



Avenue du Ciseau 15
1348 Louvain-la-Neuve

0xEntropy

Conception et réalisation d'un générateur matériel de nombres aléatoires
portatif basé sur un phénomène physique non-prédictible

Travail de fin d'études présenté en vue de l'obtention du diplôme de bachelier en
Technologies de l'informatique

DE HENAU Dudley

ANNEXES

Année académique 2025 - 2026

Rapporteur :
Yves DELVIGNE

Table des matières

Annexe A : Méthodologie et grille d'évaluation des sources d'entropie	2
Annexe B : Résultats de l'évaluation statistique par la suite Dieharder	4
Annexe C : Code source du projet	6
1. Racine du projet	6
A. Fichier <code>0xEntropy.c</code> (Main)	6
2. Dossier Entropy	8
A. Fichier <code>entropy.c</code>	8
B. Fichier <code>entropy.h</code>	10
C. Fichier <code>noise.pio</code>	10
D. Fichier <code>sha256.c</code>	10
E. Fichier <code>sha256.h</code>	13
3. Dossier Health	13
A. Fichier <code>health_test.c</code>	13
B. Fichier <code>health_test.h</code>	14
C. Fichier <code>health_warn.c</code>	14
D. Fichier <code>health_warn.h</code>	16
4. Dossier Stats	16
A. Fichier <code>stats.c</code>	16
B. Fichier <code>stats.h</code>	18
5. Dossier UI	19
A. Fichier <code>Screen_app.c</code>	19
B. Fichier <code>Screen_app.h</code>	27
D. Fichier <code>Screen_tactile.h</code>	29

*

Annexe A : Méthodologie et grille d'évaluation des sources d'entropie matérielle

Afin de sélectionner la source d'entropie la plus adaptée au cahier des charges de ce Travail de Fin d'Études (conception d'un générateur d'aléa matériel sous forme de clé USB, basé sur un composants discrets), une grille d'évaluation quantifiable a été établie. Le tableau comparatif (voir Tableau 4 du Rapport) repose sur cinq critères stricts décrit ci-dessous.

1. Définition des paramètres de la grille d'évaluation

A. Compatibilité USB (Contrainte d'intégration énergétique)

Ce critère évalue la faisabilité d'alimenter le circuit générateur d'aléa via le standard USB (5V DC, courant limité à environ 500 mA).

- **Oui** : Le phénomène physique peut être exploité sous 5V directement, ou via un circuit élévateur de tension (boost converter) simple et peu consommateur (ex: 12V-18V pour une jonction en avalanche).
- **Partiel** : L'alimentation est théoriquement possible mais requiert des courants trop élevés ou des composants trop encombrants.
- **Non** : Le phénomène exige des tensions ou des puissances incompatibles avec le standard USB (ex: le tube Geiger-Müller nécessite de 400V à 1000V, ce qui pose un problème d'isolation et d'encombrement).

B. Signal exploitable / Observabilité (Contrainte d'auditabilité)

L'objectif du dispositif est de garantir une totale transparence. Il doit être possible de vérifier physiquement la présence d'entropie à la source.

- **Oui** : Le phénomène produit un signal analogique clair (amplitude > 10 mV) avant toute numérisation. Il peut être observé et validé directement à l'oscilloscope, prouvant ainsi le caractère chaotique du signal.
- **Partiel** : Le signal existe mais est difficilement isolable sans équipement de laboratoire très haute fréquence, ou se trouve enfoui au sein d'une puce (ex: conversion ADC interne d'un microcontrôleur).
- **Non** : Le signal initial est d'une amplitude trop faible (ordre du μV pour le bruit thermique) nécessitant une forte amplification qui risque d'ajouter du bruit parasite, ou il s'agit d'un état statique non observable en continu (startup SRAM).

C. Flux (Contrainte de disponibilité)

- **Oui** : Génération d'un flux continu et spontané de données aléatoires avec un débit suffisant (au minimum plusieurs kbps) sans nécessiter d'intervention externe.
- **Partiel** : Processus nécessitant une horloge ou un déclenchement périodique (ex: forcer un état métastable et attendre sa résolution).
- **Non** : Aléa généré de manière unique à la mise sous tension (SRAM) ou phénomène physique trop lent (variations de température ambiante).

D. Robustesse (Contrainte de sécurité matérielle et d'indépendance)

Ce critère juge l'immunité de la source face aux perturbations environnementales (CEM, diaphonie) et aux attaques (injection de signal, variation d'alimentation).

- **Élevée** : Le phénomène tire sa source de la physique quantique ou atomique dans des conditions isolées (Avalanche sous tension de claquage, Radioactivité). Une tentative de manipulation par un champ externe échoue à imposer un état prédictible.

- **Moyenne** : Le phénomène est valide mais vulnérable s'il est implémenté sur un circuit imprimé classique (PCB). Les pistes de cuivre peuvent agir comme des antennes, captant le bruit ambiant ou provoquant une synchronisation parasite (verrouillage de phase sur des oscillateurs discrets).
- **Faible** : Le capteur réagit par définition à son environnement (antennes RF, photodiodes, capteurs Hall, MEMS). Un attaquant peut influencer la sortie en émettant une onde radio, de la lumière ou un champ magnétique. Technologie inadaptée pour de la sécurité cryptographique.

2. Synthèse : Le calcul de la Pertinence globale (Score sur 5)

La note de **Pertinence** n'évalue pas la technologie dans l'absolu, mais son adéquation avec les contraintes strictes de notre cahier des charges (circuit sur PCB, alimentation USB, audibilité analogique, résistance aux attaques).

- **Score 1 à 2 (Technologies rejetées)** : Soit un critère éliminatoire est présent (Pas de flux continu pour la SRAM [2], pas d'USB pour le Geiger [1]), soit la robustesse est jugée trop faible car la source est manipulable de l'extérieur (Capteurs environnementaux, RF, magnétiques [1 à 1.5]).
- **Score 2 à 3 (Technologies imparfaites sur PCB)** : Le bruit thermique [2] et le Shot noise [2] sont de bonnes sources théoriques, mais délicates à exploiter sur un PCB standard sans que le bruit de l'alimentation 5V (ordre du mV) ne vienne masquer le signal aléatoire (ordre du μV).
- **Score 3.5 à 4 (Les standards intégrés)** : Le *Jitter* [3.5] et les *Ring Oscillators* [4] offrent d'excellentes performances au sein de puces silicium (ASIC, FPGA). Cependant, en implémentation **discrète sur PCB**, ils ne permettent pas de prouver facilement la source du chaos à l'oscilloscope et risquent de se synchroniser mutuellement via des couplages parasites.
- **Score 4.5 (Le choix optimal)** : Le **Bruit d'avalanche** se démarque nettement. L'effet de multiplication quantique génère un signal d'amplitude suffisamment élevée. Il est immunisé contre les variations de l'environnement externe, ne peut être manipulé par des ondes radio, et produit un bruit blanc analogique facilement observable. C'est la solution qui répond le mieux au besoin d'un dispositif matériel transparent, robuste et vérifiable.
- **Score 5 (L'idéal théorique / Le choix utopique)** : Ce niveau de pertinence absolu n'existe pas avec les composants discrets actuels. S'il devait exister, ce générateur parfait fonctionnerait nativement sous la tension standard de 5V (USB) et produirait spontanément un bruit non-déterministe pleine échelle (*rail-to-rail*, variant directement de 0V à 5V) avec une bande passante et une vitesse quasi infinies. Ce signal serait directement interfaçable avec les broches de lecture de n'importe quel microcontrôleur, sans nécessiter le moindre étage d'amplification ou de conditionnement préalable.

Annexe B : Résultats de l'évaluation statistique par la suite Dieharder

Afin de valider formellement la qualité de l'entropie brute générée par le circuit matériel conçu (double voie d'amplification du bruit d'avalanche), le flux de données a été soumis à la suite d'outils statistiques *Dieharder* (version 3.31.1). Le fichier d'entrée binaire (`random_data.bin`) soumis à l'évaluation contenait approximativement 6 Go de données brutes, extraites directement du circuit sans subir le moindre post-traitement ou algorithme de blanchiment (hachage, correction de biais).

1. Interprétation des résultats et *p-values*

La suite *Dieharder* évalue la robustesse des nombres aléatoires en soumettant la séquence à diverses épreuves mathématiques. Pour chaque épreuve, elle calcule une *p-value* représentant la probabilité que la séquence testée provienne d'un générateur véritablement aléatoire. Si la distribution globale des tests respecte un comportement uniforme et ne présente pas de défaillance critique, le test est considéré comme réussi.

