ESCOLA
SUPERIOR
DE MÚSICA
E ARTES
DO ESPETÁCULO
POLITÉCNICO
DO PORTO

P.PORTO

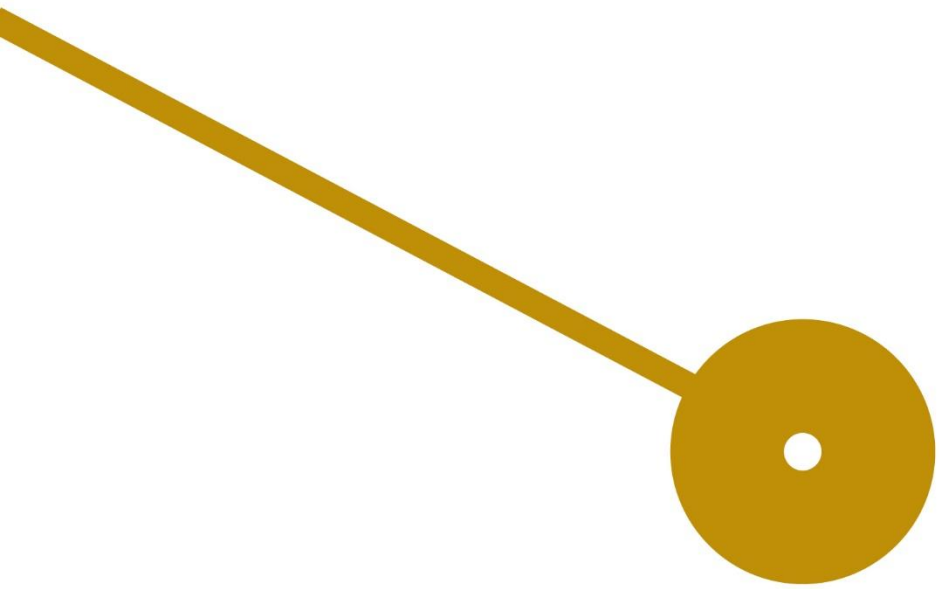
M

MESTRADO
ARTES E TECNOLOGIAS DO SOM

VÆTTIR

Diogo José Vieira Nóbrega

06/**2026**



ESMAE
ESCOLA SUPERIOR DE MÚSICA E ARTES DO ESPETÁCULO
POLITÉCNICO DO PORTO

P.PORTO

M MESTRADO
ARTES E TECNOLOGIAS DO SOM

VÆTTIR

Diogo José Vieira Nóbrega

Projeto apresentada à Escola Superior de Música e Artes do Espetáculo como requisito parcial para obtenção do grau de Mestre em Artes e Tecnologias do Som

Profesores Orientadores
Filipe Lopes
Bruno Pereira

06/**2026**

VÆTTIR

Organic Chaos Synthesizer

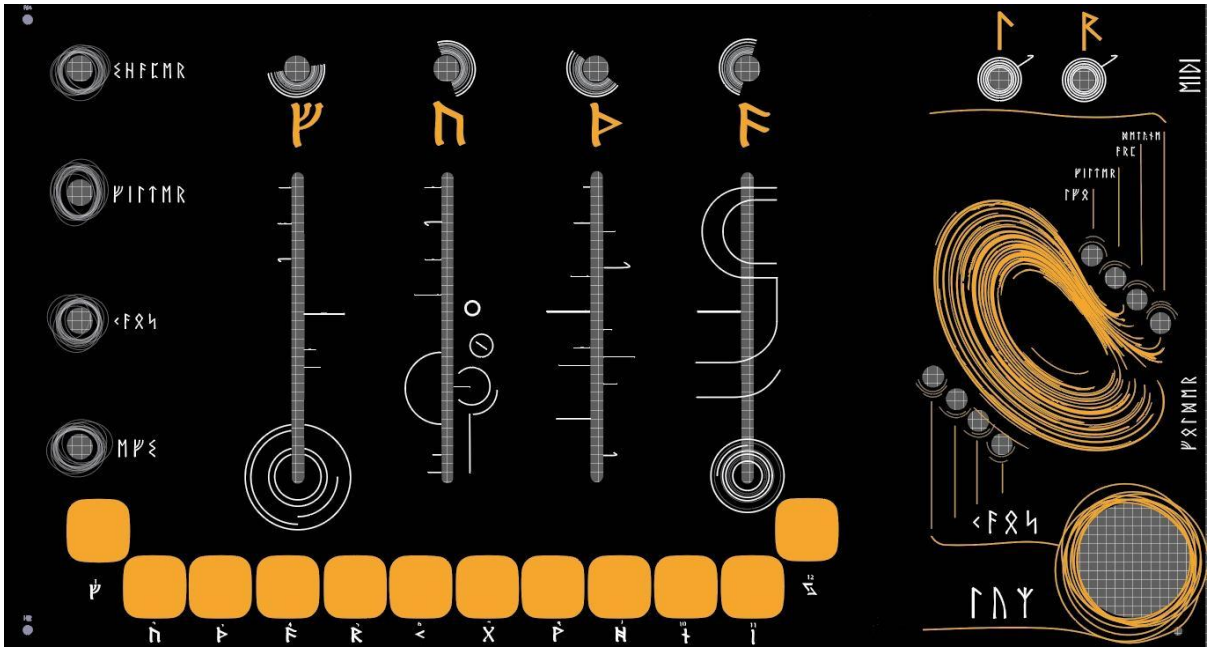


Figura 1- Interface do VÆTTIR. As quatro runas centrais (ᚠ ᚢ ᚦ ᚨ) identificam as quatro vozes de drone; à direita, o atractor de Lorenz marca a patchbay do caos; em baixo, os 12 pads capacitivos da escala asiática.

Resumo

O VÆTTIR é um sintetizador digital construído de raiz sobre a plataforma Daisy Seed, concebido como instrumento de performance para música experimental, drone e ambient. O sistema integra quatro vozes de drone sinusoidal (cada uma com três osciladores internos), um gerador de caos determinístico baseado no atractor de Lorenz, uma patchbay digital com quatro pontos de modulação, filtro SVF, waveshaper com três modos de dobragem, superfície táctil capacitiva de 12 elétrodos (MPR121) afinada numa escala de inspiração Hirajoshi, sensor de luminosidade ambiental (BH1750), MIDI bidirecional completo e efeitos stereo (reverb + delay ping-pong). A identidade visual e conceptual do instrumento ancora-se no sistema rúnico do Elder Futhark e na mitologia nórdica dos vættir, espíritos da natureza que habitam o limiar entre o controlado e o selvagem. O trabalho explora a tensão produtiva entre a previsibilidade das equações e a imprevisibilidade da prática, entre o instrumento como ferramenta e o instrumento como colaborador, e entre a construção técnica e a identidade artística.

Palavras-chave: Sintetizador, caos, atractor de Lorenz, instrumentos DIY, Daisy Seed

Abstract

VÆTTIR is a custom-built digital synthesizer developed on the Daisy Seed platform, designed as a performance instrument for experimental music, drone, and ambient. The system integrates four sinusoidal drone voices (each with three internal oscillators), a deterministic chaos generator based on the Lorenz attractor, a digital patchbay with four modulation points, an SVF filter, a waveshaper with three folding modes, a 12-electrode capacitive touch surface (MPR121) tuned to a Hirajoshi-inspired scale, an ambient light sensor (BH1750), full bidirectional MIDI, and stereo effects (reverb + ping-pong delay). The visual and conceptual identity of the instrument is grounded in the Elder Futhark runic system and the Norse mythology of the vættir, spirits of nature inhabiting the threshold between the controlled and the wild. The work explores the productive tension between the predictability of equations and the unpredictability of practice, between the instrument as tool and the instrument as collaborator, and between technical construction and artistic identity.

Keywords: Synthesizer, chaos, Lorenz attractor, DIY instruments, Daisy Seed

Índice

Índice de Figuras

Introdução

Há uma tensão que atravessa toda a história dos instrumentos musicais eletrónicos: a tensão entre o controlo e o acaso, entre a precisão da engenharia e a imprevisibilidade da expressão. Os primeiros sintetizadores analógicos (os Moog, os Buchla) prometiam um domínio total sobre o som. Mas os músicos que os abraçaram descobriram rapidamente que a riqueza desses instrumentos residia não naquilo que controlavam, mas naquilo que escapava ao controlo: as tensões que derivavam com a temperatura, os osciladores que nunca se fixavam completamente numa frequência, os filtros que respondiam ao toque de formas que os manuais não descreviam. Era nessa margem de imprevisibilidade que o instrumento respirava. Há uma certa honestidade nos instrumentos que falham. É precisamente nessa instabilidade que o *VÆTTIR* nasce. O problema central que o VÆTTIR procura responder é simultaneamente técnico e estético: como construir um sistema de síntese sonora que seja imprevisível sem ser arbitrário? Como incorporar o caos não como ornamento, mas como motor? Como criar uma interface que convide ao gesto e à descoberta em vez de impor uma gramática de uso? Estas questões cruzam-se com problemas mais amplos sobre a relação entre o músico e o instrumento, sobre o papel da instabilidade na expressão musical, e sobre o que significa construir um objeto sonoro com identidade própria. O VÆTTIR não foi concebido como uma ferramenta de precisão musical, mas foi pensado, desde o início, como um objeto vivo, um sistema que respira, que reage ao ambiente, que carrega consigo a história das suas componentes e das decisões improvisadas que o foram moldando ao longo do processo. É simultaneamente instrumento musical, escultura sonora e interface de performance audiovisual. A ideia central é criar um sistema de geração sonora que seja ao mesmo tempo controlável e indomável.

Este documento descreve o processo de construção do VÆTTIR em todas as suas dimensões: o contexto estético e conceptual que o enquadra, as referências técnicas e artísticas que o informaram, a arquitetura do firmware e o mapa completo das funcionalidades, e a identidade visual e simbólica que o torna um objeto coerente. O objetivo não é apenas documentar um objeto construído, mas articular as escolhas que o tornaram o que é e, através dessas escolhas, esboçar uma posição sobre o que a síntese sonora pode ser quando deixa de ser apenas produção de som e passa a ser um modo de habitar o tempo.

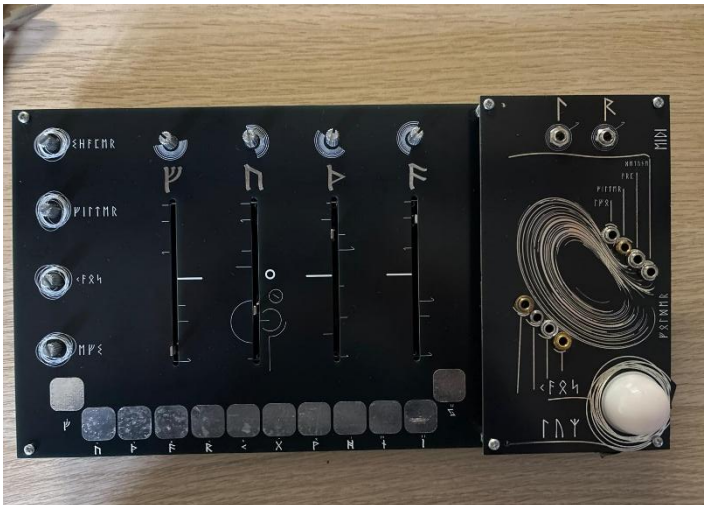


Figura 2- Vista geral do VÆTTIR montado.

Enquadramento e Contexto

O desenvolvimento de instrumentos musicais digitais baseados em microcontroladores de baixo custo acelerou significativamente na última década, impulsionado por plataformas como Arduino, Teensy e, mais recentemente, a Daisy Seed da Electro-Smith. Mathews (1963) foi pioneiro na síntese sonora por computador com o sistema MUSIC, estabelecendo os fundamentos matemáticos que continuam a informar a síntese digital contemporânea. A transição desses princípios para hardware embarcado de baixo custo é documentada por Boulanger e Lazzarini (2011), que traçam a democratização das ferramentas de síntese ao longo de cinco décadas.

A plataforma Daisy Seed representa um caso particularmente relevante para este projeto pois oferece processamento de áudio a 48 kHz/24 bits com latência abaixo de 1 ms, num formato do tamanho de um cartão de crédito. O ecossistema de software associado (libDaisy e DaisySP) fornece todas as implementações necessárias de osciladores, filtros, efeitos e interfaces de comunicação que reduzem significativamente a barreira de entrada para a construção de instrumentos personalizados (Electro-Smith, 2023). É precisamente sobre esta plataforma que o VÆTTIR assenta.

Neste contexto, é importante referir a Synthux Academy (www.synthux.academy). Trata-se de uma organização sem fins lucrativos que se situa na interseção do design de hardware, software open-source e educação, com a missão declarada de acreditar que os melhores instrumentos são os que ainda não existem. A Synthux desenvolve plataformas como o Simple Designer (um sistema de prototipagem rápida baseado na Daisy Seed) e disponibiliza os seus projetos em licença aberta para que qualquer pessoa possa construir, modificar ou aprender a partir deles. O VÆTTIR partilha desta filosofia de construção como prática artística e pedagógica, ainda que seja um objeto singular e não uma plataforma aberta e inspirou-se em alguns dos exemplos de interfaces e implementações de código.

Caos Determinístico em Síntese Sonora

A utilização de sistemas dinâmicos caóticos, em particular o atractor de Lorenz, como fonte de variação paramétrica em síntese sonora tem uma história documentada, embora relativamente especializada. Lorenz (1963) descreveu o sistema que leva o seu nome no contexto da modelação atmosférica, introduzindo no pensamento científico e artístico a noção de sensibilidade às condições iniciais, o célebre "efeito borboleta", como propriedade fundamental de certas classes de sistemas determinísticos.

No contexto da música experimental, a exploração de sistemas caóticos como geradores de material composicional é anterior à sua formalização matemática: a aleatoriedade estruturada de John Cage pode ser reinterpretada retroativamente como uma intuição artística sobre comportamentos caóticos determinísticos (Nyman, 1999). Mais recentemente, Sanfilippo e Valle (2013) exploraram sistematicamente os sistemas de feedback, dos quais o atractor de Lorenz é um caso especial, como material sonoro e compositivo, argumentando que a fronteira entre estabilidade e instabilidade é o espaço expressivo mais rico desses sistemas. Já Kim-Boyle (2010) documentou a utilização de autómatos celulares e operações de acaso em espacialização sonora, aproximando-se conceptualmente da abordagem do VÆTTIR, embora com implementações tecnicamente distintas.

A prática do circuit bending foi sistematizada por Ghazala (2005), que a enquadra numa filosofia de descoberta acidental e subversão criativa da tecnologia de consumo. Collins (2006) expande este enquadramento numa análise abrangente da cultura DIY em eletrónica musical, documentando desde os noise boxes dos anos 70 até às comunidades atuais de makers e os seus instrumentos de construção própria. A relação entre imperfeição

tecnológica e valor estético é teorizada por Cascone (2000) no seu ensaio seminal sobre a estética do fracasso na música computacional pós-digital.

Universo dos Fabricantes Contemporâneos

O universo dos sintetizadores modulares Eurorack, e em particular a tradição da síntese West Coast associada a Don Buchla, serviu como inspiração e enquadramento técnico próximo do VÆTTIR. Pinch e Trocco (2002) documentam a história paralela dos sistemas Moog (East Coast, baseados em filtros subtrativos) e Buchla (West Coast, baseados em waveshaping e processamento não-linear), estabelecendo uma genealogia que continua a influenciar o design de instrumentos contemporâneos. Neste universo, vários fabricantes informaram diretamente as decisões de design do VÆTTIR, cada um com uma filosofia distinta e identificável. A Make Noise Music, fundada por Tony Rolando, é atualmente um dos fabricantes mais influentes no espaço modular, que me cativou nomeadamente pela sua filosofia de design que valoriza a interatividade, a imprevisibilidade controlada e a expressividade táctil. As interfaces da Make Noise com os seus painéis graficamente densos, a ausência de hierarquias visuais óbvias, e a ideia de que o utilizador descobre o instrumento em vez de o aprender foi uma das influências estéticas mais diretas na conceção visual do VÆTTIR. O *0.COAST* (Make Noise Music, 2022) foi a principal inspiração, tanto sonoramente como esteticamente, por encarnar exatamente a tensão entre estrutura e caos que o VÆTTIR explora. A SOMA Laboratory, fundada em 2016 por Vlad Kreimer, traz uma posição radicalmente diferente, mas igualmente inspiradora. A sua filosofia, que a própria empresa designa como "romantic engineering", parte do princípio de que os melhores instrumentos não são os mais convenientes, mas os mais surpreendentes. Os seus instrumentos (o LYRA-8, o PULSAR-23, o TERRA, o FLUX) rejeitam sistematicamente as convenções de interface. O LYRA-8 usa transistors de efeito de campo e responde ao toque através de uma superfície resistiva em vez de um teclado convencional; o PULSAR-23 organiza os seus osciladores segundo uma lógica biológica e mecânica em vez da linear. O VÆTTIR partilha desta estranheza deliberada na escala asiática nos pads em vez do teclado cromático, as runas em vez dos nomes de notas convencionais, a patchbay como interface em vez de menus digitais. É a estranheza como método, não como ornamento. A Teenage Engineering, por seu lado, representa um polo estético oposto, mas igualmente relevante com design clean, minimalista, quase industrial, de instrumentos como o OP-1 ou o Pocket Operator. O VÆTTIR aprendeu com a Teenage Engineering a legibilidade visual e a ideia de que um instrumento deve ser imediatamente inteligível no que oferece, mesmo que complexo no que produz. A ausência de elementos decorativos supérfluos no painel do VÆTTIR, a lógica de colunas paralelas para os controlos, a clareza com que cada elemento ocupa o seu espaço, devem algo a esta estética. A Erica Synths, fabricante letão com raízes na cena modular europeia, contribuiu com uma perspetiva sobre a construção DIY de alta qualidade. Os seus módulos combinam performance técnica rigorosa com uma estética de hardware visível, as PCBs à vista como parte da identidade do objeto. Esta é uma escolha que o VÆTTIR adota literalmente, mantendo as duas PCBs expostas como declaração estética e como vínculo à tradição do circuit bending e da construção artesanal. Num mundo de sintetizadores modulares onde a proliferação de módulos é frequentemente sinal de poder e versatilidade, o VÆTTIR resiste deliberadamente a essa lógica de acumulação. A eletrónica à vista não é apenas uma escolha estética, mas também uma declaração de filiação. O instrumento não esconde o que é.

