ARCHITECTURE RISC 32 BITS · COMPILATEUR C · NOYAU AUTO-HÉBERGÉ

LA BIBLE DE C OPERATING SYSTEM

Du silicium FPGA au système d'exploitation complet
conçu de zéro pour Sipeed Tang Nano 20K



coslogo.img · Mode GFX 320x240 RGB332

COS-32 SoC

GW2AR-LV18QN88C8/I7

CPU COS-32 @ 54 MHz + Timer IRQ
8 Mo SDRAM · Cache BSRAM 4 Ko
HDMI 640x480 · GFX 256c + Audio
FAT32 SPI 27 MHz · AZERTY CH9350
cc / as auto-hébergés · USB 921k

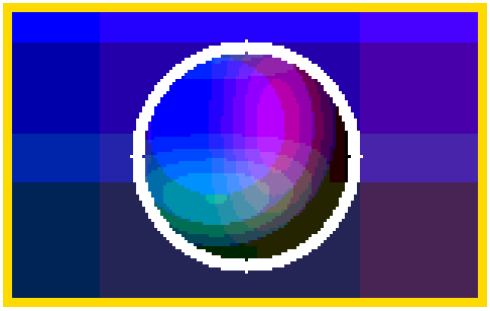
- **5 PARTIES · 19 CHAPITRES**
  Référence intégrale issue des 19 documents du projet
- **SYSTÈME & MATÉRIEL**
  ISA COS-32, MMIO, RTL Verilog, compilateur C & noyau
- **GUIDES & DÉPLOIEMENT**
  Installation Linux clé en main & guide développeur

SIPEED TANG NANO 20K · ÉDITION DE RÉFÉRENCE COMPLÈTE

COS v0.1 · 2026

La Bible de C Operating System

Ouvrage de référence complet compilant l'ensemble de la documentation technique, des guides utilisateur/développeur et des manuels en ligne du projet COS sur Sipeed Tang Nano 20K.



Logo officiel de COS (media/coslogo.img, 160×100 pixels RGB332).

À propos de cet ouvrage

Ce livre rassemble de façon structurée les 19 fichiers Markdown officiels du dépôt COS : présentation générale et bilan FPGA, guides d'installation pour débutant Linux et développeur, manuels complets du shell et des 28 utilitaires, spécification du jeu d'instructions COS-32, carte mémoire SDRAM 8 Mo, registres MMIO, brochage matériel et jalons de conception.

> Convention des liens internes : dans la version PDF, chaque entrée du Sommaire ainsi que les références croisées vers d'autres chapitres sont directement cliquables.

Fiche technique synthétique

Composant	Spécification de référence
Système	COS (C Operating System) v0.1 — auto-hébergé (C + Verilog)
Cible matérielle	Sipeed Tang Nano 20K (FPGA Gowin GW2AR-LV18QN88C8/I7)
Processeur	COS-32 RISC 32 bits @ 54 MHz (16 registres, MUL/DIV matériel, IRQ 50 ms)
Mémoires	Boot ROM BSRAM 64 Ko (65 525 o) · SDRAM interne 8 Mo · Cache L1 4 Ko
Noyau étendu SD	/boot/kernel.bin (140 288 o / 208 Ko max) · FAT32 8.3 & lecture VFAT LFN
Vidéo & Audio	HDMI 640×480 @ 60 Hz (Texte 80×30 / GFX 320×240 RGB332) + APU 4 canaux 32 kHz
Périphériques	Clavier USB AZERTY CH9350L (broche 76) · SD SPI 27 MHz · USB-UART 921 600 bauds
Contenu de ce livre	5 parties · 19 chapitres (intégralité des 19 fichiers Markdown du projet)

Sommaire

PARTIE I — Vision, Objectifs & Vue d'ensemble 4

01. **Présentation complète, objectifs et bilan FPGA** (PUBLICATION.md) 5

02. **Vue d'ensemble du dépôt et démarrage rapide** (README.md) 12

03. **Manuel général du système COS** (docs/man/cos.md) 16

PARTIE II — Installation & Développement 18

04. **Guide d'installation Débutant sous Linux** (docs/GUIDE_DEBUTANT_LINUX.md) 19

05. **Guide Développeur & Contribution** (docs/GUIDE_DEVELOPPEUR.md) 23

PARTIE III — Manuel Utilisateur, Shell & Outils 26

06. **Le Shell COS et l'environnement interactif** (docs/man/shell.md) 27

07. **Pipes RAM, redirections, jokers et scripts sh** (docs/man/sh.md) 29

08. **Commandes système et utilitaires /bin** (docs/man/fs.md) 31

09. **Éditeur plein écran ed avec coloration C** (docs/man/ed.md) 32

10. **Mode graphique HDMI 320×240 et images CIMG** (docs/man/gfx.md) 33

11. **Audio HDMI 4 canaux et lecteur WAV** (docs/man/audio.md) 34

12. **Compilateur Mini-C auto-hébergé (cc)** (docs/man/cc.md) 35

13. **Assembleur et éditeur de liens COS-32 (as)** (docs/man/as.md) 36

PARTIE IV — Architecture Matérielle, ISA & Noyau 37

14. **Spécification du jeu d'instructions COS-32** (docs/isa.md) 38

15. **Carte mémoire, registres MMIO, format .cos et ABI** (docs/memmap_abi.md) 44

16. **Brochage Tang Nano 20K et chaîne FPGA** (docs/pinout.md) 50

17. **Périphériques du SoC et API C <cos.h>** (docs/man/hw.md) 52

18. **Noyau C, Boot ROM et /boot/kernel.bin** (docs/man/kernel.md) 53

PARTIE V — Feuille de route & Jalons 54

19. **Plan d'implémentation et jalons du projet** (PLAN.md) 55

PARTIE I

Vision, Objectifs & Vue d'ensemble

Architecture globale, philosophie from scratch, bilan matériel FPGA et présentation générale de COS.

Chapitre 01 — Présentation complète, objectifs et bilan FPGA
Document de référence pour la publication, comparatif des éditions et feuille de route matérielle [PUBLICATION.md] · p. 5

Chapitre 02 — Vue d'ensemble du dépôt et démarrage rapide
Capacités clés du système, arborescence des sources et lancement dans l'émulateur [README.md] · p. 12

Chapitre 03 — Manuel général du système COS
Carte mémoire synthétique, organisation de la carte SD FAT32 et index des manuels [docs/man/cos.md] · p. 16

CH. 01 | Présentation complète, objectifs et bilan FPGA

Document de référence pour la publication, comparatif des éditions et feuille de route matérielle · Source : PUBLICATION.md

COS (*C Operating System*) est un système d'exploitation complet, accompagné de son propre microprocesseur RISC 32 bits (**COS-32**), de son propre compilateur C, de son assembleur/éditeur de liens et de son jeu d'utilitaires système — le tout conçu *from scratch* en Verilog et en C pour la carte FPGA **Sipeed Tang Nano 20K** (`Gowin GW2AR-LV18QN88C8/I7`).

L'objectif du projet est d'offrir une machine autonome et éducative sur une simple carte à ~30 € : branchez un écran **HDMI**, une carte **microSD**, un clavier USB **AZERTY** (via un module CH9350) et vous pouvez éditer, compiler, lier et exécuter des programmes C, écrire des scripts shell, afficher des graphismes 256 couleurs et lire de la musique sur HDMI — **directement sur le FPGA, sans PC**, tout en gardant un déploiement ultra-simple depuis Linux par un unique câble USB-C.

---

1. Objectifs du projet

1. **Partir réellement de zéro (*from scratch*)** : aucun cœur RISC-V/ARM tiers ni libc externe ; le CPU (`cos32_cpu`), l'architecture d'instructions (ISA COS-32 optimisée pour le C compilé), le compilateur (`cc`), l'assembleur (`as`), la bibliothèque C standard (`libc`) et le noyau sont entièrement documentés et lisibles.
2. **Atteindre l'auto-hébergement complet** : le compilateur `cc` et l'assembleur `as` sont eux-mêmes compilés pour COS (`/bin/cc` et `/bin/as`), permettant d'écrire un fichier `.c` avec l'éditeur plein écran `ed` sur HDMI, de le compiler sur le Tang Nano 20K et même de recompiler le noyau étendu `/boot/kernel.bin` sur la carte.
3. **Rester compact en Flash/BRAM, extensif sur SD** :
4. La **Boot ROM** interne (`64 Ko` en BSRAM FPGA) reste minimale (`65 525 / 65 536 octets`) pour stabiliser définitivement le bitstream matériel.
5. Au démarrage, si `/boot/kernel.bin` existe sur la carte microSD, la Boot ROM charge automatiquement le **noyau étendu SD** (`140 288 octets` actuellement pour une fenêtre de `208 Ko`) en SDRAM. Une mise à jour de l'OS ne demande donc même plus de reflasher le FPGA !
6. **Expérience Unix-like réactive avec matériel accessible** : shell avec pipelines en RAM, scripts structurés (`if`/`for`/`while`), éditeur C colorisé, affichage HDMI texte/graphique, synthèse sonore + lecture WAV sur HDMI, et transfert PC→SD à `921 600` bauds via un seul câble USB.

---

2. Capacités et architecture actuelles

Logiciel & système d'exploitation

- **28 commandes `/bin/*` préinstallées** :

- Système de fichiers & disque : `ls`, `cd`, `md`, `rd`, `cp`, `rm`, `find`, `df`, `free`, `format` (formatage FAT32 directement sur le FPGA).
- Filtres & texte Unix : `cat`, `more`, `grep`, `head`, `tail`, `wc`, `sort`, `uniq`, `hexdump`, `clear`, `sleep`, `uptime`, `echo`, `man`.
- Édition & développement : `ed` (éditeur plein écran 80×30 avec coloration syntaxique C, recherche `^F`/`^N`/`^P` surlignée, pagination `^B`/`^V`), `cc`, `as`, `/lib/libc.lib`, en-têtes `/include/*.h`, code source complet de l'OS dans `/src`.
- Multimédia : `view` (images `.img` CIMG avec RLE et transparence) et `play` (lecture WAV au premier plan ou en tâche de fond via DMA RAM / streaming SD continu).
- **Shell étendu (`/boot/kernel.bin`)** :
- Redirections (`>`, `>>`, `<`) et pipelines `|` sur double tampon SDRAM `2 × 256 Ko` (zéro usure SD).
- Enchaînements `;`, `&&`, `||`, jokers `*` et `?`, variables d'environnement (`VAR=val`, `set`, `unset`, `$?`, `PATH`), arithmétique `$((expr))`.
- Scripts `.sh` (et démarrage automatique `/boot/init.sh`) avec `if`/`elif`/`else`/`fi`, `for`/`in`/`do`/`done`, `while`/`do`/`done`, `break`/`continue`, `test`/`[ ... ]`.
- Complétion `Tab`, historique persistant (`/history`, flèches Haut/Bas).
- Préemption matérielle par **IRQ Timer 50 ms** : `Ctrl-C` interrompt tout programme (même une boucle CPU pure sans appel système) et `Ctrl-X` / `F12` coupe immédiatement l'audio.

Architecture mémoire et matérielle

- **CPU COS-32 @ 54 MHz** : RISC 32 bits, 16 registres généraux (`fp=r13`, `sp=r14`, `lr=r15`), multiplicateur et diviseur matériels, interruptions matérielles (`EI`/`DI`/`RETI`, vecteur `0x0000_000C`).
- **SDRAM interne 8 Mo (`0x0010_0000 – 0x008F_FFFF`)** avec cache direct-mapped 4 Ko et arbitre équitable 3 ports (CPU / Vidéo DMA / Audio DMA) :
- `0x0010_0000 – 0x0010_7FFF` : table vecteurs RAM & données Boot ROM.
- `0x0010_8000 – 0x0013_BFFF` : noyau étendu `/boot/kernel.bin` (**208 Ko max**, actuellement 140 288 o).
- `0x0013_C000 – 0x0013_FFF0` : pile IRQ (4 Ko) et pile noyau (12 Ko).
- `0x0014_0000 – 0x0073_FFF0` : mémoire programmes utilisateur (**6 Mo** pour code, `malloc`/`sbrk` et pile).
- `0x0074_0000 – 0x007B_FFFF` : ramdisk des pipes (`PIPE_BUF0` / `PIPE_BUF1`, 2 × 256 Ko).
- `0x007C_0000 – 0x008B_FFFF` : anneau DMA audio (1 Mo).
- `0x008C_0000 – 0x008F_FFFF` : double framebuffer graphique 320×240 RGB332.
- **Vidéo & Audio HDMI 640×480 @ 60 Hz** : mode texte 80×30 (16 couleurs, police CP437 avec accents français) ou mode graphique 320×240 256 couleurs sans déchirement (*tear-free* sur VSYNC), plus APU 4 canaux 32 kHz (3 canaux PSG + 1 canal PCM 16 bits DMA) encapsulé dans les Data Islands HDMI.
- **Clavier & USB-UART** : module clavier USB AZERTY CH9350L sur la broche 76 + passerelle USB-UART BL616 (`115 200` → `921 600` bauds lors des transferts `sync`, FIFO RX matériel 2 Ko en BSRAM).

3. Occupation exacte du FPGA (`GW2AR-LV18QN88C8/I7`)

Voici le relevé exact issu de la dernière synthèse Gowin (`yosys` + `nextpnr-himbaechel` + `gowin_pack`) :

Ressource FPGA	Utilisé / Total	Taux d'occupation	Disponible restant
LUT4 (logique)	`9 867 / 20 736`	`47 %`	**`10 869 LUT4` libres (53 %)**
DFF (bascules)	`3 617 / 15 552`	`23 %`	**`11 935 DFF` libres (77 %)**
ALU (chaînes de retenue)	`1 872 / 15 552`	`12 %`	**`13 680 ALU` libres (88 %)**
MUX2_LUT5	`706 / 10 368`	`6 %`	`9 662` libres
RAM16SDP4 (RAM distribuée en LUT)	`72 / 648`	`11 %`	**`576 blocs` libres (89 %)**
BSRAM (Block RAM interne)	**`45 / 46`**	**`97 %`**	**`1 bloc` libre (point critique)**
DSP `MULT36X36`	`1 / 12`	`8 %`	**`11` multiplieurs 36×36 libres**
DSP `MULT9X9`	`3 / 96`	`3 %`	**`93` blocs libres**
DSP `MULT18X18` / `ALU54D`	`0 / 48` · `0 / 24`	`0 %`	**100 % libres**
IOB (broches E/S utilisées)	`47 / 66`	`71 %`	**`19 broches` libres**

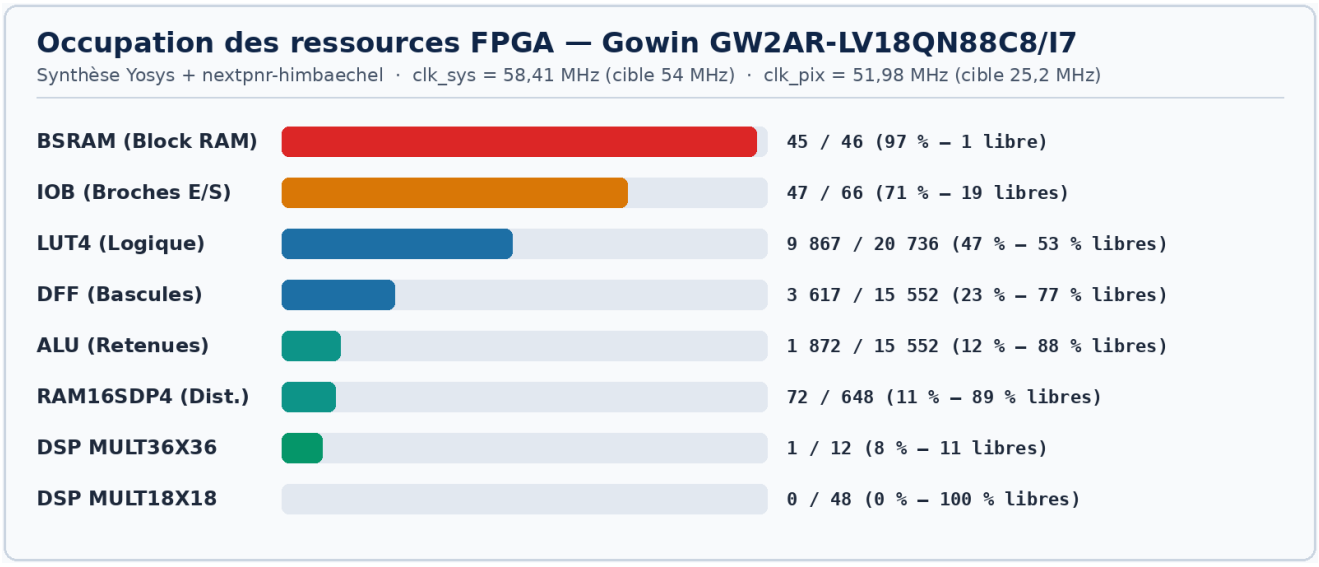


Figure 1 — Bilan d'occupation des ressources sur Gowin GW2AR-LV18QN88C8/I7 après routage.