Nom du test	Principe mathématique et caractéristique évaluée	Résultat
diehard_birthdays	Basé sur le paradoxe des anniversaires. Mesure l'espacement moyen entre l'apparition de valeurs identiques.	PASSED
diehard_operm5	Analyse la distribution des permutations sur des fenêtres glissantes de 5 mots pour traquer des motifs récurrents.	PASSED
diehard_rank	Construit des matrices (32x32 et 6x8) avec les bits générés et calcule leur rang algébrique (indépendance linéaire).	PASSED
diehard_bitstream	Recherche l'occurrence de "mots" de 20 bits se chevauchant pour s'assurer qu'aucun modèle ne fait défaut.	PASSED
diehard_opso/oqso	(<i>Overlapping Pairs/Quadruples Sparse Occupancy</i>). Vérifie l'apparition exhaustive de combinaisons complexes de bits.	PASSED
diehard_dna	Traite les paires de bits comme des bases nucléiques (A, C, G, T) et vérifie la distribution des "mots génétiques".	PASSED
diehard_count_1s	Compte le ratio de bits à '1' par rapport aux '0'. Test crucial pour vérifier l'absence de biais de tension continu.	PASSED
diehard_parking_lot	Simule le placement de cercles dans un espace défini. Vérifie si leur répartition (sans chevauchement) est homogène.	PASSED
diehard_2d/3dsphere	Génère des coordonnées spatiales en 2D et 3D au hasard et s'assure que la distance minimale entre les points est cohérente.	PASSED
diehard_squeeze	Multiplie itérativement des valeurs flottantes aléatoires pour évaluer la vitesse à laquelle le produit converge vers zéro.	PASSED
diehard_sums	Additionne des séries de nombres pour vérifier que la somme des distributions tend bien vers une courbe de Gauss (loi normale).	PASSED
diehard_runs	Analyse les séries ("vagues") de valeurs consécutives croissantes et décroissantes (détecte les oscillations parasites du circuit).	PASSED
diehard_craps	Simule 200 000 parties du jeu de dés "Craps" et vérifie que la distribution statistique des victoires/défaites correspond à la théorie.	PASSED
marsaglia_tsang_gcd	Calcule le Plus Grand Commun Diviseur (PGCD) sur de grandes séquences. Test lourd recherchant des corrélations profondes.	PASSED

Table 1: Résultats de l'évaluation de l'entropie matérielle par la suite Dieharder

Annexe C : Code source du projet 0xEntropy

Cette annexe regroupe l'intégralité du code source développé pour le projet. Le code est organisé selon l'arborescence du projet pour en faciliter la compréhension.

1. Racine du projet

La racine contient le point d'entrée principal du programme.

A. Fichier 0xEntropy.c (Main)

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include "pico/stdlib.h"
#include "pico/multicore.h"
#include "hardware/watchdog.h"
#include "hardware/pwm.h"

// Mes modules pour les stats et l'entropie
#include "Stats/stats.h"
#include "Entropy/entropy.h"

// Le reste des trucs pour l'interface et le hardware
#include "UI/Melody/melody.h"
#include "UI/Screen/Driver/LCD_1in69.h"
#include "UI/Screen/Driver/Touch_1in69.h"
#include "UI/Screen/Driver/GUI_Paint.h"
#include "UI/Screen/Driver/DEV_Config.h"
#include "UI/Screen/Screen_app.h"
#include "Health/health_test.h"
#include "Health/health_warn.h"
#include "UI/Screen/Screen_tactile.h"

#define BUZZER_PIN 19

// Variables pour le tactile et la navigation
Touch_1IN69_XY XY;
PageSysteme page_actuelle = PAGE_DASHBOARD;
bool demander_rafraichissement = true;

// =====

int main() {
    stdio_init_all();

    // Apres 2 jours de debug, fin du biais de l'octets 10 générant un octets 13 pour
    ↪ signaler le retour a la ligne que provoque SDK quand on transmet donnée par port
    ↪ série ... une ligne ....
    stdio_set_translate_crlf(&stdio_usb, false);

    UWORD *BlackImage = NULL;

    // --- On prepare l'ecran ---
    if (DEV_Module_Init() == 0) {
        DEV_SET_PWM(0); // On commence dans le noir pour eviter flash blanc
    ↪ d'initialisation
        LCD_1IN69_Init(VERTICAL);
        LCD_1IN69_Clear(BLACK);
        Touch_1IN69_init(1);
    }
```

```

// Calcul de la taille du buffer pour l'image
uint32_t Imagesize = (uint32_t)LCD_1IN69_HEIGHT * LCD_1IN69_WIDTH * 2;
BlackImage = (UWORD *)malloc(Imagesize);

if(BlackImage != NULL) {
    // Configuration de la zone de dessin
    Paint_NewImage((UBYTE *)BlackImage, LCD_1IN69_WIDTH, LCD_1IN69_HEIGHT, 0,
↪ WHITE);
    Paint_SetScale(65);
    Paint_Clear(BLACK);
    Paint_SetRotate(ROTATE_0);

    // Petit logo au demarrage pour faire propre
    dessiner_logo_entropy(LCD_1IN69_WIDTH, LCD_1IN69_HEIGHT);
    LCD_1IN69_Display(BlackImage);
    DEV_SET_PWM(100); // On allume le retroeclairage
} else {
    printf("Gros souci : impossible d'allouer la memoire pour l'image\r\n");
}
}

// Petite musique d'intro (qui laisse les condensateurs se charger au passage le temps
↪ de l'exécution)
play_star_trek(BUZZER_PIN);

watchdog_enable(2000, 1);

// On lance la generation de nombres aleatoires sur le second coeur
multicore_launch_core1(core1_trng_task);

uint32_t dernier_calcul = to_ms_since_boot(get_absolute_time());
bool alarme_declenchee = false;

while (true) {

    // On donne signe de vie au watchdog
    watchdog_update();

    // Verification des alertes de sante du systeme
    Check_et_alerte(BlackImage);

    // --- Gestion des stats toutes les 2 secondes environ ---
    uint32_t temps_actuel = to_ms_since_boot(get_absolute_time());
    uint32_t delta_temps = temps_actuel - dernier_calcul;

    if (delta_temps > 2000) {
        calculer_statistiques_trng(delta_temps);
        dernier_calcul = temps_actuel;

        // Si on est sur l'ecran principal, faut mettre a jour les chiffres
        if (page_actuelle == PAGE_DASHBOARD) {
            demander_rafraichissement = true;
        }
    }

    // Si on n'a pas d'image en memoire, on ne peut rien faire
    if (BlackImage == NULL) continue;

    // --- Mise a jour de l'affichage ---
    if (demander_rafraichissement) {
        Paint_Clear(BLACK); // On nettoie avant de redessiner

        // Choix du contenu selon la page ou on se trouve
        if (page_actuelle == PAGE_DASHBOARD) {

```



```

        dessiner_page_dashboard(BlackImage);
    } else if (page_actuelle == PAGE_REGLAGES) {
        dessiner_page_reglages(BlackImage);
    } else if (page_actuelle == PAGE_ABOUT) {
        dessiner_page_about(BlackImage);
    }

    // On envoie le tout a l'ecran
    LCD_1IN69_Display(BlackImage);
    demander_rafraichissement = false;
}

// On regarde si l'utilisateur touche l'ecran
gerer_tactile();

// Petite pause pour laisser le processeur souffler
sleep_ms(1);
}
}

```

2. Dossier Entropy/

Ce dossier contient les fichiers gérant l'acquisition du bruit matériel, l'extraction de l'entropie et le hachage cryptographique.

A. Fichier entropy.c

```

#include "entropy.h"
#include <stdio.h>
#include <stdbool.h>
#include "pico/stdlib.h"
#include "hardware/pio.h"
#include "noise.pio.h"

#include "Health/health_test.h"
#include "Stats/stats.h"
#include "sha256.h"

#define PIN_BRUIT 16

int trng_ratio = 2; // A ajuster si besoin

void entropy_init(PIO pio, uint sm, uint pin) {
    uint offset = pio_add_program(pio, &noise_program);
    noise_program_init(pio, sm, offset, pin, 125.0f); // Reviens à 1.0f comme l'ancien code
}

uint32_t entropy_get_data(PIO pio, uint sm) {
    return pio_sm_get_blocking(pio, sm);
}

void core1_trng_task(void) {
    PIO pio = pio0;
    uint sm = 0;

    entropy_init(pio, sm, PIN_BRUIT);

    uint8_t buffer_brut[32] = {0};
    uint8_t hash_out[32];
    int bit_index = 0;
    int byte_index = 0;
}

```

```

while (true) {
    // Le PIO lit les broches 16 et 17 en un coup (si configuré avec 'in pins, 2')
    uint32_t raw_data = entropy_get_data(pio, sm);

    // On traite les 2 bits capturés l'un après l'autre (bit 0 = pin 16, bit 1 = pin
    // 17)
    for (int i = 0; i < 32; i++) {
        uint8_t bit = (raw_data >> i) & 0x01;

        if (bit) {
            buffer_brut[byte_index] |= (1 << bit_index);
        } else {
            buffer_brut[byte_index] &= ~(1 << bit_index);
        }

        bit_index++;
        if (bit_index >= 8) {
            bit_index = 0;
            byte_index++;

            if (byte_index >= 32) {
                byte_index = 0;

