ChatGPT

Electrosmith Daisy Seed: Technical Overview

Figure: The Electrosmith Daisy Seed development board (51×18 mm), showing the STM32H7 MCU and supporting chips.

Hardware Capabilities and Interfaces

The **Daisy Seed** is a compact audio-focused embedded board built around a high-performance ARM Cortex-M7 microcontroller. Its hardware is tailored for **digital signal processing (DSP)** in music and audio applications. Key hardware features include:

- **Microcontroller**: 480 MHz ARM Cortex-M7 MCU (STM32H7 family) with floating-point unit, providing ample processing power for real-time audio DSP [1].
- **Memory**: 64 MB of external SDRAM and 8 MB QSPI flash for program storage and audio buffers (supports ~10-minute audio loops in RAM) [1]. The MCU also has internal SRAM for low-latency tasks.
- **Audio I/O**: Onboard stereo audio codec (AKM AK4556) supporting 24-bit, 96 kHz stereo audio I/O (line-level, AC-coupled) by default [2] [3]. (The codec can be configured up to 192 kHz sample rate for advanced use cases [4].) Audio input and output are available on dedicated analog pins (ADC_IN, DAC_OUT).
- **Analog Inputs (ADC)**: 12 (up to 14) general-purpose **analog input channels** accessible at 16-bit resolution [5] [6]. These can connect to knobs, CVs, etc., and are read via the MCU's ADC.
- **Analog Outputs (DAC)**: 2 dedicated **analog output channels** (12-bit DAC, DC-coupled) for control voltage or low-frequency output [6]. These 0–5V outputs are primarily for CV or slow signals (they are DC-coupled and won't natively drive high-fidelity audio).

- **Digital GPIO**: 31 multifunction **GPIO pins** for digital input/output [6]. These 3.3V logic pins can be used for switches, LEDs, triggers, etc., and many are multi-purpose (SPI, I2C, UART pins overlap with GPIO).
- **Serial Interfaces**: Supports **I²C**, **SPI**, and **UART** serial buses via the MCU. Multiple I2C and SPI peripherals are available for connecting sensors, displays, MIDI adapters, etc. (e.g., pins for I2C_SCL/SDA, SPI SCK/MISO/MOSI/SS are provided on the header [7]). The Daisy's MCU also provides **SAI/I2S** interface for the audio codec and can be used for other digital audio streams.
- **MIDI**: Dedicated UART pins for MIDI DIN input and output are available (with appropriate level shifting). A TRS MIDI jack can be attached via the provided pins [8].
- **USB Connectivity**: Micro USB port for power, programming, and USB communication. Daisy can enumerate as a USB device (supports USB MIDI, serial, and even USB Audio in firmware) [9]. This allows direct connection to a PC for audio/MIDI streaming or debugging.
- **SD Card**: Built-in SD card interface (SDMMC) for reading/writing SD cards [10]. This enables sample playback, recording, or file storage using FATFS in firmware.
- **Power Supply**: Powered via USB 5V or an external VIN pin (accepts ~9V–12V DC, regulated on-board to 3.3V) [11]. On-board regulators provide 3.3V for the MCU and codec. A dedicated VBAT pin is available to power the real-time clock if needed.
- **Physical Form**: 20×2 DIP pin format (0.1″ spacing) for easy breadboard or PCB mounting. The board includes a user LED and a reset button. A **BOOT0** button is provided to enter DFU bootloader mode for firmware flashing.

Supported Interfaces Summary: In total, Daisy offers a rich set of I/O: stereo audio ADC/DAC (on codec), 2 extra 12-bit DAC outs, up to 14 ADC inputs, 31 GPIOs, multiple I2C/SPI/UART ports, PWM timers, and SDIO for SD card [10] [12]. This allows interfacing with various sensors, controls, and peripherals needed in synthesizers and effects units.

Supported Development Environments

One of Daisy's strengths is its **flexible programming support** – you can develop in several environments depending on your preference:

- **C++ with libDaisy** – The native **libDaisy** C++ framework provides low-level control and optimized audio handling. You write firmware in C/C++ (typically using the GCC/ARM toolchain or STM32CubeIDE) and link against libDaisy and DaisySP. This gives the best performance and full hardware control. (ElectroSmith provides many C++ examples and a template for setting up the toolchain [13].)
- **Arduino (DaisyDuino)** – Daisy can be programmed through the familiar **Arduino IDE** using the **DaisyDuino** library. This library wraps Daisy's functionality into Arduino-friendly APIs [14]. After installing DaisyDuino, you can write sketches in C++ and use `DAISY.init()` and `DAISY.begin()` to set up audio, read sensors, etc., just like on an Arduino, but with Daisy's advanced audio features [15] [16]. This is great for beginners and quick prototyping.
- **Max/MSP Gen~ (Oopsy)** – For visual programming enthusiasts, Daisy supports **Gen~** patches from Cycling '74 Max. The open-source **Oopsy** toolchain can compile Gen~ patchers to Daisy firmware with no manual coding [17]. This allows using Max's high-level DSP graph environment and deploying the result to the Daisy. Development is done in Max; Oopsy handles code generation, compilation, and loading onto Daisy [18].

- **Pure Data (pd2dsy / PlugData)** – Daisy also supports Pure Data patches via tools like *pd2dsy*. You can create Pure Data (PD) patches and use the pd2dsy converter (or the PlugData environment) to generate Daisy-compatible code. This effectively lets you run PD audio patches on the hardware. (It's recommended to use the newer *PlugData* workflow for better reliability [19].)
- **CircuitPython** – A CircuitPython firmware is available for the Daisy Seed [20]. This lets you program the Daisy in Python for rapid prototyping, using Adafruit's extensive library ecosystem. (CircuitPython support includes the basic hardware interfaces, though real-time audio processing in Python is limited.)
- **Others**: The Daisy can be used with **Faust** (a DSP language) and other audio frameworks with some community effort. For example, DaisyDuino can integrate with Arduino AudioTools or Faust libraries [21]. Additionally, standard **STMicroelectronics development tools** (STM32CubeIDE, etc.) can be used since Daisy's MCU is an STM32H7 – advanced users can develop at the HAL or register level if needed, using the Daisy's schematic for reference [22].

Note: All official Daisy firmware and libraries are open-source (MIT license) [23]. Electrosmith provides a web-based **Daisy Web Programmer** for flashing firmware via USB, and maintains an active forum and Discord for support.