Fréquences maximales atteintes (clôture temporelle vérifiée) :

- `clk_sys` (CPU, SDRAM, bus) : **58,41 MHz** atteints *(cible 54,00 MHz — PASS)*
- `clk_pix` (pixel HDMI 640×480) : **51,98 MHz** atteints *(cible 25,20 MHz — PASS)*

> **Point clé d'architecture** : il reste plus de la moitié de la logique (**10 869 LUT4**), **89 %** des registres/RAM distribuée (**576 `RAM16SDP4`**), plus de **90 %** des blocs DSP et **19 GPIO**, mais une seule **BSRAM** disponible (`45/46`). Toute

> extension matérielle future doit soit consommer **0 BSRAM** (logique/DSP/`RAM16SDP4`/SDRAM), soit libérer de la BSRAM (par ex. en réduisant la Boot ROM).

---

4. Les deux éditions téléchargeables

Pour répondre aux besoins de chacun, deux archives ZIP autonomes sont générées par `make release` :

Caractéristique	Édition Débutant Linux (cos-tangnano20k-beginner-linux.zip)	Édition Confirmé / Développeur (cos-tangnano20k-dev-source.zip)
Public cible	Découverte & utilisation immédiate sur matériel neuf	Modification de l'OS, du compilateur, des outils ou du RTL Verilog
Contenu	Bitstream FPGA prêt à flasher (`cos.fs`), arborescence complète `sdcard/` (noyau SD, 28 binaires, `/src`, `/docs`, `/media`), script `./install.sh`, outils USB et `img2cos.py`	Code source complet (`rtl/`, `kernel/`, `libc/`, `toolchain/`, `bin/`, `examples/`, `tests/`, `tools/`, `docs/`), `Makefile` et guides
Prérequis PC Linux	`python3` + `openFPGALoader`	`gcc`, `python3`, `mtools`, **OSS CAD Suite** (`yosys`, `nextpnr-himbaechel`, `gowin_pack`, `iverilog`, `openFPGALoader`)
Tang neuf + SD vierge	**1 seule commande `./install.sh`** par câble USB-C	`make deploy` ou `make sync`

Édition 1 : Utilisateur débutant sous Linux (`cos-tangnano20k-beginner-linux.zip`)

Prévue pour une carte **Tang Nano 20K neuve** équipée d'une **carte microSD vierge** (même sans lecteur de carte SD sur le PC) : tout se fait en **un seul câble USB-C** entre le PC Linux et le port USB-C du Tang Nano 20K.

Ce que fait automatiquement l'installateur `./install.sh`

1. **Vérifie votre PC Linux** (`python3`, `openFPGALoader`, détection du port `/dev/ttyUSB*` et des permissions utilisateur).
2. **Flashe le FPGA en Flash SPI persistante** avec le processeur COS-32 et la Boot ROM (`openFPGALoader -b tangnano20k -f bitstream/cos.fs`).
3. **Formate automatiquement la carte microSD vierge en FAT32** si elle ne contient pas encore de partition FAT32 valide (aucun adaptateur SD nécessaire sur le PC !).
4. **Accélère automatiquement le transfert (amorçage en 2 étapes)** : envoie d'abord `/boot/kernel.bin` à `115 200` bauds, active immédiatement le noyau SD pour basculer le port USB-UART à **`921 600` bauds**, puis transfère les **106 fichiers** du système (`/bin/*`, `/lib/*`, `/include/*`, `/docs/*`, `/src/*`, `/media/*`, `/boot/init.sh`) avec vérification CRC-16.
5. **Démarre COS** : exécute `/boot/init.sh` et vous place devant l'invite `cos:/$`.

Procédure rapide sous Linux

```
# 1. Installer les deux prérequis (exemple Debian/Ubuntu)
sudo apt update && sudo apt install -y python3 openfpgaloader

# 2. Autoriser l'accès USB sans sudo (déconnectez/reconnectez votre session après la commande)
sudo usermod -aG dialout,plugdev "$USER"

# 3. Insérer la microSD dans le Tang Nano 20K, brancher le câble USB-C au PC, puis lancer :
unzip cos-tangnano20k-beginner-linux.zip
cd cos-tangnano20k-beginner-linux
./install.sh
```

Options utiles : `./install.sh --format` (forcer un reformatage FAT32), `./install.sh --sync-only` (mettre à jour la SD sans reflasher le FPGA), `./install.sh --port /dev/ttyUSB1`. Voir le guide détaillé dans `docs/GUIDE_DEBUTANT_LINUX.md` (`README_INSTALLATION.md` dans le ZIP).

Édition 2 : Utilisateur confirmé / Développeur (`cos-tangnano20k-dev-source.zip`)

Contient l'intégralité des sources pour modifier l'architecture matérielle (Verilog), le jeu d'instructions, le compilateur C, les pilotes noyau ou les applications utilisateur.

Ce que contient l'archive

- `rtl/` : cœur `cos32_cpu.v`, cache SDRAM (`sdram_cache.v`, `sdram_ctrl.v`), contrôleur HDMI texte/graphique (`hdmi_text.v`, `hdmi_audio_tmds.v`), APU 4 canaux (`audio_apu.v`), SPI SD (`sd_spi.v`), Timer IRQ (`timer.v`), UART FIFO (`uart_debug.v`, `uart_rx_fifo.v`).
- `toolchain/` & `tools/` : compilateur mini-C (`cc.c`), assembleur/linker 2 passes (`as.c`), émulateur complet avec contrôleur SD/UART (`emu.c`), scripts USB/image/release.
- `kernel/`, `libc/`, `bin/`, `examples/`, `docs/` : OS complet, bibliothèque C, 28 utilitaires et exemples.
- `tests/` : **5 suites de tests automatisées** (`asm`, `cc`, `rtl` avec `iverilog`, `kernel`, `bin`).

Commandes principales

```
unzip cos-tangnano20k-dev-source.zip
cd cos-tangnano20k-dev-source

make all sd-kernel   # Compile outils PC, libc, Boot ROM (64 Ko), noyau SD et binaires /bin
make test            # Exécute les 5 suites de non-régression (CPU, C, RTL Icarus, Kernel, Binaires)
make synth           # Synthèse Yosys + P&R nextpnr-himbaechel + bitstream build/cos.fs
make deploy          # Flashe le FPGA + formate/déploie la carte SD par USB (FAST=1 pour incrémental)
make sync            # Met à jour uniquement les fichiers modifiés sur SD à 921 600 bauds (0 reflash)
make release         # Régénère les deux archives ZIP de publication dans dist/
```

Voir le guide technique complet dans `docs/GUIDE_DEVELOPPEUR.md` (`README_DEVELOPPEUR.md` dans le ZIP).

---

5. Suggestions d'améliorations et d'extensions (adaptées aux ressources FPGA restantes)

Les pistes ci-dessous tiennent directement compte du bilan matériel réel (**45/46 BSRAM**, mais **10 869 LUT4**, **11 935 DFF**, **576 RAM16SDP4**, **>90 % DSP** et **19 GPIO** encore disponibles).

A. Extensions purement logicielles (0 BSRAM, 0 LUT)

Le noyau SD dispose encore d'environ **72 Ko** libres dans sa fenêtre (`140 Ko / 208 Ko`) et les programmes utilisateur ont **6 Mo de SDRAM** :

1. **Multitâche coopératif / préemptif léger par Timer IRQ** : le CPU dispose déjà de l'IRQ Timer 50 ms (`epc`, `RETI`). Ajouter une table de 4 à 8 contextes (`r1..r15`, `pc`, pile dédiée en SDRAM) permettrait des commandes `jobs`, `&` en arrière-plan ou `watch` sur l'état du système.
2. **Enregistrement LFN en écriture (FAT32)** : `fat32.c` lit déjà les noms longs VFAT dans le noyau SD ; étendre `fat32_open`/`fat32_mkdir` pour écrire des entrées LFN permettrait de créer des fichiers avec noms longs/minuscules directement depuis COS.
3. **Nouveaux utilitaires & mini-interpréteurs auto-hébergés** :
4. Outils système : `diff`, `tar` (archivage sur SD), `watch`, transfert fichier XMODEM/YMODEM directement depuis le shell.
5. Langages embarqués sur la carte : portage d'un interpréteur **Forth**, **TinyBASIC** ou **Lisp**, ou compilation sur COS de jeux 2D 320×240 avec bande-son WAV DMA.
6. **Améliorations de `cc` et `ed`** : ajout du type `long long` ou virgule fixe 16.16 dans `cc`, sélection/copier-coller (`^K`/`^Y`) et onglets multiples dans `ed`.

B. Extensions matérielles RTL à 0 BSRAM (exploitant les `10 869 LUT4`, `576 RAM16SDP4`, `48 MULT18X18` et `19 GPIO` libres)

1. **Blitter / accélérateur DMA 2D** (~400–800 LUT4, 0 BSRAM) : un petit automate branché en 4e port de l'arbitre SDRAM pourrait copier des rectangles avec transparence (`blit`), remplir des zones (`fillrect`) et tracer des lignes à la vitesse SDRAM au lieu d'une boucle CPU — idéal pour animer des sprites à 60 Hz en 320×240.
2. **Co-processeur arithmétique / virgule fixe Q16.16 & Q15 audio** (~300 LUT4 + `MULT18X18` / `ALU54D`, 0 BSRAM) : les **48 multiplieurs** `MULT18X18` et **24** `ALU54D` sont aujourd'hui inutilisés à `100 %` ! Un périphérique MMIO ou de nouvelles instructions ISA offriraient `mul_q16`, `sqrt`, `sin`/`cos` (table CORDIC pure logique) et du mixage/filtrage audio temps réel pour la démo graphique 3D wireframe ou le rendu fractal.
3. **Exploitation des 19 broches E/S et composants de la carte Tang Nano 20K** (~150–400 LUT4, 0 BSRAM) :
4. Pilote matériel **LED RGB WS2812** sur la broche 79 de la carte (compteur précis en cycles 54 MHz).
5. Contrôleur **I2C / SPI maître** sur les broches libres : lecture de date/heure réelle avec une **RTC DS3231** (pour horodater les fichiers FAT32), capteurs, ou connexion d'une **manette NES/SNES** (3 broches seulement).
6. **Réseau / transfert sans fil via module ESP-AT UART** ou **Ethernet SPI (W5500)** sur port PMOD.
7. **Optimisation de performance CPU via `RAM16SDP4`** (~600 LUT4 + ~128 `RAM16SDP4`, 0 BSRAM) : les **576 blocs** `RAM16SDP4` disponibles permettent d'ajouter un petit **cache d'instructions séparé (L1-I)** ou une table de

prédiction de branchement/pile de retour pour réduire l'arbitrage SDRAM sans toucher à aucune BSRAM.

C. Extensions nécessitant de libérer de la BSRAM (stratégie Boot ROM 64 Ko → 32 Ko)

Aujourd'hui, `boot_rom.v` occupe 16 384 mots de 32 bits (`64 Ko`, soit ~32 BSRAM). Plus de **30 %** de ce code vient du formatage FAT32/éditeur de boot en ROM. En déplaçant certaines commandes non critiques vers `/boot/kernel.bin` pour ramener la Boot ROM sous `32 Ko` (`boot_rom` 8K mots), on récupère d'un coup **~16 BSRAM libres**, ouvrant la porte à :

- Un **cache SDRAM direct-mapped 8 Ko ou 16 Ko** (ou 2 voies) augmentant nettement la vitesse d'exécution de `cc`/`as`.
- Un **mode texte couleur RGB étendu ou sprites matériels en BRAM** dans `hdmi_text.v` (64 sprites 16×16 superposés à l'écran HDMI sans bande passante SDRAM).
- Un **double-tampon DMA secteur SD matériel** permettant des lectures/écritures multi-blocs continues à plein débit SPI.

CH. 02 | Vue d'ensemble du dépôt et démarrage rapide

Capacités clés du système, arborescence des sources et lancement dans l'émulateur · Source : README.md

COS est un système d'exploitation 32 bits complet, écrit de zéro (*from scratch*) en C et en Verilog pour la carte FPGA **Sipeed Tang Nano 20K** (`Gowin GW2AR-LV18QN88C8/I7`). Il inclut son propre processeur RISC 32 bits (**COS-32**), une chaîne complète de compilation C/assembleur auto-hébergée, un éditeur de texte, un shell Unix-like avec scripts, un mode graphique 256 couleurs double-bufferisé et de l'audio numérique sur HDMI.

Vue d'ensemble : ce que l'on peut faire avec COS

1. Travailler directement sur la carte (auto-hébergement complet)

- **Éditer du code sur HDMI 80×30** avec l'éditeur plein écran `ed` : coloration syntaxique C, recherche (`^F` / `^N` / `^P` avec surlignage), pagination (`^B` / `^V`, `PgUp` / `PgDn`), aller à la ligne (`^G`), clavier USB AZERTY (module CH9350L) ou terminal UART.
- **Compiler et exécuter des programmes C directement sur le FPGA** sans PC : `text  cos:/src/examples$ cc hello.c  cos:/src/examples$ hello` `cc` compile un sous-ensemble complet du C (pointeurs, structures, tableaux, préprocesseur) vers l'assembleur COS-32 et appelle automatiquement `as` lié à la bibliothèque standard `/lib/libc.lib`.
- **Recompiler le noyau lui-même depuis COS** : le noyau C complet (`/src/kernel`) se recompile et s'installe en `/boot/kernel.bin` sans reflasher le bitstream FPGA.