                // Vérification de base
                bool ligne_morte = true;
                for(int j = 1; j < 32; j++) {
                    if(buffer_brut[j] != buffer_brut[0]) {
                        ligne_morte = false;
                        break;
                    }
                }

                if (buffer_brut[0] != 0x00 && buffer_brut[0] != 0xFF) {
                    ligne_morte = false;
                }

                verifier_sante_flux_brut((const uint32_t *)buffer_brut, 8);

                if (!capteur_suspect && !ligne_morte && !alerte_sante_trng) {
                    // 1. On HASH le flux brut d'abord
                    sha256_hash(buffer_brut, 32, hash_out);

                    // 2. On fait les STATS sur le hash final (et pas avant !)
                    stats_add_data(hash_out, 32);

                    // 3. On envoie
                    fwrite(hash_out, 1, 32, stdout);
                    fflush(stdout);
                }
            }
        }
    }

    // Le sleep est ici ! On attend que les capteurs physiques (broches 16 et 17)
    // génèrent un nouveau bruit avant de demander la prochaine lecture au PIO.
    sleep_us(trng_ratio);
}
}

```

B. Fichier entropy.h

```
#ifndef ENTROPY_H
#define ENTROPY_H

#include "pico/stdlib.h"
#include "hardware/pio.h"

extern int trng_ratio;

void core1_trng_task(void);

void entropy_init(PIO pio, uint sm, uint pin);
uint32_t entropy_get_data(PIO pio, uint sm);

#endif
```

C. Fichier noise.pio

```
.program noise

; Boucle infinie qui lit 2 bits (broches 16 et 17) et les pousse
loop:
    in pins, 2
    jmp loop

% c-sdk {
static inline void noise_program_init(PIO pio, uint sm, uint offset, uint pin, float
↪ clk_div) {
    pio_sm_config c = noise_program_get_default_config(offset);

    // On configure pour 2 broches consecutives (ex: 16 et 17)
    sm_config_set_in_pins(&c, pin);
    pio_sm_set_consecutive_pindirs(pio, sm, pin, 2, false);

    pio_gpio_init(pio, pin);
    pio_gpio_init(pio, pin + 1); // Initialisation de la deuxieme broche !

    // Configure le décalage (32 bits = 16 lectures de 2 bits)
    sm_config_set_in_shift(&c, false, true, 32);
    sm_config_set_clkdiv(&c, clk_div);

    pio_sm_init(pio, sm, offset, &c);
    pio_sm_set_enabled(pio, sm, true);
}
%}
```

D. Fichier sha256.c

Ce code est tiré du dépôt GitHub de Brad Conte sur les algorithmes cryptographiques, et a été adapté par mes soins sur la base de la documentation du NIST.

```
#include "sha256.h"
#include <string.h>

// Quelques macros pour les rotations et les fonctions logiques de base de SHA-256
#define ROTLEFT(a,b) (((a) << (b)) | ((a) >> (32-(b))))
#define ROTRIGHT(a,b) (((a) >> (b)) | ((a) << (32-(b))))
```

```

#define CH(x,y,z) (((x) & (y)) ^ (~(x) & (z)))
#define MAJ(x,y,z) (((x) & (y)) ^ ((x) & (z)) ^ ((y) & (z)))
#define EPO(x) (ROTRIGHT(x,2) ^ ROTRIGHT(x,13) ^ ROTRIGHT(x,22))
#define EP1(x) (ROTRIGHT(x,6) ^ ROTRIGHT(x,11) ^ ROTRIGHT(x,25))
#define SIG0(x) (ROTRIGHT(x,7) ^ ROTRIGHT(x,18) ^ ((x) >> 3))
#define SIG1(x) (ROTRIGHT(x,17) ^ ROTRIGHT(x,19) ^ ((x) >> 10))

// Les constantes k definies par le standard fips (racines cubiques des nombres premiers)
static const uint32_t k[64] = {
    0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1, 0x923f82a4,
    ↪ 0xab1c5ed5,
    0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe, 0x9bdc06a7,
    ↪ 0xc19bf174,
    0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc,
    ↪ 0x76f988da,
    0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147, 0x06ca6351,
    ↪ 0x14292967,
    0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354, 0x766a0abb, 0x81c2c92e,
    ↪ 0x92722c85,
    0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624, 0xf40e3585,
    ↪ 0x106aa070,
    0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f,
    ↪ 0x682e6ff3,
    0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2
};

// C'est ici qu'on fait le gros du travail de compression sur un bloc de 512 bits
static void sha256_transform(SHA256_CTX *ctx, const uint8_t data[]) {
    uint32_t a, b, c, d, e, f, g, h, i, j, t1, t2, m[64];

    // On prepare le message schedule
    for (i = 0, j = 0; i < 16; ++i, j += 4)
        m[i] = (data[j] << 24) | (data[j + 1] << 16) | (data[j + 2] << 8) | (data[j + 3]);
    for (; i < 64; ++i)
        m[i] = SIG1(m[i - 2]) + m[i - 7] + SIG0(m[i - 15]) + m[i - 16];

    // On initialise nos variables de travail avec l'etat actuel
    a = ctx->state[0]; b = ctx->state[1]; c = ctx->state[2]; d = ctx->state[3];
    e = ctx->state[4]; f = ctx->state[5]; g = ctx->state[6]; h = ctx->state[7];

    // La boucle principale avec ses 64 rounds
    for (i = 0; i < 64; ++i) {
        t1 = h + EP1(e) + CH(e,f,g) + k[i] + m[i];
        t2 = EPO(a) + MAJ(a,b,c);
        h = g; g = f; f = e; e = d + t1;
        d = c; c = b; b = a; a = t1 + t2;
    }

    // On rajoute le resultat a l'etat precedent
    ctx->state[0] += a; ctx->state[1] += b; ctx->state[2] += c; ctx->state[3] += d;
    ctx->state[4] += e; ctx->state[5] += f; ctx->state[6] += g; ctx->state[7] += h;
}

// Initialise la structure pour un nouveau calcul de hash
void sha256_init(SHA256_CTX *ctx) {
    ctx->datalen = 0;
    ctx->bitlen = 0;
    /* Valeurs initiales standard (h0 a h7) */
    ctx->state[0] = 0x6a09e667; ctx->state[1] = 0xbb67ae85;
    ctx->state[2] = 0x3c6ef372; ctx->state[3] = 0xa54ff53a;
    ctx->state[4] = 0x510e527f; ctx->state[5] = 0x9b05688c;
    ctx->state[6] = 0x1f83d9ab; ctx->state[7] = 0x5be0cd19;
}

```

```

// On ajoute des donnees au fur et a mesure
void sha256_update(SHA256_CTX *ctx, const uint8_t data[], size_t len) {
    uint32_t i;
    for (i = 0; i < len; ++i) {
        ctx->data[ctx->datalen] = data[i];
        ctx->datalen++;
        // Des qu'on a un bloc complet de 64 octets, on transforme
        if (ctx->datalen == 64) {
            sha256_transform(ctx, ctx->data);
            ctx->bitlen += 512;
            ctx->datalen = 0;
        }
    }
}

// On termine le hash et on recupere le resultat
void sha256_final(SHA256_CTX *ctx, uint8_t hash[]) {
    uint32_t i;
    i = ctx->datalen;

    // etape de padding : on ajoute un bit a 1 (0x80) puis des zeros
    if (ctx->datalen < 56) {
        ctx->data[i++] = 0x80;
        while (i < 56) ctx->data[i++] = 0x00;
    } else {
        ctx->data[i++] = 0x80;
        while (i < 64) ctx->data[i++] = 0x00;
        sha256_transform(ctx, ctx->data);
        memset(ctx->data, 0, 56);
    }

    // On enregistre la taille totale du message en bits a la fin
    ctx->bitlen += ctx->datalen * 8;
    ctx->data[63] = ctx->bitlen; ctx->data[62] = ctx->bitlen >> 8;
    ctx->data[61] = ctx->bitlen >> 16; ctx->data[60] = ctx->bitlen >> 24;
    ctx->data[59] = ctx->bitlen >> 32; ctx->data[58] = ctx->bitlen >> 40;
    ctx->data[57] = ctx->bitlen >> 48; ctx->data[56] = ctx->bitlen >> 56;
    sha256_transform(ctx, ctx->data);