A patchbay do VÆTTIR, em particular, deve muito à filosofia Eurorack e à ideia de que o sinal de um sistema pode ser fisicamente mudado para outro através de um cabo é central à filosofia modular. Não há routing fixo, tudo é configurável. No VÆTTIR, a patchbay é digital (não é através de voltagem, mas inputs digitais que ativam conexões entre o atractor de Lorenz e os parâmetros que ele pode modular), mas o princípio e o gesto performativo são os mesmos. A utilização de superfícies tácteis capacitivas em instrumentos musicais digitais foi extensivamente estudada desde a popularização dos controladores multi-touch. O MPR121, o controlador utilizado no VÆTTIR para os pads, permite o instrumento ter esta

dimensão performativa através de doze elétrodos independentes sem a complexidade (e o custo) de um ecrã táctil. Truax (2001) argumenta que a relação física entre performer e instrumento, nomeadamente o gesto, é importante na expressividade musical, e deve preservar tanto quanto possível a relação entre intenção e resultado

A integração de síntese sonora com geração visual em tempo real tem uma longa história na arte eletrónica, desde os pioneiros da videoarte (Nam June Paik, Steina Vasulka) até às práticas contemporâneas de live coding e VJing. Softwares como o TouchDesigner (Derivative, 2023) representam o estado da arte em plataformas de performance audiovisual em tempo real, com capacidades de processamento de sinal MIDI, geração procedural de imagem e integração com hardware externo. A utilização do MIDI como protocolo de comunicação entre instrumentos musicais e sistemas visuais está bem documentada (Collins, 2006), embora a extensão do MIDI OUT a todos os parâmetros internos de um instrumento seja uma abordagem menos comum que o VÆTTIR adota.

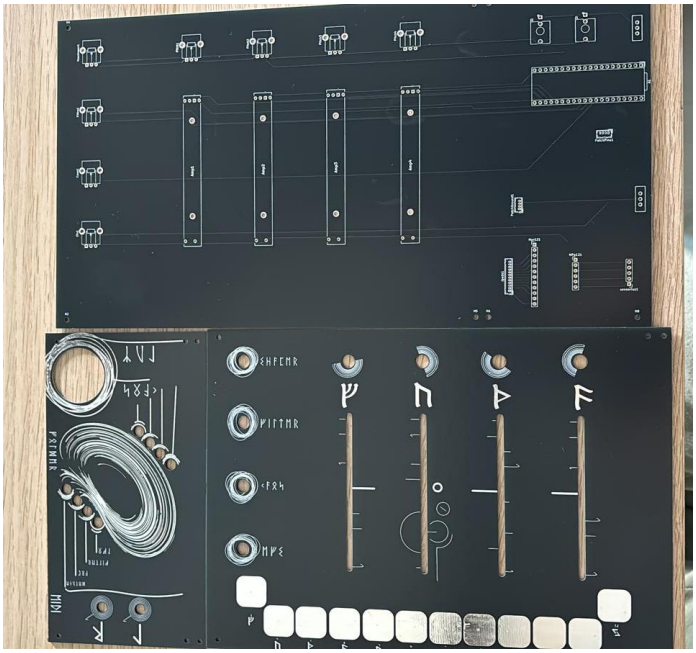


Figura 3- Placas principais

O Nome VÆTTIR: Etimologia, Simbolismo e Identidade

O nome vem da mitologia nórdica antiga: *Vættir* (singular *vættr*) são os espíritos da natureza na cosmologia nórdica pré-cristã: entidades que que habitam lugares, objetos e fenómenos, forças que não são nem completamente controláveis nem completamente imprevisíveis. Não são deuses, nem humanos, nem mortos. São algo intermédio, descritos como forças imanentes à natureza que têm vontade própria, mas respondem à relação que com elas se estabelece.

Num sentido profundo, o VÆTTIR como instrumento aspira a esse estatuto: não é uma ferramenta passiva nem um sistema autónomo. É algo intermédio, um sistema que tem a sua própria lógica interna, mas que responde ao toque, ao espaço e à intenção do performer. Não é uma reinterpretação de um instrumento existente nem uma tentativa de emular os clássicos. É uma proposta sobre o que um instrumento pode ser quando é pensado desde o início para um tipo específico de escuta e de performance. A escolha deste nome para o sintetizador não

é arbitrária. O instrumento tem uma lógica interna própria que não é completamente controlável pelo performer. É, nesse sentido, um *vættr*: tem vontade, tem carácter, responde ao contexto. A relação entre performer e instrumento é de diálogo.

Referências Estéticas e Técnicas

A referência musical mais próxima e mais direta é *keinseier* (keinseier.bandcamp.com), músico eletrónico baseado em Hamburgo cujo trabalho explora os limites entre glitch, estrutura e texturas contrastantes, frequentemente com configurações reduzidas e fontes sonoras estritamente limitadas. O som de keinseier define-se pela contenção, clareza estrutural e intimidade sónica, qualidades que informam diretamente o VÆTTIR. A abordagem de keinseier a álbuns como *Januarstücke* (2026), gravado em capturas únicas sem multitracks nem revisões, ou *Particles* (2024), com os seus drones rítmicos construídos a partir de sintetizadores esticados no tempo e gravações de campo reimaginadas como texturas, ecoa diretamente a filosofia do VÆTTIR: a performance como documento e a imperfeição como método. A ideia de que cada execução é um acontecimento único e irrepetível, que o instrumento deve responder ao momento em vez de reproduzir um resultado fixo, está no centro de ambos os trabalhos.

Alva Noto (Carsten Nicolai) é outra referência central, não tanto sonoramente, mas conceptualmente e como posição estética. Nicolai, figura seminal na música eletrónica minimal e glitch, trabalha na intersecção do som, dos dados e das estruturas visuais. A sua exploração de uma estética minimalista usando cliques digitais, glitches e ondas sinusoidais como materiais primários, informou a escolha de tornar visível o motor caótico do VÆTTIR em vez de o esconder. A ideia de que o glitch não é uma falha, mas um vocabulário expressivo, que os artefactos dos processos digitais têm as suas próprias qualidades tímbricas e texturais, vem diretamente do trabalho de Nicolai e do ensaio fundacional de Cascone (2000). O VÆTTIR não esconde o atractor de Lorenz: expõe-no na interface, faz dele o centro visual e conceptual do instrumento. *Ryoji Ikeda* representa a extremidade mais austera desta linhagem com os seus trabalhos em ambiente de concerto (*test pattern*, *data.matrix*, *supercodex*). Ikeda não procura a beleza no som, mas a precisão, e descobre que a precisão extrema se torna sublime.

La Monte Young é uma referência conceptual e de pensamento mais do que uma influência sonora direta. Young é o patriarca do drone como forma musical autónoma, não como textura de fundo, mas como objeto de contemplação. As suas peças mais longas (*The Well-Tuned Piano*, *Dream House*) estabelecem que o som sustentado no tempo é suficiente como proposição musical, que a perceção da mudança dentro da continuidade é em si mesma uma experiência estética completa. Esta ideia fundamenta a dimensão de drone do VÆTTIR. O instrumento pode existir apenas nesse modo, gerando texturas que mudam tão lentamente que a perceção da mudança é quase impercetível até que, olhando para trás, algo se transformou profundamente. Phill Niblock e as suas texturas de microdesafinação, Alvin Lucier e o espaço acústico como instrumento (*I Am Sitting in a Room* como exemplo extremo de como o espaço físico é um processador de som), e todo o panorama do noise japonês (Merzbow, Masonna, Government Alpha) como exploração de sistemas eletrónicos empurrados para regimes caóticos, são referências que situam o VÆTTIR numa tradição de música experimental que valoriza o processo, a imprevisibilidade e a fisicalidade do som sobre a composição prévia e a reprodutibilidade.

O Instrumento em Sistema: Solo e em Conjunto

O VÆTTIR foi concebido para funcionar tanto como instrumento autónomo como nó num sistema maior. Em modo solo, o instrumento é suficiente. As quatro vozes de drone, a patchbay, os pads capacitivos e o sensor de luz criam um universo sonoro completo que pode sustentar uma performance de duração arbitrária sem necessitar de elementos externos. A amplitude sonora suave de drone a noise denso garante que o instrumento solo não é mono-dimensional[1].

Em sistemas com outros instrumentos[2], o VÆTTIR comporta-se como um *centro gravitacional* através da sua natureza de drone, a sua tendência para preencher o espaço de forma contínua, tornando-o naturalmente um instrumento de fundo sobre o qual outros instrumentos podem atuar. Experiências em contexto de performance com outros sintetizadores e integrado em sistemas completos, revelaram que o VÆTTIR cria um campo harmónico e tímbrico que outros instrumentos habitam em vez de competirem com ele. A escala Hirajoshi dos pads, por ser pentatónica e de tensão não resolvida, harmoniza com um espectro largo de contextos tonais sem impor uma tónica forte.

O MIDI OUT (que transmite todos os parâmetros internos do instrumento) transforma o VÆTTIR num controlador[3]. Em integração com TouchDesigner[4], os valores caóticos do atractor podem conduzir geradores de partículas, deformar geometrias, ou controlar a velocidade de processos generativos visuais, criando uma correspondência entre o som e a imagem que não é de sincronização, mas de coevolução, o mesmo sistema caótico que governa o som governa a imagem.

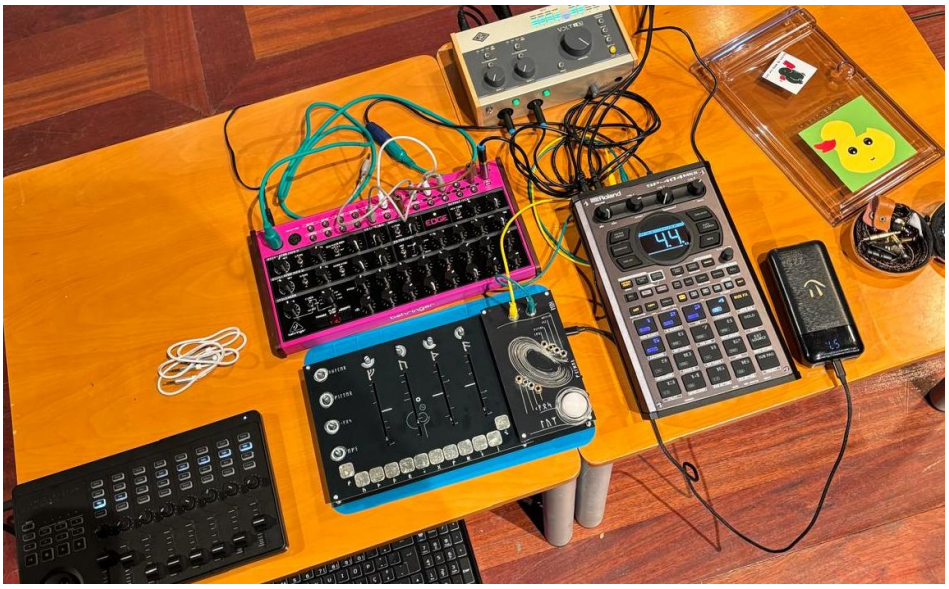


Figura 4- VÆTTIR integrado num sistema com outros instrumentos

[1] Vídeo de demonstração: VÆTTIR em modo solo: https://youtu.be/nT3orADI0fA
[2] VÆTTIR integrado num sistema com outros instrumentos: https://youtu.be/SR0KWz-v5a4
[3] VÆTTIR integrado num sistema como instrumento e controlador MIDI em simultâneo: https://youtu.be/ofJ4EJXamII
[4] VÆTTIR como controlador MIDI em sistema com TouchDesigner: https://youtu.be/ZNLXLTqrqcY

Objetivos do Projeto

O VÆTTIR define-se em torno de três eixos fundamentais que coexistem em tensão produtiva.

O primeiro eixo é expressivo: criar um instrumento que permita gerar texturas sonoras com uma qualidade particular, densa, imersiva, ligeiramente instável, adequada tanto para composição em estúdio como para performance ao vivo. O instrumento deve ser físico, táctil e imediato na resposta.

O segundo eixo é técnico: construir um sistema de síntese digital com firmware próprio, executado numa Daisy Seed, capaz de gerir quatro vozes de drone (cada uma com três osciladores internos), filtro SVF, waveshaping, efeitos stereo, sensor de luminosidade, superfície táctil, MIDI bidirecional e um sistema de caos determinístico baseado no atractor de Lorenz.

O terceiro eixo é identitário: criar uma identidade visual e filosófica coerente com a sonoridade do instrumento, ancorada num sistema rúnico próprio e num nome que carrega significado mitológico e simbólico.

Teoria do Caos, Atractores e o Efeito Borboleta

A palavra caos carrega no uso quotidiano uma conotação de desordem total, de ausência de regras, de ruído puro. Mas na matemática e na física, o caos tem um significado radicalmente diferente e muito mais interessante. O caos determinístico é o comportamento de sistemas governados por leis exatas e completamente definidas, cujas trajetórias são, no entanto, impossíveis de prever a longo prazo porque são extremamente sensíveis às condições iniciais.

Um sistema caótico determinístico tem três propriedades fundamentais:

é determinístico: dado exatamente o mesmo estado inicial, o sistema seguirá sempre exatamente a mesma trajetória;

é sensível às condições iniciais: duas trajetórias que comecem em estados quase idênticos divergem exponencialmente ao longo do tempo;

tem estrutura: a trajetória do sistema no espaço de estados não é aleatória, mas descreve padrões reconhecíveis, nunca se repetindo exatamente, mas nunca sendo completamente imprevisível (Strogatz, 2015).

O Atractor de Lorenz

Um atractor é a região do espaço de estados para a qual um sistema dinâmico converge a longo prazo. Para um pêndulo simples, o atractor é um ponto (repouso). Para um oscilador periódico, é um ciclo. Para sistemas caóticos, os atractores têm geometria muito mais complexa: são fractais, com dimensão não-inteira, estruturas que combinam confinamento e não-repetição de formas matematicamente extraordinárias. A trajetória é atraída para uma região do espaço de estados e não a escapa nem diverge para infinito, mas nunca se fecha sobre si mesma, nunca repete exatamente o mesmo estado. Esta região é um atractor estranho (*strange attractor*): uma estrutura geométrica de dimensão fractal que combina confinamento e não repetição de formas matematicamente extraordinárias (Gleick, 1987).

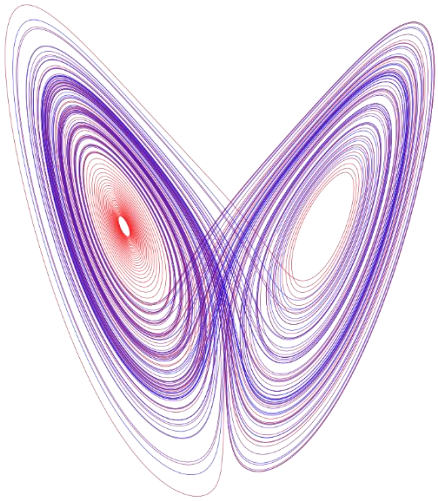


Figura 5- Lorenz Attractor

O atractor de Lorenz é o exemplo mais icónico. Descrito por Edward Lorenz em 1963 a partir de um modelo simplificado de convecção atmosférica, é governado por três equações diferenciais ordinárias com apenas três parâmetros (σ, ρ, β):

$$dx/dt = \sigma(y - x)$$

$$dy/dt = x(\rho - z) - y$$

$$dz/dt = xy - \beta z$$

A trajetória do sistema orbita em torno de dois pontos centrais, as célebres "asas da borboleta" na visualização clássica, alternando de forma aparentemente aleatória entre eles, sem nunca repetir exatamente o mesmo caminho, mas sem nunca sair da estrutura global do atractor.