2. Exploiter un shell productif (`sh`, pipes, redirections, historique)

- **Historique persistant & complétion** : rappel des 16 dernières commandes distinctes avec les flèches `Haut`/`Bas` (sauvegardées dans `/history`) et complétion intelligente par `Tab` sur les commandes et chemins.
- **Enchaînement, jokers, structures et pipelines RAM** : opérateurs `;`, `&&`, `||`, jokers `*`/`?`, blocs `if`/`elif`/`else`/`fi`, boucles `for`/`while`, `test`/`[`, arithmétique `$((...))`, variables d'environnement (`VAR=val`, `set`, `unset`, `$?`, `PATH`), redirections (`>`, `>>`, `<`) et pipelines sur RAM (`2 × 256 Ko` en SDRAM) sans usure SD.
- **Raccourcis globaux (IRQ Timer 50 ms)** : `Ctrl-C` interrompt tout programme/script (même une boucle CPU pure) et `Ctrl-X` / `F12` coupe immédiatement le son.
- **Utilitaires système & filtres Unix** : `ls`, `cat`, `more`, `grep`, `head`, `tail`, `wc`, `sort`, `uniq`, `find`, `hexdump`, `clear`, `sleep`, `uptime`, `cd`, `md`, `rd`, `cp`, `rm`, `df`, `free`, `format`, `man`, `echo`.

3. Afficher des graphismes 256 couleurs et des images (`gfx`, `view`)

- **Mode texte 80×30** : 16 couleurs CGA/VGA, accents français CP437, curseur matériel clignotant (`examples/video.c`).
- **Mode graphique 320×240 RGB332** : double-buffering matériel en SDRAM (`0x008C0000` / `0x008E0000`), commutation sans déchirement (*tear-free* sur VSYNC), primitives (`gfx_line`, `gfx_rect`, `gfx_circle`,

`gfx_fill_*`), police bitmap 8×8 (`gfx_puts`).

- **Visionneuse d'images** : `view /media/coslogo.img [x] [y]` affiche n'importe quelle image `.img` (brut ou compressée PackBits RLE avec transparence) à l'endroit choisi (`bin/view.c`).
- **Convertisseur PC fourni** : `tools/scripts/img2cos.py` convertit PNG/JPG/BMP/GIF en format COS (`CIMG`) avec tramage Floyd-Steinberg et compression `--rle`.

4. Générer des sons et lire des fichiers WAV sur HDMI (`audio`, `play`)

- **APU 4 canaux à 32 kHz via Data Islands HDMI 1.3a** :
- **3 canaux PSG** (carré, triangle, dent de scie, bruit LFSR) avec durée et décroissance matérielle (`examples/sound.c`).
- **1 canal PCM 16 bits + DMA SDRAM autonome** jouant l'audio en arrière-plan pendant une animation graphique (`examples/gfx.c`).
- **Lecteur WAV (premier plan, RAM DMA ou streaming SD longue durée)** : `play /media/demo.wav` lit un WAV PCM (8/16 bits, mono/stéréo) ; `play -b [-l] fichier.wav` joue en tâche de fond sans bloquer le shell (en **RAM DMA** jusqu'à 1 Mo, ou automatiquement en **streaming continu depuis la carte SD** sur anneau DMA au-delà de 1 Mo sans limite de durée ; `play -q` pour l'état, `play -s` pour arrêter).

5. Déployer et synchroniser en USB-UART à 921 600 bauds

- Sans sortir la carte microSD : tapez `sync` dans COS puis `make sync` sur PC. Le protocole négocie automatiquement **921 600 bauds** (FIFO RX matériel 2 Ko en BSRAM), vérifie les hash locaux ou CRC-16 sur SD, et ne transfère que les fichiers modifiés.

Architecture matérielle du SoC

Bloc	Spécification
CPU	COS-32 RISC 32 bits, 16 registres, **54 MHz** (`clk_sys`), multiplicateur/diviseur matériel
Mémoire ROM	**64 Ko** Boot ROM en BSRAM (`0x0000_0000 – 0x0000_FFFF`)
Mémoire SDRAM	**8 Mo** interne GW2AR-18 (`0x0010_0000 – 0x008F_FFFF`) avec **cache direct-mapped 4 Ko** et arbitre 3 ports (CPU / Vidéo DMA / Audio DMA)
Vidéo & Audio	HDMI 640×480 @ 60 Hz (`clk_pix = 25,2 MHz`) : Texte 80×30 ou Graphique 320×240 RGB332 + LPCM 32 kHz
Stockage	MicroSD SPI 27 MHz avec moteur bloc matériel 512 o + FAT32 (noms courts 8.3 & lecture LFN dans `/boot/kernel.bin`)
Entrées / Debug	Clavier USB via CH9350L (AZERTY, broche 76) + USB-UART BL616 (115 200 → 921 600 bauds, FIFO 2 Ko)

Pour les détails techniques complets :

- Spécification ISA COS-32

- Carte mémoire, registres MMIO & ABI
- Brochage Tang Nano 20K & synthèse FPGA
- Manuels en ligne (`docs/man/` & commande `man <sujet>` sous COS) : `cos`, `shell`, `sh`, `fs`, `ed`, `cc`, `as`, `gfx`, `audio`, `hw`, `kernel`.

Démarrage rapide (Build, Simulation, FPGA)

Prérequis sur PC

- `gcc`, `python3`, `mtools` (`/sbin/mkfs.fat`, `mcopy`, `mdir`), `Pillow` (`img2cos.py`). Note : `deploy_usb.py` utilise uniquement la bibliothèque standard Python (`termios`).
- Pour le FPGA et la simulation RTL : OSS CAD Suite (`yosys`, `nextpnr-himbaechel`, `gowin_pack`, `openFPGALoader`, `iverilog`) dans `~/oss-cad-suite/bin`.

Commandes principales

```
# 1. Compiler les outils PC (cos-cc, cos-as, cos-emu), la libc, le noyau Boot ROM,
#    le noyau étendu SD (/boot/kernel.bin) et tous les binaires /bin/*
make all sd-kernel

# 2. Exécuter les 5 suites de tests automatisées (asm, cc, rtl, kernel, bin)
make test

# 3. Synthétiser le bitstream FPGA (.fs)
make synth

# 4. Déployer bitstream + contenu complet sur la carte Tang Nano 20K via USB
make deploy

# 5. Mises à jour incrémentales rapides vers la SD (après `sync` dans COS)
make sync
```

Lancer dans l'émulateur sur PC (`cos-emu`)

```
truncate -s 34M /tmp/cos_sd.img
/sbin/mkfs.fat -F 32 -n COS /tmp/cos_sd.img
bash tools/scripts/deploy.sh /tmp/cos_sd.img
./build/cos-emu --rom build/kernel.bin --sd /tmp/cos_sd.img
```

Arborescence du dépôt

```
├── bin/          Utilitaires utilisateur (/bin/* : ls, cat, more, ed, view, play, man, ...)
├── docs/         Spécifications techniques et pages de manuel (/docs/*.md)
├── examples/     Programmes d'exemple en C et scripts shell (.sh)
├── kernel/       Noyau C (Boot ROM 64 Ko et noyau étendu SD -D SD_KERNEL)
├── libc/         Runtime startup (crt0.s) et bibliothèque C standard + APIs gfx/snd
├── media/        Ressources multimédias de démonstration (coslogo.img, demo.wav)
├── rtl/          Sources Verilog du SoC COS-32 (CPU, cache SDRAM, HDMI, APU, SD, UART)
├── tests/        Suites de tests non-régression (asm, cc, rtl, kernel, bin)
├── toolchain/    Compilateur mini-C (cc.c) et assembleur/linker 2 passes (as.c)
└── tools/        Émulateur PC (emu.c) et scripts (deploy_usb.py, img2cos.py, gen_font.py)
```

CH. 03 | Manuel général du système COS

Carte mémoire synthétique, organisation de la carte SD FAT32 et index des manuels · Source : docs/man/cos.md

COS est un systeme d'exploitation 32 bits auto-heberge concu from scratch pour la carte FPGA **Sipeed Tang Nano 20K** (`GW2AR-LV18QN88C8/I7`).

Architecture materielle & memoire

- **Processeur COS-32 @ 54 MHz** : 16 registres 32 bits (`r0`..`r15`), `fp=r13`, `sp=r14`, `lr=r15`.
- **Boot ROM (64 Ko)** : `0x00000000 - 0x0000FFFF` (noyau minimal en BSRAM).
- **SDRAM interne (8 Mo, cache BSRAM 4 Ko)** : `0x00100000 - 0x008FFFFF`
- `0x00100000 - 0x00107FFF` : vecteurs RAM + donnees Boot ROM.
- `0x00108000 - 0x0013BFFF` : noyau etendu `/boot/kernel.bin` (208 Ko max).
- `0x0013C000 - 0x0013FFF0` : pile noyau & pile IRQ (16 Ko).
- `0x00140000 - 0x0073FFF0` : espace programme utilisateur + tas (`sbrk`/`malloc`) + pile.
- `0x00740000 - 0x007BFFFF` : ramdisk de pipe SDRAM (`PIPE_BUF0` / `PIPE_BUF1`, 2 x 256 Ko).
- `0x007C0000 - 0x008BFFFF` : tampon circulaire DMA audio tache de fond (`AUDIO_BG_ADDR`, 1 Mo).
- `0x008C0000 - 0x008FFFFF` : 2 pages graphiques 320x240 RGB332 (`GFX_PAGE0` et `GFX_PAGE1`).

Arborescence de la carte SD (FAT32)

- `/bin` : commandes et outils natifs (`ls`, `cat`, `more`, `grep`, `head`, `tail`, `wc`, `sort`, `uniq`, `find`, `hexdump`, `clear`, `sleep`, `uptime`, `cd`, `md`, `rd`, `cp`, `rm`, `ed`, `df`, `free`, `format`, `man`, `view`, `play`, `cc`, `as`).
- `/boot/kernel.bin` : noyau C etendu (LFN, historique, Tab, pipes, redirections, `sh`, sync 921600 bauds).
- `/boot/init.sh` : script execute automatiquement au demarrage par le noyau SD.
- `/include` : en-tetes C standard + `<cos.h>` (`gfx_*`, `snd_*`, console, SD).
- `/lib` : runtime `crt0.s`, `libc.s` et bibliotheque binaire indexee `libc.lib`.
- `/docs` : pages d'aide Markdown consultables via `man`.
- `/media` : images `.img` et fichiers audio `.wav` de demonstration.
- `/src` : code source complet (`bin`, `libc`, `tools`, `kernel`, `examples` dont `demo.sh`).

Pages de manuel disponibles

Tapez `man <sujet>` pour lire :

- `shell` : invite, historique `/history`, completion `Tab`, resolution des binaires
- `sh` : pipes `|`, redirections `>`/`>>`/`<`, `cat`, `more`, `grep`, `head`, `tail`, `wc`, `echo`, scripts `.sh`

- `fs` : gestion fichiers/disque (`ls`, `cat`, `more`, `grep`, `head`, `tail`, `wc`, `cd`, `md`, `rd`, `cp`, `rm`, `df`, `free`, `format`)
- `cc` : compilateur mini-C et enchainement automatique `cc hello.c`
- `as` : assembleur COS-32 (`as prog.s` -> `prog.bin` avec `libc.lib`)
- `ed` : editeur plein ecran avec coloration syntaxique C
- `gfx` : mode graphique 320x240 256 couleurs, double-buffer, primitives, `view` et format CIMG
- `audio` : synthese 3 canaux + DMA PCM sur HDMI 32 kHz et lecteur `play` (WAV)
- `hw` : peripheriques HDMI, audio, clavier CH9350 AZERTY, SD SPI, UART, timer
- `kernel` : architecture du noyau C, appels systeme et `/boot/kernel.bin`

PARTIE II

Installation & Développement

Deux parcours complets : installation USB clé en main pour débutant Linux et environnement complet pour développeur.

Chapitre 04 — Guide d'installation Débutant sous Linux
Initialisation d'un Tang Nano 20K neuf et d'une microSD vierge par un seul câble USB-C [docs/GUIDE_DEBUTANT_LINUX.md] · p. 19

Chapitre 05 — Guide Développeur & Contribution
Chaîne de compilation, ajout de commandes, modification du noyau SD et du RTL Verilog [docs/GUIDE_DEVELOPPEUR.md] · p. 23

CH. 04 | Guide d'installation Débutant sous Linux

Initialisation d'un Tang Nano 20K neuf et d'une microSD vierge par un seul câble USB-C · Source : docs/GUIDE_DEBUTANT_LINUX.md

Ce guide explique pas à pas comment installer et démarrer **COS (*C Operating System*)** à partir de zéro avec :

- une carte **Sipeed Tang Nano 20K neuve**,
- une carte **microSD vierge** (insérée directement dans la carte Tang Nano 20K),
- un **seul câble USB-C** relié à votre PC Linux (pas besoin de lecteur de carte SD sur le PC !).

---

1. Matériel nécessaire

1. **Sipeed Tang Nano 20K** (`GW2AR-LV18QN88C8/I7`).
2. **Carte microSD** (format microSDHC/SD, par ex. 1 Go à 32 Go — même totalement vierge).
3. **Câble USB-C** (données + alimentation) relié à votre PC Linux.
4. **Écran / TV HDMI** + câble HDMI (adaptateur mini-HDMI ou câble selon connecteur de la carte).
5. **Clavier USB AZERTY** connecté à un **module CH9350** câblé sur la **broche 76** du Tang Nano 20K *(optionnel : vous pouvez aussi piloter intégralement COS depuis votre PC via le terminal série USB dès la fin de l'installation)*.

---

2. Préparer votre PC Linux (une seule fois)

Ouvrez un terminal Linux et installez **Python 3** et **openFPGALoader** :

- **Ubuntu / Debian / Linux Mint** : `bash  sudo apt update  sudo apt install -y python3 openfpgaloader unzip`
- **Fedora** : `bash  sudo dnf install -y python3 openFPGALoader unzip`
- **Arch Linux / Manjaro** : `bash  sudo pacman -S python openfpgaloader unzip`

Autoriser l'accès USB sans `sudo`

Pour éviter les erreurs `Permission denied` sur le port série `/dev/ttyUSB*` et le programmateur JTAG :

```
sudo usermod -aG dialout,plugdev "$USER"
```

> **Important** : après cette commande, **fermez votre session Linux et reconnectez-vous** (ou redémarrez le PC) pour que les nouveaux groupes soient actifs.

Si `openFPGALoader` signale un problème d'accès USB-JTAG, ajoutez la règle udev standard :

```
echo 'ATTRS{idVendor}=="0403", ATTRS{idProduct}=="6010", MODE="0666"
ATTRS{idVendor}=="0403", ATTRS{idProduct}=="6014", MODE="0666"
SUBSYSTEM=="tty", ATTRS{idVendor}=="0403", MODE="0666"' | sudo tee /etc/udev/rules.d/99-tangnano20k.rules
sudo udevadm control --reload-rules && sudo udevadm trigger
```

(Les cartes récentes utilisent un pont BL616 USB-JTAG/UART — les groupes `dialout` et `plugdev` suffisent généralement sur Debian/Ubuntu/Arch).

---

3. Branchement & installation automatique en 1 commande

1. **Insérez la carte microSD vierge** dans le lecteur de la face arrière du Tang Nano 20K (jusqu'au clic).
2. *(Optionnel pour voir l'affichage tout de suite)* Branchez le câble **HDMI** vers un écran.
3. **Branchez le câble USB-C** entre votre PC Linux et le connecteur USB-C du Tang Nano 20K.
4. Décompressez l'archive et lancez l'installateur :

```
unzip cos-tangnano20k-beginner-linux.zip
cd cos-tangnano20k-beginner-linux
./install.sh
```

Que fait `./install.sh` exactement ?

