EPFL

MICROCONTRÔLEURS – Enseignant : Pr. Schmid Alexandre

# Rapport : Stations de Pointage de Satellites

**Jean Aboutboul (375057)** **Lars Clausen (399191)**

Génie Microtechnique | Groupe : 111 | 26 mai 2026

## 1. Résumé

Le présent projet s'inspire des technologies spatiales et robotiques pour implémenter une station de pointage d'antenne satellitaire à deux axes. Directement inspirée par le réseau ESTRACK de l'ESA, notre station a pour objectif d'optimiser le bilan de liaison avec des satellites (hypothétiques) en orbite.

L'intérêt fondamental de ce dispositif réside dans sa capacité à orienter une antenne directive pour concentrer un signal, maximisant ainsi l'énergie transmise et reçue. Au-delà du simple ciblage, le système est conçu pour pallier la perte de coordonnées orbitales en détectant activement les satellites dans son environnement, puis en mémorisant leur position — garantissant le maintien d'une communication stable et efficace.

### 1. Description générale

Le système repose sur une architecture embarquée pilotée par un ATmega128L (STK-300, 4 MHz). Deux axes contrôlent le pointage : l'**azimut** (rotation libre [0°, 360°]) assuré par un stepper X27 à haute résolution angulaire (1/3° par pas), et l'**élévation** ([0°, 90°]) contrôlée par un servo S3003 via PWM matériel autonome. L'interface utilisateur s'articule autour d'un LCD 2×16 mappé en mémoire, du décodage par interruption du protocole infrarouge RC5 issu de la télécommande, et d'un lien UART optionnel permettant le contrôle depuis un terminal PC.

Le code est structuré autour d'une **FSM à deux couches** imbriquées — navigation (sélection de mode) et exécution matérielle — séparées en `drivers/` (accès matériel) et `libs/` (logiciel pur). Cette conception modulaire découple la gestion des entrées/sorties physiques de la logique applicative. Trois modes opératoires définissent les capacités du système :

— **Mode MANUEL** — Contrôle direct de la cinématique de l'antenne. Un système d'accélération algorithmique à trois paliers adapte dynamiquement la vitesse de balayage en fonction de la durée de maintien de la touche (1°, 5° puis 10° par répétition). Une fonction de sauvegarde à la volée permet d'enregistrer toute position dans la base de données.

— **Mode AUTO** — Système de persistance CRUD exploitant l'EEPROM interne pour gérer une base de 32 positions satellites (dont 10 pré-enregistrées). Lors de la sélection d'une cible, la station calcule le chemin le plus court (modulo 360°) et exécute un *Go To* séquentiel automatique (azimut puis élévation), minimisant le temps de latence avant l'établissement de la liaison.

— **Mode SCAN** — Balayage autonome de 360° autour de l'élévation courante. L'algorithme compare chaque degré parcouru aux positions de la base de données (pré-chargée en cache SRAM) et restitue en fin de cycle un rapport de détection filtré, navigable comme en mode AUTO.

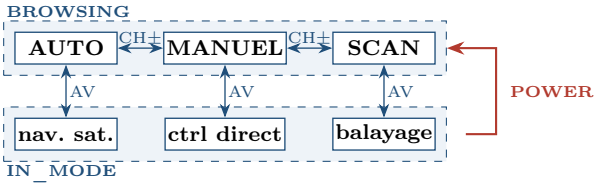
# 2. Mode d'emploi

## 1. Matériel

Carte STK-300 (ATmega128L, RC interne 4 MHz). Télécommande IR Vivanco UR Z2 (mode TV, RC5) sur PE7. Stepper JST X27 (azimut) sur PORTD[0 :3]. Servo Futaba S3003 (élévation) sur PB5/OC1A. Buzzer piézo sur PE2. LCD HD44780 16×2 mappé sur bus externe (0x8000 cmd / 0xC000 data). UART USART0 sur PE0/PE1, 19 200 bps.

## 2. Architecture de navigation

Deux couches superposées. POWER reste prioritaire en toutes circonstances.