    // On remplit le tableau de sortie en convertissant l'etat interne (big-endian)
    for (i = 0; i < 4; ++i) {
        hash[i] = (ctx->state[0] >> (24 - i * 8)) & 0x000000ff;
        hash[i + 4] = (ctx->state[1] >> (24 - i * 8)) & 0x000000ff;
        hash[i + 8] = (ctx->state[2] >> (24 - i * 8)) & 0x000000ff;
        hash[i + 12] = (ctx->state[3] >> (24 - i * 8)) & 0x000000ff;
        hash[i + 16] = (ctx->state[4] >> (24 - i * 8)) & 0x000000ff;
        hash[i + 20] = (ctx->state[5] >> (24 - i * 8)) & 0x000000ff;
        hash[i + 24] = (ctx->state[6] >> (24 - i * 8)) & 0x000000ff;
        hash[i + 28] = (ctx->state[7] >> (24 - i * 8)) & 0x000000ff;
    }
}

// Petite fonction pour generer le hash d'un coup
void sha256_hash(const uint8_t *data, size_t len, uint8_t hash[SHA256_BLOCK_SIZE]) {
    SHA256_CTX ctx;
    sha256_init(&ctx);
    sha256_update(&ctx, data, len);
    sha256_final(&ctx, hash);
}

```

E. Fichier sha256.h

```
#ifndef SHA256_H
#define SHA256_H

#include <stdint.h>
#include <stddef.h>

#define SHA256_BLOCK_SIZE 32

typedef struct {
    uint8_t data[64];
    uint32_t datalen;
    unsigned long long bitlen;
    uint32_t state[8];
} SHA256_CTX;

void sha256_init(SHA256_CTX *ctx);
void sha256_update(SHA256_CTX *ctx, const uint8_t data[], size_t len);
void sha256_final(SHA256_CTX *ctx, uint8_t hash[]);

void sha256_hash(const uint8_t *data, size_t len, uint8_t hash[SHA256_BLOCK_SIZE]);

#endif
```

3. Dossier Health/

Ce dossier est dédié aux tests de santé en continu (Health Tests) permettant de s'assurer de la qualité de l'entropie générée.

A. Fichier health_test.c

```
#include "health_test.h"
#include "pico/stdlib.h"
#include "pico/time.h" // Utile pour gerer le timeout de 10 secondes

volatile bool alerte_sante_trng = false;
volatile bool capteur_suspect = false;

// On garde une trace des stats sur une petite fenetre
static uint32_t total_ones_acc = 0;
static uint32_t total_bits_acc = 0;
#define HEALTH_CHECK_WINDOW 8192 // On analyse par paquets de 1024 octets

// Pour gerer le chrono du mode default
static bool en_default = false;
static absolute_time_t debut_default_time;

bool verifier_sante_flux_brut(const uint32_t *raw_buffer, size_t mots_a_lire) {
    uint32_t raw_ones = 0;

    // Comptage des bits a '1' dans le bloc qui vient d'arriver
    for (size_t i = 0; i < mots_a_lire; i++) {
        raw_ones += __builtin_popcount(raw_buffer[i]);
    }

    total_ones_acc += raw_ones;
    total_bits_acc += (mots_a_lire * 32);

    // Une fois qu'on a assez de matiere, on fait le bilan
    if (total_bits_acc >= HEALTH_CHECK_WINDOW) {
```

```

uint32_t total_zeros_acc = total_bits_acc - total_ones_acc;

// --- TEST : est-ce que le flux est naze ? (moins de 5% de 1 ou de 0) ---
if (total_ones_acc < (total_bits_acc / 20) || total_zeros_acc < (total_bits_acc /
↪ 20)) {

    if (!en_defaut) {
        // C'est le début d'un souci, on lance le chrono
        en_defaut = true;
        debut_defaut_time = get_absolute_time();
    } else {
        // Ca continue de deconner... on regarde si ca depasse les 10 secondes
        if (absolute_time_diff_us(debut_defaut_time, get_absolute_time()) >=
↪ 10000000) {
            alerte_sante_trng = true; // Alerte rouge declenchée
            return false;
        }
    }
} else {
    // Tout va bien, on reset l'état de default s'il y en avait un
    en_defaut = false;
}

// On remet les compteurs a zero pour la prochaine analyse
total_ones_acc = 0;
total_bits_acc = 0;
}

return true;
}

```

B. Fichier health_test.h

```

#ifndef HEALTH_TEST_H
#define HEALTH_TEST_H

#include <stdint.h>
#include <stdbool.h>
#include <stddef.h>

extern volatile bool alerte_sante_trng;
extern volatile bool capteur_suspect;

bool verifier_sante_flux_brut(const uint32_t *raw_buffer, size_t mots_a_lire);

#endif

```

C. Fichier health_warn.c

```

#include "health_warn.h"

// faut que alerte_sante_trng soit bien declaree dans ce header sinon ca va planter
#include "Health/health_test.h"

// les trucs de base pour le pico (pwm, gpio, watchdog et compagnie)
#include "pico/stdlib.h"
#include "hardware/pwm.h"
#include "hardware/watchdog.h"
#include "hardware/gpio.h"

```

```

// drivers pour l'ecran waveshare et la gestion du tactile
#include "UI/Screen/Driver/DEV_Config.h"
#include "UI/Screen/Driver/LCD_1in69.h"
#include "UI/Screen/Driver/Touch_1in69.h"
#include "UI/Screen/Screen_app.h"

#define BUZZER_PIN 19

// un petit flag pour eviter que l'alarme se relance en boucle
bool alarme_declenchee = false;

// la fonction qui verifie si ca a pete et balance l'alerte
void Check_et_alerte(UWORD *image_buffer) {
    if (alerte_sante_trng && !alarme_declenchee) {
        alarme_declenchee = true;

        // on met la luminosite a fond
        DEV_SET_PWM(100);

        // on dessine l'ecran de la mort en rouge si le buffer est pret
        if (image_buffer != NULL) {
            dessiner_page_fatal(image_buffer);
            LCD_1IN69_Display(image_buffer);
        }

        // on configure le buzzer pour qu'il siffle a environ 1.25 khz
        gpio_set_function(BUZZER_PIN, GPIO_FUNC_PWM);
        uint slice_num = pwm_gpio_to_slice_num(BUZZER_PIN);
        pwm_set_clkdiv(slice_num, 10.0f);
        pwm_set_wrap(slice_num, 10000);
        pwm_set_chan_level(slice_num, pwm_gpio_to_channel(BUZZER_PIN), 5000);
        pwm_set_enabled(slice_num, true);

        bool son_active = true;

        // boucle infinie pour bloquer le systeme et attendre un appui tactile
        while(true) {
            watchdog_update(); // on oublie pas de nourrir le chien pour eviter le reboot

            if (son_active) {
                // on regarde si quelqu'un touche l'ecran
                if (DEV_Digital_Read(DEV_I2C_INT) == 0) {
                    Touch_1IN69_XY struct_tc = Touch_1IN69_Get_Point();
                    if (struct_tc.x_point != 0 && struct_tc.y_point != 0) {

                        // tactile detecte : on coupe le pwm
                        pwm_set_enabled(slice_num, false);

                        // on remet la pin en sortie normale a 0 pour etre sur que ca se
                        // taise
                        gpio_set_function(BUZZER_PIN, GPIO_FUNC_SIO);
                        gpio_set_dir(BUZZER_PIN, GPIO_OUT);
                        gpio_put(BUZZER_PIN, 0);

                        son_active = false;
                    }
                }
            }
            sleep_ms(100); // petite pause pour pas faire chauffer le processeur
        }
    }
}

```



```
}
```

D. Fichier health_warn.h

```
#ifndef HEALTH_WARN_H
#define HEALTH_WARN_H

#include <stdbool.h>
#include "UI/Screen/Driver/GUI_Paint.h"

void Check_et_alerte(UWORD *image_buffer);

#endif
```

4. Dossier Stats/

Ce dossier gère le traitement et l'affichage des statistiques liées à la génération des nombres aléatoires.

A. Fichier stats.c

```
#include "stats.h"
#include <stdio.h>
#include <math.h>
#include "pico/stdlib.h"
#include "pico/critical_section.h" // Pour proteger les variables entre les 2 coeurs

#include "Entropy/entropy.h"
#include "Entropy/sha256.h"
#include "Health/health_test.h"

// Variables partagees
volatile uint32_t hc_compteur_octets[256] = {0};
volatile uint32_t hc_compteur_bits_1 = 0;
volatile uint32_t hc_total_octets = 0;

typedef struct {
    uint32_t compteurs_octets[256];
    uint32_t total_octets;
    uint32_t total_bits_1;
} BlocHistorique;

#define NB_BLOCS_HISTORIQUE 40

static BlocHistorique historique[NB_BLOCS_HISTORIQUE] = {0};
static int index_bloc_actuel = 0;

float ui_entropie = 0.0f;
float ui_ratio_bits_1 = 50.0f;
uint32_t ui_debit_o_s = 0;

// Une section critique pour empecher les coeurs de se marcher dessus
static critical_section_t stats_crit_sec;
static bool crit_sec_initialized = false;

void stats_add_data(const uint8_t *data, size_t len) {
    if (!crit_sec_initialized) {
        critical_section_init(&stats_crit_sec);
        crit_sec_initialized = true;
    }
}
```