Do ponto de vista prático, o atractor de Lorenz é amplamente utilizado em áreas que vão da modelação climática à engenharia de comunicações seguras, da análise de séries temporais biológicas à geração de variação paramétrica em síntese sonora. A sua utilidade nestas aplicações deriva exatamente das propriedades que o tornam musicalmente interessante: produz variação contínua, estruturada, sem repetição, com sensibilidade às condições iniciais. No VÆTTIR, o atractor corre em tempo real no firmware, integrado a cada iteração do loop principal (≈1 kHz), e os seus valores normalizados alimentam a patchbay.

A sensibilidade às condições iniciais que caracteriza os sistemas caóticos foi popularizada por Lorenz com a metáfora do efeito borboleta. A ideia é simples na formulação, mas profunda

nas implicações, famosamente descrita como “O bater de asas de uma borboleta na China pode causar um furacão nos EUA”. Esta metáfora descreve matematicamente o crescimento exponencial de pequenas diferenças iniciais. Se dois estados de um sistema caótico diferem de ε na sua condição inicial, essa diferença cresce aproximadamente como e^(λt), onde λ é o expoente de Lyapunov. Para o atractor de Lorenz com parâmetros clássicos, λ ≈ 0.9, o que significa que dois estados que diferem de 0.001 serão completamente distintos ao fim de apenas alguns segundos (Strogatz, 2015). Na prática, isto significa que *nunca* é possível conhecer exatamente o estado inicial de um sistema real e há sempre alguma incerteza de medição associada, por mais pequena que seja. Em sistemas caóticos, essa incerteza, por menor que seja, cresce até inundar toda a previsibilidade.

Caos na Natureza e na Música

O caos determinístico é omnipresente na natureza. a turbulência em fluidos, as variações climáticas, o batimento cardíaco, os padrões de respiração, o fumo de uma vela que sobe em espiral suave e de repente explode em redemoinhos imprevisíveis, a corrente de um rio que alterna entre fluxo laminar e turbulento, nenhum destes é periódico puro, mas caótico no sentido técnico. Um coração com ritmo perfeitamente periódico é, paradoxalmente, um sinal de patologia (Goldberger et al., 2002). A ordem perfeita é uma abstração que a natureza raramente realiza. Por baixo dessa aparente desordem há uma estrutura e padrões que se repetem em escalas diferentes, mas nunca exatamente da mesma forma.

A relação entre o caos e a música é mais antiga do que a teoria matemática que o descreve. Os músicos sempre souberam intuitivamente, antes de terem palavras para o formular, que a música viva não é exatamente periódica. O vibrato de uma voz humana, a variação de intensidade de um instrumentista ao longo de uma nota sustentada, as micro-variações de tempo num ensemble que toca em conjunto sem ser metronomicamente preciso, todas estas são formas de caos controlado que distinguem a música viva da música mecânica.

A tradição musical ocidental desenvolveu instrumentos e práticas que aproximam o som da periodicidade perfeita: o temperamento igual, o metrónomo, a afinação de referência a 440 Hz. E ao mesmo tempo, os músicos resistem sistematicamente a essa perfeição: o vibrato, o portamento, o rubato, a microtonalidade, são todos desvios deliberados da periodicidade que tornam a música expressiva e humana em vez de mecânica.

No contexto da música experimental e eletrónica, a relação com o caos tornou-se mais explícita. Os sistemas de feedback são sistemas caóticos que os músicos aprenderam a domar parcialmente e a usufruir da sua riqueza tímbrica e imprevisibilidade controlada (Sanfilippo & Valle, 2013). Todo o panorama do noise japonesa (Merzbow, Masonna, Government Alpha) é, entre outras coisas, uma exploração de sistemas eletrónicos empurrados para regimes caóticos onde o circuito já não opera no seu regime linear.

O VÆTTIR Como Sistema Caótico Musical

O VÆTTIR não simula o caos, mas incorpora-o como sistema dinâmico real. O atractor de Lorenz que corre no firmware não é uma lista de números aleatórios, nem uma função de aleatoriedade pseudoaleatória. É um sistema dinâmico genuíno, integrado em tempo real, cujas propriedades matemáticas se traduzem diretamente em qualidades sonoras e performativas específicas.

A sensibilidade às condições iniciais manifesta-se na impossibilidade de replicar exatamente uma performance ou uma peça porque o estado do atractor no início de uma sessão depende de toda a história de operação do sistema, e pequenas diferenças no momento exato em que se liga o instrumento, na temperatura dos componentes, ou na ordem em que os patches são

inseridos, amplificam-se para produzir trajetórias caóticas distintas. Duas performances com exatamente as mesmas posições de knobs nunca soam exatamente igual, não porque haja aleatoriedade, mas porque há sensibilidade às condições iniciais. O facto do knob de caos escalar σ, ρ e β de valores clássicos para valores extremos cria uma progressão musical do ordenado para o caótico que não é linear, mas exponencial. Com o knob a 10%, o atractor está próximo da bifurcação, o comportamento é quase periódico, com uma regularidade que pode ser sentida, mas não exatamente prevista. Com o knob a 50%, o caos está estabelecido e as trajetórias exploram todo o espaço do atractor. Com o knob a 100%, os parâmetros extremos criam um atractor muito mais veloz e denso, com transições mais rápidas e variações mais abruptas. O efeito borboleta tem uma expressão direta na patchbay quando por exemplo P3 (detune spread) e P4 (arpeggiator) estão ativos simultaneamente com caos alto, pequenas variações no valor do atractor propagam-se para múltiplos parâmetros sonoros em simultâneo e com dinâmicas diferentes. A interação entre estas modulações paralelas cria uma complexidade sonora que não pode ser descrita como a soma das suas partes porque emerge da interação não-linear entre elas, exatamente como o comportamento caótico emerge da interação não-linear das equações de Lorenz e também do performer que interage com o sistema. Há também uma dimensão que vai além da técnica, sendo ela uma qualidade orgânica, a sensação de que o instrumento está de alguma forma vivo e tem a sua própria respiração.

Um Instrumento Construído Para Si Próprio

O Problema dos Sintetizadores Comerciais

Há uma frustração que sinto como músico que trabalha com síntese durante o tempo suficiente para começar a reconhecer a sensação de que os instrumentos do mercado são, em certa medida, todos iguais. Não no sentido superficial (os knobs estão em sítios diferentes, as interfaces têm layouts distintos) mas num sentido mais profundo. São instrumentos desenhados para satisfazer um perfil de utilizador médio, um conjunto de casos de uso que cobre o suficiente para justificar o investimento de produção em massa, mas que raramente corresponde exatamente a quem os vai usar. Comprar um sintetizador comercial é, muitas vezes, comprar um conjunto de presets de fábrica com alguma capacidade de modificação à volta. O som característico do instrumento, a sua "voz própria", é ao mesmo tempo o seu ponto forte e o seu limite mais intransponível. Um Minimoog vai sempre soar a Minimoog: extraordinário no que faz, mas limitado ao que faz.

Para um performer que trabalha em múltiplos contextos sonoros e que numa sessão precisa de gerar um drone imersivo de vinte minutos, noutra de tocar linhas melódicas com sensibilidade táctil, noutras ainda de criar noise denso ou texturas ambientes delicadas, nenhum desses instrumentos serve plenamente. Serve em parte. Cobre alguns cenários e deixa outros completamente a descoberto. E a solução habitual (comprar mais instrumentos, um para cada contexto) é ao mesmo tempo financeiramente absurda e artisticamente insatisfatória, porque o vocabulário dos instrumentos nunca é exatamente o vocabulário que o performer precisa. Esta frustração foi o ponto de partida real do VÆTTIR. Não um interesse abstrato em síntese digital, embora esse interesse exista e seja genuíno, mas a necessidade muito prática de ter um instrumento que correspondesse exatamente ao que é necessário especificamente a este performer, nestas performances, com este vocabulário sonoro específico.

Construído Para Este Corpo

O VÆTTIR foi desenhado de dentro para fora, a partir de um conjunto muito concreto de perguntas: que sons é que preciso? Como é que os quero controlar fisicamente? Que gestos performativos são centrais ao meu trabalho? O que é que cada controlo deve fazer quando o toco em cena, com as mãos que tenho, na posição em que costumo estar?

A resposta a essas perguntas determinou cada decisão de design. O layout dos pads capacitivos, alinhados em baixo, com dimensão adequada para serem tocados sem olhar para eles, evocando a estética clássica de um piano, a posição dos knobs de amplitude e pitch um por voz, em colunas paralelas, responde ao facto de que os ajustes de volume e afinação são os gestos mais frequentes durante uma performance, e devem estar acessíveis imediatamente, sem procura e até patchbay, evocando as interfaces de eurorack. Mesmo a escolha da escala asiática tipo Hirajoshi para os pads não foi feita por exotismo, mas porque é a escala que, depois de testes, melhor funciona sobre as texturas de drone que o instrumento gera. Tem tensão suficiente para ser interessante, está harmonicamente relacionada com a afinação base, e tem a qualidade particular de nunca soar completamente resolvida, o que a torna ideal para música que não quer resolver. Este nível de especificidade é simplesmente impossível de obter num produto comercial. Não porque os fabricantes não o queiram oferecer, mas porque um produto de massa não pode ser desenhado para um único utilizador.

O VÆTTIR pode, e foi.

Amplitude Sonora

Um dos requisitos centrais era que o instrumento não ficasse preso a um único modo de ser. Os sintetizadores comerciais têm, na sua maioria, uma "personalidade sonora" definida que os torna facilmente reconhecíveis e facilmente previsíveis. O VÆTTIR foi concebido com a preocupação inversa de ser capaz de habitar territórios sonoros muito diferentes sem nunca soar exatamente igual a si próprio, e acima de tudo sem soar como outro instrumento que já existe. No modo de drone puro, com os quatro osciladores ativos, o spread de detune mínimo e o caos inativo, o VÆTTIR gera texturas sinusoidais lentas e densas que se aproximam de Phill Niblock ou das peças mais longas de La Monte Young, sendo sons que existem no tempo e que mudam tão lentamente que a perceção da mudança é quase impercetível até que, olhando para trás, se percebe que algo se transformou profundamente. É um som que convida à imersão. Com os pads capacitivos, o instrumento transforma-se num espaço de ressonância entre as notas que lembra um piano numa sala muito reverberante, ou um vibrafone sem pedal. Aqui o VÆTTIR comporta-se como um instrumento melódico, ainda que com a sua própria qualidade tímbrica que o distingue do convencional. Com o wavefolder no máximo e o caos total ativado através dos quatro patches simultaneamente, o instrumento converte-se em algo completamente diferente. Noise denso, onde as quatro vozes de drone explodem em frequências que o wavefolder transforma em distorção agressiva e o arpegiador fragmenta em padrões rítmicos não repetíveis. Entre estes extremos existem todos os estados intermédios: texturas ambient com tremolo lento e reverb longo; linhas politónicas em modo MIDI; drones com modulação lenta do filtro que criam a sensação de respiração; pads que criam camadas de textura em que o melódico e o textural coexistem sem hierarquia clara.

Esta amplitude sonora foi arquitetada deliberadamente sendo que cada sistema foi pensado para se relacionar musicalmente com os outros, de modo que as combinações produzem resultados que são mais do que a soma das partes.

A Interface Como Linguagem Física

Uma decisão que foi ponderada longamente foi a de incorporar no mesmo instrumento três paradigmas de interface que pertencem a tradições instrumentais distintas. Os potenciómetros rotativos, doze no total, organizados em quatro colunas de três, pertencem à linguagem dos sintetizadores analógicos clássicos: Moog, Roland, Korg. São a interface com que qualquer músico de sintetizadores cresceu a trabalhar, com a sua qualidade de controlo contínuo, o feedback táctil da resistência à rotação, a capacidade de fazer ajustes precisos ou rápidos com o mesmo gesto. No VÆTTIR, os knobs controlam os parâmetros que requerem precisão e ajuste contínuo como amplitudes, frequências, quantidade de processamento. São a camada de controlo "estável" do instrumento.

Os pads capacitivos evocam outra tradição: a dos controladores MIDI tácteis, mas também, mais proximamente, a dos teclados de piano. A disposição linear dos doze pads, com as runas dos nomes das notas abaixo de cada um, sugere conscientemente um teclado. Esta escolha de design cria uma interface que é simultaneamente familiar (para quem tem background de teclado) e estranha (a sensação táctil do pad capacitivo é completamente diferente de uma tecla mecânica, a escala não é a escala cromática standard), convidando a descobrir novas formas de tocar.

A patchbay com os quatro inputs de patch que ligam o sistema de caos a diferentes parâmetros pertence ao universo dos sintetizadores modulares Eurorack. A ideia de que o sinal de um sistema pode ser fisicamente roteado para outro através de um cabo é central à filosofia modular pois não há routing fixo, tudo é configurável. No VÆTTIR, a patchbay é interna e discreta, não são jacks de 3.5 mm através dos quais fluem tensões analógicas, mas inputs digitais que ativam ou desativam conexões entre o atractor de Lorenz e os parâmetros que ele pode modular, mas o princípio é o mesmo. O performer decide em performance quais as conexões que existem, e a configuração da patchbay é em si mesma uma decisão compositiva.

A coexistência destes três paradigmas num único instrumento compacto é, talvez, a síntese mais clara da filosofia do VÆTTIR. A decisão de não escolher entre tradições, mas criar um espaço onde elas coexistem e se informam mutuamente. O performer pode usar o instrumento como um sintetizador clássico de knobs, como um controlador de pads, como um módulo modular ou como todos eles em simultâneo, dependendo do contexto e do momento da performance.

Anatomia do VÆTTIR: Como Funciona o Instrumento

Esta secção descreve o funcionamento do instrumento[5] do ponto de vista do performer, o que cada controlo faz, como os elementos interagem, e como a interface física se relaciona com o sistema interno.

As Quatro Vozes de Drone e as Runas Centrais — ᚠ ᚢ ᚦ ᚨ

O coração sonoro do VÆTTIR são as quatro vozes de drone, identificadas na interface pelas runas centrais ᚠ, ᚢ, ᚦ e ᚨ. As quatro runas grandes que dominam a interface são do Elder Futhark, o mais antigo alfabeto rúnico documentado (século II–VIII d.C.), com correspondências tanto fonéticas como simbólicas. A sua escolha para identificar as quatro vozes de drone é uma correspondência com o significado simbólico:

ᚠ **Fehu** ("riqueza, energia, criação") - A primeira voz opera na região grave, com frequência base em 55 Hz (A1). É a fundação harmónica do sistema, a raiz da tríade menor. Quando o caos atua sobre ela, a sua energia lenta e densa pode expandir-se ou contrair-se de formas que sugerem respiração ou pulsação geológica. O nome Fehu, associado à energia primordial, ao gado, à riqueza como força viva, é adequado para uma voz que é simultaneamente fundação e motor.

ᚢ **Uruz** ("força, poder, resistência") - A segunda voz opera na terça menor acima da raiz (aproximadamente 65.4 Hz transposto para o registo de 100-300 Hz conforme a configuração), conferindo ao sistema a tensão harmónica característica do modo menor. Uruz, a runa do auroque selvagem, traz uma qualidade de força não domesticada, algo que empurra contra os limites e tem resistência própria.

ᚦ **Thurisaz** ("caos, defesa, transformação") - A terceira voz opera na quinta justa (250–450 Hz), criando o espaço harmónico. Thurisaz é a runa do gigante, do limiar, da transformação violenta e a quinta tem essa qualidade. É simultaneamente estável (é o segundo harmónico mais forte) e estranha, capaz de abrir espaços harmónicos que a raiz e a terça por si só não criam.

ᚨ **Ansuz** ("comunicação, inspiração, conhecimento") - A quarta voz opera na oitava (400–800 Hz), dobrando a raiz base num registo superior. Ansuz é a runa de Odin, do sopro criador, da palavra que dá vida, e a oitava tem essa qualidade de repetição com transformação, de ser simultaneamente a mesma nota e algo diferente, mais próximo do éter.

Cada voz tem dois controlos diretos: um potenciómetro de **amplitude** (A0–A3) e um potenciómetro de **pitch**(A4–A7). O potenciómetro de pitch controla um desvio de ±9 semitons em relação à frequência base da voz. Adicionalmente, o sensor de luminosidade BH1750 introduz um desvio suplementar de ±1 semitom consoante a luminosidade do ambiente sendo uma forma subtil de fazer o instrumento respirar com o espaço em que atua.