## 3. Mise sous tension

Au reset : LCD ligne 1 = `MODE: AUTO`, ligne 2 = `AZ:000 EL:000` (positions courantes). Servo mis à 0° (OCR1A=500, 1000 µs). Stepper à 0 (mettre à la main la flèche, elle doit pointer vers la carte perpendiculaire à celle-ci). Timer2 tick 1 ms. UART OFF. Couche BROWSING.

## 4. Mode MANUEL

Contrôle direct des axes. L1 : `MANUEL MOVING`. L2 : `AZ:xxx EL:xxx`.

| | |
|---|---|
| `CH+ / CH−` | Azimut ±1° par appui |
| `VOL+ / VOL−` | Élévation ± 1 cran (≈ 2°) |
| `GUIDE` | Sauvegarder la position courante |
| `AV` | Sortir du mode |

**Accélération par maintien :** $< 1{,}1$ s → 1°/rep · $< 3{,}4$ s → 5°/rep · $\geq 3{,}4$ s → 10°/rep. Azimut circulaire ; élévation bornée $[0°, 90°]$.

**Sauvegarder :** GUIDE → `SAVE:PRESS GUIDE` → GUIDE×2 confirme (1er slot libre, max 32). ou → CANCEL n'importe quelle autre touche.

## 5. Mode AUTO

Navigation dans la base satellites. Affichage : `Sat:<nom>` / `AZ:xxx EL:xx`.

| | |
|---|---|
| `CH+ / CH−` | Satellite suivant / précédent |
| `0-9` | Accès direct au slot |
| `GUIDE` | Lancer GO TO |
| `DEL` ×2 | Supprimer (affiche `DEL? PUSH DEL`) |
| `AV` | Sortir |

**GO TO :** azimut en premier (chemin court ±180°), puis élévation. Toute touche bloquée sauf POWER.

## 6. Mode SCAN

Balayage 360° pour détecter les satellites enregistrés visibles.

**IDLE** `MODE: SCAN` / `SCAN: PUSH GUIDE`. GUIDE lance le balayage.

**SCANNING** Affichage `SCANNING AZ:xxx`. Rotation 1°/cycle (3 pas partiels). Détection : AZ exact *et* EL ±10° par rapport à l'EL au lancement. Hit → chirp + `N SATS DETECTED` + pause 1 s. Fin auto après 360° (1080 pas).

**NO RESULTS** `NO SATS FOUND` ; AV pour sortir.

**RESULTS** Navigateur filtré (comme AUTO mais seuls les hits) ; GUIDE = GO TO, AV = sortir.

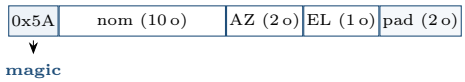
## 7. Contrôle UART

USART0 à 19 200 bps. Bascule via `MUTE` (télécommande) ou `U` (clavier) → affiche `UART ON` / `UART OFF`. OFF au boot. Quand actif, le terminal affiche en miroir l'état du LCD + couche + mode + position + activité (codes ANSI VT100).

| | | | |
|---|---|---|---|
| `W/S` | VOL± | `M` | AV |
| `A/D` | CH± | `X` | DEL |
| `Space` | GUIDE | `P` | POWER |
| `0-9` | slot direct | `L` | lister sats |
| | | `U` | UART ON/OFF |

## 8. Base satellites (EEPROM)

32 slots × 16 o. Structure d'un slot :




Slot vide : magique ≠ 0x5A. Suppression : magique → 0xFF. Préprogrammés : ISS, Hubble, Tiangong, Starlink, Envisat, NOAA-19, Iridium, Terra, Aqua, Suomi NPP.

## 9. Signaux sonores

| | |
|---|---|
| `1 bip (2 notes)` | Confirmation (AV, sauvegarde) |
| `1 chirp` | Satellite détecté pendant scan |
| `Double bip` | Arrêt d'urgence POWER |

**POWER — arrêt d'urgence.** Prioritaire en toutes circonstances. Coupe immédiatement le servo et le stepper, émet un double bip, retour BROWSING. Reste actif pendant les déplacements automatiques (GO TO, SCAN).