Key APIs and Functions in libDaisy (C++) and DaisyDuino (Arduino)

Daisy's software libraries provide a rich API to configure hardware and handle audio. Below is a **catalog of important classes, functions, and commands** for both the low-level libDaisy (C++) and the Arduino environment:

libDaisy C++ API (Core Classes & Functions)

- **DaisySeed** – The central class representing the Daisy board. You typically create a `DaisySeed hw;` object in `main()`. Important methods:
- `hw.Configure()` – (Optional) Set custom hardware configs before init (e.g. enable/disable peripherals).
- `hw.Init(bool boost)` – Initialize the hardware (boost mode true enables 480 MHz vs default lower clock) [24]. This sets up SDRAM, flash, audio codec (AK4556 at 48kHz by default), USB, onboard LED, etc. [25]. After `Init()`, all core hardware is ready.
- `hw.SetAudioSampleRate(rate)` – Change audio sample rate (must call *before* starting audio, or stop and reinit) [26]. Supported rates include 48k, 96k, etc.
- `hw.SetAudioBlockSize(size)` – Set the number of samples per audio callback (per channel) [27]. Default is 48 samples (for 1 kHz callback rate at 48kHz sampling), but you can reduce this for lower latency or increase for efficiency.
- `hw.StartAudio(callback)` – Start the audio engine and begin calling the specified audio callback function in real-time [28]. (Overloads allow **interleaved** or **multi-channel** buffer callbacks [29].)
- `hw.StopAudio()` – Stop the audio processing if needed [30].
- `hw.AudioSampleRate()` – Get the current sample rate in Hz [31], and `hw.AudioBlockSize()` to get block length [32] (also `AudioCallbackRate()` for the callback frequency [33]).
- `hw.StartAdc()` – Begin analog-to-digital conversions on the configured ADC pins [34]. (By default, if you've configured any analog inputs, call this before reading them, especially if audio is running concurrently.)

- `hw.DelayMs(ms)` – Simple utility to delay (busy-wait) for a number of milliseconds [35].
- DaisySeed also provides helpers like `GetPin(pin_number)` to get a `Pin` handle for GPIO, and handles initialization of the built-in LED and test point.

- **GPIO (General-Purpose I/O)** – libDaisy uses a `daisy::Gpio` class to configure digital pins. Example workflow: define a `Pin p = seed.GetPin(n);` (using the Daisy pin index from the pinout). Then do:

  ```
  daisy::Gpio gpio;
  gpio.Init(p, daisy::Gpio::Mode::OUTPUT);
  gpio.Write(true); // set HIGH
  ```

  Modes can be `OUTPUT`, `INPUT`, `INPUT_PULLUP`, etc., and `Read()`/`Write()` functions get or set the pin state [36] [37]. (There are also shorthand constants for onboard LED pins and such.)

- **Analog Inputs (ADC)** – libDaisy provides two ways to read analog: low-level ADC handles and a high-level `AnalogControl`.

- *Low-level:* Configure ADC channels with `AdcChannelConfig` (assign a hardware pin to an ADC), then call `seed.adc.Init(configs, num_channels)` and `seed.adc.Start()` to begin continuous conversions. You can fetch values via `seed.adc.Get(channel)` which gives raw ADC counts.
- *High-level:* The `daisy::AnalogControl` class wraps an ADC channel with scaling and filtering (used for knobs/CV). You initialize it with an ADC channel and parameters (like slew rate or hysteresis for smoothing). Then call `analogCtrl.Process()` each loop to update its value, and `analogCtrl.Value()` to get a normalized 0–1 float [38]. This is useful for reading potentiometers smoothly.

- **Note:** ADC readings are typically updated at audio rate (if tied into the audio callback) or in a fast loop. In Daisy's example code, after starting audio and ADC, you might read the latest ADC value inside the audio callback or a control loop. Be mindful to call `StartAdc()` or the ADC will read as constant if audio is started (since the ADC is not started by default when audio begins) [39] [40].

- **DAC Outputs** – The two 12-bit DAC channels on Daisy are accessed via `daisy::DacHandle`. You can initialize the DAC handle and use `DacHandle.Write(channel, value)` to output a value (0–4095 for 0–3.3V or scaled to 0–5V via op-amp if on Daisy Pod/Patch). In practice, you can also use higher-level functions or in Arduino simply use `analogWrite()` (see Arduino section). The DACs are typically updated at control rate. For CV outputs, you might update them in a slower loop or on knob changes. (Driving them at audio-rate is possible but not ideal – see notes in "Known Issues".)

- **Audio and DSP** – libDaisy's **Audio** subsystem is initialized by `StartAudio()` as above. You provide an audio callback function, e.g.:

```
void AudioCallback(float **in, float **out, size_t frames) {
    // process `frames` samples for each channel
    // e.g., out[0][i] = in[0][i] * gain;
}
```

libDaisy handles filling `in`/`out` buffers via DMA from the codec. You can change the block size for different latency/CPU trade-offs. For generating or processing audio, you will often use the **DaisySP** library (described below) which provides DSP modules like oscillators and filters. For example, you might instantiate `daisysp::Oscillator osc; osc.Init(sampleRate); osc.SetWaveform(daisysp::Oscillator::WAVE_SAW);` and then in the callback call `out[0][i] = osc.Process();`.

- **MIDI** – The `daisy::MidiHandler` class handles MIDI input/output over UART or USB. You can initialize it for UART1 (the hardware MIDI RX/TX pins) or USB, and then call `midi.StartReceive()` to begin listening [34]. In your main loop (or a dedicated MIDI loop), call `midi.Listen()` and then process incoming messages via `while(midi.HasEvents()) { auto msg = midi.PopEvent(); ... }` [41]. Daisy's MIDI library can parse common MIDI messages (note on/off, CC, etc.) and provides them as event objects (see `daisy::MidiEvent` classes).

- **Serial/UART** – For debugging or other communication, Daisy's USB interface can be used as a serial port (CDC). libDaisy's `daisy::Logger` or `daisy::UsbHandle` can facilitate printouts. There's also `seed.StartLog()` to enable USB serial logging, after which you can use `printf`/`Logger` macros to output to the PC over USB [9]. (In Arduino, you would use `Serial.begin()` instead.)