Vous n'avez rien d'autre à taper pendant l'opération :

1. **Flashage persistant du FPGA (`bitstream/cos.fs`)** : écrit le processeur COS-32 et la Boot ROM 64 Ko dans la mémoire Flash du Tang Nano 20K (il démarrera automatiquement à chaque mise sous tension).
2. **Détection et formatage automatique de la carte microSD vierge** : si aucune partition FAT32 valide n'est trouvée, COS formate la microSD en **FAT32** directement depuis le FPGA via le câble USB !
3. **Amorce rapide à 921 600 bauds** : copie d'abord `/boot/kernel.bin` (le noyau étendu), redémarre le noyau étendu en SRAM, bascule la liaison USB-UART à **`921 600` bauds**, puis transfère les **106 fichiers** du système (`/bin/*`, `/lib/*`, `/include/*`, `/docs/*`, `/src/*`, `/media/*`, `/boot/init.sh`).
4. **Démarrage final** : recharge la carte pour exécuter `/boot/init.sh` et vous placer devant l'invite `cos:/$`.

Options utiles de `./install.sh`

- `./install.sh --format` : force un reformatage propre en FAT32 avant de tout copier (efface le contenu de la microSD).
- `./install.sh --sync-only` : met à jour la carte microSD sans reflasher le FPGA (ultra rapide après une première installation).
- `./install.sh --port /dev/ttyUSB1` : choisit explicitement le port série si plusieurs périphériques USB-série sont branchés.

- `./install.sh --sram` : charge le bitstream uniquement en SRAM volatile (test sans écrire dans la Flash SPI).

---

4. Premiers pas sous COS

Vous pouvez interagir soit **directement sur la carte** (écran HDMI + clavier AZERTY CH9350 sur broche 76), soit **depuis le PC Linux** via un terminal série à `115 200` bauds :

```
# Exemple depuis votre PC Linux (remplacez /dev/ttyUSB1 par le port affiché par install.sh)
picocom -b 115200 /dev/ttyUSB1
# (ou avec minicom : minicom -D /dev/ttyUSB1 -b 115200 ; avec screen : screen /dev/ttyUSB1 115200)
```

À l'invite `cos:/$`, essayez ces commandes :

```
ls                          # Lister les dossiers (/bin, /boot, /docs, /media, /src...)
man cos                     # Ouvrir l'aide générale plein écran sur HDMI (q pour quitter)
man -p sh                   # Afficher le manuel du shell directement dans le terminal
view /media/coslogo.img     # Afficher l'image logo COS en mode graphique 320x240
play -b /media/demo.wav     # Jouer une musique WAV en tâche de fond sur HDMI
play -s                     # Couper le son (ou appuyer sur Ctrl-X / F12)
cc /src/examples/hello.c    # Compiler un programme C directement sur le FPGA !
hello                       # Exécuter le programme compilé
ed /monprog.c               # Ouvrir l'éditeur C plein écran (^S sauver, ^Q quitter)
find /docs -name "*.md"     # Rechercher des fichiers sur la carte SD
hexdump -n 32 /media/coslogo.img
```

> **Astuce pratique** : `Ctrl-C` stoppe immédiatement un programme ou un script en cours (même une boucle infinie), et `Ctrl-X` (ou `F12` au clavier USB) coupe immédiatement toute lecture audio.

Convertir vos propres images pour COS

Si `Pillow` est installé sur votre PC (`pip install pillow`), utilisez l'outil fourni dans le dossier `tools/` :

```
python3 tools/img2cos.py ma_photo.png ma_photo.img --width 320 --height 240 --rle
```

Copiez ensuite le fichier `.img` dans `sdcard/media/` et lancez `./install.sh --sync-only`, puis `view /media/ma_photo.img` sous COS.

---

5. Dépannage

Symptôme	Cause probable et solution
`Aucun port serie USB detecte (/dev/ttyUSB*)`	Vérifiez que votre câble USB-C transporte bien les données (certains câbles ne font que charger) et que la carte est alimentée. Vérifiez avec `ls /dev/ttyUSB*`.
`Permission denied` sur `/dev/ttyUSB1`	Exécutez `sudo usermod -aG dialout,plugdev $USER`, puis **déconnectez-vous et reconnectez votre session** Linux.
`openFPGALoader` non trouvé	Installez `openfpgaloader` via le gestionnaire de paquets de votre distribution (voir section 2).
Le terminal série affiche `TangNano20K />`	C'est l'invite du contrôleur USB-UART BL616 de la carte. Tapez simplement `choose uart` puis `Entrée` pour retrouver l'invite `cos:/$`. (`./install.sh` gère cela automatiquement).
La carte SD n'est pas détectée	Retirez et réinsérez la microSD fermement jusqu'au clic, puis redémarrez ou tapez `mount` dans COS. Évitez certaines vieilles cartes SDSC non compatibles SPI standard (privilégiez une microSDHC 2–32 Go).
Pas de sortie audio HDMI sur l'écran	Assurez-vous que le moniteur/TV HDMI supporte l'audio HDMI PCM (un écran PC DVI sans haut-parleurs n'a pas de son ; le son fonctionne sur TV/moniteur avec sortie audio ou prise casque).

CH. 05 | Guide Développeur & Contribution

Chaîne de compilation, ajout de commandes, modification du noyau SD et du RTL Verilog · Source : docs/GUIDE_DEVELOPPEUR.md

Ce guide s'adresse aux utilisateurs confirmés souhaitant modifier le code C de l'OS, ajouter des commandes, adapter le compilateur mini-C / assembleur ou modifier l'architecture Verilog du processeur **COS-32**.

1. Prérequis sur Linux

Compilation logicielle & tests émulateur

- `gcc`, `make`, `python3`, `bash`
- `mtools` (`mcopy`, `mdir`, `mtype`, `minfo`) et `dosfstools` (`/sbin/mkfs.fat`) pour générer/vérifier les images FAT32 des tests.

Synthèse FPGA & simulation RTL

- **OSS CAD Suite** installée dans `~/oss-cad-suite/bin` (ou variable `OSS_CAD=/chemin/vers/bin`) :
- `yosys`, `nextpnr-himbaechel`, `gowin_pack`, `openFPGALoader`
- `iverilog` et `vvp` (utilisés par `make test-rtl`).

2. Organisation du dépôt

Dossier / Fichier	Rôle
`rtl/cpu/cos32_cpu.v`	Processeur COS-32 multicyle (Fetch/Decode/Exec/Mul/Div/Mem/Halt + IRQ `EI`/`DI`/`RETI`)
`rtl/cos_soc.v`, `rtl/top_tang20k.v`	Interconnexion bus MMIO, arbitre SDRAM 3 ports, PLL 54 MHz / 25,2 MHz, top-level Tang Nano 20K
`rtl/periph/`	HDMI texte/graphique (`hdmi_text.v`), TMDS + audio Data Islands (`hdmi_audio_tmds.v`), APU 4 canaux (`audio_apu.v`), SD SPI (`sd_spi.v`), Timer IRQ (`timer.v`), UART FIFO (`uart_debug.v`, `uart_rx_fifo.v`)
`rtl/mem/`	Boot ROM BSRAM 64 Ko (`boot_rom.v`), cache SDRAM 4 Ko (`sdram_cache.v`), contrôleur SDRAM (`sdram_ctrl.v`)
`toolchain/cc/cc.c`	Compilateur mini-C (produit de l'assembleur COS-32 ; auto-compilé dans `/bin/cc`)

Dossier / Fichier	Rôle
`toolchain/as/as.c`	Assembleur/linker 2 passes (produit formats ELF-like `.cos`, raw `.bin`, `.lib`, `.hex` ; auto-compilé dans `/bin/as`)
`kernel/`	Noyau C dual-target : sans `-D SD_KERNEL` (Boot ROM ≤ 64 Ko) et avec `-D SD_KERNEL` (`/boot/kernel.bin` chargé en `0x00108000`, fenêtre 208 Ko)
`libc/`	Runtime `crt0.s` et bibliothèque C standard + appels système (`cos.h`)
`bin/`	Programmes utilisateur `/bin/*` (compilés avec `build/cos-cc` et liés à `build/libc.s`)
`docs/`	Spécifications ISA/ABI/brochage (`isa.md`, `memmap_abi.md`, `pinout.md`) et pages `man` (`docs/man/*.md`)
`examples/`	Exemples C (`hello.c`, `gfx.c`, `sound.c`, `video.c`, ...) et scripts shell (`init.sh`, `demo.sh`)
`tests/`	5 suites de non-régression : `asm`, `cc`, `rtl`, `kernel`, `bin`
`tools/`	Émulateur PC (`tools/emu/emu.c`), scripts USB/SD (`deploy_usb.py`, `deploy.sh`, `flash.sh`), `img2cos.py`, `make_release.py`

3. Flux de compilation, test et déploiement

```
# Compiler outils PC + libc + Boot ROM (64 Ko) + noyau SD (/boot/kernel.bin) + binaires /bin
make all sd-kernel

# Lancer les 5 suites complètes (recommandé avant tout déploiement)
make test

# Lancer une suite ciblée
make test-asm
make test-cc
make test-rtl
make test-kernel
make test-bin

# Synthétiser le bitstream FPGA (Vérifiez clk_sys >= 54 MHz, clk_pix >= 25,2 MHz, BSRAM <= 46)
make synth

# Déployer bitstream + carte SD complète par USB (FAST=1 pour sync incrémentale)
make deploy FAST=1 VERIFY=1 PORT=/dev/ttyUSB1

# Synchronisation incrémentale rapide (uniquement logiciels/noyau SD/docs modifiés, sans reflasher)
make sync PORT=/dev/ttyUSB1

# Générer les archives ZIP débutant & développeur dans dist/
make release
```

4. Recettes courantes de modification

A. Ajouter une nouvelle commande `/bin/<nom>`

1. Créez `bin/<nom>.c` (contrainte de `cos-cc` : pas de `float`/`double`, pas de `long long`, pas de `goto`, pas de macros à paramètres).
2. Ajoutez `<nom>` dans `BIN_NAMES` du `Makefile` et dans la vérification `tests/bin/run_tests.sh`.
3. Ajoutez l'entrée correspondante dans `bin/man.c` et la documentation dans `docs/man/`.
4. Exécutez `make bins test-bin` puis `make sync` pour l'envoyer sur la carte.

B. Modifier le noyau SD (`/boot/kernel.bin`) sans toucher au FPGA

- La fenêtre réservée en SDRAM pour `/boot/kernel.bin` est `0x00108000 – 0x0013BFFF` (`212 992` octets max, pile IRQ à `0x0013D000` et pile noyau à `0x0013FFF0`).
- **Attention à la taille de la Boot ROM (`build/kernel.bin`)** : elle doit rester **≤ 65 536 octets** (actuellement `65 525` octets). Gardez toute nouvelle fonctionnalité shell/OS uniquement sous `#ifdef SD_KERNEL` afin de ne pas faire déborder la Boot ROM.
- Après modification dans `kernel/` : `make sd-kernel && make test-bin && make sync`.

C. Modifier l'ISA, le CPU ou les périphériques RTL

1. Mettez à jour `rtl/cpu/cos32_cpu.v` (et si besoin `toolchain/as/as.c` et `tools/emu/emu.c` pour garder l'émulateur synchronisé avec le matériel).
2. Ajoutez un test matériel dans `tests/rtl/soc_test.c` et vérifiez avec `make test-rtl`.
3. Lancez `make synth` et contrôlez dans les logs que :
4. `BSRAM` reste `<= 46 / 46` (actuellement `45 / 46`),
5. `clk_sys >= 54.00 MHz` et `clk_pix >= 25.20 MHz` sont respectés.
6. Déployez le nouveau bitstream avec `make deploy FAST=1 VERIFY=1`.

D. Tester localement dans l'émulateur `cos-emu`

```
truncate -s 34M /tmp/cos_sd.img
/sbin/mkfs.fat -F 32 -n COS /tmp/cos_sd.img
bash tools/scripts/deploy.sh /tmp/cos_sd.img
./build/cos-emu --rom build/kernel.bin --sd /tmp/cos_sd.img
```

Option `--uart-stdio` : connecte directement stdin/stdout PC à l'UART du SoC (pratique pour automatiser ou utiliser `deploy_usb.py --emu-cmd ...`).

PARTIE III

Manuel Utilisateur, Shell & Outils

Prise en main complète de COS sur carte : shell Unix-like, utilitaires, éditeur C, multimédia et chaîne de compilation embarquée.

Chapitre 06 — Le Shell COS et l'environnement interactif
Historique persistant, complétion Tab, /boot/init.sh et résolution des commandes [docs/man/shell.md] · p. 27

Chapitre 07 — Pipes RAM, redirections, jokers et scripts sh
Pipelines SDRAM 2×256 Ko, contrôle if/for/while, test/[, arithmétique et IRQ 50 ms [docs/man/sh.md] · p. 29

Chapitre 08 — Commandes système et utilitaires /bin
Navigation, filtres Unix, find, hexdump, diagnostic mémoire et formatage FAT32 [docs/man/fs.md] · p. 31

Chapitre 09 — Éditeur plein écran ed avec coloration C
Raccourcis clavier, recherche avec surlignage, pagination et édition de code source [docs/man/ed.md] · p. 32

Chapitre 10 — Mode graphique HDMI 320×240 et images CIMG
Double buffering RGB332 sans déchirement, visionneuse view, img2cos.py et API C [docs/man/gfx.md] · p. 33

Chapitre 11 — Audio HDMI 4 canaux et lecteur WAV
Synthèse PSG 3 canaux, canal PCM DMA SDRAM, streaming SD continu play -b et contrôles [docs/man/audio.md] · p. 34

Chapitre 12 — Compilateur Mini-C auto-hébergé (cc)
Sous-ensemble C supporté, préprocesseur, options et enchaînement automatique avec as [docs/man/cc.md] · p. 35

Chapitre 13 — Assembleur et éditeur de liens COS-32 (as)
Formats .cos, .bin, .lib, .hex, options de base mémoire et résumé des instructions [docs/man/as.md] · p. 36

CH. 06 | Le Shell COS et l'environnement interactif

Historique persistant, complétion Tab, /boot/init.sh et résolution des commandes · Source : docs/man/shell.md

Le shell minimal reside en Boot ROM (`kernel.bin`), tandis que le noyau etendu `/boot/kernel.bin` sur carte SD ajoute historique, completion `Tab`, pipes `|`, redirections `>`, `>>`, `<`, `echo` et execution de scripts `.sh`.

Commandes internes du noyau

- `cd [chemin]` : change le repertoire courant (supporte `/`, `.`, `..`).
- `set [NOM=VAL]` / `unset NOM` : gere les variables d'environnement et `$PATH` (noyau SD).
- `echo [-n] [args...]` : affiche des arguments sur `stdout` (`-n` sans retour a la ligne, noyau SD).
- `sh <script.sh> [args...]` : execute un script shell `.sh` avec arguments `$1..$9` (noyau SD).
- `mount` : reinitialise et monte la carte SD FAT32.
- `sync` : active le serveur USB-UART rapide (jusqu'a 921600 bauds) pour `make sync`.
- `halt` : arrete le processeur.