### Limites mécaniques.

**Azimut :** rotation libre 360° continu, wrap-around modulo 1080 pas.

**Élévation :** bornée $[500, 900]$ ticks = $[1000, 1800]$ µs = $[0°, 90°]$. Commandes hors limite ignorées.

**Pendant GO TO ou SCAN :** navigation bloquée, seul POWER actif.

# 3. Description Technique

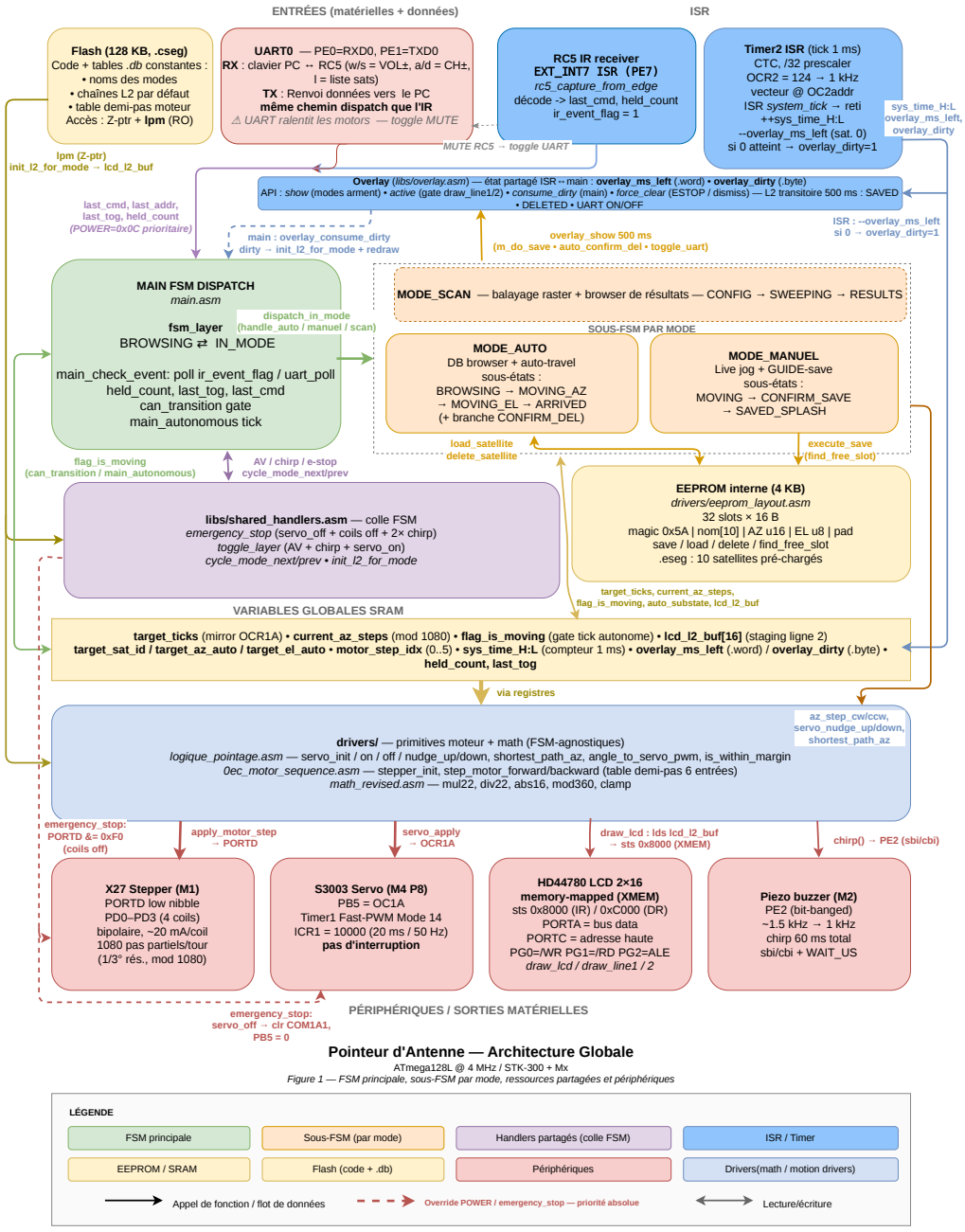



*Figure 1 — FSM principale, sous-FSM par mode, ressources partagées et périphériques*