- **I2C & SPI** – libDaisy provides `I2CHandle` and `SpiHandle` classes to interface with external devices (sensors, displays, codecs, etc.). To use, configure the handle with the port (e.g., `I2CHandle::Config cfg; cfg.periph = I2CHandle::Config::I2C1;` and the SDA/SCL pins, speed, address mode, etc.), then call `I2C.Init(cfg)`. You can then perform transactions like `I2C.Transmit(addr, data, length, timeout)` or receive, etc. Similar for SPI: configure the peripheral (SPI1/2), SCLK/MISO/MOSI pins, mode (clock polarity/phase), and call `SpiHandle::BlockingTransmit` or use DMA with callbacks for asynchronous transfer [42] [43]. Daisy's library also has helpers for specific devices (e.g., `daisy::Icm20948` IMU driver over SPI/I2C, `daisy::OledDisplay` for SSD1306 OLED over I2C, etc., as seen in the device classes list [44] [45]). These make it easier to integrate common modules.

- **SD Card & Filesystem** – DaisySeed's SD slot can be accessed via the `SdmmcHandler` (for raw SD card interface) in combination with FatFS for a filesystem. The libDaisy has examples where after initializing `SdmmcHandler`, you mount a FAT filesystem and read/write WAV files or patches from SD. (This is more advanced, involving third-party FATFS code integration).

- **Other Hardware Abstractions**: libDaisy also includes classes for **timers**, **PWM** (for e.g. LED dimming or generating low-frequency signals), **Watchdog** timers, etc., and high-level **UI helpers** in the `daisy::UI` module (for example, debouncing switches with `daisy::Switch` class, reading multiplexed controls, or managing an on-screen menu). There are **LED** classes for smart LEDs (e.g., DotStar or NeoPixel) and even a `daisy::Encoder` class for rotary encoders with push buttons [46]. These ready-made classes simplify implementing user interfaces (HID – human interface devices).

- **DaisySP Library**: While not part of *libDaisy* per se, **DaisySP** is the standard DSP library provided for audio algorithms. It includes modules like oscillators, filters, envelopes, delays, reverbs, FFT, and even physical modeling elements. It's heavily inspired by Mutable Instruments and Csound/Soundpipe modules [47]. In C++, you include `<daisysp.h>` (or specific module headers) and instantiate objects (e.g., `daisysp::ReverbSc reverb; reverb.Init(sampleRate);`). DaisySP is optimized for the Daisy's ARM M7 and uses efficient algorithms for real-time audio. *Examples:* `daisysp::PolyOsc` for polyblep oscillators, `daisysp::MoogLadder` for ladder filter, `daisysp::AdEnv` for ADSR envelope, etc. This library greatly accelerates audio development by providing ready-to-use building blocks.

DaisyDuino (Arduino) API

When using the Arduino IDE with DaisyDuino, much of the above functionality is exposed in an **Arduino-friendly** way. After installing the DaisyDuino library, you'll write an Arduino sketch for Daisy. Key elements of the DaisyDuino API:

- **Initialization**: In your `setup()` function, initialize the Daisy hardware with `DAISY.init()` and start audio with `DAISY.begin()`. For example:

```
#include "DaisyDuino.h"
DaisyHardware hw;
void setup() {
    hw = DAISY.init(DAISY_SEED,
AUDIO_SR_48K);  // init Daisy Seed at 48kHz sample rate
    DAISY.begin(AudioCallback);                // start audio processing
with callback
}
```

`DAISY_SEED` is a constant specifying the board type (others include DAISY_POD, etc., which configure built-in controls). The `hw` (of type `DaisyHardware`) returned can be used to query things like `hw.num_channels` (number of audio channels, e.g. 2) or other hardware info [48] [49].

- **Audio Callback**: Define an audio callback function in your sketch with the signature `void AudioCallback(float **in, float **out, size_t size)`. Inside, process `size` samples for each channel. For example, a pass-through callback:

```
void AudioCallback(float **in, float **out, size_t size) {
    for (size_t i = 0; i < size; i++) {
        out[0][i] = in[0][i];
        out[1][i] = in[1][i];
    }
}
```

DaisyDuino will call this continuously at the audio rate (by default 48kHz, block size 48). You can use global objects from **DaisySP** in the callback as well – DaisyDuino includes DaisySP, so you can instantiate oscillators, effects, etc. (For example, `Oscillator osc; osc.Init(AUDIO_SR_48K);` globally, then `out[0][i] = osc.Process();` in the callback.) Make sure any DaisySP objects are initialized *after* calling `DAISY.init()`, so you have the correct sample rate.

- **Analog and Digital I/O**: In Arduino, you can use standard `pinMode()`, `digitalWrite()`, `digitalRead()`, `analogRead()` on Daisy's pins (the DaisyDuino core uses the STM32 Arduino core). The pin numbering corresponds to the Daisy Seed's pin numbers as defined in the variant (check DaisyDuino docs for mapping). For example, `pinMode(LED_BUILTIN, OUTPUT); digitalWrite(LED_BUILTIN, HIGH);` will control the Daisy's built-in LED, as shown in Arduino examples [50] [51]. Reading a knob connected to an ADC pin can be done with `analogRead(A0)` (if A0 maps to that pin) which returns 0–1023 by default (10-bit Arduino analog resolution). However, DaisyDuino also provides a `seed.analogRead12(pin)` for 12-bit reads, if enabled. In practice, many use the higher-level DaisySP controls or do scaling in code.

- **DAC Outputs (CV out)**: Arduino's `analogWrite(pin, value)` can be used on the two DAC-capable pins (the ones tied to the DAC outputs on Daisy, typically pin 25 & 26 or similar – check documentation). By default, `analogWrite()` expects an 8-bit value (0–255) corresponding to 0–3.3V on the DAC [52]. For example, to output a control voltage: `analogWrite(CV1_PIN, 128)` sets ~1.65V. DaisyDuino's examples (e.g., **CV_Out** example) show how to map audio to DAC: you might take a float -1.0 to +1.0 audio signal and convert to 0–255 (with offset) for analogWrite [52]. Keep in mind the DAC outputs are unipolar 0–+5V (on Daisy Patch, etc.) and are not updated at audio-rate by hardware, so outputting fast audio waveforms through them may result in aliasing or unexpected behavior [53] [54]. They are mainly intended for low-frequency or DC signals (LFOs, CV, etc.).

- **MIDI (Arduino)**: DaisyDuino can use the **FortySevenEffects MIDI Library** (if included) for MIDI over UART or USB [55]. Alternatively, you can use `Serial1` for hardware MIDI (UART) at 31250 baud. For example, `Serial1.begin(31250);` and then use `Serial1.read()`, etc., to handle DIN MIDI messages. USB MIDI devices can be accessed via the Arduino MIDIUSB library if needed.