Crucialmente, cada voz não é um único oscilador, mas três osciladores sinusoidais ligeiramente desafinados entre si. A diferença de afinação entre eles (*detune spread*) é normalmente mínima. Quando o caos atua através do patch P3, este spread pode expandir-se drasticamente. Esta arquitetura de osciladores em clusters é a fonte da qualidade orgânica

[5] Tutorial completo e explicação do VÆTTIR: https://youtu.be/FMjcOkXzoQE

e ligeiramente instável do som do VÆTTIR em repouso, mesmo sem caos ativo, o instrumento respira.

Os Quatro Knobs da Direita

Na coluna direita da interface, quatro potenciómetros controlam o processamento global do sinal. Dde cima para baixo na interface física:

Waveshaper (A8) - Este controlo governa o *waveshaper*, que transforma a forma de onda dos osciladores de sinusoidal em formas progressivamente mais ricas em harmónicos. Com o knob no mínimo, o sinal passa inalterado. À medida que o knob aumenta, um ganho de até 40× é aplicado ao sinal antes de este ser saturado entre -1 e +1 introduzindo harmónicos ímpares e pares que transformam a qualidade sinusoidal limpa num som mais agressivo. O *waveshaper* é misturado com o sinal original na proporção definida pelo próprio knob, permitindo um controlo suave da quantidade de harmónicos adicionados. Existem também três modos de wavefold selecionáveis por botão do lado direito do instrumento: *average* (média simples, sem fold), *sine wave fold* (dobra sinusoidal, mais suave) e *triangle wave fold* (dobra triangular, mais agressiva com harmónicos mais fortes).

Filtro (A9) - Um filtro *state variable filter* (SVF) passa-baixo com frequência de corte variável entre 100 Hz e 9 kHz. O SVF é um dos filtros mais utilizados em síntese analógica porque oferece múltiplas saídas (passa-baixo, passa-alto, passa-banda) a partir do mesmo circuito de processamento. No VÆTTIR, apenas a saída passa-baixo é utilizada, com ressonância fixa em 0.3, suficiente para um ligeiro realce à frequência de corte sem criar auto-oscilação. Com o patch P2 ativo e caos presente, a frequência de corte é modulada em tempo real pelo atrator de Lorenz.

Chaos / Lorenz (A10) - Este é o knob mais crítico da interface pois controla a **intensidade do atractor de Lorenz**, que é o motor de variação interna de todo o sistema. Com o knob no mínimo, o atrator existe, mas produz valores próximos de zero e o instrumento comporta-se de forma estática. À medida que o knob aumenta, os parâmetros do atractor (sigma, rho, beta) escalam para valores progressivamente mais extremos, resultando num comportamento cada vez mais errático. O valor normalizado do atractor (-1 a +1) alimenta, via patchbay, o tremolo (P1), a modulação do filtro (P2), o spread de detune (P3) e o arpegiador (P4). Este knob é, em essência, o controlo do *quanto caos* existe no sistema.

FX / Reverb + Delay Ping-Pong (A11) - O último knob controla a mistura entre o sinal seco e o sinal processado pelos efeitos stereo. Uma reverberação de longa duração (ReverbSc, com feedback de 70%) combinada com um delay ping-pong assimétrico (450 ms no canal esquerdo, 600 ms no direito, feedback de 45%). Com o knob no mínimo, o sinal é completamente seco.

A Patchbay do Caos e o Símbolo do Atractor de Lorenz

À direita da interface, marcada pelo diagrama do atractor de Lorenz, situa-se a **patchbay**. Quatro entradas de patch que ligam o sistema de caos interno a diferentes parâmetros sonoros. Cada entrada é lida como sinal digital (ativo/inativo), e a sua ativação, combinada com o nível de caos definido pelo knob A10, determina o comportamento em cada dimensão.

P1 - Tremolo: Com um cabo de patch inserido em P1 e o knob de caos acima de zero, o atractor de Lorenz modula a amplitude global do sinal de saída a uma taxa que varia entre 0.2 Hz e 15 Hz conforme a intensidade do caos. O efeito é um tremolo (variação rítmica de amplitude). A profundidade do tremolo escala de 30% (caos baixo) até 100% (caos máximo), podendo criar desde uma pulsação subtil até um corte total do sinal em momentos de caos extremo.

P2 - Modulação do Filtro: Com P2 ativo, o valor instantâneo do atractor de Lorenz modula a frequência de corte do filtro SVF, adicionando ao valor definido pelo potenciómetro um desvio de até ±9000 Hz. Na prática, isto significa que o timbre do instrumento oscila continuamente entre mais brilhante e mais escuro.

P3 - Detune Spread: Com P3 ativo, o spread de desafinação entre os três osciladores de cada voz expande-se de 2 cents (valor base) até 36 semitons no máximo de caos. Este é, possivelmente, o patch mais dramaticamente transformador. Com caos baixo, o efeito é uma textura ligeiramente mais rica. Com caos alto, as vozes explodem em frequências que se sobrepõem e interferem entre si de formas que não são musicais.

P4 - Arpeggiator: Com P4 ativo, o atractor de Lorenz controla um arpeggiator que distribui a atenção entre as quatro vozes. Quando o valor caótico é positivo, o arpeggiator avança para a frente (voz seguinte); quando é negativo, recua. As vozes não selecionadas ficam atenuadas para 15% da sua amplitude, criando um acento rítmico que varia de tempo e direção. Em momentos de caos elevado, o arpeggiator pode avançar dois passos de cada vez.

O **diagrama do atractor de Lorenz** que ocupa visualmente a zona da patchbay é uma representação matematicamente precisa da trajetória do sistema no plano XZ, gerada pela integração das equações com os parâmetros clássicos ($\sigma = 10$, $\rho = 28$, $\beta = 8/3$). As duas espirais interligadas que nunca se fecham são a "assinatura visual" do caos determinístico. A sua presença na interface comunica ao performer a natureza do motor interno do instrumento.

Figura 6- A patchbay do caos com o diagrama do atractor de Lorenz. Os quatro pontos de patch (P1–P4) emergem visualmente das espirais do atractor.

Os Pads Capacitivos e a Escala Hirajoshi

Os 12 pads capacitivos na zona inferior, controlados pelo MPR121, estão afinados numa escala de inspiração Hirajoshi em A (55 Hz, transposta uma oitava), com os intervalos em semitons: 0, 2, 3, 7, 8, 12, 14, 15, 19, 20, 24, 26. Esta escala pentatónica japonesa tem uma qualidade de tensão não resolvida que complementa as texturas de drone sem as dominar. Cada pad tem ataque lento (300 ms) e libertação muito longa (2 s), criando uma qualidade de ressonância que lembra instrumentos de percussão de grande câmara. Cada pad está etiquetado com uma runa do sistema próprio do VÆTTIR, criando uma correspondência entre o símbolo visual e o som que representa (não fonética, mas simbólica).

Figura 7- Os 12 pads capacitivos do VÆTTIR com as etiquetas rúnicas da escala Hirajoshi.

Sensor de Luz BH1750

O sensor de luminosidade BH1750 introduz o ambiente físico como parâmetro do instrumento. Mede a luminosidade em lux até 2000 lux e converte esse valor num desvio de afinação de ±1 semitom aplicado igualmente a todas as vozes. A suavização exponencial (alpha = 0.05) garante que as transições de afinação ocorrem lentamente. Em ambientes onde a luz muda dramaticamente, este sensor torna o instrumento sensível ao espaço que habita.

Modo MIDI IN

Ativando o switch D7 (MIDI ON), o VÆTTIR entra em modo MIDI[6], onde as frequências das quatro vozes passam a ser controladas por notas MIDI recebidas nos canais 1-4. Cada canal corresponde a uma das quatro vozes (𝄢, 𝅝, 𝄡, 𝄞), com polifonia de até quatro vozes por canal. As envolventes de amplitude são implementadas por software (ataque de 2 ms, libertação de 30 ms), eliminando cliques. O sistema de desvio de afinação (sensor de luz, potenciómetros de pitch, atractor de Lorenz via patches) continua ativo em modo MIDI, o que significa que as notas recebidas são sujeitas ao mesmo caos que governa o modo drone.

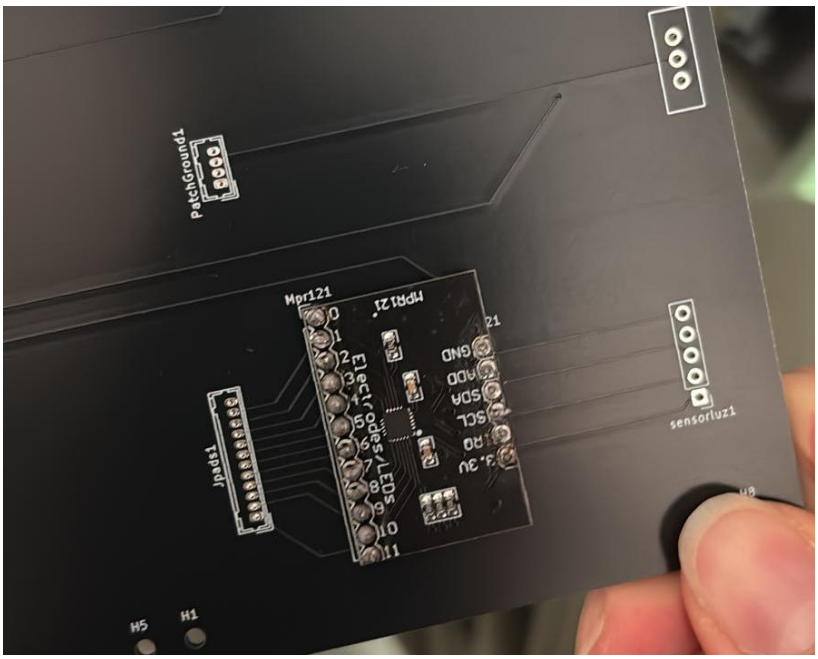


Figura 8- Detalhes do MPR121 na PCB principal do circuito

[6] Tutorial completo e explicação da implementação dos modos MIDI no VÆTTIR: https://youtu.be/WxxWoIvJ4Cw

Desenvolvimento Técnico

O código-fonte integral do firmware do VÆTTIR encontra-se no Anexo 1 deste documento. As secções seguintes descrevem a arquitetura e a lógica do sistema.

Arquitetura Geral

O firmware do VÆTTIR organiza-se em dois contextos de execução que correm em paralelo: o **audio callback** e o **loop principal**.

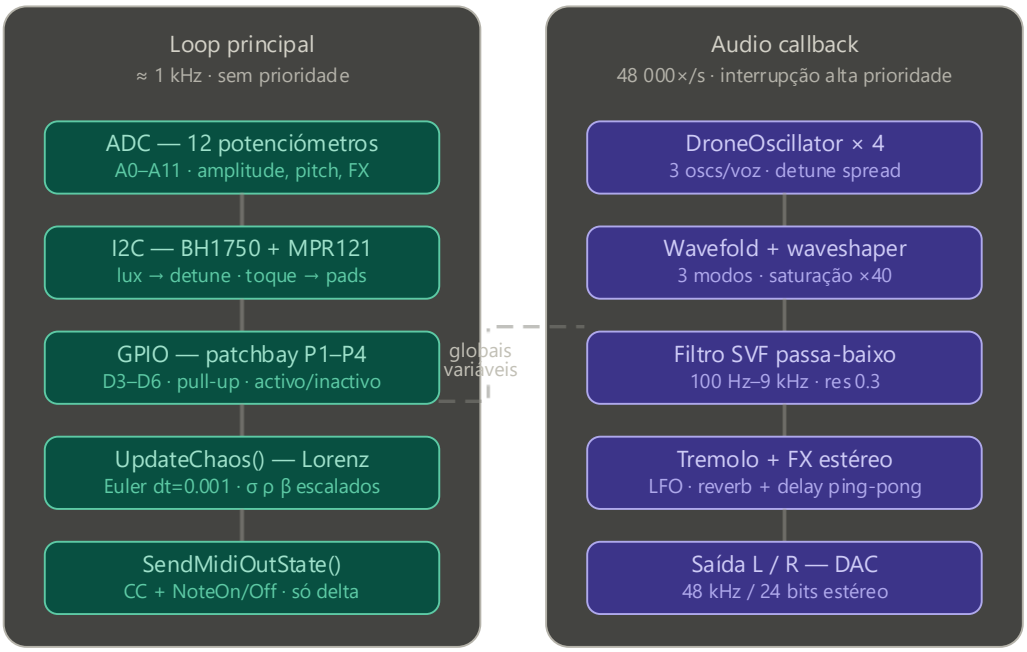


Figura 9- Arquitetura geral. O VÆTTIR corre dois contextos em paralelo

O audio callback é executado em interrupção de alta prioridade pelo hardware de áudio da Daisy Seed, chamado exatamente 48000 vezes por segundo (ou em blocos de tamanho configurável). É aqui que toda a geração de som acontece. Os osciladores são processados amostra a amostra, o waveshaping e o filtro são aplicados, os efeitos stereo são calculados e o resultado é enviado para os conversores. Este contexto deve ser o mais eficiente possível pois qualquer atraso produz artefactos audíveis. O loop principal corre no contexto normal do processador, sem prioridade especial, a aproximadamente 1 kHz (iteração a cada 1 ms). É aqui que acontece tudo o resto desde a leitura dos potenciómetros, polling dos sensores I2C, leitura dos inputs digitais (patchbay), processamento dos switches, atualização do atractor de Lorenz e envio de MIDI OUT. Os valores calculados no loop principal são escritos em variáveis globais que o audio callback lê em cada amostra. Esta separação é importante para a qualidade de áudio: o audio callback nunca espera por operações I2C lentas nem por transmissões MIDI; lê apenas variáveis que foram atualizadas pelo loop principal no ciclo anterior. A implementação completa de ambos os contextos, pode ser consultada no **Anexo 1**, nas funções audio_callback() e main().

A Classe DroneOscillator

A classe DroneOscillator é o bloco de geração sonora. Cada instância contém três osciladores sinusoidais independentes, um valor de amplitude e limites de frequência. O método set_semitone calcula as três frequências a partir da base e do spread global (g_detune_spread), que varia entre 0.02 cents (estático) e 36 semitons (caos máximo). O método process() devolve a média dos três osciladores multiplicada pela amplitude. O sistema mantém um banco de g_osc[NUM_TONES] [MAX_POLY], quatro vozes, cada uma com quatro slots de polifonia MIDI. Em modo drone, apenas o slot 0 de cada voz é usado; em modo MIDI, todos os quatro slots podem estar ativos simultaneamente

O Atractor de Lorenz no Firmware

O atractor de Lorenz é governado por três equações diferenciais ordinárias:

$$dx/dt = \sigma\,(y - x) \quad | \quad dy/dt = x\,(\rho - z) - y \quad | \quad dz/dt = xy - \beta z$$

O atractor é integrado a cada iteração do loop principal (≈1 kHz) usando o método de Euler com passo dt = 0.001. (ver função UpdateChaos() no **Anexo 1**). Os parâmetros σ, ρ e β são escalados pelo knob de caos: com parâmetros clássicos (σ=10, ρ=28, β=8/3) o atractor exibe o comportamento documentado por Lorenz (1963). À medida que σ e ρ aumentam, o sistema acelera e a trajetória no espaço de fase torna-se mais densa e errática. Com σ=90, ρ=168, a 1 kHz de integração corresponde a dezenas de "órbitas completas" por segundo, produzindo variação extremamente rápida que se manifesta como ruído quase-branco nos parâmetros modulados.