## 1. Fonctionnement top-down

Le programme suit une **boucle événementielle coopérative** à entrées mixtes. L'IR est capturé par interruption (INT7) et signalé via un flag ; l'UART est interrogé par polling non-bloquant. Chaque itération de `main` exécute :

1. **Overlay** — `overlay_consume_dirty` teste si le Timer2 a expiré un message transitoire. Si oui, `init_l2_for_mode` restaure l'affichage normal.
2. **UART** — `uart_poll` teste `RXC0` (non-bloquant). Si un octet est disponible, il est traduit en commande RC5 équivalente (`last_cmd`).
3. **IR** — test de `ir_event_flag` (levé par l'ISR INT7). Si = 1, le flag est acquitté et le dispatch commence.
4. **Tick autonome** — si aucune entrée mais `flag_is_moving` = 1, un pas de mouvement GO TO ou SCAN est exécuté.
5. **Dispatch par priorité** —
   — POWER → `emergency_stop` (single-fire)
   — Overlay dismiss (appui frais → annulation immédiate)
   — MUTE → bascule UART
   — AV → `toggle_layer`

— BROWSING : CH± → cycle mode
— IN_MODE : forward au handler actif

6. **Rendu** — `draw_lcd` + miroir UART.

L'UART et l'IR convergent vers la même variable `last_cmd`, unifiant le dispatch sans duplication. Pendant un overlay actif, le dispatch des handlers de mode est gelé pour protéger le contenu de `lcd_l2_buf`.

## 2. Interruptions et concurrence

Le système utilise deux sources d'interruption :

| Source | Vecteur | Rôle |
|---|---|---|
| INT7 | `INT7addr` | Capture RC5 sur front descendant PE7 |
| Timer2 | `OC2addr` | Tick 1 ms, décompte overlay |

**INT7 (RC5)** est configurée sur front descendant (`ISC71` dans `EICRB`). L'ISR `EXT_INT7` sauvegarde 9 registres, appelle `rc5_capture_from_edge` (25 ms bloquant en ISR), valide les bits de start, extrait commande/toggle, met à jour `held_count`, puis lève `ir_event_flag`. Le contexte complet (SREG, r16–r19, b0–b2, u) est restauré avant `reti`. Pendant ces 25 ms les interruptions globales sont masquées (comportement par défaut AVR : `cli` implicite à l'entrée d'ISR). Les occurrences Timer2 lèvent le flag `OCF2` mais l'ISR `system_tick` ne s'exécute qu'au `reti` d'INT7. Cela peut retarder un décompte overlay d'environ 25 ms — acceptable pour les durées d'overlay de 500 ms utilisées.

**Timer2 (overlay)** décrémente `overlay_ms_left` (16 bits, atomique car en ISR). Quand le compteur atteint 0, `overlay_dirty` est levé à 1. La boucle principale le consomme via `overlay_consume_dirty` et restaure l'affichage normal. Les accès 16 bits depuis le main sont protégés par `cli/sei`.

## 3. FSM à deux couches

La navigation repose sur deux couches encodées par `fsm_layer` :

**BROWSING (0)** Sélection du mode par CH± sans activer les moteurs. AV entre dans le mode.

**IN_MODE (1)** Toutes les touches sont déléguées au handler du mode courant. AV revient à BROWSING.

POWER force le retour à BROWSING depuis n'importe quel état, coupe les deux moteurs, efface tout overlay actif, et remet les sous-états à zéro. Le verrou `can_transition` (test de `flag_is_moving`) bloque tout changement de couche ou de mode pendant un déplacement automatique.