- **Using DaisySP**: As noted, DaisyDuino incorporates the DaisySP DSP library. You can declare DaisySP objects in Arduino sketches directly. For instance:

```
#include <DaisyDuino.h>
Oscillator osc;
void setup() {
    hw = DAISY.init(DAISY_SEED, AUDIO_SR_48K);
    osc.Init(hw.sample_rate);
    osc.SetWaveform(Oscillator::WAVE_SIN);
    DAISY.begin(AudioCallback);
}
```

```
void AudioCallback(float **in, float **out, size_t size) {
    for(size_t i=0; i<size; i++) {
        out[0][i] = osc.Process();
        out[1][i] = out[0][i];
    }
}
```

This would generate a sine wave tone on both left and right outputs. The DaisySP modules (oscillators, filters, reverbs, etc.) run efficiently on the Daisy's MCU even in the Arduino context, making it easy to create synths and effects without writing DSP from scratch.

- **Miscellaneous**: In Arduino, you still have access to lower-level if needed. For example, you can call `DAISY.setAudioBlockSize(n)` to change block size (this is available as of DaisyDuino 1.6.0 and later). You can also retrieve `DAISY.get_samplerate()` [49] or `DAISY.get_blocksizerate()`, etc., if needed. The DaisyHardware `hw` also may contain convenience references (for Daisy Petal, etc., it has `hw.knob1` analogControl objects, etc., but for the Seed it's mostly basic). Standard Arduino functions like `millis()`, `delay()`, `Wire` library (for I2C), `SPI` library, etc., all work since Daisy is supported by STM32 Arduino core. Just be cautious using `delay()` or heavy loops in the audio callback – keep real-time constraints in mind (see Best Practices below).

In summary, **libDaisy (C++)** offers fine-grained control and maximum performance with an extensive API for hardware, while **DaisyDuino (Arduino)** provides a faster-to-learn interface that still exposes Daisy's audio and MIDI capabilities. Both can leverage **DaisySP** for DSP. All APIs are well-documented in the Daisy reference and examples (with many code examples provided in the official DaisyExamples repository).

Best Practices for Firmware Development and Audio Performance

Developing DSP firmware for the Daisy Seed requires some attention to real-time performance and audio quality. Here are some best practices and tips to get the most out of the hardware:

- **Keep the Audio Callback Efficient:** The audio callback runs at a high rate (e.g., 1kHz if 48-sample blocks at 48kHz). Do all heavy DSP math in a lean way – avoid memory allocations, complex branch logic, or any I/O (like `printf`/`Serial` prints) inside the callback. Expensive operations in the callback can lead to audio glitches or a high-frequency "callback buzz" if the timing varies. If you need to communicate or log data, set a flag in the callback and handle it in the main loop (or at a slower rate). A good pattern is to use a **control loop** (running, say, at 1ms or 10ms interval) for non-audio tasks and keep the audio callback focused on sample processing.

- **Optimize Block Size vs Latency:** Daisy's block size is configurable. Smaller audio blocks reduce latency (and push any callback-induced noise to higher frequency) but increase CPU interrupt overhead [56]. Larger blocks are more CPU-efficient but add latency. If you hear a ~1kHz tone (callback "whine") in the output, it's likely due to block processing causing current transients at the block rate [57] [58]. To mitigate this, you can **reduce the block size** (e.g., to 4 or 2 samples) which pushes that noise out of audible range (e.g., 24kHz for a 2-sample block) [56]. The trade-off is higher CPU usage. Adjust block size to balance latency and performance: for critical low-latency apps (e.g., live FX), you might use 16-sample blocks; for heavy DSP where some latency is tolerable, 48 or 64 may be better.

- **Manage CPU Usage and Avoid Dropouts:** If doing intense DSP (FFT, large convolution, etc.), consider using **idle time** outside the audio ISR. One technique to avoid audio dropouts is to distribute work across multiple callbacks or use background processing. For instance, if an effect can be processed in chunks (partitioned FFT), do a little each callback. Keep audio ISR computation consistent to avoid varying CPU load. In some cases, adding a small background load *outside* the callback can smooth current draw and reduce audio whining noise [59] (this is an odd case where the *lack* of CPU use between audio blocks causes noise due to ground bounce; a consistent load can help stability).

- **Memory: Internal vs External** – Utilize the 64MB SDRAM for large buffers (delays, reverbs, granular sample memory), but be aware that external RAM has more latency. Try to keep **time-critical data in internal RAM** (DTCM or SRAM) – for example, DSP state variables, filter coefficients, or small delay lines. If you have a large structure (like the ReverbSC effect with big delay lines), you can place only the delay line in SDRAM and keep the core processing in internal memory [60]. Use the `DSY_SDRAM_BSS` macro (in C++) to place large arrays in SDRAM. In practice, Daisy's external RAM can handle audio, but performance-sensitive code may need cache enabled and 32-byte aligned bursts for best speed. If you experience glitches when using SDRAM, consider splitting the memory or enabling the CPU data cache (Daisy's init enables caches by default). Testing and profiling will help – e.g., the ReverbSC in DaisySP can run from SDRAM, but under Arduino it had issues, suggesting overhead in that environment [61] [62].

- **Use DMA and Peripherals**: Offload work from the CPU by using hardware features. The audio codec I/O is already via DMA. You can similarly use DMA for other tasks – e.g., SPI transfers to OLED or SD card can use non-blocking DMA so that drawing to a screen doesn't stall audio. The libDaisy SPI and SDMMC handlers support DMA modes. Also use timers for precise scheduling if needed (for example, a Timer or the built-in HAL SAI can generate a timer tick at audio rate). Taking advantage of the MCU's **DSP instructions (MAC, etc.)** via ARM CMSIS or compiler intrinsics can massively speed up processing (the M7 DSP instructions can do parallel multiplies, etc.). If using Arm's CMSIS library or ARM FFT, ensure to enable floating-point hardware and optimize with `-O2` or `-O3` flags in compilation.

- **Audio Quality and Noise**: To achieve the best audio quality, follow analog design best practices. Use proper buffering and filtering on analog inputs/outputs – Daisy Seed by itself is line-level; if interfacing with eurorack or guitars, use op-amp circuits (ElectroSmith provides reference designs in the Daisy datasheet and examples [63]). Keep analog cables short and use shielded wiring for audio I/O to prevent noise pickup. Ground loops can introduce hum – ensure a solid common ground. The Daisy Seed's codec audio I/O is AC-coupled (capacitively) – it's designed for ~±2V line level signals. If you need a different level (instrument or modular levels), hardware amplification/attenuation is needed. In code, avoid DC biases in audio signals (the codec cannot output DC due to AC coupling, so any DC component will bleed out).

