analog voltage (often phototransistors are used in a voltage divider). For example, a phototransistor might be arranged with a resistor so that output voltage increases with more IR reflected. The output is analog, varying as the eye moves [47] . With two sensors, you can do differential measurement in software: e.g., compute (Left - Right) or so. In practice, you might just feed both into two ADC channels on Arduino and do a simple comparison. The signals might need some filtering – if the LED is always on, ambient IR (sunlight) or quick changes might affect it. Some designs use a **modulated IR** approach: drive the LED with a 1 kHz square wave and use synchronous detection to filter out ambient light. But that's complex; many just assume indoor conditions where IR noise is limited.

Using such IR sensors, you can get a coarse eye position. It's not super high resolution but enough for e.g. detecting gaze direction (left/right) or maybe rudimentary tracking. It's been used in projects like controlling a mouse cursor: The resolution might allow, say, 3–5 discrete positions (left, center, right). For finer control, calibration and linearization might be needed.

Processing and Integration: With analog IR sensor outputs going into microcontroller ADCs, you can stream those to a PC or directly use them to control something. For instance, an Arduino could read the two sensor values and then act as a USB joystick or mouse by converting that into cursor movement (Arduino Leonardo can emulate a mouse and move it based on analog input). Or send the data to Processing which then moves a visual element. If controlling actual mouse pointer, one has to handle the centering and drift (some use a self-centering approach where when you look straight, Arduino continually adjusts to treat that as "center").

Camera-based Eye Tracking: Though not analog electronics per se, worth mentioning: using a Raspberry Pi with an IR camera and some open-source computer vision (OpenCV algorithms for pupil detection) can achieve eye tracking. You'd illuminate the eye with IR (to not distract the user) and use image processing to find the pupil center. This can give true 2D gaze tracking, but it's much more CPU-intensive. The analog methods above are simpler hardware-wise but give less precise info.

Use case example: A DIY drowsiness detector in a car: EOG could detect slow rolling eye movements or long blink durations. Alternatively, a pair of IR sensors mounted on glasses can detect if the eyes move erratically or not at all (blink patterns). The microcontroller can then trigger an alarm if signs of microsleeps are detected (like eyes drifting and blinking slowly).

Another example: An eye-controlled assistive device: An EOG setup with Arduino processes eye gestures – two quick glances to the left could mean "next", two right glances mean "select", enabling a user with limited mobility to communicate. The Arduino could interpret these and send keystrokes to a PC (e.g. via Bluetooth or USB). In fact, systems like the EyeWriter (low-cost eye tracker for paralyzed artist) combined glasses with IR sensors and software to allow eye-drawing.

Key takeaways: Eye tracking with analog means teaches you to amplify very small signals (EOG microvolts) and deal with offsets (there is often a slow drift in EOG baseline, so you might implement high-pass filtering or auto-zero algorithms). It also highlights the challenge of calibration: every user might have slightly different signal magnitude, so your system should have a way to calibrate (maybe ask the user to look left/right during setup to set the range). On the integration side, these eye signals often will be turned into control signals for software – here is where knowing about serial, or HID (human interface device) profiles, or OSC can be powerful. For example, you might send an OSC message `/eye/X 0.2` for left, `/eye/X -0.2` for right, to a Processing app that controls a game.

In conclusion, the above case studies demonstrate how to bring together the fundamentals and techniques discussed: from understanding component behavior (Ohm's law, filtering, amplification) to designing the analog front-end tailored to the signal, and finally interfacing that to digital systems (microcontrollers and PCs) using the appropriate communication and software tools. Whether it's capturing the whisper of neural signals, rocking out with interactive guitars, or guiding technology with the gaze of an eye, analog hardware design combined with modern microcontrollers opens up a world of creative electronics projects.

References: The information in this guide draws from electronics textbooks and application notes, such as Horowitz & Hill's *The Art of Electronics* (a comprehensive resource on analog design), and various manufacturer application guides (Analog Devices' **Analog Dialogue** articles on ECG front-ends [12] [14], TI's op amp design guides, etc.), as well as practical tutorials and community projects (e.g. SparkFun tutorials on analog concepts [48], Instructables projects for EEG [13] [42], guitar pedal analyses [44], and DIY eye tracker builds [47]). These resources, alongside datasheets for components like instrumentation amplifiers and op amps, are invaluable for deeper study. As you embark on building your own circuits, refer to those resources for schematics and detailed theory – but also be prepared for hands-on learning, as real-world analog behavior often teaches lessons beyond theory (like the art of grounding and the nuance of component tolerances). Happy building, and may your analog adventures interface smoothly with the digital world! [13] [7]

---

1 Ohm’s law | Physics, Electric Current, Voltage | Britannica
https://www.britannica.com/science/Ohms-law

2 3 8 Analog Circuits: Ohm's Law, Basic Concepts & Examples - NI
https://www.ni.com/en/shop/data-acquisition/measurement-fundamentals/analog-fundamentals/basic-analog-circuits.html?srsltid=AfmBOorHWtVx-Ft7ny7GUX8aGZwUVbDoPNswjxoMfCSDOqir3xPK-44K