```

uint32_t local_compteur_bits_1 = 0;
uint32_t local_compteur_octets[256] = {0};

// On prepare tout en local pour ne pas bloquer le systeme
for (size_t i = 0; i < len; i++) {
    uint8_t octet = data[i];
    local_compteur_octets[octet]++;

    uint8_t temp = octet;
    while (temp) {
        temp &= (temp - 1);
        local_compteur_bits_1++;
    }
}

// On verrouille brievement pour transferer au Core 0
critical_section_enter_blocking(&stats_crit_sec);
for (int i = 0; i < 256; i++) {
    if (local_compteur_octets[i] > 0) {
        hc_compteur_octets[i] += local_compteur_octets[i];
    }
}
hc_total_octets += len;
hc_compteur_bits_1 += local_compteur_bits_1;
critical_section_exit(&stats_crit_sec);
}

void calculer_statistiques_trng(uint32_t delta_temps) {
    if (!crit_sec_initialized) return;

    // 1. On recupere les donnees en verrouillant brievement
    critical_section_enter_blocking(&stats_crit_sec);

    if (hc_total_octets > 0) {
        if (capteur_suspect) {
            ui_debit_o_s = 0;
        } else if (delta_temps > 0) {
            ui_debit_o_s = (hc_total_octets * 1000) / delta_temps;
        }
    } else {
        ui_debit_o_s = 0;
    }

    // On transfere dans le bloc actuel
    for (int i = 0; i < 256; i++) {
        historique[index_bloc_actuel].compteurs_octets[i] = hc_compteur_octets[i];
        hc_compteur_octets[i] = 0;
    }
    historique[index_bloc_actuel].total_octets = hc_total_octets;
    hc_total_octets = 0;
    historique[index_bloc_actuel].total_bits_1 = hc_compteur_bits_1;
    hc_compteur_bits_1 = 0;

    critical_section_exit(&stats_crit_sec);

    // 2. On avance la fenetre glissante
    index_bloc_actuel = (index_bloc_actuel + 1) % NB_BLOCS_HISTORIQUE;
    historique[index_bloc_actuel].total_octets = 0;
    historique[index_bloc_actuel].total_bits_1 = 0;
    for (int i = 0; i < 256; i++) {
        historique[index_bloc_actuel].compteurs_octets[i] = 0;
    }
}

```

```

// 3. Calcul de l'entropie et du RATIO sur l'ensemble
uint32_t somme_compteurs[256] = {0};
uint32_t somme_octets = 0;
uint32_t somme_bits_1 = 0;

for (int b = 0; b < NB_BLOCS_HISTORIQUE; b++) {
    somme_octets += historique[b].total_octets;
    somme_bits_1 += historique[b].total_bits_1;
    for (int i = 0; i < 256; i++) {
        somme_compteurs[i] += historique[b].compteurs_octets[i];
    }
}

if (somme_octets > 0) {
    ui_ratio_bits_1 = ((float)somme_bits_1 / (float)(somme_octets * 8)) * 100.0f;

    float entropie_temp = 0.0f;
    for (int i = 0; i < 256; i++) {
        if (somme_compteurs[i] > 0) {
            float probabilite = (float)somme_compteurs[i] / (float)somme_octets;
            entropie_temp -= probabilite * log2f(probabilite);
        }
    }
    ui_entropie = entropie_temp;
}

void stats_reset(void) {
    if (crit_sec_initialized) {
        critical_section_enter_blocking(&stats_crit_sec);
    }
    hc_compteur_bits_1 = 0;
    hc_total_octets = 0;
    for (int i = 0; i < 256; i++) {
        hc_compteur_octets[i] = 0;
    }
    if (crit_sec_initialized) {
        critical_section_exit(&stats_crit_sec);
    }

    for (int b = 0; b < NB_BLOCS_HISTORIQUE; b++) {
        historique[b].total_octets = 0;
        historique[b].total_bits_1 = 0;
        for (int i = 0; i < 256; i++) {
            historique[b].compteurs_octets[i] = 0;
        }
    }
    index_bloc_actuel = 0;

    ui_entropie = 0.0f;
    ui_ratio_bits_1 = 50.0f;
    ui_debit_o_s = 0;
}

```

B. Fichier stats.h

```

#ifndef STATS_H
#define STATS_H

#include <stdint.h>

```

```

#include <stddef.h>

extern float ui_entropie;
extern float ui_ratio_bits_1;
extern uint32_t ui_debit_o_s;

void stats_reset(void);

void stats_add_data(const uint8_t *data, size_t len);

void calculer_statistiques_trng(uint32_t delta_temps);

#endif // STATS_H

```

5. Dossier UI/

Ce dossier regroupe les éléments liés à l'interface utilisateur, incluant l'écran tactile.

A. Fichier Screen/Screen_app.c

```

#include "Screen_app.h"
#include "pico/stdlib.h"
#include <stdio.h>
#include "../Entropy/entropy.h"
#include <stdbool.h>
#include <string.h>

extern bool intro_sound_enabled;
extern bool always_on;
extern float ui_entropie;
extern float ui_ratio_bits_1;
extern uint32_t ui_debit_o_s;

// MATRICES PIXEL ART

const char* design_logo_entropy[8] = {
    " 000      EEEEE      t",
    "0  0      E          t",
    "0 00 x   x E      nnnn ttttt rrrr   ooo pppp y y",
    "0 0 0 x x EEEE n  n  t  r  r o  o p  p y y",
    "00 0  x  E      n  n  t  r      o  o p  p y y",
    "0  0 x x E      n  n  t  r      o  o pppp  yyy",
    " 000 x  x EEEEE n  n  t  r      ooo p      y",
    "                                p      yy "
};

const char* design_logo_settings[5] = {
    " SSSS EEEEE TTTT TTTT I N  N GGGG SSSS ",
    "S      E      T      T  I NN  NG  S      ",
    " SSS EEEEE  T      T  I N N NG  GG  SSS ",
    "   S E      T      T  I N  NN G  G    S ",
    "SSSS EEEEE  T      T  I N  N GGGG SSSS "
};

const unsigned char Uptime[] = {
    //Economie d'impression. contenu bitmap d'une image
};

const unsigned char Ratio[] = {
    //Economie d'impression
};

```

```

const unsigned char Health[] = {
    //Economie d'impression
};

const unsigned char Bitrate[] = {
    //Economie d'impression
};

const unsigned char Settings[] = {
    //Economie d'impression
};

const unsigned char Arrow[] = {
    //Economie d'impression
};

const unsigned char Lune[] = {
    //Economie d'impression
};

const unsigned char Lunebarre [] = {
    //Economie d'impression
};

const unsigned char Qrcode[] = {
    //Economie d'impression
};

const unsigned char Qrcodefatal[] = {
    //Economie d'impression
};

// fonctions pour le dessin

void dessiner_rectangle_arrondi(uint16_t x1, uint16_t y1, uint16_t x2, uint16_t y2,
    ↪ uint16_t rayon, uint16_t couleur) {
    // on trace deux rectangles croisés et des cercles aux coins pour l'effet arrondi
    Paint_DrawRectangle(x1 + rayon, y1, x2 - rayon, y2 + 1, couleur, DRAW_FILL_FULL,
    ↪ DOT_PIXEL_1X1);
    Paint_DrawRectangle(x1, y1 + rayon, x2 + 1, y2 - rayon, couleur, DRAW_FILL_FULL,
    ↪ DOT_PIXEL_1X1);
    Paint_DrawCircle(x1 + rayon, y1 + rayon, rayon, couleur, DRAW_FILL_FULL,
    ↪ DOT_PIXEL_1X1);
    Paint_DrawCircle(x2 - rayon, y1 + rayon, rayon, couleur, DRAW_FILL_FULL,
    ↪ DOT_PIXEL_1X1);
    Paint_DrawCircle(x1 + rayon, y2 - rayon, rayon, couleur, DRAW_FILL_FULL,
    ↪ DOT_PIXEL_1X1);
    Paint_DrawCircle(x2 - rayon, y2 - rayon, rayon, couleur, DRAW_FILL_FULL,
    ↪ DOT_PIXEL_1X1);
}

void dessiner_rectangle_arrondi_creux(uint16_t x1, uint16_t y1, uint16_t x2, uint16_t y2,
    ↪ uint16_t rayon, uint16_t epaisseur, uint16_t couleur) {
    // on dessine le fond puis on vide le milieu avec un rectangle noir
    dessiner_rectangle_arrondi(x1, y1, x2, y2, rayon, couleur);
    dessiner_rectangle_arrondi(x1 + epaisseur, y1 + epaisseur, x2 - epaisseur, y2 -
    ↪ epaisseur,
                                (rayon > epaisseur) ? (rayon - epaisseur) : 1, BLACK);
}

void dessiner_barre_pilule(uint16_t x, uint16_t y, uint16_t largeur, uint16_t hauteur,
    ↪ uint8_t pourcentage, uint16_t couleur_gauche, uint16_t couleur_droite) {
    uint16_t epaisseur_bord = 1;
    uint16_t couleur_contour = DARK_GRAY;

```