- **Firmware Structure**: It's often useful to separate your code into an **audio thread and a control thread**. For example, run audio continuously via the callback, and use the main loop to handle user input, OLED updates, MIDI, etc., at a slower rate (e.g., check knobs every few milliseconds). This prevents control handling from interfering with the strict timing of audio. libDaisy's `System::Delay()` or `daisy::ElapsedTime` can help implement timed loops. In Arduino, the `loop()` function can run fast but you might deliberately add a small delay or use a millis-timer to service UI at, say, 100 Hz.

- **Avoid Blocking Operations**: Do not perform long blocking operations (like file loading from SD, lengthy calculations, or `delay()` calls) in the audio thread. If you need to load a sample from SD card, consider doing it in small chunks or while audio is stopped, or double-buffer the audio so one buffer plays while the next loads. The Daisy's 64MB SDRAM can hold large audio files, so a common technique is to preload as much as possible into memory before playback. If using Arduino and calling functions like `delay()` or `Serial.print()` inside the audio callback, expect problems – instead set a flag and handle those in `loop()`.

- **Use of Boost Mode**: By default, Daisy may run the MCU at a slightly reduced clock (e.g., 400 MHz) to save power. If you need maximum CPU headroom, enable the **boost** flag on init (`hw.Init(true)`) to run at 480 MHz [24]. This gives ~20% more processing power at the cost of higher power consumption and heat. Monitor the MCU temperature if pushing it; the STM32H7 has an internal temp sensor if needed.

- **Profiling and Benchmarks**: It's good practice to measure CPU usage. In C++, you can toggle a GPIO at the start and end of your audio callback and observe it on an oscilloscope to measure how much time the ISR takes (the duty cycle of the toggled pin indicates CPU load in the callback). In Arduino, you might use `micros()` to check how long the callback took (careful to do it in a not always-on way). Keep the audio callback CPU usage ideally below 70-80% to leave room for system tasks and avoid underruns.

- **Leverage Examples and Templates**: The DaisyExamples repo and Electrosmith's documentation contain many optimized code examples (synth voices, effects, MIDI handlers, etc.). Use them as a starting point – for instance, see how they implement polyphony or efficient filter control. Often these examples illustrate best practices like precomputing coefficient tables, using integer fixed-point for control rates, etc.

Following these practices will help ensure your Daisy Seed projects run reliably with high audio quality and minimal glitches. The Daisy community (forums and Discord) is also a great resource to learn about specific optimizations and to get help tuning performance.

Libraries and Tools for Daisy Development

A variety of open-source libraries and tools are available to streamline development on the Daisy Seed:

- **libDaisy (Hardware API Library)** – **libDaisy** is the official C++ hardware abstraction library for Daisy [9]. It provides drivers for all peripherals (audio, ADC, DAC, GPIO, UART, I2C, SPI, timers) and helpers for common tasks (like debouncing switches or reading analog controls). This is the core library you use in C++ projects. *Repo:* electro-smith/libDaisy on GitHub. The documentation is available online (Doxygen style) and covers all classes and functions.

- **DaisySP (DSP Library)** – **DaisySP** is an open-source DSP library in C++ with a comprehensive collection of audio processing modules [64]. It includes oscillators, noise, LFOs, filters (IIR and FIR), envelope generators, delays, reverbs, chorus/flanger, bitcrushers, and more. Many modules are ports from *Soundpipe*, *Csound*, or Mutable Instruments algorithms [47]. DaisySP is optimized for embedded use and forms the building blocks of most Daisy examples. *Repo:* electro-smith/DaisySP (plus DaisySP-LGPL for a few GPL-licensed modules). Documentation includes a list of modules and example usage for each.

- **DaisyDuino (Arduino Library)** – **DaisyDuino** is the Arduino library that brings Daisy support to the Arduino IDE [14]. It wraps libDaisy and DaisySP functionality into an easy-to-use package. DaisyDuino includes the core audio/MIDI features, and you install it via Arduino Library Manager. Alongside the library, a **Daisy Seed board definition** is provided (using STM32 Arduino core) so that the Arduino IDE knows how to flash the board. The DaisyDuino GitHub contains example sketches (for synths, effects, CV out, etc.) to demonstrate usage. Notably, DaisyDuino merges what used to be separate ArduinoAudio and ArduinoDaisySP libraries – so you only need to include **<DaisyDuino.h>** to get everything [65].

- **Oopsy (Max/MSP Gen~ Tool)** – **Oopsy** is the code generation tool that integrates with Max/MSP. It allows you to write Gen~ patchers and export them as firmware for Daisy. Oopsy is itself open-source (JavaScript in Max and some Python/C++ under the hood). It supports multiple Daisy hardware platforms (Seed, Pod, Patch, etc.), letting you target whichever hardware in the Gen patch by selecting an appropriate template. *Repo:* electro-smith/oopsy on GitHub. There is also a detailed *Oopsy Wiki* and tutorials from Cycling '74 [66], and even YouTube videos, since it opens up Daisy to the large Max community.

- **Petal, Patch, Field Support** – If you are using Daisy Seed in one of the Electrosmith products (like Daisy Petal guitar pedal, Daisy Patch eurorack, Daisy Field handheld), libDaisy includes specific classes for those (e.g., `DaisyPetal`, `DaisyPatch`) which pre-configure all the controls and codec routing for those boards [67]. The DaisyExamples repo has example code tailored to each platform. While not a separate library, it's worth noting these in case you expand to using the Seed in different form factors.

- **Daisy Web Programmer** – A browser-based tool (at **flash.daisy.audio**) that can flash firmware onto the Daisy via the DFU bootloader. It's handy for quickly deploying compiled binaries without using command-line DFU utilities. You simply connect the Daisy in bootloader mode and upload or choose a pre-made program. It's especially useful for beginners or sharing demo firmware (Electrosmith provides a set of example binaries on that site too).