Resolution des commandes et scripts

Lorsque vous tapez `<nom> [args...]`:

1. S'il n'y a pas de `/`, le shell cherche `/bin/<nom>`, puis `<nom>`, `<nom>.bin` et `<nom>.sh`.
2. S'il y a un `/`, il cherche `<nom>`, `<nom>.bin`, `<nom>.sh`.

Historique et completion (`Tab`)

- **Rappel de commandes** : touches `Haut` / `Bas` pour parcourir les 16 dernieres commandes distinctes (sauvegardees dans `/history`).
- **Completion `Tab`** : complete les commandes internes ou `/bin/*` en debut de ligne, ou les chemins de fichiers/dossiers ensuite ; liste les candidats s'il y a plusieurs choix.

Script de demarrage automatique (`/boot/init.sh`)

- Apres montage de la carte SD et chargement du noyau `/boot/kernel.bin`, s'il existe un fichier `/boot/init.sh`, le shell l'execute automatiquement avant d'afficher l'invite interactive.
- Utile pour afficher un message d'accueil, changer de dossier (`cd`) ou lancer un fond sonore (`play -b /media/demo.wav`).

Edition de ligne

- Clavier USB via module **CH9350L** (broche 76, AZERTY) ou terminal UART.
- `Backspace`, `Ctrl-U` (effacer la ligne), `Entree` (valider).
- Voir aussi `man sh` pour les pipes `|`, redirections et scripts.

CH. 07 | Pipes RAM, redirections, jokers et scripts sh

Pipelines SDRAM 2×256 Ko, contrôle if/for/while, test/[, arithmétique et IRQ 50 ms · Source : docs/man/sh.md

Avec le noyau SD `/boot/kernel.bin`, le shell COS gere les redirections de flux, les pipelines et les scripts `.sh`.

Redirections et pipes (`|`)

- `cmd > fichier` : redirige `stdout` vers `fichier` (cree ou ecrase).
- `cmd >> fichier` : redirige `stdout` en ajout a la fin de `fichier`.
- `cmd < fichier` : lit `stdin` depuis `fichier`.
- `cmd1 | cmd2 | cmd3` : enchaine la sortie de chaque etape vers l'entree de la suivante via deux tampons RAM de 256 Ko en SDRAM (`0x00740000` et `0x00780000`, zero ecriture SD ; repli sur `/_p0.tmp`/`/_p1.tmp` en cas de script imbrique).
- `stderr` (`fprintf(stderr, ...)`) reste toujours affiche a l'ecran.

Enchainement, jokers et variables (noyau SD)

- `cmd1 ; cmd2` : execute sequentiellement `cmd1` puis `cmd2`.
- `cmd1 && cmd2` : execute `cmd2` uniquement si `cmd1` reussit (code `0`).
- `cmd1 || cmd2` : execute `cmd2` uniquement si `cmd1` echoue (code `!= 0`).
- **Jokers (globbing)** : `*` et `?` (hors guillemets) sont etendus en liste triee de fichiers correspondants dans le repertoire cible (ex. `ls /src/bin/*.c`, `echo /docs/s*.md`). S'il n'y a aucune correspondance, le motif reste tel quel.
- `NOM=VAL` / `set [NOM=VAL]` / `unset NOM` : definit ou supprime des variables d'environnement (24 max) ; `$NOM`, `${NOM}`, `$?` et l'arithmetique `$((expr))` (`+`, `-`, `*`, `/`, `%`) sont etendus.
- `PATH` (defaut `/bin:.`) : liste des dossiers de recherche separes par `:`.
- Raccourcis globaux : `Ctrl-C` interrompt un programme/script (y compris boucle CPU grace a l'IRQ timer 50 ms) ; `Ctrl-X` ou `F12` coupe immediatement l'audio.

Structures de controle et `test` / `[`

- `if cond; then ... [elif cond; then ...] [else ...] fi`
- `for var [in liste...]; do ... done` (ou arguments `$1..` sans `in`)
- `while cond; do ... done` (avec `break` et `continue`)
- `test expr` ou `[ expr ]` :
- Negation `!`, chaines (`-z`, `-n`, `=`, `!=`), entiers (`-eq`, `-ne`, `-lt`, `-le`, `-gt`, `-ge`), fichiers (`-e`, `-f`, `-d`, `-s`).

Commandes associees

- `cat [-n] [fichier...]` : affiche des fichiers ou `stdin` (`-`). `-n` numerote les lignes.
- `more [fichier...]` : affiche page par page (29 lignes). Touches : `Espace` (page), `Entree` (1 ligne), `q` (quitter).
- `grep [-i] [-n] [-v] [-c] <motif> [fichier...]` : filtre les lignes selon un motif (`-i` casse, `-n` numeros, `-v` inverser, `-c` compter).
- `head [-n N | -N] [fichier...]` : affiche les `N` premieres lignes (defaut 10).
- `tail [-n N | -N] [fichier...]` : affiche les `N` dernieres lignes (defaut 10).
- `wc [-l] [-w] [-c] [fichier...]` : compte lignes (`-l`), mots (`-w`) et octets (`-c`).
- `sort [-r] [-n] [-u] [fichier...]` : trie les lignes (`-r` inverse, `-n` numerique, `-u` unique).
- `uniq [-c] [-d] [-u] [fichier]` : filtre les doublons adjacents (`-c` compteur, `-d` doublons, `-u` uniques).
- `clear`, `sleep <ms|Ns>`, `uptime`, `echo [-n] [args...]`, `test` / `[`.

Scripts shell (`sh`)

- Lancement : `sh script.sh [arg1 arg2 ...]` ou directement `script.sh [args...]`.
- Syntaxe supportee dans les scripts `.sh` :
- Commentaires commencant par `#` (hors guillemets `'...'` et `"..."`).
- Variables positionnelles : `$0` (script), `$1`..`$9` (arguments), `$#` (nombre d'arguments).
- Pipes, redirections, commandes internes (`echo`, `cd`, ...) et programmes `/bin/*`.
- Exemple fourni : `sh /src/examples/demo.sh Bonjour` (voir `/src/examples/demo.sh`).

CH. 08 | Commandes système et utilitaires /bin

Navigation, filtres Unix, find, hexdump, diagnostic mémoire et formatage FAT32 · Source : docs/man/fs.md

Le noyau Boot ROM gere les noms courts FAT32 **8.3** ; le noyau SD `/boot/kernel.bin` ajoute la lecture des noms longs **VFAT LFN** (ex. `ls`, `open`, `stat`).

Navigation et consultation

- `ls [-a] [-l] [chemin...]` : liste un repertoire (ou un fichier) avec taille en octets et `<DIR>`. Par defaut liste `.`.
- `cat [-n] [fichier...]` : affiche le contenu d'un ou plusieurs fichiers (ou `stdin` / `-`). `-n` numerote les lignes.
- `more [fichier...]` : filtre page par page (29 lignes). `Espace` = page suivante, `Entree` = 1 ligne, `q` = quitter.
- `grep [-i] [-n] [-v] [-c] <motif> [fichier...]` : recherche de texte par ligne.
- `head [-n N] [fichier...]` / `tail [-n N] [fichier...]` : debut ou fin d'un fichier / flux.
- `wc [-l] [-w] [-c] [fichier...]` : compte lignes, mots et octets.
- `sort [-r] [-n] [-u] [fichier...]` / `uniq [-c] [-d] [-u]` : tri et deduplication de lignes.
- `find [chemin] [-name motif] [-type f|d] [-maxdepth N]` : parcours recursif FAT32 avec filtrage par motif/type.
- `hexdump [-C] [-n len] [-s off] [fichier]` : dump hexadecimal et ASCII par blocs de 16 octets (ou `stdin`).
- `clear`, `sleep <ms|Ns>`, `uptime` : effacement d'ecran, pause et temps d'activite.
- `cd [repertoire]` : change le repertoire courant.
- `md <dir>...` : cree un ou plusieurs repertoires.
- `rd <dir>...` : supprime un ou plusieurs repertoires vides.
- `cp [-r] <src>... <dst>` : copie un fichier (ou un dossier avec `-r`).
- `rm [-r] [-f] <cible>...` : supprime un fichier (ou un dossier recursivement avec `-r`).

Administration & diagnostic

- `free` : affiche l'utilisation des 8 Mo de SDRAM interne (noyau, utilisateur, libre).
- `df` : lit le BPB et la FAT pour afficher l'espace FAT32 total, utilise et disponible (avec duree SD).
- `format [-y] [-s secteurs] [-c spc] [-L label]` : formate la carte SD en FAT32.
- `man [-p] [sujet]` : consulte la documentation `/docs/*.md`.

CH. 09 | Éditeur plein écran ed avec coloration C

Raccourcis clavier, recherche avec surlignage, pagination et édition de code source · Source : docs/man/ed.md

`ed` edite des fichiers texte en mode plein ecran 80x30 sur HDMI avec coloration syntaxique C automatique pour les fichiers `.c` et `.h`.

Lancement

- `ed fichier.c` : ouvre (ou cree) `fichier.c`.
- `ed` : ouvre un tampon vierge (le nom est demande lors du premier `^S`).

Raccourcis clavier

- `Ctrl-S` (`^S`) : enregistrer le fichier sur la carte SD.
- `Ctrl-Q` (`^Q`) : quitter (demande confirmation si modifie ; `^Q` force, `^S` sauve et quitte).
- `Ctrl-F` (`^F`) : rechercher un texte (avec surlignage des occurrences ; Entree vide re-cherche le motif precedent).
- `Ctrl-N` (`^N`) / `Ctrl-P` (`^P`) : occurrence suivante / precedente.
- `Ctrl-G` (`^G`) : aller a un numero de ligne.
- `Ctrl-B` (`^B`) / `PgUp` et `Ctrl-V` (`^V`) / `PgDn` : page precedente / suivante (indicateur `P:courante/total` dans la barre d'etat).
- `Ctrl-T` (`^T`) / `Ctrl-E` (`^E`) : debut / fin du fichier.
- `Ctrl-K` (`^K`) : supprimer la ligne courante.
- Fleches, `Home`, `End`, `Suppr`, `Backspace`, `Tab` (4 espaces).

Coloration syntaxique C

- Cyan : mots-cles et types C (`int`, `struct`, `return`, `uint32_t`, ...).
- Vert : commentaires `//` et `/* ... */` multi-lignes.
- Jaune : chaines `"..."` et caracteres `'...'`.
- Rouge clair : constantes numeriques.
- Magenta : directives preprocesseur `#include`, `#define`, etc.

CH. 10 | Mode graphique HDMI 320×240 et images CIMG

Double buffering RGB332 sans déchirement, visionneuse view, img2cos.py et API C · Source : docs/man/gfx.md

COS dispose d'un mode graphique **320x240 en 256 couleurs (RGB332)** avec **double buffering** sans dechirement (sync sur VSYNC), police bitmap 8x8 integree et affichage d'images.

Commande `view`

- `view <image.img> [x] [y]` : affiche une image au format `.img` (`CIMG`) aux coordonnees `(x, y)` (centree par defaut), et attend l'appui d'une touche pour revenir en mode texte.
- Exemple : `view /media/coslogo.img` ou `view /media/coslogo.img 20 30`.

Format COS Image (`CIMG`, en-tete 12 octets LE)

- `0..3` : `"CIMG"` ; `4..5` : `width` (1..320) ; `6..7` : `height` (1..240)
- `8` : `format` (`0` = RGB332 brut, `1` = RGB332 compresse PackBits RLE)
- `9` : `has_trans` (`1` = couleur `trans_key` ignoree) ; `10` : `trans_key` ; `11` : `0`.
- Sur PC : `python3 tools/scripts/img2cos.py entree.png sortie.img [--size LxH] [--fit] [--rle]`

API C (`<cos.h>`)

- Mode & pages : `gfx_mode(en)`, `gfx_set_draw_page(p)`, `gfx_set_disp_page(p)`, `gfx_wait_vsync()`, `gfx_flip()` (`GFX_PAGE0 = 0x008C0000`, `GFX_PAGE1 = 0x008E0000`).
- Dessin : `gfx_rgb(r,g,b)`, `gfx_clear(c)`, `gfx_pixel(x,y,c)`, `gfx_hline`, `gfx_vline`, `gfx_line`, `gfx_rect`, `gfx_fill_rect`, `gfx_circle`, `gfx_fill_circle`.
- Texte & images : `gfx_putc(x,y,ch,fg,bg)` (`bg=-1` transparent), `gfx_puts`, `gfx_draw_img(path,x,y)`.
- Voir `/src/examples/gfx.c` et `/src/examples/video.c` (texte 80x30).

CH. 11 | Audio HDMI 4 canaux et lecteur WAV

Synthèse PSG 3 canaux, canal PCM DMA SDRAM, streaming SD continu play -b et contrôles · Source : docs/man/audio.md

Le SoC integre une APU (`0x10050000`) qui injecte du son LPCM **32 kHz / 16 bits stereo** directement dans le flux HDMI via les *Data Islands* TMDS.

Commande `play`

- `play <fichier.wav>` : lecture directe (bloquante) d'un WAV PCM 8/16-bit mono/stereo (`32`, `16` ou `8 kHz`). Appuyez sur une touche pour stopper.
- `play -b [-l] [-S] <fichier.wav>` : lecture en **tache de fond** (`-l` = boucle). Jusqu'a 1 Mo, le WAV est charge integralement en SDRAM (`0x007C0000`, mode **RAM DMA**). Au-dela de 1 Mo (ou avec `-S`), le noyau SD lit le fichier **en streaming continu** dans un tampon circulaire DMA, sans limite de duree.
- `play -q` : affiche l'etat (`idle`, `playing (RAM DMA)` ou `streaming (SD DMA, refills)`).
- `play -s` : arrete la lecture en tache de fond.
- Exemple : `play -b -l /media/demo.wav`.

Canaux disponibles

- **Canaux 0..2 (PSG)** : generateurs de formes d'onde avec duree materielle en ms et atténuation automatique (`WAVE_SQUARE=0`, `WAVE_TRI=1`, `WAVE_SAW=2`, `WAVE_NOISE=3`).
- **Canal 3 (PCM FIFO + DMA SDRAM)** : FIFO materielle `1024 x 16 bits` en BSRAM, alimentable manuellement ou en arriere-plan par DMA depuis la SDRAM (CPU libre pour le graphisme ou les calculs).

API C (`<cos.h>`)

- Synthese : `snd_tone(ch, freq_hz, dur_ms, wave, vol, decay)`, `snd_stop(ch)`, `snd_stop_all()`, `snd_beep(freq_hz, dur_ms)`.
- PCM FIFO : `snd_pcm_clear(div)` (`div`: `1`=32kHz, `2`=16kHz, `4`=8kHz), `snd_pcm_push(sample16)`.
- DMA SDRAM : `snd_dma_play(buf, len_bytes, is_16bit, div, loop)`, `snd_dma_stop()`, `snd_dma_busy()`.
- Voir `/src/examples/sound.c` et `/src/examples/gfx.c`.

CH. 12 | Compilateur Mini-C auto-hébergé (cc)

Sous-ensemble C supporté, préprocesseur, options et enchaînement automatique avec as · Source : docs/man/cc.md

`cc` traduit un sous-ensemble du langage C vers l'assembleur COS-32 et enchaine automatiquement `/bin/as` sous COS pour produire un executable `.bin` avec chronometrage des etapes et des temps d'E/S SD (`sys_sd_stats`).