## 4. Sous-FSM par mode

| | |
|---|---|
| **AUTO** | BROWSING → MOVING_AZ → MOVING_EL → ARRIVED. Branche CONFIRM_DEL (double-appui). |
| **MANUEL** | JOGGING (0) → CONFIRM_SAVE (1) → `execute_save` + overlay 500 ms → retour JOGGING. Accélération par paliers via `held_count`. |
| **SCAN** | IDLE → SWEEPING (360°, 3 pas/deg) → NO_RESULTS ou RESULTS_BROWSE (navigateur filtré par bitmap 32 bits). |

Le GO TO (AUTO) déplace l'azimut en premier via `shortest_path_az` (±180° chemin court), puis l'élévation. Le SCAN pré-charge les 32 slots EEPROM dans un cache SRAM de 100 o pour éviter ∼46 k lectures pendant le balayage.

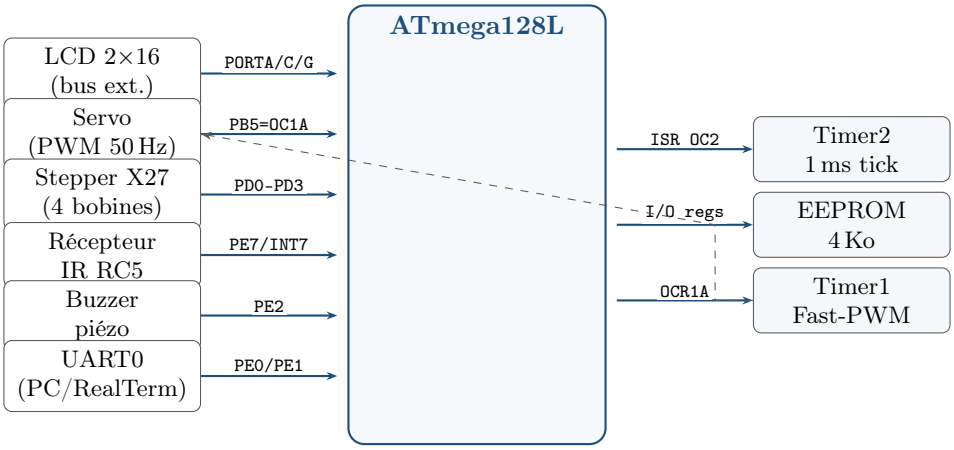
## 5. Présentation des modules

Le code (∼5 600 lignes, 20 fichiers) est structuré en deux répertoires reflétant la séparation matériel/logiciel :

| Fichier | Responsabilité |
|---|---|
| `drivers/` — accès matériel direct | |
| `rc5_logic.asm` | Capture RC5 14 bits, extraction champs |
| `timer_manager.asm` | Init Timer2 CTC, ISR `system_tick` |
| `logique_pointage.asm` | Servo init/on/off/nudge, `shortest_path_az` |
| `0ec_motor_seq.asm` | Table demi-pas 6 motifs, step fwd/bwd |
| `buzzer.asm` | Chirp bit-bang 2 notes (PE2) |
| `uart_control.asm` | Polling RX, traduction clavier, miroir VT100 |
| `eeprom_layout.asm` | CRUD 32 slots, `find_free_slot`, `execute_save` |
| `math_revised.asm` | `mul22`, `div22`, `abs16`, `mod360`, `clamp` |
| `lcd.asm` | Driver LCD XMEM (sts 0x8000/0xC000) |
| `libs/` — logiciel pur (FSM, affichage, glue) | |
| `overlay.asm` | Overlay temporisé L2 (show / active / consume) |
| `shared_handlers.asm` | `emergency_stop`, `toggle_layer`, `is_slot_visible` |
| `display.asm` | Rendu LCD L1/L2, formatage dynamique |
| `mode_auto.asm` | Navigation DB, GO TO séquentiel, suppression |
| `mode_manuel.asm` | Move AZ/EL, sauvegarde GUIDE |
| `mode_scan.asm` | Balayage 360°, cache, navigateur résultats |

**Découplage.** Les `drivers/` exposent des primitives réutilisables (`servo_nudge_up`, `step_motor_forward`, `chirp`) sans connaissance de la FSM. Les `libs/` orchestrent ces primitives selon leur sous-FSM. Les variables SRAM partagées (`target_ticks`, `current_az_steps`, `flag_is_moving`, `ir_event_flag`) sont toutes déclarées dans le `.dseg` de `main.asm`, formant le contrat d'interface unique entre les deux couches.