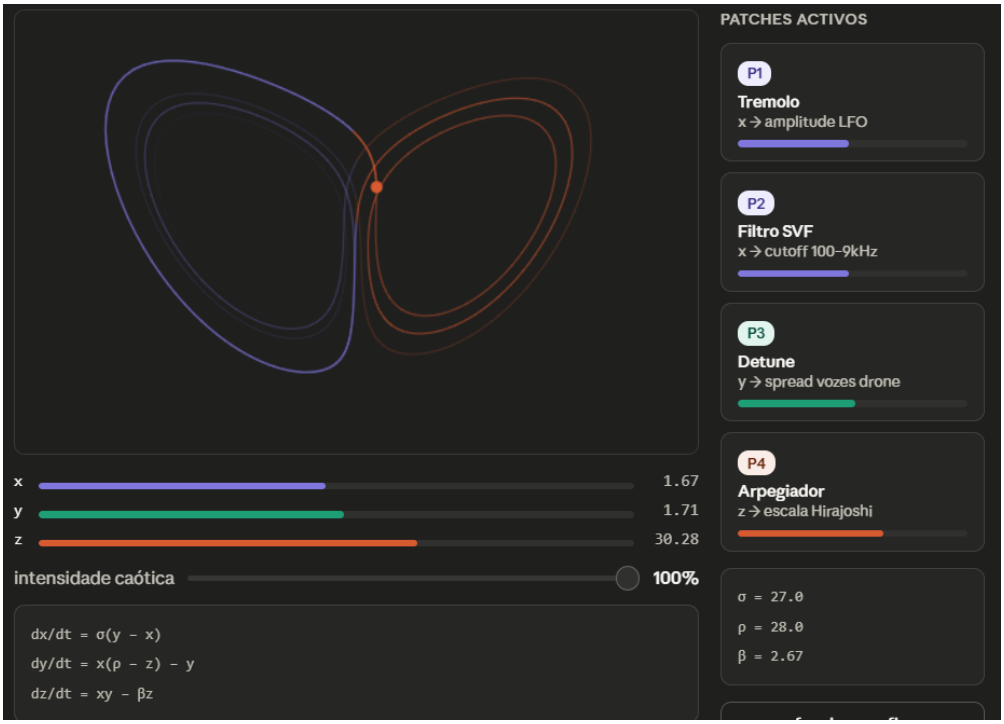


Figura 10- Influencia e mapeamento do atractor de Lorenz e mapeamento do caos

Fluxo de Sinal no AudioCallback

Para cada amostra, o processamento segue a cadeia:

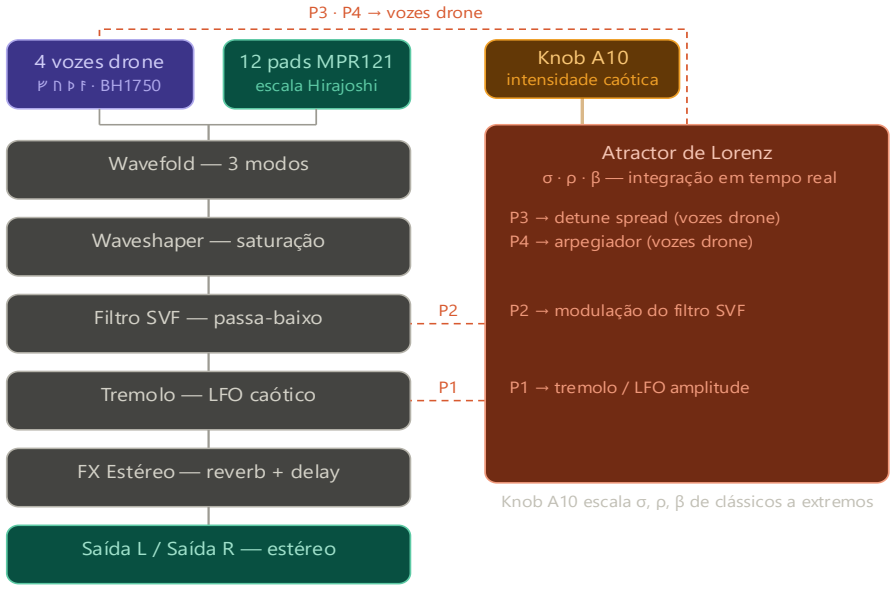


Figura 11- Diagrama do fluxo de sinal áudio do VÆTTIR

Em detalhe, para cada amostra:

Geração das vozes de drone: Cada DroneOscillator é processado. Se o arpegiador (P4) estiver ativo, o ganho de cada voz é escalado por g_arp_gain[t].
Geração dos pads: As 12 vozes do MPR121 são processadas com as suas amplitudes individuais.
Soma/Wavefold: As saídas são somadas e o modo de fold é aplicado: divisão simples (AVERAGE), sin(x)(SINE_WAVE_FOLD) ou função triangular periódica (TRIANGLE_WAVE_FOLD).
Waveshaper: Se g_waveshaper_amount > 0, o sinal é misturado com uma versão saturada na proporção definida pelo knob.
Filtro SVF: O sinal passa pelo filtro passa-baixo com a frequência de corte atual (modificada pelo caos se P2 ativo).
Tremolo: Se P1 e caos ativos, um LFO sinusoidal modula o ganho do master.
FX stereo: O sinal mono é alimentado nas linhas de delay (L e R com tempos diferentes) e no reverb, misturado com o sinal seco na proporção do knob de FX.

Evolução do Firmware

Versão 1.0 Prova de Conceito: Quatro osciladores simples, oito potenciómetros (amplitude e pitch), saída mono. Sem filtro, sem efeitos, sem caos.

Versão 1.5 Processamento: Adição do filtro SVF e do waveshaper. concluiu-se que a ordem do processamento importa musicalmente. O Waveshaper antes do filtro produz resultado diferente do filtro antes do waveshaper.

Versão 2.0 Caos: Introdução do atrator de Lorenz e da patchbay.

Versão 2.5 Sensores: Integração I2C com BH1750 e MPR121.

Versão 3.0 MIDI Completo: MIDI IN/OUT bidirecional, modo MIDI com quatro vozes polifónicas por canal, MIDI OUT de todos os estados internos. O principal problema foi que o envio MIDI a cada 10 ms introduzia jitter. Decidiu-se enviar apenas CCs com valor alterado desde o ciclo anterior (ver função SendMidiOutState() no **Anexo 1**).

Versão 3.5 (atual): Otimizações gerais, calibração dos parâmetros do Lorenz para comportamento musical em toda a gama do knob, ajuste dos thresholds MPR121, calibração dos pads.

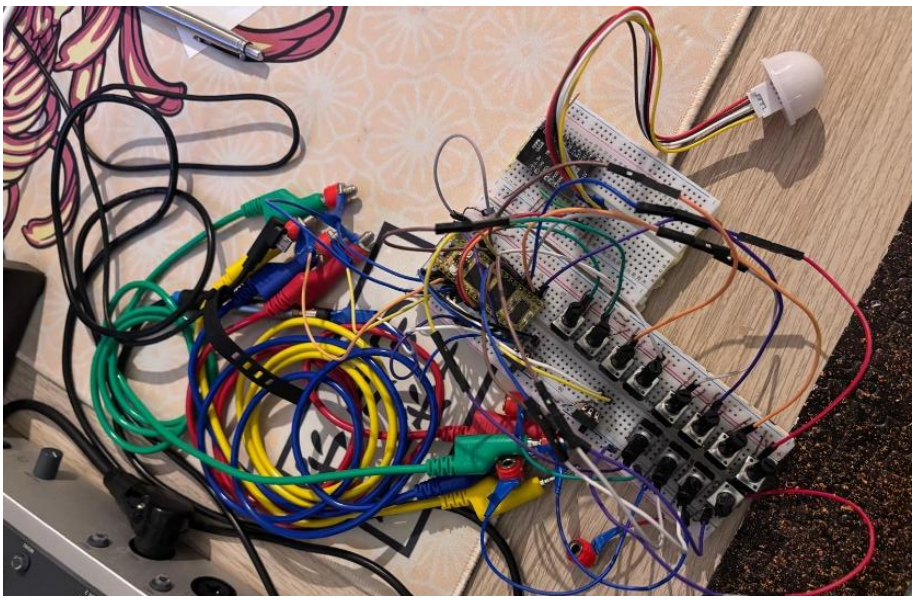


Figura 12- Documentação do processo de construção do VÆTTIR: protoboard inicial

Mapa MIDI Completo do VÆTTIR

O VÆTTIR implementa MIDI bidirecional completo. As tabelas seguintes documentam todos os eventos MIDI gerados e recebidos pelo instrumento.

MIDI OUT - Controlos Contínuos (CC)

Enviados a cada ≈10 ms apenas quando o valor mudou, no canal MIDI 1.

CC	Controlo Físico	Pino ADC	Gama de Valores	Parâmetro Interno
20	Knob Amplitude ᚠ	A0 (D15)	0–127	Amplitude da voz 1 (drone ᚠ)
21	Knob Amplitude ᚢ	A1 (D16)	0–127	Amplitude da voz 2 (drone ᚢ)
22	Knob Amplitude ᚦ	A2 (D17)	0–127	Amplitude da voz 3 (drone ᚦ)
23	Knob Amplitude ᚨ	A3 (D18)	0–127	Amplitude da voz 4 (drone ᚨ)
24	Knob Pitch ᚠ	A4 (D19)	0–127	Pitch ᚠ: 64 = centro, ±9 semitons
25	Knob Pitch ᚢ	A5 (D20)	0–127	Pitch ᚢ: 64 = centro, ±9 semitons
26	Knob Pitch ᚦ	A6 (D21)	0–127	Pitch ᚦ: 64 = centro, ±9 semitons
27	Knob Pitch ᚨ	A7 (D22)	0–127	Pitch ᚨ: 64 = centro, ±9 semitons
28	Knob Waveshaper	A8 (D23)	0–127	Intensidade do waveshaper (0=limpo)
29	Knob Filtro	A9 (D24)	0–127	Freq. de corte LP: 0=100Hz, 127=9kHz
30	Knob Chaos	A10 (D25)	0–127	Intensidade do atractor de Lorenz
31	Knob FX	A11 (D28)	0–127	Mix reverb + delay ping-pong

MIDI OUT - Patchbay (CC)

Enviados apenas quando o estado muda, canal MIDI 1.

CC	Entrada	Pino Digital	Valor OFF	Valor ON	Função quando ativo
40	P1	D3	0	127	Ativa tremolo
41	P2	D4	0	127	Ativa modulação do filtro
42	P3	D5	0	127	Ativa spread do detune
43	P4	D6	0	127	Ativa arpeggiator

MIDI OUT - Pads Capacitivos MPR121 (Note On/Off)

Enviados quando o estado de toque de cada elétrodo muda, canal MIDI 1.

Nota MIDI	Nota	Pad	Semitom (rel. A2)	Frequência aprox.	Símbolo Rúnico
60	C4	1	0 (A)	110 Hz	ᚠ
61	C#4	2	2 (B)	123 Hz	ᚢ
62	D4	3	3 (C)	130 Hz	ᚦ
63	D#4	4	7 (E)	165 Hz	ᚨ
64	E4	5	8 (F)	175 Hz	ᚱ
65	F4	6	12 (A oct.)	220 Hz	ᚲ
66	F#4	7	14 (B oct.)	247 Hz	ᚷ
67	G4	8	15 (C oct.)	261 Hz	ᚹ

68	G#4	9	19 (E oct.)	330 Hz	ᚺ
69	A4	10	20 (F oct.)	349 Hz	ᚾ
70	A#4	11	24 (A 2ª oct.)	440 Hz	ᛁ
71	B4	12	26 (B 2ª oct.)	494 Hz	ᛃ

Velocity fixa em 100 para Note On; velocity 0 para Note Off.

MIDI IN - Notas (Modo MIDI Ativo)

Recebidos quando o botão D7 (MIDI ON) está pressionado.

Canal MIDI	**Voz**	**Runa**	**Registo Preferencial**	**Polifonia**
1	Voz 1	ᛦ	Grave (40–200 Hz)	4 vozes
2	Voz 2	ᚢ	Médio-grave (100–300 Hz)	4 vozes
3	Voz 3	ᚦ	Médio (250–450 Hz)	4 vozes
4	Voz 4	ᚨ	Médio-agudo (400–800 Hz)	4 vozes

Em modo MIDI, as notas recebidas substituem as frequências base das vozes, mas todos os processamentos (detune spread, modulação do Lorenz via patches, sensor de luz) continuam ativos. O All Notes Off (CC 123) em qualquer canal ativa a libertação suave de todas as notas desse canal.

Diagrama de Fluxo MIDI

ENTRADAS FÍSICAS MIDI OUT (Canal 1)

═══════════════ ═══════════════

Knob A0 (Amp ᚠ) ──────────► CC 20 (0–127)

Knob A1 (Amp ᚢ) ──────────► CC 21

Knob A2 (Amp ᚦ) ──────────► CC 22

Knob A3 (Amp ᚨ) ──────────► CC 23

Knob A4 (Pitch ᚠ) ──────────► CC 24

Knob A5 (Pitch ᚢ) ──────────► CC 25

Knob A6 (Pitch ᚦ) ──────────► CC 26

Knob A7 (Pitch ᚨ) ──────────► CC 27

Knob A8 (Waveshape) ──────────► CC 28

Knob A9 (Filtro) ──────────► CC 29

Knob A10 (Chaos) ──────────► CC 30

Knob A11 (FX) ──────────► CC 31

Patch P1 ativo ──────────► CC 40 (0 ou 127)

Patch P2 ativo ──────────► CC 41

Patch P3 ativo ──────────► CC 42

Patch P4 ativo ──────────► CC 43

Pad 1 tocado ──────────► Note On 60, vel=100

Pad 1 solto ──────────► Note Off 60

Pad 2 tocado ──────────► Note On 61, vel=100

...

Pad 12 tocado ──────────► Note On 71, vel=100

MIDI IN (Canais 1–4) VOZES INTERNAS

Note On Ch.1 → voz ᵸ ——————► g_osc [0] [slot]

Note On Ch.2 → voz ᑎ ——————► g_osc [1] [slot]

Note On Ch.3 → voz ᑭ ——————► g_osc [2] [slot]

Note On Ch.4 → voz ᖴ ——————► g_osc [3] [slot]

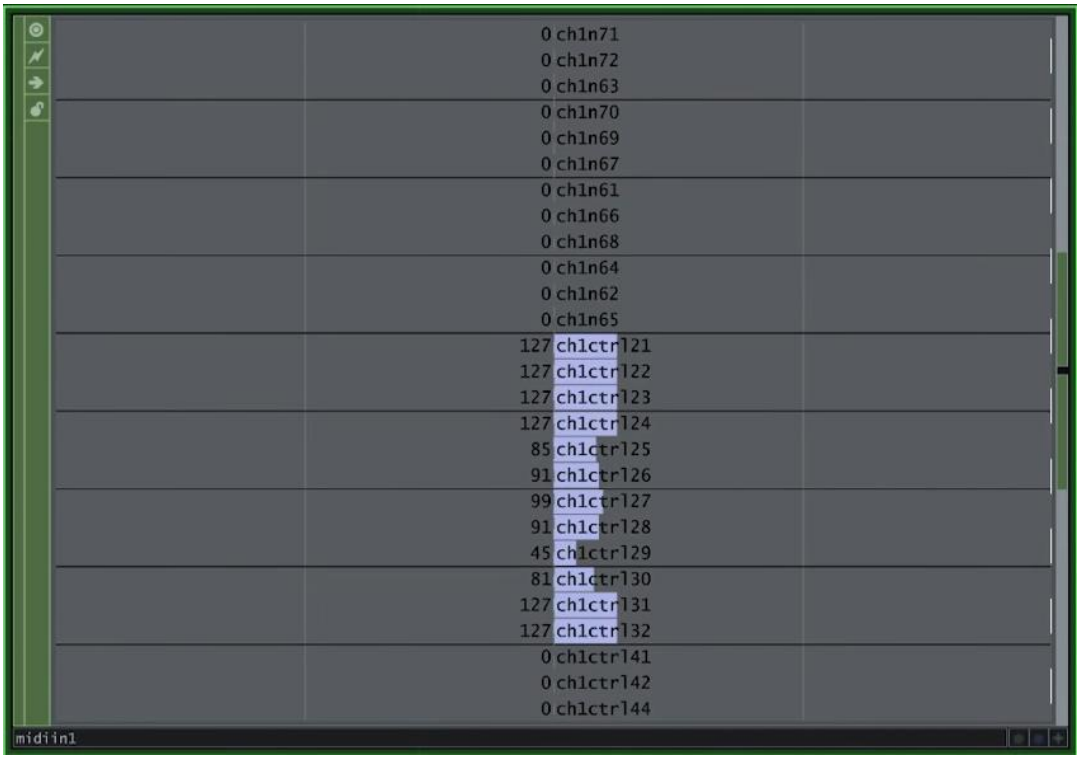


Figura 13- sinais MIDI recebidos do instrumento no TouchDesigner

Conclusão: O Instrumento como Colaborador

Um instrumento que incorpora caos determinístico não responde apenas à pergunta técnica sobre como gerar variação sonora não-repetitiva. Responde a uma pergunta mais profunda sobre a natureza do controlo na prática musical. Operar o VÆTTIR é operar num sistema que se entende parcialmente: as equações são conhecidas, os parâmetros são ajustáveis, mas o resultado específico de cada configuração de patch e cada posição de knob não é completamente previsível. É precisamente essa parcialidade que cria o espaço para a surpresa e é no espaço da surpresa que acontece a música que não poderia ter sido composta antecipadamente.

O instrumento que resultou deste processo é, em certo sentido, um argumento de que a instabilidade e a imprevisibilidade não são limitações a superar, mas qualidades a cultivar. Que a relação entre o músico e o instrumento é mais rica quando é de diálogo do que quando é de dominação. Estas não são ideias novas, estão implícitas em toda a tradição da música experimental, de John Cage a Alvin Lucier, de La Monte Young ao noise japonês. O VÆTTIR não as inventa, tenta realizá-las numa forma específica e pessoal, construída de raiz para este performer, para estas performances, para este vocabulário sonoro.