```

    if (pourcentage > 100) pourcentage = 100;

    // calcul des dimensions du contour
    uint16_t x_contour = x - 1;
    uint16_t y_contour = y;
    uint16_t x2_contour = x + largeur;
    uint16_t y2_contour = y + hauteur - 1;

    uint16_t w_contour = x2_contour - x_contour;
    uint16_t h_contour = y2_contour - y_contour;
    uint16_t rayon_contour = h_contour / 2;

    dessiner_rectangle_arrondi_creux(x_contour, y_contour, x2_contour, y2_contour,
↪ rayon_contour, epaisseur_bord, couleur_contour);

    // zone de remplissage a l'interieur
    uint16_t marge = epaisseur_bord + 1;
    uint16_t x_in = x_contour + marge;
    uint16_t y_in = y_contour + marge;
    uint16_t w_in = w_contour - (marge * 2);
    uint16_t h_in = h_contour - (marge * 2);

    uint16_t largeur_remplissage = (w_in * pourcentage) / 100;
    uint16_t r_in = h_in / 2;

    // on dessine la barre pixel par pixel pour bien gerer l'arrondi
    for (uint16_t i = 0; i < w_in; i++) {
        uint16_t dx = 0;
        if (i < r_in) {
            dx = r_in - 1 - i;
        } else if (i >= w_in - r_in) {
            dx = i - (w_in - r_in);
        }

        uint16_t max_dy = 0;
        while ((dx * dx + (max_dy + 1) * (max_dy + 1)) <= (r_in * r_in)) {
            max_dy++;
        }

        uint16_t decalage_y = r_in - max_dy;
        uint16_t couleur_ligne = (i < largeur_remplissage) ? couleur_gauche :
↪ couleur_droite;

        uint16_t y_start = y_in + decalage_y;
        uint16_t y_end = y_in + h_in - 1 - decalage_y;

        if (y_start <= y_end) {
            Paint_DrawLine(x_in + i, y_start, x_in + i, y_end, couleur_ligne,
↪ DOT_PIXEL_1X1, LINE_STYLE_SOLID);
        }
    }
}

void dessiner_icone_40x40(uint16_t x, uint16_t y, const unsigned char* bitmap, uint16_t
↪ couleur_icone) {
    // affichage d'une icone bitmap simple
    uint8_t octets_par_ligne = 5;
    for (uint16_t j = 0; j < 40; j++) {
        for (uint16_t i = 0; i < 40; i++) {
            uint16_t index_octet = (j * octets_par_ligne) + (i / 8);
            uint8_t bit_index = 7 - (i % 8);
            if ((bitmap[index_octet] & (1 << bit_index)) == 0) {
                Paint_SetPixel(x + i, y + j, couleur_icone);
            }
        }
    }
}

```

```

    }
}

}

void dessiner_qrcode_100x100(uint16_t x, uint16_t y, const unsigned char* bitmap) {
    // rendu du QR code
    uint8_t octets_par_ligne = 13;

    for (uint16_t j = 0; j < 100; j++) {
        for (uint16_t i = 0; i < 100; i++) {
            uint16_t index_octet = (j * octets_par_ligne) + (i / 8);
            uint8_t bit_index = 7 - (i % 8);

            if ((bitmap[index_octet] & (1 << bit_index)) == 0) {
                Paint_SetPixel(x + i, y + j, BLACK);
            } else {
                Paint_SetPixel(x + i, y + j, WHITE);
            }
        }
    }
}

void dessiner_fond_grille(uint16_t largeur_ecran, uint16_t hauteur_ecran) {
    // petit quadrillage de fond pour le style
    Paint_DrawRectangle(0, 0, largeur_ecran, hauteur_ecran, BLACK, DRAW_FILL_FULL,
    ↪ DOT_PIXEL_1X1);
    uint16_t espacement = 12;
    for (uint16_t x = 0; x < largeur_ecran; x += espacement) {
        Paint_DrawLine(x, 0, x, hauteur_ecran, DARK_GRAY, DOT_PIXEL_1X1, LINE_STYLE_SOLID);
    }
    for (uint16_t y = 0; y < hauteur_ecran; y += espacement) {
        Paint_DrawLine(0, y, largeur_ecran, y, DARK_GRAY, DOT_PIXEL_1X1, LINE_STYLE_SOLID);
    }
}

void dessiner_logo_entropy(uint16_t largeur_ecran, uint16_t hauteur_ecran) {
    // le logo fait avec des petits cubes
    dessiner_fond_grille(largeur_ecran, hauteur_ecran);
    uint16_t lignes = 8, colonnes = 52, taille_cube = 3, taille_gap = 1;
    uint16_t pas = taille_cube + taille_gap;
    uint16_t start_x = (largeur_ecran - (colonnes * pas)) / 2;
    uint16_t start_y = (hauteur_ecran - (lignes * pas)) / 2;

    for (uint16_t y = 0; y < lignes; y++) {
        for (uint16_t x = 0; x < colonnes; x++) {
            if (design_logo_entropy[y][x] != ' ') {
                uint16_t px = start_x + (x * pas);
                uint16_t py = start_y + (y * pas);
                Paint_DrawRectangle(px, py, px + pas - 1, py + pas - 1, BLACK,
    ↪ DRAW_FILL_FULL, DOT_PIXEL_1X1);
                Paint_DrawRectangle(px, py, px + taille_cube - 1, py + taille_cube - 1,
    ↪ WHITE, DRAW_FILL_FULL, DOT_PIXEL_1X1);
            }
        }
    }
}

void dessiner_entete_dashboard(uint16_t largeur_ecran, uint16_t hauteur_ecran) {
    // header du dashboard principal
    dessiner_fond_grille(largeur_ecran, hauteur_ecran);
    uint16_t lignes = 8, colonnes = 52, taille_cube = 2, taille_gap = 1;
    uint16_t pas = taille_cube + taille_gap;
    uint16_t start_x = (largeur_ecran - (colonnes * pas)) / 2;

```

```

uint16_t start_y = 15;

for (uint16_t y = 0; y < lignes; y++) {
    for (uint16_t x = 0; x < colonnes; x++) {
        if (design_logo_entropy[y][x] != ' ') {
            uint16_t px = start_x + (x * pas);
            uint16_t py = start_y + (y * pas);
            Paint_DrawRectangle(px, py, px + pas - 1, py + pas - 1, BLACK,
↪ DRAW_FILL_FULL, DOT_PIXEL_1X1);
            Paint_DrawRectangle(px, py, px + taille_cube - 1, py + taille_cube - 1,
↪ WHITE, DRAW_FILL_FULL, DOT_PIXEL_1X1);
        }
    }
}

void dessiner_entete_settings(uint16_t largeur_ecran, uint16_t hauteur_ecran) {
    // header pour la page reglages
    dessiner_fond_grille(largeur_ecran, hauteur_ecran);
    uint16_t colonnes = 45, lignes = 5, taille_cube = 2, taille_gap = 1;
    uint16_t pas = taille_cube + taille_gap;
    uint16_t start_x = ((largeur_ecran - (colonnes * pas)) / 2) + 3;
    uint16_t start_y = 15;

    for (uint16_t y = 0; y < lignes; y++) {
        for (uint16_t x = 0; x < colonnes; x++) {
            if (design_logo_settings[y][x] != ' ') {
                uint16_t px = start_x + (x * pas);
                uint16_t py = start_y + (y * pas);
                Paint_DrawRectangle(px, py, px + pas - 1, py + pas - 1, BLACK,
↪ DRAW_FILL_FULL, DOT_PIXEL_1X1);
                Paint_DrawRectangle(px, py, px + taille_cube - 1, py + taille_cube - 1,
↪ WHITE, DRAW_FILL_FULL, DOT_PIXEL_1X1);
            }
        }
    }
}

void dessiner_page_dashboard(UWORD *BlackImage) {
    dessiner_entete_dashboard(240, 280);

    uint16_t marge_x = 10, hauteur_box = 36, espace_y = 8, espace_x = 5, start_y = 55;
    uint16_t x1_carre = marge_x, x2_carre = x1_carre + hauteur_box;
    uint16_t x1_rect = x2_carre + espace_x, x2_rect = 240 - marge_x;
    uint16_t epaisseur_bord = 1, rayon_bord = 5;