# 4. Accès aux Périphériques et Références




Architecture matérielle — connexion des périphériques à l'ATmega128L.
Pointillée : Timer1 pilote PB5/OC1A directement via le hardware, sans intervention logicielle continue.

---

## 1. LCD — Interface mémoire externe

Le LCD n'est pas connecté sur un port GPIO standard (DDRX/PORTX) mais exploite le bus SRAM externe de l'ATmega : données et adresses basses multiplexées sur PORTA, adresses hautes sur PORTC, signaux de contrôle (`/WR`, `/RD`, `ALE`) sur PORTG. Activer le bit `SRE` de `MCUCR` délègue la gestion de ces broches au hardware XMEM dès le boot.

Le LCD est ainsi vu comme deux cases mémoire : 0x8000 (registre de commandes, IR) et 0xC000 (registre de données, DR). Une simple instruction `sts` vers l'une de ces adresses génère automatiquement les signaux `/WR` et `ALE` — aucun bit-bang manuel requis.

Avant toute écriture, le busy flag (bit 7 de l'IR) est sondé : tant qu'il est à 1, toute écriture serait ignorée.

```
LCD_wr_ir:            ; in: w = commande
  lds  u, LCD_IR      ; lit IR (bit7=busy)
  sbrc u, 7           ; saute si flag = 0
  rjmp LCD_wr_ir      ; boucle si occupé
  rcall lcd_4us       ; délai DDRAM
  sts  LCD_IR, w      ; envoie la commande
```

L'initialisation envoie 4 commandes : nettoyage (0x01), mode d'entrée (0x06), affichage actif (0x0C), interface 8 bits 2 lignes (0x38). Le curseur se positionne par `0x80 | adresse`, avec `0x00-0x0F` (ligne 1) et `0x40-0x4F` (ligne 2).

## 2. Servo — Timer1 Fast-PWM (PB5)

Le servo est connecté sur PB5 = OC1A. Il est contrôlé par la télécommande RC5 via les boutons VOL+ / VOL-, qui modifient la valeur `OCR1A` - le registre qui encode directement la largeur d'impulsion PWM. C'est ce paramètre seul qui détermine l'angle de la tête.

Le Timer1 en mode 14 (Fast-PWM, TOP = ICR1) génère un signal à 50 Hz en autonome. Avec un prescaler de 8, chaque tick vaut 2 µs. `ICR1` = 10 000 donne une période de 20 ms.

| OCR1A (ticks) | Angle servo |
|---|---|
| 500 (1,0 ms) | 0° (bas) |
| 750 (1,5 ms) | 45° (centre) |
| 900 (1,8 ms) | 90° (haut) |

```
; VOL+ appuyé -> servo_nudge_up:
;   target_ticks += TICKS_STEP
;   clamp à TICKS_MAX
out OCR1AH, r25    ; écriture double-
out OCR1AL, r24    ; buffered (anti-glitch)
```

> **PWM matériel :** une fois Timer1 configuré, le hardware génère le signal sans ISR ni bit-bang. Le CPU reste libre pour réagir aux trames RC5 — zéro jitter sur l'angle.

L'arrêt d'urgence (bouton POWER) masque le bit `COM1A1` dans `TCCR1A`, déconnectant OC1A de PB5 et forçant la broche à 0. Le servo relâche son couple en ∼50 ms.

## 3. Stepper — GPIO et table LUT (PD0–PD3)

Le stepper contrôle l'azimut de l'antenne et est piloté depuis PD0–PD3. Côté télécommande, les boutons CH+ / CH- déclenchent respectivement une rotation horaire ou antihoraire — chaque appui commandant un nombre de pas proportionnel

à la durée de maintien.