Syntaxe

- `cc fichier.c` : compile vers `fichier.s` puis lie avec `/bin/as` pour produire `fichier.bin`.
- `cc -o monprog.bin fichier.c` : produit l'executable `monprog.bin`.
- `cc -S [-o sortie.s] fichier.c` : genere uniquement l'assembleur `.s` sans assembler.
- `cc -I rep -D NOM=VAL fichier.c` : ajoute un dossier d'include ou une macro (`-D NOM` avec espace). (`/include` est inclus par defaut sous COS ; les tampons d'entree/sortie sont bufferises 2 Ko.)

Sous-ensemble C supporte

- **Types** : `void`, `char`, `short`, `int`, `unsigned`, `uint8_t`, `uint16_t`, `uint32_t`, `int32_t`, `size_t`, pointeurs, tableaux multi-dimensions, `struct`, `union`, `enum`, `typedef`, pointeurs de fonctions.
- **Controle** : `if`/`else`, `while`, `do`/`while`, `for`, `switch`/`case`/`default`, `break`, `continue`, `return`.
- **Preprocesseur** : `#include`, `#define` (constantes/objets), `#ifdef`/`#ifndef`/`#if`/`#elif`/`#else`/`#endif`.
- **Limitations connues** : pas de `float`/`double`, pas de `long long` (64 bits), pas de `goto`, pas de champs de bits, pas de macros avec parametres `FN(x)`.

Exemple rapide

```
#include <stdio.h>
int main(void) { printf("Bonjour COS!\n"); return 0; }
```

Tapez `cc hello.c` puis `hello`.

CH. 13 | Assembleur et éditeur de liens COS-32 (as)

Formats .cos, .bin, .lib, .hex, options de base mémoire et résumé des instructions · Source : docs/man/as.md

`as` est un assembleur 2 passes ciblant l'ISA 32 bits COS-32, avec affichage de la progression et des durees d'E/S SD.

Syntaxe sous COS

- `as prog.s` : si `crt0.s` est absent et que `/lib/libc.lib` existe, `as` lie automatiquement l'index binaire `/lib/libc.lib` (sinon `/lib/crt0.s` + `/lib/libc.s`) et produit `prog.bin`.
- `as -o sortie.bin /lib/crt0.s /lib/libc.s prog.s` : assemblage explicite complet.
- `as --nostdlib -f raw --base 0x0 -o rom.bin ...` : assemblage brut sans libc.

Options principales

- `-o <fichier>` : fichier de sortie (`.bin` ou `a.cos`).
- `-f cos|raw|hex|lib` : format executable COS (`.cos`), binaire plat (`raw`), mots hexa (`hex`), ou bibliotheque binaire indexee (`CLIB`).
- `--base <addr>` : adresse de chargement `.text` (defaut `0x00140000`).
- `--data-base <addr>` : adresse VMA separee pour `.data` (ex. ROM en `0x0`, `.data` en `0x00100010`).
- `--entry <symbole>` : point d'entree (defaut `_start`).
- `--nostdlib` : desactive la liaison automatique de `libc.lib` / `crt0.s`.

Jeu d'instructions resume

- Arithmetique : `ADD`, `SUB`, `MUL`, `DIV`, `MOD`, `AND`, `OR`, `XOR`, `SHL`, `SHR`, `SAR`, `ADDI`, `LI`, `LA`, `MOV`.
- Memoire : `LDW`, `LDH`, `LDB`, `LDBU`, `LDHU`, `STW`, `STH`, `STB`, `PUSH`, `POP`.
- Controle : `SEQ`, `SNE`, `SLT`, `SLTU`, `SGE`, `SGEU`, `BZ`, `BNZ`, `BEQ`, `BNE`, `BLT`, `BGE`, `JMP`, `CALL`, `CALLR`, `RET`, `SYSCALL`, `HALT`.

PARTIE IV

Architecture Matérielle, ISA & Noyau

Spécifications bas niveau : jeu d'instructions COS-32, carte mémoire 8 Mo, registres MMIO, brochage FPGA et noyau C.

Chapitre 14 — Spécification du jeu d'instructions COS-32
Banc de registres 32 bits, formats R/I/J, encodage complet et interruptions EI/DI/RETI [docs/isa.md] · p. 38

Chapitre 15 — Carte mémoire, registres MMIO, format .cos et ABI
Adressage physique, découpage SDRAM 8 Mo, périphériques mappés et appels système [docs/memmap_abi.md] · p. 44

Chapitre 16 — Brochage Tang Nano 20K et chaîne FPGA
Affectation des broches .cst, SDRAM interne GW2AR-18 et commandes Yosys / nextpnr [docs/pinout.md] · p. 50

Chapitre 17 — Périphériques du SoC et API C <cos.h>
Vue synthétique des contrôleurs matériels et interface de programmation utilisateur [docs/man/hw.md] · p. 52

Chapitre 18 — Noyau C, Boot ROM et /boot/kernel.bin
Architecture dual-target, chainloading depuis SD, VFAT LFN et recompilation sur carte [docs/man/kernel.md] · p. 53

CH. 14 | Spécification du jeu d'instructions COS-32

Banc de registres 32 bits, formats R/I/J, encodage complet et interruptions EI/DI/RETI · Source : docs/isa.md

1. Principes généraux

- **Largeur de mot** : 32 bits.
- **Adressage** : à l'octet (*byte-addressed*), *little-endian*.
- **Instructions** : largeur fixe de **32 bits**, alignées sur 4 octets.
- **Implémentation RTL (`clk_sys = 54 MHz`)** : exécution multi-cycles (`S_FETCH -> S_DECODE -> S_EXEC -> S_MEM_WAIT`), avec un étage dédié `S_MUL` (1 cycle supplémentaire) pour la multiplication matérielle `MUL`.
- **PC à la mise sous tension (mode système)** : `0x00000000`.

2. Banc de registres (16 registres × 32 bits)

Numéro	Nom	Alias	Rôle / Convention d'appel C
`r0`	`r0`	`zero`	Câblé à `0` en lecture ; toute écriture est ignorée
`r1`	`r1`	`rv`	Valeur de retour / 1er arg syscall / temporaire (`caller-saved`)
`r2`	`r2`	—	Temporaire / 2e arg syscall (`caller-saved`)
`r3`	`r3`	—	Temporaire / 3e arg syscall (`caller-saved`)
`r4`	`r4`	—	Temporaire / 4e arg syscall (`caller-saved`)
`r5`	`r5`	—	Temporaire (`caller-saved`)
`r6`	`r6`	—	Temporaire (`caller-saved`)
`r7`	`r7`	—	Temporaire / numéro de syscall (`caller-saved`)
`r8`	`r8`	—	Variable locale conservée (`callee-saved`)
`r9`	`r9`	—	Variable locale conservée (`callee-saved`)
`r10`	`r10`	—	Variable locale conservée (`callee-saved`)
`r11`	`r11`	—	Variable locale conservée (`callee-saved`)
`r12`	`r12`	—	Variable locale conservée (`callee-saved`)
`r13`	`r13`	`fp`	Pointeur de cadre (*Frame Pointer*, `callee-saved`)
`r14`	`r14`	`sp`	Pointeur de pile (*Stack Pointer*, décroissant, aligné 4 octets)
`r15`	`r15`	`lr`	Adresse de retour (*Link Register*)

3. Formats d'instructions

Format R (Registre-Registre)

```
31    26 25  22 21  18 17  14 13        6 5      0
+--------+------+------+------+-----------+--------+
|   op   |  rd  |  ra  |  rb  |   00000   |  func  |
+--------+------+------+------+-----------+--------+
  6 bits   4 b    4 b    4 b      8 bits    6 bits
```

Format I (Immédiat, Mémoire, Branchement)

```
31    26 25  22 21  18 17 16 15                 0
+--------+------+------+-----+--------------------+
|   op   |  rd  |  ra  | 00  |       imm16        |
+--------+------+------+-----+--------------------+
  6 bits   4 b    4 b    2 b        16 bits
```

Format J (Saut long / Appel / Syscall)

```
31    26 25                                     0
+--------+-----------------------------------------+
|   op   |                 imm26                   |
+--------+-----------------------------------------+
  6 bits                 26 bits
```

Pour les sauts et appels relatifs au PC :

- Format J : `target = PC + 4 + (sext26(imm26) << 2)`
- Format I (branchements) : `target = PC + 4 + (sext16(imm16) << 2)`

4. Encodage complet des instructions

4.1 Format R (`op = 0x00`)

func	Mnémonique	Syntaxe	Sémantique
`0x00`	`ADD`	`ADD rd, ra, rb`	`rd = ra + rb`
`0x01`	`SUB`	`SUB rd, ra, rb`	`rd = ra - rb`

func	Mnémonique	Syntaxe	Sémantique
0x02	AND	AND rd, ra, rb	rd = ra & rb
0x03	OR	OR rd, ra, rb	rd = ra | rb
0x04	XOR	XOR rd, ra, rb	rd = ra ^ rb
0x05	SHL	SHL rd, ra, rb	rd = ra << (rb & 31)
0x06	SHR	SHR rd, ra, rb	rd = (uint32)ra >> (rb & 31)
0x07	SAR	SAR rd, ra, rb	rd = (int32)ra >> (rb & 31)
0x08	MUL	MUL rd, ra, rb	rd = (uint32)((int32)ra * (int32)rb)
0x09	DIV	DIV rd, ra, rb	Signé : rb == 0 ? -1 : (ra == INT_MIN && rb == -1 ? INT_MIN : ra / rb)
0x0A	MOD	MOD rd, ra, rb	Signé : rb == 0 ? ra : (ra == INT_MIN && rb == -1 ? 0 : ra % rb)
0x0B	DIVU	DIVU rd, ra, rb	Non signé : rb == 0 ? 0xFFFFFFFF : ra / rb
0x0C	MODU	MODU rd, ra, rb	Non signé : rb == 0 ? ra : ra % rb
0x0D	NOT	NOT rd, ra	rd = ~ra (rb = 0)
0x10	SEQ	SEQ rd, ra, rb	rd = (ra == rb) ? 1 : 0
0x11	SNE	SNE rd, ra, rb	rd = (ra != rb) ? 1 : 0
0x12	SLT	SLT rd, ra, rb	rd = ((int32)ra < (int32)rb) ? 1 : 0
0x13	SLTU	SLTU rd, ra, rb	rd = ((uint32)ra < (uint32)rb) ? 1 : 0
0x14	SGE	SGE rd, ra, rb	rd = ((int32)ra >= (int32)rb) ? 1 : 0
0x15	SGEU	SGEU rd, ra, rb	rd = ((uint32)ra >= (uint32)rb) ? 1 : 0
0x20	JR	JR ra	PC = ra (rd = 0, rb = 0)
0x21	CALLR	CALLR ra	tmp = ra; lr = PC + 4; PC = tmp (rd = 0, rb = 0)
0x3F	HALT	HALT	Arrête le processeur

4.2 Format I — ALU et immédiats

op	Mnémonique	Syntaxe	Sémantique
0x01	ADDI	ADDI rd, ra, imm16	rd = ra + sext16(imm16)
0x02	ANDI	ANDI rd, ra, imm16	rd = ra & zext16(imm16)

op	Mnémonique	Syntaxe	Sémantique
0x03	ORI	ORI rd, ra, imm16	rd = ra | zext16(imm16)
0x04	XORI	XORI rd, ra, imm16	rd = ra ^ zext16(imm16)
0x05	SHLI	SHLI rd, ra, imm16	rd = ra << (imm16 & 31)
0x06	SHRI	SHRI rd, ra, imm16	rd = (uint32)ra >> (imm16 & 31)
0x07	SARI	SARI rd, ra, imm16	rd = (int32)ra >> (imm16 & 31)
0x08	LUI	LUI rd, imm16	rd = zext16(imm16) << 16 (ra = 0)
0x09	SLTI	SLTI rd, ra, imm16	rd = ((int32)ra < sext16(imm16)) ? 1 : 0
0x0A	SLTIU	SLTIU rd, ra, imm16	rd = ((uint32)ra < zext16(imm16)) ? 1 : 0

4.3 Format I — Accès mémoire (`addr = ra + sext16(imm16)`)

op	Mnémonique	Syntaxe	Sémantique
0x10	LDB	LDB rd, [ra + imm16]	rd = sext8(mem8[addr])
0x11	LDBU	LDBU rd, [ra + imm16]	rd = zext8(mem8[addr])
0x12	LDH	LDH rd, [ra + imm16]	rd = sext16(mem16[addr])
0x13	LDHU	LDHU rd, [ra + imm16]	rd = zext16(mem16[addr])
0x14	LDW	LDW rd, [ra + imm16]	rd = mem32[addr]
0x18	STB	STB rd, [ra + imm16]	mem8[addr] = rd[7:0]
0x19	STH	STH rd, [ra + imm16]	mem16[addr] = rd[15:0]
0x1A	STW	STW rd, [ra + imm16]	mem32[addr] = rd

4.4 Format I — Branchements conditionnels

op	Mnémonique	Syntaxe	Sémantique
0x20	BEQ	BEQ rd, ra, label	if (rd == ra) PC = PC + 4 + (sext16(imm16) << 2)
0x21	BNE	BNE rd, ra, label	if (rd != ra) PC = PC + 4 + (sext16(imm16) << 2)
0x22	BLT	BLT rd, ra, label	if ((int32)rd < (int32)ra) PC = target
0x23	BGE	BGE rd, ra, label	if ((int32)rd >= (int32)ra) PC = target
0x24	BLTU	BLTU rd, ra, label	if ((uint32)rd < (uint32)ra) PC = target
0x25	BGEU	BGEU rd, ra, label	if ((uint32)rd >= (uint32)ra) PC = target

4.5 Format J — Sauts et appel système

op	Mnémonique	Syntaxe	Sémantique
`0x28`	`JMP`	`JMP label`	`PC = PC + 4 + (sext26(imm26) << 2)`
`0x29`	`CALL`	`CALL label`	`lr = PC + 4; PC = PC + 4 + (sext26(imm26) << 2)`
`0x30`	`SYSCALL`	`SYSCALL`	Mode système : `lr = PC + 4; PC = 0x00000008` (`imm26 = 0`)
`0x31`	`RETI`	`RETI`	Retour d'interruption : `PC = EPC; irq_en = 1`
`0x32`	`EI`	`EI`	Active les interruptions matérielles (`irq_en = 1`)
`0x33`	`DI`	`DI`	Désactive les interruptions matérielles (`irq_en = 0`)

5. Pseudo-instructions de l'assembleur (`as`)

Pseudo-instruction	Traduction
`NOP`	`ADD r0, r0, r0`
`MOV rd, rs`	`ADD rd, rs, r0`
`NEG rd, rs`	`SUB rd, r0, rs`
`BZ rs, label`	`BEQ rs, r0, label`
`BNZ rs, label`	`BNE rs, r0, label`
`RET`	`JR lr`
`PUSH rs`	`ADDI sp, sp, -4` ; `STW rs, [sp + 0]`
`POP rd`	`LDW rd, [sp + 0]` ; `ADDI sp, sp, 4`
`LA rd, label`	`LUI rd, (addr >> 16) & 0xFFFF` ; `ORI rd, rd, addr & 0xFFFF`
`LI rd, val`	`ADDI rd, r0, val` (si `val` résolu en passe 1 dans `[-32768, 32767]`), `LUI rd, val >> 16` (si `val & 0xFFFF == 0`), sinon `LUI` + `ORI`

6. Convention d'appel C

1. **Passage des arguments** : empilés de **droite à gauche** (le 1er argument est au sommet de la pile à l'entrée de la fonction).
2. **Prologue** : `asm PUSH lr PUSH fp MOV fp, sp ADDI sp, sp, -N_LOCALS` *(Avec ce prologue, `[fp + 0] = saved_fp`, `[fp + 4] = saved_lr`, `[fp + 8] = arg0`, `[fp + 12] = arg1`, ..., et `[fp - 4]`, `[fp - 8]`, ... = variables locales.)*

3. **Épilogue** : `asm MOV sp, fp POP fp POP lr RET`
4. **Retour** : valeur dans `r1` (`rv`). L'appelant nettoie les arguments (`ADDI sp, sp, 4 * N_ARGS`).