Há algo de paradoxal em construir um instrumento para descobrir o que ele pode fazer. O processo de desenvolvimento foi tanto de engenharia quanto de escuta e cada versão do firmware revelou possibilidades que a anterior não antecipava, e algumas das decisões de design mais significativas resultaram de acidentes que o processo transformou em funcionalidades. A ordem do processamento, a escolha da escala, o comportamento interno, nenhum destes foi planeado. Foram descobertos. Esta é talvez a lição mais importante do projeto. Os instrumentos ensinam-nos a tocá-los, e os melhores instrumentos ensinam-nos coisas que não sabíamos que queríamos saber.

O VÆTTIR não está acabado. Provavelmente nunca estará. Mas essa incompletude não é uma falha do processo, é a sua condição natural. Como os *vættir* da mitologia nórdica, o instrumento habita o espaço entre o controlado e o selvagem, e é precisamente nesse espaço que reside o que é mais interessante nele.

Bibliografia

Boulanger, R., & Lazzarini, V. (Eds.). (2011). The audio programming book. MIT Press.

Cascone, K. (2000). The aesthetics of failure: "Post-digital" tendencies in contemporary computer music. Computer Music Journal, 24(4), 12–18. https://doi.org/10.1162/014892600559489

Collins, N. (2006). Handmade electronic music: The art of hardware hacking. Routledge.

Electro-Smith. (2023). Daisy seed — hardware reference and libDaisy documentation. https://electro-smith.github.io/libDaisy/

Ghazala, R. (2005). Circuit-bending: Build your own alien instruments. Wiley Publishing.

Gleick, J. (1987). Chaos: Making a new science. Viking.

Goldberger, A. L., et al. (2002). Fractal dynamics in physiology: Alterations with disease and aging. PNAS, 99 (Suppl. 1), 2466–2472.

keinseier. (2024). Particles [album]. Bandcamp. https://keinseier.bandcamp.com/

keinseier. (2026). Januarstücke [album]. Bandcamp. https://keinseier.bandcamp.com/

Kim-Boyle, D. (2010). Sound spatialization with chance operations and cellular automata. Journal of New Music Research, 39(2), 139–153.

Lorenz, E. N. (1963). Deterministic nonperiodic flow. Journal of Atmospheric Sciences, 20(2), 130–141.

Make Noise Music. (2022). The *0.COAST* — operation guide. https://www.makenoisemusic.com/

Mathews, M. V. (1963). The digital computer as a musical instrument. Science, 142(3592), 553–557.

Nicolai, C. [Alva Noto]. (2000). Prototypes [album]. Mille Plateaux.

Nyman, M. (1999). Experimental music: Cage and beyond (2nd ed.). Cambridge University Press.

Pinch, T., & Trocco, F. (2002). Analog days: The invention and impact of the Moog synthesizer. Harvard University Press.

Roads, C. (2002). Microsound. MIT Press.

Russ, M. (2009). Sound synthesis and sampling (3rd ed.). Focal Press.

Sanfilippo, D., & Valle, A. (2013). Feedback systems: An analytical framework. Computer Music Journal, 37(2), 12–27.

Smalley, D. (1997). Spectromorphology: Explaining sound-shapes. Organised Sound, 2(2), 107–126.

SOMA Laboratory. (2016–2024). Instruments documentation. https://somasynths.com/

Strogatz, S. H. (2015). Nonlinear dynamics and chaos (2nd ed.). Westview Press.

Synthux Academy. (2024). About — Synthux instruments. https://www.synthux.academy/about

Truax, B. (2001). Acoustic communication (2nd ed.). Ablex Publishing.

Anexo 1 - Código Completo do Firmware

O código-fonte integral do firmware do VÆTTIR Inclui a totalidade das implementações referenciadas ao longo da Secção 6, nomeadamente:

Declaração e inicialização de todos os periféricos (ADC, I2C, MIDI USB, GPIO)
Classe DroneOscillator completa com os três osciladores por voz e o sistema de detune
Função UpdateChaos() - integração das equações de Lorenz
Função audio_callback() - geração e processamento de sinal
Função HandleMidiUsb() - eventos MIDI IN
Função SendMidiOutState() - MIDI OUT
Função ArpAdvance() - lógica do arpeggiator
Loop principal main() - leitura de sensores, atualização de parâmetros e coordenação geral
Rotinas de comunicação I2C com BH1750 e MPR121
Configuração de todos os mapeamentos de pinos e constantes de sistema