- **Community Libraries/Tools**: Beyond official libraries, there are community-driven tools: for example, *Arduino AudioTools* integration for Daisy (enabling high-level audio pipeline design in Arduino) [68], or *Faust* improvements for Daisy (some have created Faust targets to output Daisy code, although not officially maintained). There's also **PlugData**, a standalone Visual Programming environment based on Pure Data, which can export to Daisy (bypassing some of the complexity of pd2dsy) [19]. Enthusiasts have ported code like Mutable Instruments synth engines to Daisy, often shared on GitHub. For instance, there are Daisy projects incorporating the Plaits oscillator code and Clouds (granular) code using DaisySP or custom ports.

- **Daisy Examples Repository** – The **DaisyExamples** GitHub repo is an invaluable resource. It contains numerous example projects in C++ covering everything from a basic "Hello, World" (blinking LED) to complex polyphonic synths and effects chains. These examples demonstrate how to use libDaisy and DaisySP in practice. There are sub-folders for different hardware (Seed, Pod, Patch, etc.) and for different features (MIDI, CV, sampling, etc.). This is often the best starting point to find reference code for a particular feature you want to implement.

- **Development Tools**: Daisy is compatible with standard ARM development tools. Many developers use **VS Code** with the Cortex-Debug extension or **PlatformIO** for a more polished IDE experience (PlatformIO has Daisy board definitions as well) [69] . You can also use **STM32CubeIDE** – Electrosmith provides a .ioc (CubeMX configuration) file for Daisy Seed to generate projects, which some users leverage [70] . When working in those environments, you'll link against libDaisy (as a source or archive). For debugging, Daisy has a SWD port (the 4 small pins on the board) which can be connected to a programmer like ST-Link or J-Link for step debugging and flashing outside of USB DFU.

All the above libraries are actively maintained on GitHub. It's recommended to keep your local copies updated, as the Daisy ecosystem is evolving (with frequent improvements and bug fixes). The Daisy Forum and Discord often announce updates – for example, updates to DaisySP or DaisyDuino that add new features or improve performance [71] .

Known Issues and Common Questions

Like any platform, the Daisy Seed has some known issues and common pitfalls. Here are some that developers often encounter, along with solutions or workarounds:

- **USB not recognized / DFU mode issues**: When entering DFU bootloader mode (by holding BOOT and pressing RESET), Daisy should enumerate as an STM DFU device. On Windows, you may need the STM DFU driver (Zadig can install this) [72] [73] . If the Daisy isn't showing up at all (or you get "USB device malfunctioned"), first check the USB cable (use a data-capable cable and avoid USB hubs). Ensure you followed the correct button sequence. If one Daisy Seed works and another doesn't on the same setup, the second unit could be faulty [74] [75] – contact Electrosmith in that case. In general, try the Web Programmer: if Daisy appears as "DFU mode" there, you are connected. If not, double-check drivers. (On Linux/macOS, no driver is needed, but on Windows you might use **STM32CubeProgrammer** as an alternative flashing tool if the browser fails.) Once programmed, if the device is not responding, you can always re-enter bootloader and flash a known-good example to verify the hardware.

- **Firmware compatibility (old vs new Seeds)**: In early 2022 a hardware revision of the Daisy Seed was released, which required an updated core library. Binaries compiled with old libDaisy (for the original revision) might not run correctly on newer Seeds (and vice-versa) [76] . The fix is to recompile your project with the latest libDaisy, which auto-detects the hardware. (One change was related to the SDRAM timing and flash; the updated libraries work on both revisions transparently [77] .) So if you downloaded a precompiled Daisy program from 2020–2021 and it "does nothing" on a 2023 Daisy, this is likely why. Rebuild from source with current libraries to resolve.

- **Audio "whine" or noise at block rate**: As discussed earlier, a steady high-pitched tone (e.g., ~1kHz) in the audio output usually indicates **callback timing noise** [57] [58]. It's not digital bit noise but rather an analog artifact from the processor load modulating the ground/reference. Solutions include reducing the block size (increasing the frequency of that noise out of hearing range) [56], or smoothing the processor load (e.g., adding a small busy wait outside the audio callback) [59]. Also ensure your analog output coupling and grounding are solid. Using the recommended output filter (op amp buffer and cap) from the Daisy schematic can help filter out high-frequency components. In extreme cases, if you *must* use a large block size and hear noise, consider adding a notch filter in software at that frequency or try to distribute CPU usage more evenly.

- **Distorted or noisy audio**: If your audio output is distorted, first verify you aren't clipping the signal in code (Daisy audio uses floating-point `-1.0 to +1.0` as full-scale range). Also ensure the codec was initialized to the correct sample rate and that your `AudioCallback` is actually being called (if not, you might hear a static DC or repeating buffer noise). Another gotcha: if the Daisy seems to output only silence or weird noise, check that **external SDRAM is working** – if you placed large buffers in SDRAM (using `DSY_SDRAM_BSS`) but the MCU isn't actually interfacing with the RAM (e.g., due to a mismatch in revision or a CubeMX config error), the program may behave incorrectly. Always test audio with a simple bypass or oscillator first to confirm basic function, then add complexity.

- **Analog inputs reading wrong or not updating**: If you use the analog inputs and notice you always get the same value or it doesn't change with your potentiometer, make sure you've **started the ADC** conversions. In libDaisy, call `hw.StartAdc()` after initializing the channels [34]. In DaisyDuino, analog reads should "just work" via `analogRead()`, but if you mix DaisySP's AnalogControl, ensure you call its `Process()` regularly. Also note that by default the ADC inputs expect 0–3.3V. If you feed a higher voltage (e.g., 5V CV), you risk damaging the pin or getting saturating readings. Use proper scaling or the Daisy Patch Submodule (which has protection) for higher-voltage CV. Another common issue: forgetting that some ADC pins might be used by something else (for instance, if you use the same pin for audio input and as a general ADC in code, there's a conflict). Check the pinout to ensure you're not double-using pins.

- **DAC output limitations**: The 12-bit DAC outputs on the Daisy have a 0–5V range (buffered to 0–5V on some Daisy-based boards, 0–3.3V on the Seed itself without additional circuitry). They are **DC-coupled** and intended for control signals, not high-quality audio [54]. You *can* output low-frequency waveforms or even rudimentary audio, but the update rate is limited by how fast you call `analogWrite()` or update the Dac in libDaisy. Also, the DACs are **unipolar** (no negative voltage), so any audio will be biased (e.g., a centered audio waveform needs offset). For proper audio, use the main stereo codec outputs, which are bipolar AC-coupled and much higher fidelity. A known quirk: in Arduino, `analogWrite()` on Daisy by default uses 8-bit resolution (0–255). If you need finer control, you could use STM32 HAL to write 12-bit to the DAC registers, but in most cases for CV, 8-bit (0–255) is acceptable resolution. Future library updates might allow setting analogWrite resolution.