CH. 15 | Carte mémoire, registres MMIO, format .cos et ABI

Adressage physique, découpage SDRAM 8 Mo, périphériques mappés et appels système · Source : docs/memmap_abi.md

1. Horloge système et carte mémoire physique

- **Fréquence CPU / SoC** : `clk_sys = 54 MHz` (PLL à partir du quartz 27 MHz), `clk_sdram = 54 MHz` (déphasée à 180°), `clk_pix = 25,2 MHz` (horloge pixel VGA/HDMI 640×480@60Hz).
- **Cache SDRAM** : cache à correspondance directe (*write-through*) de **4 Ko** (1024 mots de 32 bits en BSRAM) devant le contrôleur SDRAM 32 bits avec arbitre 3 ports (CPU, DMA graphique, DMA audio).

Plage d'adresses	Taille	Périphérique / Usage
`0x0000_0000 – 0x0000_FFFF`	64 Ko	**Boot ROM / Noyau minimal** (BSRAM dans le bitstream FPGA)
`0x0010_0000 – 0x008F_FFFF`	8 Mo	**SDRAM interne** (GW2AR-18)
`0x1000_0000 – 0x1000_2FFF`	12 Ko	**HDMI Texte 80×30 / Mode graphique 320×240**
`0x1001_0000 – 0x1001_00FF`	256 o	**Clavier CH9350** (UART RX + FIFO 64 o, broche 76)
`0x1002_0000 – 0x1002_03FF`	1 Ko	**Contrôleur SD SPI** (registres + tampon bloc 512 o)
`0x1003_0000 – 0x1003_00FF`	256 o	**Timer système** (millisecondes & compteur de cycles 54 MHz)
`0x1004_0000 – 0x1004_00FF`	256 o	**UART Debug** (USB-UART BL616, FIFO RX 2048 o, débit programmable)
`0x1005_0000 – 0x1005_00FF`	256 o	**APU Audio HDMI 4 canaux** (3 PSG + FIFO PCM 1024×16b + DMA SDRAM)

Vecteurs système (en Boot ROM) et bloc RAM noyau

- `0x0000_0000` : vecteur Reset (point d'entrée au démarrage FPGA).
- `0x0000_0008` : vecteur `SYSCALL` (`lr` contient `PC + 4` de l'instruction `SYSCALL`), saute vers l'adresse stockée en `0x0010_0000`.
- `0x0000_000C` : vecteur `IRQ` matérielle (`EPC` contient `PC`, `irq_en = 0`), saute vers l'adresse stockée en `0x0010_000C`.
- `0x0010_0000` : `ram_syscall_handler` (adresse du gestionnaire `SYSCALL` actif).
- `0x0010_0004` : `ram_exec_saved_sp` (pile noyau sauvegardée pendant `exec`).
- `0x0010_0008` : `ram_boot_flags` (bit 0 = `1` si déjà chainloadé depuis `/boot/kernel.bin`).
- `0x0010_000C` : `ram_irq_handler` (adresse du gestionnaire `IRQ` actif).
- `0x0010_0010` : `ram_in_user` (`1` pendant l'exécution d'un binaire utilisateur).

Découpage de la SDRAM (8 Mo)

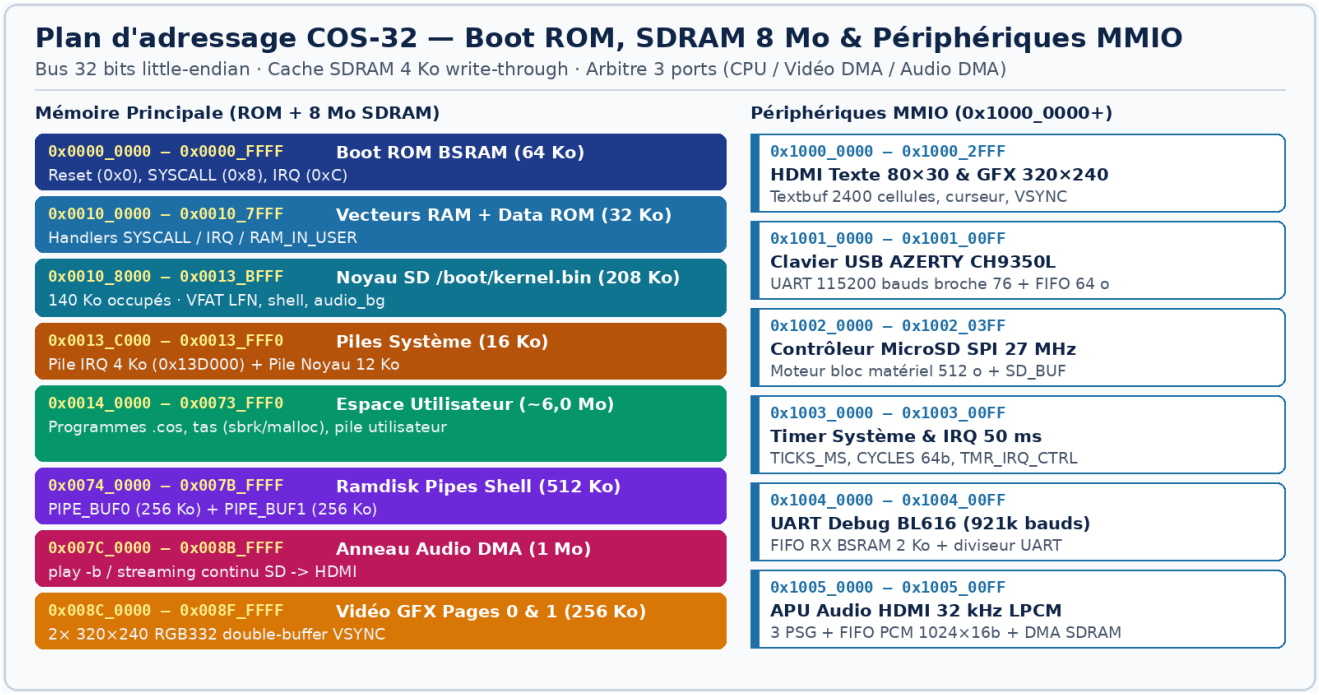


Figure 2 — Carte physique 32 bits COS-32 : Boot ROM, découpage SDRAM 8 Mo et registres MMIO.

- `0x0010_0000 – 0x0013_FFFF` (256 Ko) : vecteurs/contrôle RAM (`0x0010_0000..0x0010_001F`), `.data`/`.bss` du noyau ROM (`0x0010_0020+`), zone `/boot/kernel.bin` (**208 Ko** : `0x0010_8000–0x0013_BFFF`), pile IRQ (`0x0013_C000–0x0013_D000`) et pile noyau (`0x0013_D000–0x0013_FFF0`).
- `0x0014_0000 – 0x0073_EFF0` (~6,0 Mo) : zone de chargement des exécutables `.cos` + tas (`sbrk`) + pile utilisateur décroissante (`USER_STACK_TOP = 0x0073_FFF0`).
- `0x0074_0000 – 0x007B_FFFF` (512 Ko) : ramdisk de pipe en SDRAM (`PIPE_BUF0` 256 Ko + `PIPE_BUF1` 256 Ko).
- `0x007C_0000 – 0x008B_FFFF` (1 Mo) : tampon circulaire DMA audio en tâche de fond (`AUDIO_BG_ADDR`, utilisé par `play -b` / `play -l` et le streaming SD).
- `0x008C_0000 – 0x008D_2BFF` : **Page graphique 0** (`GFX_PAGE0`, 320×240 RGB332 = 76 800 o).
- `0x008E_0000 – 0x008F_2BFF` : **Page graphique 1** (`GFX_PAGE1`, 320×240 RGB332 = 76 800 o).

---

2. Registres MMIO

2.1 HDMI Texte & Graphique (`0x1000_0000`)

- `0x1000_0000 – 0x1000_12BF` : 2 400 cellules (`80 × 30`) de 16 bits (`LDH`/`STH` ou `LDW`/`STW` pairés).
- `[7:0]` : code caractère (CP437 / ASCII étendu + accents français).
- `[11:8]` : couleur avant-plan (0–15, palette CGA/VGA).
- `[15:12]` : couleur arrière-plan (0–15).

- `0x1000_2000` : `CURSOR_X` (0–79, R/W).
- `0x1000_2004` : `CURSOR_Y` (0–29, R/W).
- `0x1000_2008` : `CURSOR_CTRL` (bit 0 : visible, bit 1 : clignotant).
- `0x1000_200C` : `HDMI_MODE` :
- bit 0 (R/W) : `0` = mode texte 80×30, `1` = mode graphique 320×240 RGB332 (balayage DMA SDRAM via *linebuffer* BSRAM).
- bit 1 (R/W) : `gfx_page` (`0` = `0x008C_0000`, `1` = `0x008E_0000`), verrouillé sans déchirement à la ligne 523.
- bit 2 (R) : `vblank_sys` (`1` pendant le retour vertical, `v_cnt >= 480`).

2.2 CH9350 UART RX (`0x1001_0000`)

- `0x1001_0000` : `CH_STATUS` (bit 0 : `rx_ready` = FIFO non vide, bit 1 : `fifo_overflow`).
- `0x1001_0004` : `CH_DATA` (lecture : dépile un octet `[7:0]` de la FIFO ; retourne `-1` si vide).
- `0x1001_0008` : `CH_BAUD_DIV` (diviseur d'horloge UART, R/W).

2.3 SD SPI (`0x1002_0000`)

- `0x1002_0000` : `SD_DATA` (écriture : transfert SPI 8 bits ; lecture : dernier octet MISO reçu).
- `0x1002_0004` : `SD_STATUS` (bit 0 : `busy`, bit 1 : `blk_err` erreur jeton `0xFE`).
- `0x1002_0008` : `SD_CS` (bit 0 : `0` = CS actif bas, `1` = CS inactif haut).
- `0x1002_000C` : `SD_CLK_DIV` (diviseur horloge SPI : `67` ≈ 400 kHz à l'init, `0` = 27 MHz à 54 MHz).
- `0x1002_0010` : `SD_BLK_CMD` (écriture : `1` = lecture matérielle de 512 o + 2 o CRC vers `SD_BUF`, `2` = écriture matérielle de 512 o depuis `SD_BUF` ; lecture : statut moteur bloc).
- `0x1002_0200 – 0x1002_03FF` : `SD_BUF` (tampon matériel 512 o = 128 mots 32 bits little-endian).

2.4 Timer (`0x1003_0000`)

- `0x1003_0000` : `TICKS_MS` (millisecondes écoulées depuis le reset).
- `0x1003_0004` : `CYCLES_LO` (32 bits bas du compteur de cycles CPU à 54 MHz).
- `0x1003_0008` : `CYCLES_HI` (32 bits haut).
- `0x1003_000C` : `TMR_IRQ_CTRL` (`[15:0]` période IRQ en ms ou `0` pour désactiver ; écriture bit 16 = acquitter l'IRQ ; lecture bit 16 = `irq_pending`).

2.5 UART Debug (`0x1004_0000`)

- `0x1004_0000` : `UART_TX` (écriture d'un octet `[7:0]` sur TX ; en émulateur : écrit sur `stdout`).
- `0x1004_0004` : `UART_RX` (lecture d'un octet depuis la FIFO RX BSRAM 2048 o, `-1` si vide).
- `0x1004_0008` : `UART_STATUS` (bit 0 : `tx_ready`, bit 1 : `rx_ready`).
- `0x1004_000C` : `UART_BAUD_DIV` (R/W : diviseur d'horloge bit ; défaut `469` = 115200 bauds @ 54 MHz ; `59` ≈ 921600 bauds pendant `sync`).

2.6 APU Audio HDMI (`0x1005_0000`)

Sortie LPCM 2 canaux 16 bits à **32 000 Hz exacts** transportée dans les *Data Islands* HDMI 1.3a (paquets ACR, AVI InfoFrame, Audio InfoFrame, Audio Samples) :

- Canaux PSG `ch = 0..2` (base `0x1005_0000 + ch * 0x10`) :
- `+0x00` : `PHASE_INC` (`freq_hz * 524`).
- `+0x04` : `DUR_MS` (durée restante en ms ; `0` = muet).
- `+0x08` : `CTRL` (`[1:0]` forme d'onde `0`=carrée/`1`=triangle/`2`=scie/`3`=bruit LFSR, `[7:4]` volume `0..15`, `[8]` atténuation automatique).
- Canal 3 PCM / DMA SDRAM :
- `0x1005_0030` : `AUDIO_PCM_DATA` (W : pousse un échantillon 16 bits signé dans la FIFO BSRAM 1024×16b).
- `0x1005_0034` : `AUDIO_PCM_CTRL` (W : `[3:0]` diviseur de cadence `1`=32 kHz, `2`=16 kHz, `4`=8 kHz, `[16]` vider FIFO ; R : `{19'd0, empty, full, count[10:0]}`).
- `0x1005_0038` : `AUDIO_DMA_ADDR` (offset SDRAM sur 23 bits).
- `0x1005_003C` : `AUDIO_DMA_LEN` (longueur en octets sur 23 bits).
- `0x1005_0040` : `AUDIO_DMA_CTRL` (`[0]` activation, `[1]` boucle, `[2]` 16 bits signé, `[7:4]` diviseur de cadence ; R bit 0 = actif).

---

3. Format des exécutables `.cos`

En-tête fixe de **20 octets** (*little-endian*), suivi immédiatement des données à charger (`.text` aligné sur 4 octets + `.data`) :

Offset	Taille	Champ	Description
`0x00`	4	`magic`	`'C', 'O', 'S', '\0'` (`0x00534F43` en LE)
`0x04`	4	`load_addr`	Adresse de chargement en SDRAM (défaut `0x00140000`)
`0x08`	4	`entry`	Adresse du point d'entrée (`_start`)
`0x0C`	4	`load_size`	Taille en octets de `.text + .data` dans le fichier
`0x10`	4	`bss_size`	Taille de `.bss` à mettre à zéro après `load_addr + load_size`

Au démarrage d'un programme utilisateur :

- `sp` = haut de la pile après empilement de `argv` depuis `0x0073FFF0`.
- `r1` = `argc`, `r2` = `argv`.

---

4. Table des appels système (SYSCALL)

Convention : `r7` = numéro du syscall, arguments dans `r1..r4`, valeur de retour dans `r1`.