    // bloc 1 : affichage du ratio

    uint16_t y1 = start_y;
    dessiner_rectangle_arrondi_creux(x1_carre, y1, x2_carre, y1 + hauteur_box, rayon_bord,
↪ epaisseur_bord, COLOR_BOX_GREEN);
    dessiner_icone_40x40(x1_carre - 2, y1 - 2, Ratio, WHITE);
    dessiner_rectangle_arrondi_creux(x1_rect, y1, x2_rect, y1 + hauteur_box, rayon_bord,
↪ epaisseur_bord, COLOR_BOX_GREEN);

    uint16_t health_percent_1 = (uint16_t)(ui_ratio_bits_1 + 0.5f);
    if(health_percent_1 > 100) health_percent_1 = 100;
    uint16_t health_percent_0 = 100 - health_percent_1;

    char texte_0[10], texte_1[10];
    sprintf(texte_0, "%d%%", health_percent_0);
    sprintf(texte_1, "%d%%", health_percent_1);

    uint16_t hauteur_barre = 20;

```



```

uint16_t y_barre = y1 + ((hauteur_box - hauteur_barre) / 2);
Paint_DrawString_EN(x1_rect + 6, y1 + 12, texte_0, &Font12, BLACK, WHITE);

uint16_t decalage_gauche = 34, decalage_droite = 34;
uint16_t x_barre = x1_rect + decalage_gauche;
uint16_t largeur_barre = (x2_rect - x1_rect) - decalage_gauche - decalage_droite;
uint8_t tolerance = 4;
uint16_t couleur_0, couleur_1;

// change la couleur si on s'eloigne trop du 50/50
if (health_percent_0 < (50 - tolerance) || health_percent_0 > (50 + tolerance)) {
    couleur_0 = COLOR_BAR_RED; couleur_1 = COLOR_BAR_RED_FLASH;
} else {
    couleur_0 = COLOR_BAR_GREEN; couleur_1 = COLOR_BOX_BLUE;
}

dessiner_barre_pilule(x_barre, y_barre, largeur_barre, hauteur_barre,
↪ health_percent_0, couleur_0, couleur_1);
Paint_DrawString_EN(x_barre + largeur_barre + 4, y1 + 12, texte_1, &Font12, BLACK,
↪ WHITE);

// bloc 2 : debit de donnees

uint16_t y2 = y1 + hauteur_box + espace_y;

// icone a gauche
dessiner_rectangle_arrondi_creux(x1_carre, y2, x2_carre, y2 + hauteur_box, rayon_bord,
↪ epaisseur_bord, COLOR_BOX_PINK);
dessiner_icone_40x40(x1_carre - 2, y2 - 2, Bitrate, WHITE);

// boite pour la valeur
dessiner_rectangle_arrondi_creux(x1_rect, y2, x2_rect, y2 + hauteur_box, rayon_bord,
↪ epaisseur_bord, COLOR_BOX_PINK);

// conversion du debit
float debit_kbit_s = ((float)ui_debit_o_s * 8.0f) / 1000.0f;
char texte_valeur[16];
sprintf(texte_valeur, "%.1f", debit_kbit_s);

// on met le chiffre en gros
Paint_DrawString_EN(x1_rect + 16, y2 + 10, texte_valeur, &Font20, BLACK, WHITE);

// et l'unité juste a cote
uint16_t x_kbps = (x1_rect + 12) + (strlen(texte_valeur) * 14) + 4;
Paint_DrawString_EN(x_kbps, y2 + 18, "kbps", &Font12, BLACK, WHITE);

// petite zone pour le ratio TRNG
uint16_t largeur_ratio = 36;
uint16_t hauteur_ratio = 20;

// place a 5 px du bord droit
uint16_t x1_ratio = x2_rect - largeur_ratio - 5;
uint16_t y1_ratio = y2 + (hauteur_box - hauteur_ratio) / 2;

dessiner_rectangle_arrondi_creux(x1_ratio, y1_ratio, x1_ratio + largeur_ratio, y1_ratio
↪ + hauteur_ratio, 3, 1, COLOR_BOX_PINK);

char texte_ratio[10];
sprintf(texte_ratio, "%d:1", trng_ratio);

// on centre le texte a la main
uint16_t txt_ratio_w = strlen(texte_ratio) * 7;
uint16_t x_txt_ratio = x1_ratio + (largeur_ratio - txt_ratio_w) / 2;
uint16_t y_txt_ratio = y1_ratio + (hauteur_ratio - 12) / 2;

```

```

Paint_DrawString_EN(x_txt_ratio, y_txt_ratio, texte_ratio, &Font12, BLACK, WHITE);

// bloc 3 : temps d'activite (uptime)

uint16_t y3 = y2 + hauteur_box + espace_y;

dessiner_rectangle_arrondi_creux(x1_carre, y3, x2_carre, y3 + hauteur_box, rayon_bord,
↪ epaisseur_bord, COLOR_BOX_BLUE);
dessiner_icone_40x40(x1_carre - 2, y3 - 2, Uptime, WHITE);

dessiner_rectangle_arrondi_creux(x1_rect, y3, x2_rect, y3 + hauteur_box, rayon_bord,
↪ epaisseur_bord, COLOR_BOX_BLUE);

// on convertit les ms en h/m/s
uint32_t uptime_s = to_ms_since_boot(get_absolute_time()) / 1000;
uint32_t h = uptime_s / 3600;
uint32_t m = (uptime_s % 3600) / 60;
uint32_t s = uptime_s % 60;

char str_h[10], str_m[10], str_s[10];
sprintf(str_h, "%02lu", h);
sprintf(str_m, "%02lu", m);
sprintf(str_s, "%02lu", s);

// calcul pour tout centrer proprement
uint16_t w_h = strlen(str_h) * 14;
uint16_t w_m = 2 * 14;
uint16_t w_s = 2 * 14;
uint16_t w_lettre = 7;
uint16_t w_espace = 6;

uint16_t largeur_totale = w_h + w_lettre + w_espace + w_m + w_lettre + w_espace + w_s
↪ + w_lettre;

uint16_t rect_width = x2_rect - x1_rect;
uint16_t start_x_up = x1_rect + (rect_width - largeur_totale) / 2;

uint16_t y_font20 = (y3 + (hauteur_box - 20) / 2) + 2;
uint16_t y_font12 = y_font20 + 8;

// affichage Heures
Paint_DrawString_EN(start_x_up, y_font20, str_h, &Font20, BLACK, WHITE);
start_x_up += w_h;
Paint_DrawString_EN(start_x_up, y_font12, "h", &Font12, BLACK, WHITE);
start_x_up += w_lettre + w_espace;

// affichage Minutes
Paint_DrawString_EN(start_x_up, y_font20, str_m, &Font20, BLACK, WHITE);
start_x_up += w_m;
Paint_DrawString_EN(start_x_up, y_font12, "m", &Font12, BLACK, WHITE);
start_x_up += w_lettre + w_espace;

// affichage Secondes
Paint_DrawString_EN(start_x_up, y_font20, str_s, &Font20, BLACK, WHITE);
start_x_up += w_s;
Paint_DrawString_EN(start_x_up, y_font12, "s", &Font12, BLACK, WHITE);

// bloc 4 : mesure de l'entropie

uint16_t y4 = y3 + hauteur_box + espace_y;

dessiner_rectangle_arrondi_creux(x1_carre, y4, x2_carre, y4 + hauteur_box, rayon_bord,
↪ epaisseur_bord, COLOR_BOX_PURPLE);

```

```

dessiner_icone_40x40(x1_carre - 2, y4 - 2, Health, WHITE);

dessiner_rectangle_arrondi_creux(x1_rect, y4, x2_rect, y4 + hauteur_box, rayon_bord,
↪ epaisseur_bord, COLOR_BOX_PURPLE);

char texte_entropie[16];
sprintf(texte_entropie, "%.5f", ui_entropie);

// centrage du texte
uint16_t rect_width2 = x2_rect - x1_rect;
uint16_t text_width2 = strlen(texte_entropie) * 14;

uint16_t start_x2 = x1_rect + (rect_width2 - text_width2) / 2;
uint16_t start_y2 = (y4 + (hauteur_box - 20) / 2)+2;

Paint_DrawString_EN(start_x2, start_y2, texte_entropie, &Font20, BLACK, WHITE);

// icone des reglages en bas
dessiner_icone_40x40(190, 235, Settings, WHITE);
}

void dessiner_page_reglages(UWORD *BlackImage) {
    dessiner_entete_settings(240, 280);

    Paint_DrawString_EN(10, 80, "Compression Ratio:", &Font16, BLACK, WHITE);