- **MIDI UART confusion**: Daisy Seed's MIDI pins (on the 40-pin header) correspond to `USART1` on the MCU. Make sure to connect a proper MIDI transceiver circuit (opto-isolator for MIDI IN, and a line driver for MIDI OUT or a simple transistor) – the Seed pins are 3.3V logic, not directly 5V MIDI. If using DaisyDuino, `Serial1` maps to this UART. If nothing is coming in, verify your MIDI connections and that you called `Serial1.begin(31250)`. Also note that the Daisy Petal/Field hardware use a particular TRS jack standard (Type A vs Type B). If designing your own, match the wiring accordingly.

- **Hot-swapping and Power**: **Do not hot-plug** the Daisy Seed into a powered socket or base board. In other words, remove power before inserting or removing the module. Users have reported that hot-swapping can damage the Daisy (voltage surges on pins) [78]. Always power down your system before connecting/disconnecting the Seed. Additionally, be careful with power supply polarity on VIN – the Daisy Seed expects VIN up to 12V DC *center-positive*. If you plug in a reverse-polarity supply or a higher voltage, you can destroy the onboard regulator and MCU. There is limited protection on the VIN, so triple-check your power source.

- **Heat and current draw**: The STM32H7 at 480 MHz is a powerful chip and can run warm. If you run heavy DSP, it's normal for the main chip to be slightly hot to touch (50–60°C). This is within operating range. Just ensure adequate ventilation. The Daisy draws around 300 mA on 5V when running a typical audio program. If using USB power from a computer, that's fine; but if you power multiple modules or sensors, ensure the 5V rail can supply enough current.

- **"Brick" recovery**: Since Daisy runs user firmware directly, there is a possibility to flash a program that misbehaves (e.g., hangs the MCU). If the Daisy Seed's LED does not light and it doesn't enter DFU with the button, you might think it's "bricked." The remedy is to force DFU mode: hold BOOT and RESET, then release RESET while still holding BOOT, then finally release BOOT. This should put it in DFU no matter what firmware is on it (since BOOT0 pin high skips user code). Then you can re-flash a working firmware. Only if this sequence fails and the device isn't enumerating might there be a true hardware issue.

Most of these issues are edge cases; in general the Daisy Seed is a robust platform. By following guidelines (proper power, using the latest libraries, and testing incrementally), you can avoid the common pitfalls. The Electrosmith Daisy forum's **Troubleshooting** section is filled with Q&A on these topics – a quick search there can often reveal if your issue has been seen and solved by others (for example, threads on "Daisy Seed not recognized" or "ADC not working" have checklists to try) [19] [79].

Example Projects and Applications

The Daisy Seed has been used in many inspiring projects, from DIY synthesizers to art installations. Here is a selection of example projects (with sources) that showcase what you can do with Daisy:

- **Void Extrusion (Generative Sound Sculpture)** – A kinetic sound sculpture by artist [Eirik Brandal] uses the Daisy Seed as the brains of an algorithmic composition system [80]. The Daisy generates evolving ambient music in real-time, demonstrating its ability to handle algorithmic MIDI and audio synthesis. This project combines an LED light wall (driven by an Arduino) with the Daisy-based sound engine to create an immersive audiovisual piece. *(Source: Hackaday)*

- **Fish Synth – Portable DIY Synthesizer** – An instructables project for a handheld synth called "Fish Synth" uses the Daisy Seed as the core sound generator [81]. It features three knobs and four buttons for controlling effects and chords, with a built-in speaker and battery. DaisyDuino was used to program various audio effects (distortions, filters) and musical scales triggered by the buttons.

This project highlights using Daisy in an Arduino workflow to quickly implement a custom synth with onboard controls and a cute form factor. *(Source: Instructables)*

- **Open-Source Guitar Pedal Platform (Stereo FX)** – Developer **bkshepherd** designed a series of guitar pedal hardware platforms using the Daisy Seed, shared on GitHub [82]. One version is a compact stereo-in/stereo-out pedal (1590B enclosure) providing a Daisy Seed, audio I/O jacks, MIDI In/Out, and footswitches – essentially a DIY "Daisy Petal." Another is a larger pedal with an OLED screen, two footswitches, 6 knobs, and an encoder for menu navigation [83]. These pedals run Daisy firmware for various effects (reverbs, delays, loopers). The projects demonstrate integrating Daisy with analog audio circuits (input buffers, op amp mixers) and using the Seed's DAC outputs for things like expression pedal CV out or feedback control. *(Source: Daisy Forum & GitHub – DaisySeedProjects)*

- **Abacusynth (Interactive Poly Synth)** – The **Abacusynth** by [Elias Jarzombek] is a unique synthesizer that uses an abacus-like interface for control. Internally, it's powered by a Daisy Seed for the polyphonic sound generation and MIDI interfacing [84]. Four voices are controlled by sliding/spinning beads (with optical sensors read by Daisy's ADC inputs), creating a very tactile music experience. Daisy handles reading the sensors, generating wavetable synthesis for each voice, and responding to an onboard MIDI interface. *(Source: Hackaday)*

- **Eurorack Modules (Daisy Patch)** – The Daisy Seed is at the heart of Electrosmith's **Daisy Patch** module, and many hackers have created their own Eurorack modules using the Seed. For example, the *Terrarium* by PedalPCB is a Eurorack dev board for Daisy Seed, and projects like **Shell-Tethered Vesper** (a wall-hanging generative synth module) use Daisy for generating evolving CV and audio in a modular context [85]. These often leverage Daisy's CV outputs (DACs) and audio I/O to integrate digital algorithms into analog modular setups.

- **Hybrid Analog/Digital Synths** – Daisy's ability to interface with analog circuits makes it popular for hybrid designs. One project uses Daisy Seed plus analog VCF and VCA chips to create a mono synth: Daisy generates digital oscillators and envelopes, then passes them through analog filters (controlled by Daisy's CV outputs). Another project uses Daisy as a digital LFO/sequencer, outputting CV patterns to drive analog oscillators. These demonstrate how Daisy can act as the "digital modulator" in analog synth gear, thanks to its multiple ADC (for reading knobs/CV in) and DAC (CV out) channels.