O código está escrito em C++, utilizando as bibliotecas libDaisy e DaisySP da Electro-Smith.

```
#include <array>

#include <math.h>   // powf, sinf, roundf, fabsf, fmodf, expf


#include "daisy_seed.h"

#include "daisysp.h"


#ifndef PI_F

#define PI_F 3.14159265358979f

#endif


using namespace daisy;

using namespace daisysp;

using namespace daisy::seed;


DaisySeed hw;


// ---------------------- BH1750 / I2C ----------------------
```

```
I2CHandle i2c1;

// endereços I2C
constexpr uint8_t BH1750_ADDR  = 0x23; // ADDR em GND
constexpr uint8_t MPR121_ADDR  = 0x5A; // MPR121 com ADDR padrão (GND)

float g_lux              = 0.0f;
bool  g_bh1750_ok        = false;
bool  g_mpr121_ok        = false;
float g_light_detune_st  = 0.0f; // semitons suavizados (usados no pitch)
float g_light_detune_target = 0.0f; // semitons “crus” vindos da luz

// ±1 tom = ±2 semitons (mas estamos a usar LIGHT_DETUNE_DEPTH_ST = 1.0f)
constexpr float LIGHT_DETUNE_DEPTH_ST = 1.0f;
// fator de suavização (0..1) – menor = mais suave/lento
constexpr float LIGHT_SMOOTH_ALPHA   = 0.05f;

// envia 1 comando simples (1 byte) para o BH1750
bool BH1750_SendCmd(uint8_t cmd)
{
  auto res = i2c1.TransmitBlocking(BH1750_ADDR, &cmd, 1, 10);
  return res == I2CHandle::Result::OK;
}

// inicializa BH1750 em modo contínuo high-res (0x10)
bool BH1750_Init()
{
  // Power ON (0x01)
```

```
    if(!BH1750_SendCmd(0x01))
        return false;

    System::Delay(10);

    // Reset (0x07) – opcional
    BH1750_SendCmd(0x07);
    System::Delay(10);

    // Modo contínuo high-res (0x10)
    if(!BH1750_SendCmd(0x10))
        return false;

    return true;
}

// lê lux (2 bytes) do BH1750
bool BH1750_ReadLux(float &out_lux)
{
    uint8_t data[2] = {0, 0};
    auto    res     = i2c1.ReceiveBlocking(BH1750_ADDR, data, 2, 50);
    if(res != I2CHandle::Result::OK)
        return false;

    uint16_t raw = (uint16_t(data[0]) << 8) | uint16_t(data[1]);
    out_lux      = float(raw) / 1.2f; // fórmula típica do BH1750
    return true;
}
```

```
bool InitI2C1()
{
  I2CHandle::Config conf;
  conf.periph         = I2CHandle::Config::Peripheral::I2C_1;
  conf.mode           = I2CHandle::Config::Mode::I2C_MASTER;
  conf.speed          = I2CHandle::Config::Speed::I2C_100KHZ; // seguro
  conf.pin_config.scl = D11; // SCL
  conf.pin_config.sda = D12; // SDA;

  auto res = i2c1.Init(conf);
  return res == I2CHandle::Result::OK;
}

// ---------------------- MPR121 helpers ----------------------
bool MPR121_WriteRegister(uint8_t reg, uint8_t value)
{
  uint8_t buf[2] = {reg, value};
  auto    res    = i2c1.TransmitBlocking(MPR121_ADDR, buf, 2, 10);
  return res == I2CHandle::Result::OK;
}

bool MPR121_ReadTouchStatus(uint16_t &status)
{
  uint8_t reg = 0x00;
  auto    res = i2c1.TransmitBlocking(MPR121_ADDR, &reg, 1, 10);
  if(res != I2CHandle::Result::OK)
      return false;
```

```
    uint8_t buf[2] = {0, 0};
    res            = i2c1.ReceiveBlocking(MPR121_ADDR, buf, 2, 10);
    if(res != I2CHandle::Result::OK)
        return false;

    status = uint16_t(buf[0]) | (uint16_t(buf[1]) << 8);
    return true;
}

// Init básico segundo AN3944 (12 electrodes activos)
bool MPR121_Init()
{
    bool ok = true;

    // Soft reset
    ok &= MPR121_WriteRegister(0x80, 0x63);
    System::Delay(1);

    // Stop mode (ECR = 0)
    ok &= MPR121_WriteRegister(0x5E, 0x00);

    // Configuração de filtros (valores do quick-start)
    ok &= MPR121_WriteRegister(0x2B, 0x01); // MHD_R
    ok &= MPR121_WriteRegister(0x2C, 0x01); // NHD_R
    ok &= MPR121_WriteRegister(0x2D, 0x00); // NCL_R
    ok &= MPR121_WriteRegister(0x2E, 0x00); // FDL_R
```

```
  ok &= MPR121_WriteRegister(0x2F, 0x01); // MHD_F
  ok &= MPR121_WriteRegister(0x30, 0x01); // NHD_F
  ok &= MPR121_WriteRegister(0x31, 0xFF); // NCL_F
  ok &= MPR121_WriteRegister(0x32, 0x02); // FDL_F

  // Thresholds para todas as 12 entradas
  for(uint8_t i = 0; i < 12; ++i)
  {
    uint8_t touch_reg = 0x41 + i * 2;
    uint8_t rel_reg   = touch_reg + 1;
    ok &= MPR121_WriteRegister(touch_reg, 0x0F); // Touch threshold
    ok &= MPR121_WriteRegister(rel_reg, 0x0A);   // Release threshold
  }

  // Filter config
  ok &= MPR121_WriteRegister(0x5D, 0x04);

  // Auto-config (valores típicos)
  ok &= MPR121_WriteRegister(0x7B, 0x0B);
  ok &= MPR121_WriteRegister(0x7D, 0x9C);
  ok &= MPR121_WriteRegister(0x7E, 0x65);
  ok &= MPR121_WriteRegister(0x7F, 0x8C);

  // Run mode, 12 electrodes activos
  ok &= MPR121_WriteRegister(0x5E, 0x0C);

  return ok;
}
```

```
// ---------------------- Synth original ----------------------

// 4 vozes (osciladores digitais) -> DRONE / PARTES
constexpr int NUM_TONES = 4;
// polifonia por parte em modo MIDI
constexpr int MAX_POLY  = 4;

// número de vozes "pad" controladas pelo MPR121 (ELE0..ELE11)
constexpr int NUM_PIANO_VOICES = 12;

// estado de toque do MPR121 (ELE0..ELE11 -> vozes pad 0..11)
bool g_mpr_touched[NUM_PIANO_VOICES] = {false};

// 8 pots: 4 amplitudes (A0..A3) + 4 pitches (A4..A7)
// + 1 pot extra (A8)  -> waveshaper
// + 1 pot extra (A9)  -> cutoff do filtro
// + 1 pot extra (A10) -> caos
// + 1 pot extra (A11) -> FX (reverb+delay)
constexpr int NUM_POTS = NUM_TONES * 2 + 4;

// índices dos pots extra no array de ADC
constexpr int SHAPER_POT_INDEX = NUM_TONES * 2;       // A8
constexpr int FILTER_POT_INDEX = NUM_TONES * 2 + 1;   // A9
constexpr int CHAOS_POT_INDEX  = NUM_TONES * 2 + 2;   // A10
constexpr int FX_POT_INDEX     = NUM_TONES * 2 + 3;   // A11

// Mapeamento para os pinos analógicos (A0..A11)
```

```
constexpr int pot_pins[NUM_POTS] = {
  15, // A0  -> Amp 1
  16, // A1  -> Amp 2
  17, // A2  -> Amp 3
  18, // A3  -> Amp 4
  19, // A4  -> Pitch 1
  20, // A5  -> Pitch 2
  21, // A6  -> Pitch 3
  22, // A7  -> Pitch 4
  23, // A8  -> Waveshaper amount
  24, // A9  -> Low-pass cutoff
  25, // A10 -> Chaos amount
  28  // A11 -> FX (reverb + ping-pong delay)
};

struct ToneSet
{
  float m_base_frequency;
  char  m_note;
  bool  m_is_sharp;
};

constexpr int NUM_TONE_SETS = 12;

ToneSet tones_sets[NUM_TONE_SETS] = {
  {55.0f,   'A', false},
  {58.27f,  'A', true },
  {61.74f,  'B', false},
```

```
  {65.41f,  'C', false},
  {69.30f,  'C', true },
  {73.42f,  'D', false},
  {77.78f,  'D', true },
  {82.41f,  'E', false},
  {87.31f,  'F', false},
  {92.50f,  'F', true },
  {98.00f,  'G', false},
  {103.83f, 'G', true },
};

// ESCALA "ASIÁTICA" tipo Hirajoshi em A
constexpr int   NUM_ASIAN_STEPS = 12;
constexpr float asian_scale_semitones[NUM_ASIAN_STEPS] = {
  0.0f,  2.0f,  3.0f,  7.0f,  8.0f,
  12.0f, 14.0f, 15.0f, 19.0f, 20.0f,
  24.0f, 26.0f
};

enum class WAVE_SUM_TYPE
{
  AVERAGE,
  SINE_WAVE_FOLD,
  TRIANGLE_WAVE_FOLD,
};

WAVE_SUM_TYPE sum_type = WAVE_SUM_TYPE::AVERAGE;
```

```
// limites absolutos de frequência
constexpr float MIN_FREQ = 40.0f;
constexpr float MAX_FREQ = 4000.0f;

// faixas por voz (em Hz) -> DRONE
constexpr float voice_min_freq[NUM_TONES] = {  55.0f, 100.0f, 250.0f, 400.0f };
constexpr float voice_max_freq[NUM_TONES] = { 200.0f, 300.0f, 450.0f, 800.0f };

// centro (frequência base) de cada voz
float voice_base_freq[NUM_TONES] = { 100.0f, 200.0f, 300.0f, 500.0f };

// range de pitch por pot: ±9 semitons
constexpr float PITCH_RANGE_SEMITONES = 9.0f;

// ---------------------- Utils (AGORA AQUI EM CIMA) ----------------------
inline float clampf(float v, float lo, float hi)
{
  if(v < lo)
     return lo;
  if(v > hi)
     return hi;
  return v;
}

// ---------------------- Patchbay (4 entradas digitais) ----------------------
GPIO g_patch_in[4];
bool  g_patch_active[4] = {false, false, false, false};
```

```
// ---------------------- Display 7 segmentos (4 LEDs) ----------------------
class SevenSegmentDisplay
{
   static constexpr int       NUM_LEDS = 4;
   std::array<GPIO, NUM_LEDS> m_gpio_pins;

 public:
   SevenSegmentDisplay(const std::array<Pin, NUM_LEDS> &pins)
   {
      for(int p = 0; p < NUM_LEDS; ++p)
      {
         m_gpio_pins[p].Init(pins[p], GPIO::Mode::OUTPUT, GPIO::Pull::NOPULL);
      }

      for(GPIO &gpio : m_gpio_pins)
      {
         gpio.Write(false);
      }
   }

   void set_byte_mask(char mask)
   {
      int bit = 1;
      for(int p = 0; p < (NUM_LEDS - 1); ++p)
      {
         GPIO &gpio = m_gpio_pins[p];
         gpio.Write((mask & bit) != 0);
         bit <<= 1;
```

```
    }
  }

  void set_character(char character)
  {
    switch(character)
    {
      case 'B': set_byte_mask(0b01111111); break;
      case 'C': set_byte_mask(0b00111001); break;
      case 'D': set_byte_mask(0b00111111); break;
      case 'E': set_byte_mask(0b01111001); break;
      case 'F': set_byte_mask(0b01110001); break;
      case 'G': set_byte_mask(0b01111101); break;
      default:  set_byte_mask(0);          break;
    }
  }

  void set_dot(bool set) { m_gpio_pins.back().Write(set); }
};

// ---------------------- Gerador caótico Lorenz (global) ----------------------
float g_chaos_x      = 0.1f;
float g_chaos_y      = 0.0f;
float g_chaos_z      = 0.0f;
float g_chaos_value  = 0.0f;  // -1..1
float g_chaos_amount = 0.0f;  // 0..1 (knob caos)

// ---------------------- Spread de detune caótico (P3) ----------------------
```

```
constexpr float DETUNE_BASE_SPREAD  = 0.02f;  // 2 cents (±0.01)
constexpr float DETUNE_EXTRA_SPREAD = 36.0f;  // extra com caos (±18 semitons)
float           g_detune_spread     = DETUNE_BASE_SPREAD;

// forward declarations de wavefold
float triangular_wave_fold(float in);
float sin_wave_fold(float in);

// ---------------------- DroneOscillator ----------------------
class DroneOscillator
{
  Oscillator m_low_osc;
  Oscillator m_base_osc;
  Oscillator m_high_osc;
  float      m_amplitude = 0.0f;

  float      m_min_freq  = MIN_FREQ;
  float      m_max_freq  = MAX_FREQ;

  float clamp_freq(float f) const
  {
     if(f < m_min_freq)
        return m_min_freq;
     if(f > m_max_freq)
        return m_max_freq;
     return f;
  }
```

```
public:
  void initialise(float sample_rate)
  {
      auto init_osc = [sample_rate](Oscillator &osc) {
          osc.Init(sample_rate);
          osc.SetWaveform(osc.WAVE_SIN);
          osc.SetAmp(1.0f);
      };

      init_osc(m_low_osc);
      init_osc(m_base_osc);
      init_osc(m_high_osc);

      m_amplitude = 0.0f;
  }

  void set_range(float minf, float maxf)
  {
      m_min_freq = minf;
      m_max_freq = maxf;
  }

  void set_amplitude(float a) { m_amplitude = a; }

  void set_semitone(float base_frequency, float semitone)
  {
      auto semitone_to_frequency = [base_frequency](float s) -> float {
          const float freq_mult = powf(2.0f, s / 12.0f);
```

```
        return base_frequency * freq_mult;
    };

    float spread = g_detune_spread;
    float half   = spread * 0.5f;

    float base_semitone = semitone;
    float low_semitone  = base_semitone - half;
    float high_semitone = base_semitone + half;

    float f_low  = clampf(semitone_to_frequency(low_semitone), MIN_FREQ, MAX_FREQ);
    float f_mid  = clampf(semitone_to_frequency(base_semitone), MIN_FREQ,
MAX_FREQ);
    float f_high = clampf(semitone_to_frequency(high_semitone), MIN_FREQ,
MAX_FREQ);

    m_low_osc.SetFreq(f_low);
    m_base_osc.SetFreq(f_mid);
    m_high_osc.SetFreq(f_high);
  }

  float process()
  {
    const float avg_sin
      = (m_low_osc.Process() + m_base_osc.Process() + m_high_osc.Process()) / 3.0f;
    return avg_sin * m_amplitude;
  }
};
```

```
// banco de osciladores: [parte][slot polifónico]
DroneOscillator g_osc[NUM_TONES][MAX_POLY];

// ---------------------- Estado MIDI polifónico ----------------------
struct MidiNoteSlot
{
  uint8_t note    = 0;
  bool    active  = false;  // slot em uso (mesmo em release)
  bool    gate    = false;  // tecla actualmente em baixo?
  float   velocity = 0.0f;  // 0..1 (alvo quando gate ON)
  float   env      = 0.0f;  // 0..1 (envelope suavizado)
};

MidiNoteSlot g_midi_slots[NUM_TONES][MAX_POLY];

float g_midi_attack_coeff  = 0.0f;
float g_midi_release_coeff = 0.0f;

// ---------------------- *** MIDI USB IN / OUT *** ----------------------
MidiUsbHandler midi_usb;

// D7 = botão "MIDI ON", D8 = botão "MIDI OFF"
Switch g_midi_on_switch;
Switch g_midi_off_switch;

// estado global do modo MIDI
bool g_midi_mode = false;
```

```
// ---------------------- "Pad" / piano voices (MPR121) ----------------------
struct PianoVoice
{
  Oscillator osc;
  float      env; // 0..1
};

PianoVoice g_piano[NUM_PIANO_VOICES];

// envelope dos pads
float g_env_attack_step  = 0.0f;
float g_env_release_mult = 1.0f;

// ---------------------- Waveshaping ----------------------
float triangular_wave_fold(float in)
{
  const float q_in = in * 0.25f;
  return 4 * (fabsf(q_in + 0.25f - roundf(q_in + 0.25f)) - 0.25f);
}

float sin_wave_fold(float in)
{
  return sinf(in);
}

float g_waveshaper_amount = 0.0f; // 0 = off, 1 = max

float brutal_waveshaper(float in, float amount)
```

```
{
  float drive = 1.0f + amount * 40.0f;

  float x = in * drive;

  if(x > 1.0f)
     x = 1.0f;
  else if(x < -1.0f)
     x = -1.0f;

  return x;
}

// ---------- filtro low-pass SVF global ----------
Svf   svf_filter;
float g_filter_cutoff            = 9000.0f;       // Hz
constexpr float FILTER_RESONANCE  = 0.3f;          // 0..1
constexpr float FILTER_MIN_CUTOFF = 100.0f;        // Hz
constexpr float FILTER_MAX_CUTOFF = 9000.0f;       // Hz

// ---------------------- FX global (Reverb + Ping-Pong Delay) ----------------------
#define MAX_DELAY static_cast<size_t>(48000 * 0.75f)

DSY_SDRAM_BSS DelayLine<float, MAX_DELAY> g_delayL;
DSY_SDRAM_BSS DelayLine<float, MAX_DELAY> g_delayR;

ReverbSc g_reverb;
float    g_fx_amount      = 0.0f;     // 0..1
```

```
constexpr float FX_MAX_MIX     = 0.85f;
constexpr float DELAY_FEEDBACK = 0.45f;

// ---------------------- Arpegiador global (P4) ----------------------
float g_sample_rate  = 48000.0f;
float g_arp_phase    = 0.0f;
float g_arp_rate_hz  = 0.0f;
int   g_arp_index    = 0;
float g_arp_gain[NUM_TONES] = {1.0f, 1.0f, 1.0f, 1.0f};

constexpr float ARP_BG_GAIN   = 0.15f;
constexpr float ARP_MIN_RATE  = 0.5f;
constexpr float ARP_MAX_RATE  = 12.0f;

// ---------------------- Tremolo / LFO caótico global (P1) ----------------------
float g_stutter_phase   = 0.0f;
float g_stutter_rate_hz = 0.0f;
float g_stutter_depth   = 0.0f;
float g_stutter_duty    = 0.5f;

constexpr float STUTTER_MIN_RATE   = 0.2f;
constexpr float STUTTER_MAX_RATE   = 15.0f;
constexpr float STUTTER_MIN_DEPTH  = 0.3f;
constexpr float STUTTER_MAX_DEPTH  = 1.0f;

// ---------------------- MIDI OUT mapeamento e estado ----------------------
constexpr uint8_t MIDI_OUT_CHANNEL = 0; // canal 1
```

```
// CC por pot (A0..A11)
constexpr uint8_t pot_cc_numbers[NUM_POTS] = {
  20, 21, 22, 23, // A0..A3 amps
  24, 25, 26, 27, // A4..A7 pitch
  28,             // A8 waveshaper
  29,             // A9 cutoff
  30,             // A10 chaos
  31              // A11 FX
};

float g_prev_pot_cc[NUM_POTS];                // último valor MIDI (0..127) em float
bool  g_prev_patch_state[4]         = {false, false, false, false};
bool  g_prev_pad_state[NUM_PIANO_VOICES] = {false};

// ---------------------- UpdateChaos ----------------------
void UpdateChaos(float dt)
{
  float k_raw = g_chaos_amount;
  float k     = clampf(k_raw * 1.2f, 0.0f, 1.0f);

  const float sigma = 10.0f + 80.0f * k;
  const float rho   = 28.0f + 140.0f * k;
  const float beta  = 8.0f / 3.0f + 3.0f * k;

  float dx = sigma * (g_chaos_y - g_chaos_x);
  float dy = g_chaos_x * (rho - g_chaos_z) - g_chaos_y;
  float dz = g_chaos_x * g_chaos_y - beta * g_chaos_z;
```

```
  g_chaos_x += dx * dt;
  g_chaos_y += dy * dt;
  g_chaos_z += dz * dt;

  g_chaos_value = clampf(g_chaos_x * 0.06f, -1.0f, 1.0f);
}

// ---------- passo do "arpegiador" (P4) ----------
void ArpAdvance(float chaos_val)
{
  float a = fabsf(chaos_val);

  if(a < 0.05f)
  {
    for(int v = 0; v < NUM_TONES; ++v)
      g_arp_gain[v] = 1.0f;
    return;
  }

  static int dir = 1;
  if(chaos_val > 0.4f)
    dir = 1;
  else if(chaos_val < -0.4f)
    dir = -1;

  int step_size = 1;
  if(a > 0.7f)
    step_size = 2;
```

```
  g_arp_index = (g_arp_index + dir * step_size + NUM_TONES) % NUM_TONES;
  int accent  = g_arp_index;

  for(int v = 0; v < NUM_TONES; ++v)
    g_arp_gain[v] = (v == accent) ? 1.0f : ARP_BG_GAIN;

  if(a > 0.85f)
  {
    int accent2       = (accent + 2) % NUM_TONES;
    g_arp_gain[accent2] = 0.9f;
  }
}

// ---------------------- Helpers MIDI OUT (raw bytes) ----------------------
void SendCC(uint8_t channel, uint8_t cc, uint8_t value)
{
  uint8_t bytes[3];
  bytes[0] = 0xB0 | (channel & 0x0F); // Control Change
  bytes[1] = cc & 0x7F;
  bytes[2] = value & 0x7F;
  midi_usb.SendMessage(bytes, 3);
}

// envia CCs dos pots, CCs dos patchpoints, e NoteOn/Off dos pads
void SendMidiOutState()
{
  // 1) Pots -> CC 20..31
```

```
for(int i = 0; i < NUM_POTS; ++i)
{
    float v = hw.adc.GetFloat(i);
    v       = clampf(v, 0.0f, 1.0f);
    uint8_t midi_val = static_cast<uint8_t>(v * 127.0f + 0.5f);

    if(g_prev_pot_cc[i] < 0.0f
       || midi_val != static_cast<uint8_t>(g_prev_pot_cc[i]))
    {
        SendCC(MIDI_OUT_CHANNEL, pot_cc_numbers[i], midi_val);
        g_prev_pot_cc[i] = static_cast<float>(midi_val);
    }
}

// 2) Patchbay P1..P4 -> CC 40..43 (0 ou 127)
for(int i = 0; i < 4; ++i)
{
    bool s = g_patch_active[i];
    if(s != g_prev_patch_state[i])
    {
        uint8_t val = s ? 127 : 0;
        SendCC(MIDI_OUT_CHANNEL, 40 + i, val);
        g_prev_patch_state[i] = s;
    }
}

// 3) Pads MPR121 -> notas 60..71 (C4..B4) em channel 1
for(int i = 0; i < NUM_PIANO_VOICES; ++i)
```

```
  {
    bool s = g_mpr_touched[i];
    if(s != g_prev_pad_state[i])
    {
      uint8_t note = 60 + i;
      uint8_t bytes[3];

      if(s)
      {
        // NoteOn
        bytes[0] = 0x90 | (MIDI_OUT_CHANNEL & 0x0F);
        bytes[1] = note & 0x7F;
        bytes[2] = 100; // velocity
        midi_usb.SendMessage(bytes, 3);
      }
      else
      {
        // NoteOff
        bytes[0] = 0x80 | (MIDI_OUT_CHANNEL & 0x0F);
        bytes[1] = note & 0x7F;
        bytes[2] = 0; // velocity
        midi_usb.SendMessage(bytes, 3);
      }

      g_prev_pad_state[i] = s;