Pour faire tourner le rotor, les 4 bobines doivent être activées dans un ordre précis (séquence demi-pas). Plutôt que de coder ces motifs en dur, le firmware stocke une table LUT de 6 entrées en mémoire flash et la lit avec l'instruction `lpm` via le registre Z :

```
motor_sequence_table:
  .db 0x05,0x01,0x0B,0x0A,0x0E,0x04

apply_motor_step:     ; r16 = index (0..5)
  ldi ZL, low(2*motor_sequence_table)
  ldi ZH, high(2*motor_sequence_table)
  add ZL, r16         ; Z -> motif courant
  lpm r16, Z          ; lecture depuis flash
  in  r17, PORTD
  andi r17, 0xF0      ; préserve PD4-PD7
  or  r17, r16
  out PORTD, r17      ; active les bobines
```

**Résolution :** rapport 1 :180 × 6 motifs = 1080 demi-pas/tour ⇒ 1/3° par pas. La position est stockée dans `current_az_steps` (16 bits, [0, 1080)).

**Accélération au maintien :** `pick_step_size` adapte la taille du saut selon `held_count` (nombre de trames RC5 identiques consécutives) : 1° en tap, 5° après ∼1 s, 10° après ∼3 s.

## 4. Récepteur IR RC5 — Interruption INT7 (PE7)

La télécommande utilise le protocole RC5 (14 bits, encodage Manchester à 36 kHz). Le démodulateur restitue un signal numérique sur PE7 : haut au repos, front descendant = début de trame.

La réception est gérée par interruption externe INT7, configurée sur front descendant (`ISC71` dans `EICRB`). Dès qu'un front est détecté, l'ISR `EXT_INT7` est déclenchée :

```
; Configuration INT7 (dans reset) :
ldi  w, (1<<ISC71)  ; front descendant
out  EICRB, w
ori  w, (1<<INT7)
out  EIMSK, w       ; active INT7
sei                 ; interruptions ON
```

À l'intérieur de l'ISR, `rc5_capture_from_edge` est appelée : après un centrage de $T_1/4 \approx 467\,\mu$s, les 14 bits sont échantillonnés toutes les $T_1 = 1870\,\mu$s. Une fois la trame capturée et validée, la commande et le toggle bit sont extraits et stockés en SRAM, puis un **flag** `ir_event_flag` est levé à 1. La boucle principale n'attend plus sur PE7 — elle se contente de tester ce flag à chaque itération et dispatch les commandes en conséquence.

Le toggle bit change à chaque nouvel appui physique. Si ce bit reste stable d'une trame à l'autre, `held_count` s'incrémente, permettant la détection du maintien et l'accélération progressive des moteurs.

> **Interruption vs polling :** l'ISR libère la boucle principale de toute attente active sur PE7. La latence de réaction à un appui est bornée par la durée d'une itération de boucle, négligeable devant la période RC5 (∼113 ms).

## 5. Timer2 — Tick système 1 ms (CTC + ISR)

Timer2 en mode CTC (Clear Timer on Compare) déclenche une ISR toutes les millisecondes (vecteur `OC2addr`). L'ISR décrémente le compteur 16 bits `overlay_ms_left` ; lorsqu'il atteint 0, le flag `overlay_dirty` est levé à 1, signalant à la boucle principale que le message temporaire (SAVED, DELETED, UART ON/OFF…) a expiré et que l'affichage normal doit être restauré.

**Calcul :** 4 MHz ÷ 32 (prescaler) ÷ 125 (valeur OCR2) = **1 000 Hz**.

L'ISR sauvegarde et restaure `SREG` pour ne pas corrompre les flags du programme principal. Les écritures 16 bits vers `overlay_ms_left` depuis le contexte principal (`overlay_show`, `overlay_force_clear`) sont protégées par `cli/sei` pour garantir la cohérence si l'ISR s'intercalait entre les deux octets.