- **Hackaday and Community Showcase** – Many projects can be found on Hackaday.io and the Daisy forums: from drum machines to granular texture generators. For instance, a user built a **multi-FX unit** with Daisy that runs various guitar effects (phaser, delay, reverb) selectable via MIDI – showcasing Daisy's ability to store multiple "apps" and switch on the fly. Others have used Max/MSP with Oopsy to prototype complex patches (like spectral processors) and then run them on Daisy hardware in installations. The ease of swapping firmware means one piece of hardware can take on many personalities – a theme seen in Daisy-based designs.

These examples scratch the surface of what's possible. They collectively illustrate that the Daisy Seed can handle polyphonic synthesis, high-quality effects processing, interactive installations, and more. Whether you prefer coding in C++ or designing in a visual language like Max or PD, Daisy offers a path to bring your audio ideas to life in a standalone embedded device. The community actively shares code and experiences, so newcomers can build upon proven projects. By studying these examples (most have code available), you can jump-start your own Daisy Seed project – be it a novel musical interface, a boutique audio effect, or a hybrid analog-digital instrument.

Sources: The information in this overview was compiled from Electrosmith's official documentation and community resources, including the Daisy hardware reference [86] [3], library documentation [27] [34], and user-contributed knowledge from forums and projects [56] [53]. Each section contains inline citations (【source†lines】) linking to the relevant documentation or discussion for further reading. This ensures the overview is up-to-date and grounded in the current state of the Daisy platform (as of 2025).

---

[1] [2] [10] [23] Daisy Seed – Electrosmith
https://electro-smith.com/products/daisy-seed

[3] [4] [7] [8] [11] [15] [16] How to Use Electrosmith Daisy Seed: Pinouts, Specs, and Examples | Cirkit Designer
https://docs.cirkitdesigner.com/component/ddaf2794-1b5c-45eb-a113-6aa6a9a880d2/electrosmith-daisy-seed

[5] [12] [22] [86] Daisy Seed - Daisy
https://daisy.audio/hardware/Seed/

[6] [20] Daisy Seed Download
https://circuitpython.org/board/daisy_seed_with_sdram/

[9] [13] [34] [41] GitHub - electro-smith/libDaisy: Hardware Library for the Daisy Audio Platform
https://github.com/electro-smith/libDaisy

[14] [55] [64] GitHub - electro-smith/DaisyDuino: Arduino Support for the Daisy Audio Platform.
https://github.com/electro-smith/DaisyDuino

[17] Meet Oopsy: Daisy Object for Max/MSP Gen
https://forum.electro-smith.com/t/meet-oopsy-daisy-object-for-max-msp-gen/944

[18] Getting Started with Oopsy (Max/MSP Gen~) - Daisy
https://daisy.audio/tutorials/oopsy-dev-env/

[19] [72] [73] [74] [75] Unable to program new Daisy Seeds - Page 2 - Getting Started - Daisy Forums
https://forum.electro-smith.com/t/unable-to-program-new-daisy-seeds/4682?page=2

[21] [68] Arduino Audio Tools on Daisy ? · pschatzmann arduino-audio-tools · Discussion #992 · GitHub
https://github.com/pschatzmann/arduino-audio-tools/discussions/992

[24] [25] [26] [27] [28] [29] [30] [31] [32] [33] [35] Classdaisy 1 1 daisy seed - Daisy
https://daisy.audio/libDaisy/Classes/classdaisy_1_1_daisy_seed/

[36] [37] GPIO - Daisy
https://daisy.audio/tutorials/_a1_Getting-Started-GPIO/

[38] libDaisy: daisy Namespace Reference - GitHub Pages
https://electro-smith.github.io/libDaisy/namespacedaisy.html

[39] [40] [70] [79] Can't get ADC working with audio callback · Issue #277 · electro-smith/libDaisy · GitHub
https://github.com/electro-smith/libDaisy/issues/277

42 43 44 45 46 PERIPHERAL - Daisy
https://daisy.audio/libDaisy/Modules/group__peripheral/

47 DaisySP audio processing library port to Teensy Audio
https://forum.pjrc.com/index.php?threads/daisysp-audio-processing-library-port-to-teensy-audio.66747/

48 49 50 51 60 61 62 Daisy Seed loop() locks up in audio callback when processing ReverbSc in SDRAM - Arduino - Daisy Forums
https://forum.electro-smith.com/t/daisy-seed-loop-locks-up-in-audio-callback-when-processing-reverbsc-in-sdram/920

52 53 54 Access DACs using DaisyDuino - Arduino - Daisy Forums
https://forum.electro-smith.com/t/access-dacs-using-daisyduino/4955

56 57 58 59 63 Avoiding Callback Noise - Daisy
https://daisy.audio/tutorials/eliminating-callback-noise/

65 71 Improved Arduino Library, Install, and Examples! - Daisy Forums
https://forum.electro-smith.com/t/improved-arduino-library-install-and-examples/323

66 Tutorial: Oopsy Daisy — An Introduction, Part 1 | Cycling '74
https://cycling74.com/tutorials/oopsydaisy-patch-an-introduction

67 BOARDS - Daisy
https://daisy.audio/libDaisy/Modules/group__boards/

69 Daisy Seed Arduino development in VSCode?
https://forum.electro-smith.com/t/daisy-seed-arduino-development-in-vscode/3639

76 77 NOTE -- New Daisy Seeds have hardware changes that require recompiling libraries | PedalPCB Community Forum
https://forum.pedalpcb.com/threads/note-new-daisy-seeds-have-hardware-changes-that-require-recompiling-libraries.10087/

78 Warning - Don't hotswap your Daisy Seed | PedalPCB Community Forum
https://forum.pedalpcb.com/threads/warning-dont-hotswap-your-daisy-seed.4299/

80 Sound Sculpture Uses Daisy Seed To Generate Audio | Hackaday
https://hackaday.com/2023/04/02/sound-sculpture-uses-daisy-seed-to-generate-audio/

81 Build Your Own Synth! : 10 Steps (with Pictures) - Instructables
https://www.instructables.com/Build-Your-Own-Synth/

82 83 My Daisy Guitar Pedal Designs on GitHub - Guitar Pedals - Daisy Forums
https://forum.electro-smith.com/t/my-daisy-guitar-pedal-designs-on-github/3641

84 Abacus Synthesizer Really Adds Up | Hackaday
https://hackaday.com/2022/06/15/abacus-synthesizer-really-adds-up/

85 shell-tethered vesper | Hackaday.io
https://hackaday.io/project/195027-shell-tethered-vesper