    }
  }
}
```

```
// ---------------------- Handler MIDI USB (IN) ----------------------
void HandleMidiUsb()
{
  midi_usb.Listen();

  while(midi_usb.HasEvents())
  {
    MidiEvent msg = midi_usb.PopEvent();

    switch(msg.type)
    {
      case NoteOn:
      {
        auto on = msg.AsNoteOn();
        int  ch = on.channel;

        if(ch >= 0 && ch < NUM_TONES)
        {
          int part = ch;

          if(on.velocity > 0)
          {
            float vel = on.velocity / 127.0f;

            int slot = -1;
            for(int i = 0; i < MAX_POLY; ++i)
            {
```

```
            if(!g_midi_slots[part][i].active)
            {
                slot = i;
                break;
            }
        }
        if(slot < 0)
            slot = 0;

        g_midi_slots[part][slot].note     = on.note;
        g_midi_slots[part][slot].active   = true;
        g_midi_slots[part][slot].gate     = true;
        g_midi_slots[part][slot].velocity = vel;
    }
    else
    {
        for(int i = 0; i < MAX_POLY; ++i)
        {
            if(g_midi_slots[part][i].active
               && g_midi_slots[part][i].note == on.note)
            {
                g_midi_slots[part][i].gate     = false;
                g_midi_slots[part][i].velocity = 0.0f;
            }
        }
    }
}
```

```
break;

case NoteOff:
{
    auto off = msg.AsNoteOff();
    int  ch  = off.channel;

    if(ch >= 0 && ch < NUM_TONES)
    {
        int part = ch;
        for(int i = 0; i < MAX_POLY; ++i)
        {
            if(g_midi_slots[part][i].active
               && g_midi_slots[part][i].note == off.note)
            {
                g_midi_slots[part][i].gate     = false;
                g_midi_slots[part][i].velocity = 0.0f;
            }
        }
    }
}
break;

case ChannelMode:
{
    auto ev = msg.AsChannelMode();
    if(ev.event_type == ChannelModeType::AllNotesOff)
    {
```

```
                int ch = ev.channel;
                if(ch >= 0 && ch < NUM_TONES)
                {
                    int part = ch;
                    for(int i = 0; i < MAX_POLY; ++i)
                    {
                        g_midi_slots[part][i].gate     = false;
                        g_midi_slots[part][i].velocity = 0.0f;
                    }
                }
              }
            }
            break;

            default: break;
        }
    }
}

// ---------------------- Audio Callback (stereo + FX) ----------------------
void audio_callback(AudioHandle::InputBuffer  in,
                AudioHandle::OutputBuffer out,
                size_t                    size)
{
    (void)in;

    const float chaos_amount = g_chaos_amount;
    const float chaos_value  = g_chaos_value;
```

```
const bool  p1_active    = g_patch_active[0];
const bool  p2_active    = g_patch_active[1];
const bool  p4_active    = g_patch_active[3];
const bool  midi_mode    = g_midi_mode;

const float base_ws_amount = g_waveshaper_amount;
float       cutoff         = g_filter_cutoff;
float       sr             = g_sample_rate;

if(p2_active && chaos_amount > 0.001f)
{
   const float chaos_cutoff_depth = 9000.0f;
   float       chaos_mod          = chaos_value * chaos_amount;
   cutoff += chaos_mod * chaos_cutoff_depth;
}

cutoff = clampf(cutoff, FILTER_MIN_CUTOFF, FILTER_MAX_CUTOFF);

svf_filter.SetFreq(cutoff);
svf_filter.SetRes(FILTER_RESONANCE);
svf_filter.SetDrive(1.0f);

float arp_phase   = g_arp_phase;
float arp_rate_hz = g_arp_rate_hz;

float st_phase = g_stutter_phase;
float st_rate  = g_stutter_rate_hz;
float st_depth = g_stutter_depth;
```

```
float st_duty  = g_stutter_duty;


for(size_t i = 0; i < size; i++)
{
    float osc_out = 0.0f;


    if(p4_active && chaos_amount > 0.01f && arp_rate_hz > 0.0f)
    {
        arp_phase += arp_rate_hz / sr;
        if(arp_phase >= 1.0f)
        {
            arp_phase -= 1.0f;
            ArpAdvance(chaos_value);
        }
    }


    // ---------- DRONE / MIDI VOICES ----------
    if(midi_mode)
    {
        for(int t = 0; t < NUM_TONES; ++t)
        {
            for(int v = 0; v < MAX_POLY; ++v)
            {
                MidiNoteSlot &slot = g_midi_slots[t][v];


                if(!slot.active && !slot.gate && slot.env <= 0.0001f)
                    continue;
```

```
    float target = slot.gate ? slot.velocity : 0.0f;

    float env = slot.env;
    if(target > env)
    {
        env += (target - env) * g_midi_attack_coeff;
    }
    else
    {
        env += (target - env) * g_midi_release_coeff;
    }

    if(!slot.gate && env < 0.0001f)
    {
        env         = 0.0f;
        slot.active = false;
    }

    slot.env = env;

    float gate = env;

    if(p4_active && chaos_amount > 0.01f)
        gate *= g_arp_gain[t];

    float voice = g_osc[t][v].process() * gate;
    osc_out += voice;
}
```

```
    }
}
else
{
    for(int t = 0; t < NUM_TONES; ++t)
    {
        float gate = 1.0f;
        if(p4_active && chaos_amount > 0.01f)
            gate *= g_arp_gain[t];

        float voice = g_osc[t][0].process() * gate;
        osc_out += voice;
    }
}

// ---------- PADS (MPR121) ----------
float piano_out = 0.0f;
for(int v = 0; v < NUM_PIANO_VOICES; ++v)
{
    float env = g_piano[v].env;
    bool  on  = g_mpr121_ok && g_mpr_touched[v];

    if(on)
    {
        env += g_env_attack_step;
        if(env > 1.0f)
            env = 1.0f;
    }
```

```
    else
    {
        env *= g_env_release_mult;
        if(env < 0.0001f)
            env = 0.0f;
    }

    g_piano[v].env = env;

    if(env > 0.0f)
    {
        float s = g_piano[v].osc.Process();
        piano_out += s * env;
    }
    else
    {
        g_piano[v].osc.Process();
    }
}

osc_out += piano_out;

// ---------- wavesum / wavefolder ----------
switch(sum_type)
{
    case WAVE_SUM_TYPE::AVERAGE:
    {
        osc_out /= NUM_TONES;
```

```
        break;
    }
    case WAVE_SUM_TYPE::SINE_WAVE_FOLD:
    {
        osc_out = sin_wave_fold(osc_out);
        break;
    }
    case WAVE_SUM_TYPE::TRIANGLE_WAVE_FOLD:
    {
        osc_out = triangular_wave_fold(osc_out);
        break;
    }
}

// ---------- waveshaper ----------
float ws_amount = base_ws_amount;
if(ws_amount > 0.001f)
{
    float shaped = brutal_waveshaper(osc_out, ws_amount);
    osc_out      = (1.0f - ws_amount) * osc_out
                 + ws_amount * shaped;
}

// ---------- filtro ----------
svf_filter.Process(osc_out);
float filtered = svf_filter.Low();

// ---------- tremolo caótico no master ----------
```

```
float master_amp = 1.0f;
if(p1_active && chaos_amount > 0.01f && st_rate > 0.0f)
{
    st_phase += st_rate / sr;
    if(st_phase >= 1.0f)
        st_phase -= 1.0f;

    float lfo     = 0.5f * (1.0f + sinf(2.0f * PI_F * st_phase));
    float min_amp = 1.0f - st_depth;
    master_amp    = min_amp + st_depth * lfo;
}

float dry = filtered * master_amp;

// ---------- FX MONO -> STEREO ----------
float fx_mix = clampf(g_fx_amount, 0.0f, 1.0f);
fx_mix       = fx_mix * fx_mix * FX_MAX_MIX;

float fx_in = dry * 0.6f;

float dl_out = g_delayL.Read();
float dr_out = g_delayR.Read();

float fbL = fx_in + dr_out * DELAY_FEEDBACK;
float fbR = fx_in + dl_out * DELAY_FEEDBACK;

g_delayL.Write(fbL);
g_delayR.Write(fbR);
```

```
      float rv_inL = fx_in + dl_out * 0.5f;
      float rv_inR = fx_in + dr_out * 0.5f;
      float rvL, rvR;
      g_reverb.Process(rv_inL, rv_inR, &rvL, &rvR);

      rvL *= 0.7f;
      rvR *= 0.7f;

      float wetL = rvL + dl_out * 0.5f;
      float wetR = rvR + dr_out * 0.5f;

      float outL = dry * (1.0f - fx_mix) + wetL * fx_mix;
      float outR = dry * (1.0f - fx_mix) + wetR * fx_mix;

      out[0][i] = outL;
      out[1][i] = outR;
    }

    g_arp_phase     = arp_phase;
    g_stutter_phase = st_phase;
    g_stutter_duty  = st_duty;
}

// ---------------------- ADC init ----------------------
void init_adc()
{
    AdcChannelConfig adc_config[NUM_POTS];
```

```
    for(int t = 0; t < NUM_POTS; ++t)
        adc_config[t].InitSingle(hw.GetPin(pot_pins[t]));

    hw.adc.Init(adc_config, NUM_POTS);
    hw.adc.Start();
}

// ---------------------- calcular voicing menor por voz ----------------------
void update_voice_base_freqs(float root_freq)
{
    const float m3    = root_freq * powf(2.0f, 3.0f / 12.0f);
    const float fifth = root_freq * powf(2.0f, 7.0f / 12.0f);
    const float oct   = root_freq * 2.0f;

    float degrees[NUM_TONES] = {
        root_freq,
        m3,
        fifth,
        oct
    };

    for(int v = 0; v < NUM_TONES; ++v)
    {
        float f = degrees[v];

        while(f < voice_min_freq[v])
            f *= 2.0f;
```

```
        while(f > voice_max_freq[v])
            f /= 2.0f;

        if(f < MIN_FREQ)
            f = MIN_FREQ;
        if(f > MAX_FREQ)
            f = MAX_FREQ;

        voice_base_freq[v] = f;
    }
}

// ---------------------- main ----------------------
int main(void)
{
    hw.Configure();
    hw.Init();
    init_adc();

    const float sample_rate = hw.AudioSampleRate();
    g_sample_rate           = sample_rate;

    // Envelope do gate MIDI (para evitar cliques)
    {
        const float midi_attack_time  = 0.002f; // ~2 ms
        const float midi_release_time = 0.03f;  // ~30 ms
```

```
    g_midi_attack_coeff
        = 1.0f - expf(-1.0f / (midi_attack_time * sample_rate));
    g_midi_release_coeff
        = 1.0f - expf(-1.0f / (midi_release_time * sample_rate));
}

// Inicializar estados MIDI OUT
for(int i = 0; i < NUM_POTS; ++i)
    g_prev_pot_cc[i] = -1.0f;

for(int i = 0; i < 4; ++i)
    g_prev_patch_state[i] = false;
for(int i = 0; i < NUM_PIANO_VOICES; ++i)
    g_prev_pad_state[i] = false;

// ---------- MIDI USB ----------
{
    MidiUsbHandler::Config midi_cfg;
    midi_cfg.transport_config.periph = MidiUsbTransport::Config::INTERNAL;
    midi_usb.Init(midi_cfg);
    midi_usb.StartReceive();
}

// NOTA BASE FIXA: A = 55 Hz
float base_frequency = 55.0f;
update_voice_base_freqs(base_frequency);

// init drone/MIDI voices
```

```
for(int i = 0; i < NUM_TONES; ++i)
{
    for(int v = 0; v < MAX_POLY; ++v)
    {
        g_osc[i][v].initialise(sample_rate);
        g_osc[i][v].set_range(voice_min_freq[i], voice_max_freq[i]);
    }
}

// init filtro
svf_filter.Init(sample_rate);
svf_filter.SetFreq(g_filter_cutoff);
svf_filter.SetRes(FILTER_RESONANCE);
svf_filter.SetDrive(1.0f);

// init FX
g_reverb.Init(sample_rate);
g_reverb.SetFeedback(0.7f);
g_reverb.SetLpFreq(10000.0f);

g_delayL.Init();
g_delayR.Init();
g_delayL.SetDelay(sample_rate * 0.45f);
g_delayR.SetDelay(sample_rate * 0.60f);

// init envelope dos pads
const float attack_time_s  = 0.3f;
const float release_time_s = 2.0f;
```

```
g_env_attack_step  = 1.0f / (attack_time_s * sample_rate);
g_env_release_mult = powf(0.001f, 1.0f / (release_time_s * sample_rate));

// init pad voices em escala "asiática"
const float pad_root = 55.0f;
for(int i = 0; i < NUM_PIANO_VOICES; ++i)
{
    g_piano[i].osc.Init(sample_rate);
    g_piano[i].osc.SetWaveform(Oscillator::WAVE_SIN);
    g_piano[i].osc.SetAmp(1.0f);

    float semi = asian_scale_semitones[i % NUM_ASIAN_STEPS];

    float note_freq = pad_root * powf(2.0f, (semi + 12.0f) / 12.0f);

    while(note_freq > MAX_FREQ)
        note_freq *= 0.5f;
    while(note_freq < MIN_FREQ)
        note_freq *= 2.0f;

    g_piano[i].osc.SetFreq(note_freq);
    g_piano[i].env = 0.0f;
}

// I2C + BH1750 + MPR121
if(InitI2C1())
{
    g_bh1750_ok = BH1750_Init();
```

```
    g_mpr121_ok = MPR121_Init();
}
else
{
    g_bh1750_ok = false;
    g_mpr121_ok = false;
}

// Switches dos tipos de soma
Switch sum_avg_switch;
Switch sum_sin_switch;
Switch sum_tri_switch;

sum_avg_switch.Init(D2);
sum_sin_switch.Init(D1);
sum_tri_switch.Init(D0);

// Botões MIDI ON/OFF (D7/D8)
g_midi_on_switch.Init(D7);
g_midi_off_switch.Init(D8);

// Patchbay inputs (D3..D6) com pull-ups
g_patch_in[0].Init(D3, GPIO::Mode::INPUT, GPIO::Pull::PULLUP);
g_patch_in[1].Init(D4, GPIO::Mode::INPUT, GPIO::Pull::PULLUP);
g_patch_in[2].Init(D5, GPIO::Mode::INPUT, GPIO::Pull::PULLUP);
g_patch_in[3].Init(D6, GPIO::Mode::INPUT, GPIO::Pull::PULLUP);

hw.StartAudio(audio_callback);
```

```
int bh_counter   = 0;
int midi_out_div = 0;

while(1)
{
    // --------- MIDI USB IN ----------
    HandleMidiUsb();

    // Botões MIDI ON / OFF
    g_midi_on_switch.Debounce();
    g_midi_off_switch.Debounce();
    if(g_midi_on_switch.Pressed())
    {
        g_midi_mode = true;
    }
    if(g_midi_off_switch.Pressed())
    {
        g_midi_mode = false;
        for(int t = 0; t < NUM_TONES; ++t)
        {
            for(int v = 0; v < MAX_POLY; ++v)
            {
                MidiNoteSlot &slot = g_midi_slots[t][v];
                slot.active   = false;
                slot.gate     = false;
                slot.velocity = 0.0f;
                slot.env      = 0.0f;
```

```
        }
    }
}

// 0) knob do caos (A10) + Lorenz
{
    const float chaos_pot = hw.adc.GetFloat(CHAOS_POT_INDEX);
    float       k         = chaos_pot * 1.7f;
    g_chaos_amount        = clampf(k, 0.0f, 1.0f);
    UpdateChaos(0.001f);
}

// 0b) BH1750 → detune de luz
if(g_bh1750_ok)
{
    bh_counter++;
    if(bh_counter >= 50)
    {
        bh_counter = 0;
        float lux;
        if(BH1750_ReadLux(lux))
        {
            g_lux = lux;

            float max_lux = 2000.0f;
            if(lux < 0.0f)
                lux = 0.0f;
            if(lux > max_lux)
```

```
            lux = max_lux;

        float norm = lux / max_lux;

        g_light_detune_target
            = (norm - 0.5f) * 2.0f * LIGHT_DETUNE_DEPTH_ST;
      }
   }

   g_light_detune_st += LIGHT_SMOOTH_ALPHA
                     * (g_light_detune_target - g_light_detune_st);
}
else
{
   g_light_detune_target = 0.0f;
   g_light_detune_st    += LIGHT_SMOOTH_ALPHA
                        * (0.0f - g_light_detune_st);
}

// 0c) MPR121 → pads
if(g_mpr121_ok)
{
   uint16_t touch_bits = 0;
   if(MPR121_ReadTouchStatus(touch_bits))
   {
      for(int v = 0; v < NUM_PIANO_VOICES; ++v)
      {
         g_mpr_touched[v] = (touch_bits & (1 << v)) != 0;
```

```
        }
    }
}
else
{
    for(int v = 0; v < NUM_PIANO_VOICES; ++v)
        g_mpr_touched[v] = false;
}

// 1) patchbay (D3..D6)
g_patch_active[0] = !g_patch_in[0].Read();
g_patch_active[1] = !g_patch_in[1].Read();
g_patch_active[2] = !g_patch_in[2].Read();
g_patch_active[3] = !g_patch_in[3].Read();

// 2) detune spread caótico interno (P3)
if(g_patch_active[2] && g_chaos_amount > 0.01f)
{
    float a          = fabsf(g_chaos_value);
    float chaos_ener = a * g_chaos_amount;
    g_detune_spread  = DETUNE_BASE_SPREAD
                     + DETUNE_EXTRA_SPREAD * chaos_ener;
}
else
{
    g_detune_spread = DETUNE_BASE_SPREAD;
}
```

```
// 3) tremolo caótico (P1)
if(g_patch_active[0] && g_chaos_amount > 0.01f)
{
    float a         = fabsf(g_chaos_value);
    float chaos_ener = a * g_chaos_amount;

    g_stutter_rate_hz
       = STUTTER_MIN_RATE
         + (STUTTER_MAX_RATE - STUTTER_MIN_RATE) * chaos_ener;

    g_stutter_depth
       = STUTTER_MIN_DEPTH
         + (STUTTER_MAX_DEPTH - STUTTER_MIN_DEPTH) * chaos_ener;
}
else
{
    g_stutter_rate_hz = 0.0f;
    g_stutter_depth   = 0.0f;
    g_stutter_phase   = 0.0f;
}

// 4) arpegiador (P4)
if(g_patch_active[3] && g_chaos_amount > 0.01f)
{
    float a         = fabsf(g_chaos_value);
    float chaos_ener = g_chaos_amount * a;

    g_arp_rate_hz = ARP_MIN_RATE
```

```
                + (ARP_MAX_RATE - ARP_MIN_RATE) * chaos_ener;

    if(g_arp_rate_hz < ARP_MIN_RATE)
        g_arp_rate_hz = ARP_MIN_RATE;
}
else
{
    g_arp_rate_hz = 0.0f;
    for(int v = 0; v < NUM_TONES; ++v)
        g_arp_gain[v] = 1.0f;
}

// 5) pots de volume e pitch (+ detune da luz)
for(int t = 0; t < NUM_TONES; ++t)
{
    const float amp = hw.adc.GetFloat(t);

    const float pitch_pot   = hw.adc.GetFloat(NUM_TONES + t);
    const float base_detune =
        (pitch_pot - 0.5f) * 2.0f * PITCH_RANGE_SEMITONES;

    const float detune = base_detune + g_light_detune_st;

    if(g_midi_mode)
    {
        for(int v = 0; v < MAX_POLY; ++v)
        {
            g_osc[t][v].set_range(MIN_FREQ, MAX_FREQ);
```

```
        g_osc[t][v].set_amplitude(amp);

        float base_freq = voice_base_freq[t];
        if(g_midi_slots[t][v].active)
        {
            base_freq
                = mtof(static_cast<float>(g_midi_slots[t][v].note));
        }

        g_osc[t][v].set_semitone(base_freq, detune);
    }
}
else
{
    g_osc[t][0].set_range(voice_min_freq[t], voice_max_freq[t]);
    g_osc[t][0].set_amplitude(amp);

    float base_freq = voice_base_freq[t];
    g_osc[t][0].set_semitone(base_freq, detune);

    for(int v = 1; v < MAX_POLY; ++v)
    {
        MidiNoteSlot &slot = g_midi_slots[t][v];
        slot.active   = false;
        slot.gate     = false;
        slot.velocity = 0.0f;
        slot.env      = 0.0f;
    }
```

```
    }
}

// 6) waveshaper (A8)
{
    const float shaper_pot = hw.adc.GetFloat(SHAPER_POT_INDEX);
    g_waveshaper_amount    = shaper_pot;
}

// 7) cutoff (A9)
{
    const float filter_pot = hw.adc.GetFloat(FILTER_POT_INDEX);
    g_filter_cutoff        = FILTER_MIN_CUTOFF
                           + filter_pot
                               * (FILTER_MAX_CUTOFF
                                  - FILTER_MIN_CUTOFF);
}

// 8) FX mix (A11)
{
    const float fx_pot = hw.adc.GetFloat(FX_POT_INDEX);
    g_fx_amount        = fx_pot;
}

// 9) wavefolder switches
sum_avg_switch.Debounce();
sum_sin_switch.Debounce();
sum_tri_switch.Debounce();
```

```
        if(sum_avg_switch.Pressed())
            sum_type = WAVE_SUM_TYPE::AVERAGE;
        else if(sum_sin_switch.Pressed())
            sum_type = WAVE_SUM_TYPE::SINE_WAVE_FOLD;
        else if(sum_tri_switch.Pressed())
            sum_type = WAVE_SUM_TYPE::TRIANGLE_WAVE_FOLD;

        // 10) MIDI OUT: envia estado a cada ~10ms
        midi_out_div++;
        if(midi_out_div >= 10)
        {
            midi_out_div = 0;
            SendMidiOutState();
        }

        System::Delay(1);
    }
}
```

ESCOLA
SUPERIOR
DE MÚSICA
E ARTES
DO ESPETÁCULO
POLITÉCNICO
DO PORTO

P.PORTO

M

MESTRADO
ARTES E TECNOLOGIAS DO SOM

VÆTTIR

Diogo José Vieira Nóbrega