4 web.stanford.edu
https://web.stanford.edu/class/archive/engr/engr40m.1178/slides/reactives.pdf

5 6 Analog vs. Digital Signals: Uses, Advantages and Disadvantages | Article | MPS
https://www.monolithicpower.com/en/learning/resources/analog-vs-digital-signal?srsltid=AfmBOoo1G4-IZ_KdiBuVrPSFNMeKuWpiAwclj5SJkqy0Zdb_8i0OHHfd

7 12 14 ECG Front-End Design is Simplified with MicroConverter® | Analog Devices
https://www.analog.com/en/resources/analog-dialogue/articles/ecg-front-end-design-simplified.html

9 Operational Amplifier Basics, Types and Uses| Article | MPS
https://www.monolithicpower.com/en/learning/resources/operational-amplifiers?srsltid=AfmBOor8jr0e9gIQ-X_tKoF9ztmAFYeAKUPOl4U58CLYQKKMRIYRc1Mp

10 28 ElectroSmash - MXR Distortion + Circuit Analysis.
https://www.electrosmash.com/mxr-distortion-plus-analysis

11 What is a notch filter in EEG? - Quora
https://www.quora.com/What-is-a-notch-filter-in-EEG

13 16 17 Super Simple Electrocardiogram (ECG) Circuit : 11 Steps (with Pictures) - Instructables
https://www.instructables.com/Super-Simple-Electrocardiogram-ECG-Circuit/

15 I built my EEG circuit on a breadboard and wanted to see if it works
https://ez.analog.com/amplifiers/instrumentation-amplifiers/f/q-a/14251/i-built-my-eeg-circuit-on-a-breadboard-and-wanted-to-see-if-it-works

18 How to properly shield ground analog signals : r/PLC - Reddit
https://www.reddit.com/r/PLC/comments/17znx0y/how_to_properly_shield_ground_analog_signals/

19 Why grounding the shield helps reducing noise? | All About Circuits
https://forum.allaboutcircuits.com/threads/why-grounding-the-shield-helps-reducing-noise.109271/

20 [PDF] How to Exclude Interference-Type Noise Application Note (AN-347)
https://www.analog.com/AN-347

21 27 Analogue to Digital Converter (ADC) Basics - EE Times
https://www.eetimes.com/analog-to-digital-converters/

22 23 24 25 Acquiring an Analog Signal: Bandwidth, Nyquist Sampling Theorem, and Aliasing - NI
https://www.ni.com/en/shop/data-acquisition/measurement-fundamentals/analog-fundamentals/acquiring-an-analog-signal--bandwidth--nyquist-sampling-theorem-.html?srsltid=AfmBOoqFWPs_7LtjQ-3j4ybqQ0pB2MgoCigbVgNSqaub1iMpudmJp83K

26 Yet another FHT thread, the basics - Audio - Arduino Forum
https://forum.arduino.cc/t/yet-another-fht-thread-the-basics/501155

29 Serial Communication between Python and Arduino
https://projecthub.arduino.cc/ansh2919/serial-communication-between-python-and-arduino-663756

30 31 Read Streaming Data from Arduino Using Serial Port Communication - MATLAB & Simulink Example
https://www.mathworks.com/help/matlab/import_export/read-streaming-data-from-arduino.html

32 34 Max Comm Tutorial 2: Serial Communication
https://docs.cycling74.com/max5/tutorials/max-tut/communicationschapter02.html

33 Serial Data from Arduino to MAX - MaxMSP Forum | Cycling '74
https://cycling74.com/forums/serial-data-from-arduino-to-max

35 Log Temperature Data from Arduino into MATLAB - MathWorks
https://www.mathworks.com/videos/log-temperature-data-from-arduino-into-matlab-1489428648919.html

36 What kind of analog pad sensor matrices are used in midi ... - Reddit
https://www.reddit.com/r/arduino/comments/1df2nfj/what_kind_of_analog_pad_sensor_matrices_are_used/

37 In MIDI 1.0, all data was in 7-bit values. That means musical ...
https://news.ycombinator.com/item?id=22208835

38 39 Open Sound Control
https://apolloensemble.co.uk/osc.html

40 41 Analog To Midi with MKR 1000 - Arduino Documentation
https://docs.arduino.cc/tutorials/mkr-1000-wifi/analog-to-midi/

42 43 Electrooculogram with an Arduino and a Computer.
http://onloop.net/hairyplotter/

44 Vox V847 Wah-Wah Analysis - ElectroSmash
https://www.electrosmash.com/vox-v847-analysis

45 Experiment: Eye Potentials (The EOG) | BYB documentation
https://docs.backyardbrains.com/Retired/Experiments/EOG

46 47 Eye Motion Tracking Using Infrared Sensor : 5 Steps - Instructables
https://www.instructables.com/Eye-Motion-Tracking-Using-Infrared-Sensor/

48 Ohm's Law - How Voltage, Current, and Resistance Relate
https://www.allaboutcircuits.com/textbook/direct-current/chpt-2/voltage-current-resistance-relate/