N° (r7)	Nom	Arguments	Retour (r1)
0	exit	r1 = code	Ne retourne pas (rend la main au shell / quitte emu)
1	putc	r1 = ch	0 (suit la redirection stdout dans le noyau SD)
2	puts	r1 = str	0 (suit la redirection stdout dans le noyau SD)
3	getc	—	Caractère clavier bloquant (non affecté par < ou |)
4	kbhit	—	1 si touche disponible, 0 sinon
5	con_setxy	r1 = x, r2 = y	0
6	con_setattr	r1 = attr (bg<<4 | fg)	0
7	con_clear	—	0
8	con_write_cell	r1 = x, r2 = y, r3 = ch, r4 = attr	0
10	open	r1 = path, r2 = flags	fd >= 0 ou -1
11	close	r1 = fd	0 ou -1
12	read	r1 = fd, r2 = buf, r3 = len	Octets lus (0 = EOF si fd 0 redirigé) ou -1
13	write	r1 = fd, r2 = buf, r3 = len	Octets écrits (fd 1 suit redirection, fd 2 toujours console) ou -1
14	seek	r1 = fd, r2 = offset, r3 = whence	Nouvelle position ou -1
15	unlink	r1 = path	0 ou -1
16	mkdir	r1 = path	0 ou -1
17	rmdir	r1 = path	0 ou -1
18	readdir	r1 = fd, r2 = ent_buf	1 si entrée lue, 0 fin, -1 erreur
19	chdir	r1 = path	0 ou -1
20	getcwd	r1 = buf, r2 = len	buf ou 0
21	stat	r1 = path, r2 = st_buf	0 ou -1
30	sd_read_sector	r1 = lba, r2 = buf	0 ou -1
31	sd_write_sector	r1 = lba, r2 = buf	0 ou -1
32	sd_get_sectors	—	Nombre de secteurs de 512 o

N° (r7)	Nom	Arguments	Retour (r1)
33	sd_get_stats	r1 = out[4] (NULL = reset)	0 (rd_cnt, wr_cnt, rd_ms, wr_ms)
34	audio_bg	r1 = cmd, r2..r4 = args	0/état ou -1 (streaming/RAM DMA WAV, noyau SD)
40	exec	r1 = path, r2 = argc, r3 = argv	Code de sortie du fils ou -1
41	sbrk	r1 = increment	Ancien break ou -1
42	ticks_ms	—	Millisecondes depuis le boot

Drapeaux `open` et `seek`

- `O_RDONLY = 0`, `O_WRONLY = 1`, `O_RDWR = 2`, `O_CREAT = 4`, `O_TRUNC = 8`, `O_APPEND = 16`.
- `SEEK_SET = 0`, `SEEK_CUR = 1`, `SEEK_END = 2`.
- Descripteurs standard pré-ouverts : `0` (`stdin`), `1` (`stdout`), `2` (`stderr`).

CH. 16 | Brochage Tang Nano 20K et chaîne FPGA

Affectation des broches .cst, SDRAM interne GW2AR-18 et commandes Yosys / nextpnr · Source : docs/pinout.md

Cible FPGA : **Gowin GW2AR-LV18QN88C8/I7** (famille `GW2A-18C` dans `nextpnr-himbaechel` / `gowin_pack`).

1. Tableau de brochage (`.cst`)

Signal	Broche	Interface / Rôle	Contraintes recommandées
`clk_27m`	`4`	Oscillateur quartz 27 MHz	`IO_TYPE=LVCMOS33 PULL_MODE=NONE`
`btn_s1`	`88`	Bouton S1 (reset actif haut ou bas selon PCB)	`IO_TYPE=LVCMOS33 PULL_MODE=DOWN`
`btn_s2`	`87`	Bouton S2	`IO_TYPE=LVCMOS33 PULL_MODE=DOWN`
`tmds_clk_p` / `n`	`33` / `34`	Horloge différentielle HDMI (vidéo 640×480 + audio Data Islands)	`IO_TYPE=LVCMOS33D PULL_MODE=NONE`
`tmds_d_p[0]` / `n[0]`	`35` / `36`	HDMI Data 0 (Bleu)	`IO_TYPE=LVCMOS33D PULL_MODE=NONE`
`tmds_d_p[1]` / `n[1]`	`37` / `38`	HDMI Data 1 (Vert)	`IO_TYPE=LVCMOS33D PULL_MODE=NONE`
`tmds_d_p[2]` / `n[2]`	`39` / `40`	HDMI Data 2 (Rouge)	`IO_TYPE=LVCMOS33D PULL_MODE=NONE`
`ch9350_rx`	`76`	UART RX depuis module clavier CH9350	`IO_TYPE=LVCMOS33 PULL_MODE=UP`
`sd_clk`	`83`	MicroSD SPI CLK	`IO_TYPE=LVCMOS33 PULL_MODE=NONE DRIVE=8`
`sd_mosi` (`CMD`)	`82`	MicroSD SPI MOSI	`IO_TYPE=LVCMOS33 PULL_MODE=UP DRIVE=8`
`sd_miso` (`DAT0`)	`84`	MicroSD SPI MISO	`IO_TYPE=LVCMOS33 PULL_MODE=UP`
`sd_cs` (`DAT3`)	`81`	MicroSD SPI CS	`IO_TYPE=LVCMOS33 PULL_MODE=UP DRIVE=8`
`uart_tx`	`69`	UART Debug TX (vers BL616 USB-UART)	`IO_TYPE=LVCMOS33 PULL_MODE=UP DRIVE=8`
`uart_rx`	`70`	UART Debug RX (depuis BL616 USB-UART)	`IO_TYPE=LVCMOS33 PULL_MODE=UP`

Signal	Broche	Interface / Rôle	Contraintes recommandées
`led[5:0]`	`20,19,18,17,16,15`	LEDs de diagnostic	`IO_TYPE=LVCMOS33`

> **Note SDRAM interne** : la SDRAM 64 Mbit (32 bits) du GW2AR-18 est câblée en interne via les primitives dédiées de la puce (ports `O_sdram_*` / `IO_sdram_dq` déclarés dans le top-level Gowin sans `IO_LOC` externe).

---

2. Chaîne de synthèse et flashage (`~/oss-cad-suite/bin`)

```
# 1. Synthèse RTL
yosys -p "read_verilog rtl/...; synth_gowin -top top_tang20k -json build/soc.json"

# 2. Placement & routage (horloges contraintes par rtl/clocks.py : clk_sys=54 MHz, clk_pix=25,2 MHz)
nextpnr-himbaechel --pre-pack rtl/clocks.py --json build/soc.json --write build/soc_pnr.json \
  --device GW2AR-LV18QN88C8/I7 --vopt family=GW2A-18C --vopt cst=rtl/tangnano20k.cst

# 3. Génération du bitstream
gowin_pack -d GW2A-18C -o build/cos.fs build/soc_pnr.json

# 4. Chargement SRAM (volatil) ou Flash (persistant)
openFPGALoader -b tangnano20k build/cos.fs
openFPGALoader -b tangnano20k -f build/cos.fs
```

CH. 17 | Périphériques du SoC et API C <cos.h>

Vue synthétique des contrôleurs matériels et interface de programmation utilisateur · Source : docs/man/hw.md

Peripheriques mappes en memoire (SoC 54 MHz)

- **HDMI Texte & Graphique** (`0x10000000`) :
- Mode texte 80x30 (16 couleurs CGA) + curseur materiel (`0x10002000..0x10002008`).
- Mode graphique 320x240 RGB332 (256 couleurs), double buffering (`0x008C0000` / `0x008E0000`) et sync VSYNC via `HDMI_MODE` (`0x1000200C`).
- **APU Audio HDMI 32 kHz** (`0x10050000`) : 3 canaux PSG + FIFO PCM 1024 ech. + DMA SDRAM en arriere-plan (Data Islands HDMI 1.3a).
- **Clavier CH9350L** (`0x10010000`, broche 76) : UART 115200 bauds + FIFO 64 o, trames `57 AB 88` converties en AZERTY.
- **Carte SD SPI** (`0x10020000`) : horloge 27 MHz (`SD_CLK_DIV=0`), moteur bloc matériel 512 o (`SD_BLK_CMD`/`SD_BUF`) et cache FAT write-back.
- **Timer** (`0x10030000`) : `TICKS_MS`, compteur de cycles 54 MHz et interruption periodique materielle (`TMR_IRQ_CTRL` 50 ms).
- **UART Debug BL616** (`0x10040000`) : miroir console + canal `sync` avec FIFO RX BSRAM 2048 o et debit programmable (`UART_BAUD_DIV`, jusqu'a 921600 bauds).

Fonctions C dans `<cos.h>`

- Console : `sys_putc`, `sys_puts`, `sys_getc`, `sys_kbhit`, `sys_con_clear`, `sys_con_setxy`, `sys_con_setattr`, `sys_con_write_cell`.
- Fichiers & SD : `sys_open`, `sys_close`, `sys_read`, `sys_write`, `sys_seek`, `sys_unlink`, `sys_mkdir`, `sys_rmdir`, `sys_readdir`, `sys_chdir`, `sys_getcwd`, `sys_stat`, `sys_sd_stats`.
- Systeme : `sys_exec`, `sys_sbrk`, `sys_ticks_ms`, `sys_exit`.
- Graphique & Audio : `gfx_*` et `snd_*` (voir `man gfx` et `man audio`).

CH. 18 | Noyau C, Boot ROM et /boot/kernel.bin

Architecture dual-target, chainloading depuis SD, VFAT LFN et recompilation sur carte · Source : docs/man/kernel.md

Le noyau du systeme est entierement ecrit en C (repertoire `/src/kernel/`), compile avec `cos-cc` et lie avec `cos-as`.

Boot ROM (64 Ko) vs Noyau etendu SD (`-D SD_KERNEL`)

- **Boot ROM** (`0x00000000..0x0000FFFF`) : noyau minimal en BSRAM FPGA permettant de monter la SD, lancer `sync` et chainloader `/boot/kernel.bin`.
- **Noyau SD** (`/boot/kernel.bin`, charge en `0x00108000..0x0013BFFF`, 208 Ko max) : ajoute LFN, historique, `Tab`, `;`/`&&`/`||`, variables d'env + `$PATH`, pipes RAM, redirections, `sh`, IRQ timer 50 ms (`Ctrl-C`/`Ctrl-X`/`F12` globaux), streaming audio SD et `sync` 921600 bauds.

Organisation du noyau (`/src/kernel`)

- `kmain.c` : init console, clavier, SD, FAT32 et chainload `/boot/kernel.bin`.
- `shell.c` : shell, historique, `Tab`, pipes, redirections et interpreteur `sh`.
- `syscall.c` : appels systeme `0..42` (`33 = sd_get_stats`, `34 = audio_bg`), `k_exec` et redirections stdio / pipes RAM.
- `audio_bg.c` : gestionnaire tache de fond (RAM DMA <= 1 Mo et streaming continu SD sur anneau DMA).
- `sync.c` : serveur USB-UART (commandes `I`, `B` baud, `F` format, `S` stat, `R`/`W`/`D`/`M`).
- `drivers/` & `fs/fat32.c` : pilotes HDMI, CH9350, SD SPI bloc et FAT32.

Recompiler depuis le PC ou sous COS

- PC : `make sd-kernel` puis `make sync` (ou `make deploy`).
- Auto-heberge sous COS : compiler avec `cc -S -I /src/kernel -D SD_KERNEL` puis lier avec `as --nostdlib --base 0x00108000 -o /boot/kernel.bin /src/kernel/sdcrt0.s ...`.

PARTIE V

Feuille de route & Jalons

Historique de conception par phases et suivi de réalisation du projet.

Chapitre 19 — Plan d'implémentation et jalons du projet
Choix architecturaux fondateurs et bilan d'avancement des phases 0 à 5 [PLAN.md] · p. 55

CH. 19 | Plan d'implémentation et jalons du projet

Choix architecturaux fondateurs et bilan d'avancement des phases 0 à 5 · Source : PLAN.md

Ce document suit la feuille de route définie à partir de cdc.txt.

Choix architecturaux validés

1. **Processeur 32 bits (« COS-32 ») + SDRAM interne 8 Mo** : espace d'adressage plat 32 bits *little-endian* exploitant les 8 Mo de SDRAM embarquée du GW2AR-18.
2. **Clavier CH9350 (broche 76)** : réception UART + FIFO matériel en Verilog ; décodage des trames et table **AZERTY** côté noyau C.
3. **Chaîne FPGA open-source** : `~/oss-cad-suite` (`yosys`, `nextpnr-himbaechel`, `gowin_pack`, `iverilog`, `verilator`, `openFPGALoader`).
4. **Démarrage noyau** : noyau minimal flashé en BSRAM avec chargement optionnel de `/boot/kernel.bin` depuis la carte SD (repli automatique sur le noyau flashé si absent).

Phasage et avancement

☑ **Phase 0 — Spécifications & squelette**

☑ Spécification ISA COS-32 (docs/isa.md)

☑ Carte mémoire, format `.cos` et ABI syscalls (docs/memmap_abi.md)

☑ Brochage Tang Nano 20K (docs/pinout.md)

☑ `Makefile` racine

☑ **Phase 1 — Émulateur PC & chaîne de compilation croisée (`emu`, `as`, `cc`)**

☑ Émulateur PC `tools/emu/emu.c` (modes `--rom` et `--user`)

☑ Assembleur 2 passes `toolchain/as/as.c` (compatible auto-hébergement)

☑ Suite de tests ISA/ASM `tests/asm/`

☑ Compilateur mini-C `toolchain/cc/cc.c` (compatible auto-hébergement) + tests C (`libc/` + `as_native.cos`)

☑ **Phase 2 — Noyau matériel HDL (Tang Nano 20K)**

☑ Contrôleur HDMI texte 80×30 (640×480@60Hz, 16 couleurs)

☑ Cœur COS-32 + Boot ROM + UART debug (`rtl/cpu/cos32_cpu.v`, `rtl/cos_soc.v`)

☑ Contrôleur SDRAM 8 Mo interne (`rtl/mem/sdram_ctrl.v`)

☑ Contrôleur RX CH9350 (broche 76) & contrôleur SD SPI

☑ Simulation RTL (`tests/rtl/tb_soc.v`), synthèse (`build/cos.fs`) & commande `make flash` (`openFPGALoader`)

☑ **Phase 3 — Noyau logiciel minimal & FAT32**

- ☑ Pilotes HDMI, CH9350 AZERTY, SD SPI (`kernel/drivers/`)
- ☑ Pilote FAT32 complet (`kernel/fs/fat32.c`) + chargement optionnel `/boot/kernel.bin`
- ☑ Syscalls, chargeur `.cos`, mini-shell (`kernel/syscall.c`, `kernel/shell.c`, `kernel/crt0.s`) & Boot ROM 64 Ko stabilisée (`build/cos.fs`)
- ☑ **Phase 4 — Utilitaires `/bin`, éditeur `ed` & `make deploy`**
- ☑ Commandes `md`, `rd`, `cp`, `rm`, `ls`, `cd`, `format`, `free`, `df`
- ☑ Éditeur plein écran `ed` avec coloration syntaxique C
- ☑ Déploiement USB complet (`make deploy`) et sync incrémentale SD uniquement (`make sync`)
- ☐ **Phase 5 — Auto-hébergement (*Self-Hosting*)**
- ☑ Portage natif `/bin/as` et `/bin/cc` + sources sur carte SD (`/src/bin`, `/src/libc`, `/src/tools`, `/src/kernel`, `/src/examples`)
- ☐ Recompilation complète de `cc`, `as`, des utilitaires et de l'OS sur le Tang Nano 20K