    // selection du ratio avec 4 boutons
    uint16_t x_coords[] = {20, 75, 130, 185};
    uint16_t ratios[] = {2, 4, 8, 16};
    uint16_t y_reg = 110;
    uint16_t w_reg = 45;
    uint16_t h_reg = 40;

    for (int i = 0; i < 4; i++) {
        bool is_active = (trng_ratio == ratios[i]);
        char text[3];
        sprintf(text, "%d", ratios[i]);

        uint16_t offset_x = (ratios[i] < 10) ? 17 : 11;

        if (is_active) {
            // bouton actif : fond blanc, texte noir
            dessiner_rectangle_arrondi_creux(x_coords[i], y_reg, x_coords[i] + w_reg, y_reg +
↪ h_reg, 5, WHITE);
            Paint_DrawString_EN(x_coords[i] + offset_x, y_reg + 12, text, &Font16, WHITE,
↪ BLACK);
        } else {
            // bouton inactif : contour gris
            dessiner_rectangle_arrondi_creux(x_coords[i], y_reg, x_coords[i] + w_reg, y_reg
↪ + h_reg, 5, 2, DARK_GRAY);
            Paint_DrawString_EN(x_coords[i] + offset_x, y_reg + 12, text, &Font16, BLACK,
↪ WHITE);
        }
    }

    uint16_t y_btn = 175;
    uint16_t h_btn = 35;

    // bouton reset
    dessiner_rectangle_arrondi_creux(20, y_btn, 115, y_btn + h_btn, 5, 2, COLOR_BOX_PINK);
    Paint_DrawString_EN(40, y_btn + 10, "RESET", &Font16, BLACK, WHITE);

    // bouton info
    dessiner_rectangle_arrondi_creux(125, y_btn, 220, y_btn + h_btn, 5, 2, COLOR_BOX_BLUE);
}

```

```

    Paint_DrawString_EN(145, y_btn + 10, "ABOUT", &Font16, BLACK, WHITE);

    // icone de retour
    dessiner_icone_40x40(15, 235, Arrow, WHITE);

    // gestion du mode nuit/veille
    if (always_on) {
        dessiner_icone_40x40(190, 235, Lunebarre, WHITE);
    } else {
        dessiner_icone_40x40(190, 235, Lune, WHITE);
    }
}

void dessiner_page_about(UWORD *BlackImage) {
    dessiner_fond_grille(240, 280);

    Paint_DrawString_EN(70, 20, "ABOUT ME", &Font20, BLACK, WHITE);
    Paint_DrawString_EN(45, 55, "Firmware: v1.0", &Font16, BLACK, WHITE);
    Paint_DrawString_EN(45, 75, "0xEntropy.xyz", &Font16, BLACK, WHITE);

    dessiner_qrcode_100x100(70, 110, Qrcode);

    Paint_DrawString_EN(40, 250, "Tap anywhere to return", &Font12, BLACK, WHITE);
}

void dessiner_page_fatal(UWORD *BlackImage) {
    // ecran de la mort
    Paint_Clear(RED);

    Paint_DrawString_EN(10, 15, "!!! FATAL !!!", &Font24, RED, WHITE);

    Paint_DrawLine(20, 40, 220, 40, WHITE, DOT_PIXEL_1X1, LINE_STYLE_SOLID);

    Paint_DrawString_EN(8, 60, "TRNG SENSOR DEAD", &Font20, RED, WHITE);

    dessiner_qrcode_100x100(70, 100, Qrcodefatal);

    Paint_DrawString_EN(21, 230, "USB OUTPUT BLOCKED", &Font16, RED, WHITE);

    Paint_DrawString_EN(48, 260, "Press 1s to stop sound", &Font12, RED, WHITE);
}

```

B. Fichier Screen/Screen_app.h

```

#ifndef SCREEN_APP_H
#define SCREEN_APP_H

#include <stdint.h>
#include "UI/Screen/Driver/GUI_Paint.h"

typedef enum {
    PAGE_DASHBOARD,
    PAGE_REGLAGES,
    PAGE_ABOUT
} PageSysteme;

void dessiner_logo_entropy(uint16_t largeur_ecran, uint16_t hauteur_ecran);
void dessiner_icone_fleche(uint16_t x_centre, uint16_t y_centre, uint16_t couleur);
void dessiner_icone_curseurs(uint16_t x_centre, uint16_t y_centre, uint16_t couleur,
    ↵ uint16_t couleur_fond);
void dessiner_page_dashboard(UWORD *BlackImage);

```

```

void dessiner_page_reglages(UWORD *BlackImage);
void dessiner_page_about(UWORD *BlackImage);
void dessiner_page_fatal(UWORD *BlackImage);

#endif``

\subsection*[C. Fichier \texttt{Screen/Screen\_tactile.c}]
\addcontentsline{toc}{subsection}{C. Fichier \texttt{Screen\_tactile.c}}
``c
#include "pico/stdlib.h"
#include <stdbool.h>

#include "Driver/DEV_Config.h"
#include "Driver/Touch_1in69.h"
#include "Stats/stats.h"
#include "Screen_app.h"
#include "Screen_tactile.h"

// Recup des variables definies ailleurs
extern PageSysteme page_actuelle;
extern bool demander_rafraichissement;
extern int trng_ratio;

static uint32_t dernier_toucher = 0;
static bool ecran_allume = true;

bool always_on = false;

// On part sur 15 secondes avant de couper l'ecran
#define DELAI_VEILLE_MS 15000

void gerer_tactile(void) {
    uint32_t maintenant = to_ms_since_boot(get_absolute_time());

    // Gestion de la mise en veille auto
    if (!always_on && ecran_allume && (maintenant - dernier_toucher >= DELAI_VEILLE_MS)) {
        DEV_SET_PWM(0);
        ecran_allume = false;
    }

    // On regarde si on detecte un contact sur la dalle
    if (DEV_Digital_Read(DEV_I2C_INT) == 0) {
        Touch_1IN69_XY struct_tc = Touch_1IN69_Get_Point();

        // Si les coordonnees sont valides (pas 0,0)
        if (struct_tc.x_point != 0 && struct_tc.y_point != 0) {

            dernier_toucher = maintenant;

            // Si l'ecran etait eteint, le premier clic ne sert qu'a le rallumer
            if (!ecran_allume) {
                DEV_SET_PWM(100);
                ecran_allume = true;
                sleep_ms(300); // Petit delai pour eviter un clic fantome au reveil
            }

            return;
        }

        // --- Logique de navigation entre les pages ---

        if (page_actuelle == PAGE_DASHBOARD) {
            // Zone du bouton reglages sur le dashboard
            if (struct_tc.x_point >= 130 && struct_tc.x_point <= 240 &&
                struct_tc.y_point >= 190 && struct_tc.y_point <= 280) {

```

```

        page_actuelle = PAGE_REGLAGES;
        demander_rafraichissement = true;
    }
}
else if (page_actuelle == PAGE_REGLAGES) {
    // Bouton retour en bas a gauche
    if (struct_tc.x_point >= 0 && struct_tc.x_point <= 90 &&
        struct_tc.y_point >= 240 && struct_tc.y_point <= 280) {
        page_actuelle = PAGE_DASHBOARD;
        demander_rafraichissement = true;
    }
    // Selection des differents ratios (ligne horizontale)
    else if (struct_tc.y_point >= 110 && struct_tc.y_point <= 150) {
        if (struct_tc.x_point >= 20 && struct_tc.x_point <= 65) {
            trng_ratio = 2; stats_reset(); demander_rafraichissement = true;
        } else if (struct_tc.x_point >= 75 && struct_tc.x_point <= 120) {
            trng_ratio = 4; stats_reset(); demander_rafraichissement = true;
        } else if (struct_tc.x_point >= 130 && struct_tc.x_point <= 175) {
            trng_ratio = 8; stats_reset(); demander_rafraichissement = true;
        } else if (struct_tc.x_point >= 185 && struct_tc.x_point <= 230) {
            trng_ratio = 16; stats_reset(); demander_rafraichissement = true;
        }
    }
    // Les boutons RESET et ABOUT au milieu
    else if (struct_tc.y_point >= 155 && struct_tc.y_point <= 225) {
        if (struct_tc.x_point <= 135) {
            stats_reset();
            page_actuelle = PAGE_DASHBOARD;
            demander_rafraichissement = true;
        }
        else if (struct_tc.x_point > 135) {
            page_actuelle = PAGE_ABOUT;
            demander_rafraichissement = true;
        }
    }
    // Switch pour le mode Always On (ecran toujours allume)
    else if (struct_tc.y_point >= 230 && struct_tc.y_point <= 280) {
        if (struct_tc.x_point >= 160) {
            always_on = !always_on;
            demander_rafraichissement = true;
        }
    }
}

}
else if (page_actuelle == PAGE_ABOUT) {
    // N'importe quel clic sur ABOUT ramene aux reglages
    page_actuelle = PAGE_REGLAGES;
    demander_rafraichissement = true;
}

// On attend un peu pour pas compter 50 clics d'un coup
sleep_ms(200);
}
}
}

```

D. Fichier Screen/Screen_tactile.h

```

#ifndef SCREEN_TACTILE_H
#define SCREEN_TACTILE_H

```

```
void gerer_tactile(void);  
#endif
```