## 6. EEPROM — Base de données (32 slots)

Les 4 Ko d'EEPROM interne stockent jusqu'à 32 satellites, chacun dans un slot de 16 octets à l'adresse $id \times 16$ (calculée par 4 décalages `lsl/rol`). Un marqueur de validité 0x5A en offset 0 distingue les slots occupés (EEPROM effacée = 0xFF par défaut). Suit le nom ASCII (10 octets), l'azimut 16 bits little-endian et l'élévation sur 1 octet.

L'écriture impose une séquence critique de la datasheet : après avoir chargé adresse (`EEARH:L`) et donnée (`EEDR`), les bits `EEMWE` puis `EEWE` doivent être activés en moins de 4 cycles. Les interruptions sont donc coupées (`cli`) pendant ces deux instructions — toute interruption intercalée annulerait silencieusement l'écriture.

## 7. UART0 — Contrôle alternatif par PC (PE0/1)

L'UART0 permet de remplacer la télécommande RC5 par un clavier PC via RealTerm. Les caractères reçus sont traduits en commandes FSM identiques à celles du RC5 :

| Touche | Action équivalente RC5 |
|---|---|
| `a / d` | CH+ / CH- (azimut) |
| `w / s` | VOL+ / VOL- (élévation) |
| `m` | AV (changer de mode) |
| `p` | POWER (arrêt d'urgence) |
| `espace` | GUIDE (action positive) |
| `x` | DEL (suppression) |
| `0-9` | Accès direct au slot |
| `l` | Lister tous les satellites |
| `u` | Bascule UART ON/OFF |

La routine `uart_poll` est appelée à chaque itération de boucle principale : elle teste le flag `RXC0` de `UCSR0A` (polling non-bloquant), lit `UDR0` si un octet est disponible, et le convertit en `last_cmd` exactement comme l'ISR RC5. Le `held_count` est remis à 0 et le toggle bit inversé, simulant un appui frais. En sens inverse, chaque `LCD_putc` renvoie le caractère sur UART avec séquences ANSI (ESC[H/K/J) pour un miroir LCD propre sous RealTerm.

L'UART est activable/désactivable (touche `u` ou bouton MUTE du RC5). Cette bascule n'est pas anodine : chaque `LCD_putc` envoie plusieurs octets UART en polling bloquant, ce qui mobilise le CPU pendant plusieurs centaines de µs. Lorsque l'UART est coupé, ces envois sont court-circuités, libérant du temps CPU entre deux pas du stepper — ce qui se traduit par une rotation significativement plus rapide de l'azimut.

## 8. Buzzer — Génération sonore bit-bang (PE2)

Le buzzer piézo est piloté par bascule logicielle de PE2 (`sbi/cbi`) avec délais `WAIT_US` — aucun timer dédié, la fréquence résulte directement de la demi-période choisie.

La routine `chirp` joue deux notes successives de 30 ms chacune pour un total de ∼60 ms :

- 1500 Hz : demi-période de 333 µs, **45 cycles** × 666 µs = 30 ms
- 1000 Hz : demi-période de 500 µs, **30 cycles** × 1000 µs = 30 ms

Les deux notes ont la même durée car la fréquence plus basse est compensée par un nombre de cycles moindre : c'est un choix délibéré pour un son symétrique. Usage : transition de couche FSM (1 chirp), arrêt d'urgence (2 chirps), confirmation de sauvegarde (1 chirp).

## Références


[1] Atmel, *ATmega128/128L Datasheet* (Rev. 2467X-AVR-06/11).

[2] Vishay Semiconductors, *TSOP22../TSOP24../TSOP48../ IR Receiver Modules Datasheet*, www.vishay.com.

[3] Hitachi, *HD44780U LCD Controller Datasheet*, Rev. 0.0.

[4] Futaba, *S3003 Servo Specification Sheet*.

[5] Juken Swiss Technology, *X27 Stepper Motor Specification*, Doc. No. SP-X27-e-C.

[6] Vivanco, *Universal TV-DVB Controller UR Z2 Operating Instructions*, EDP-No. 34873.

[7] Philips, *RC5 IR Remote Control Protocol*, App. Note AN10003.

# 5. Annexes

## Table des matières

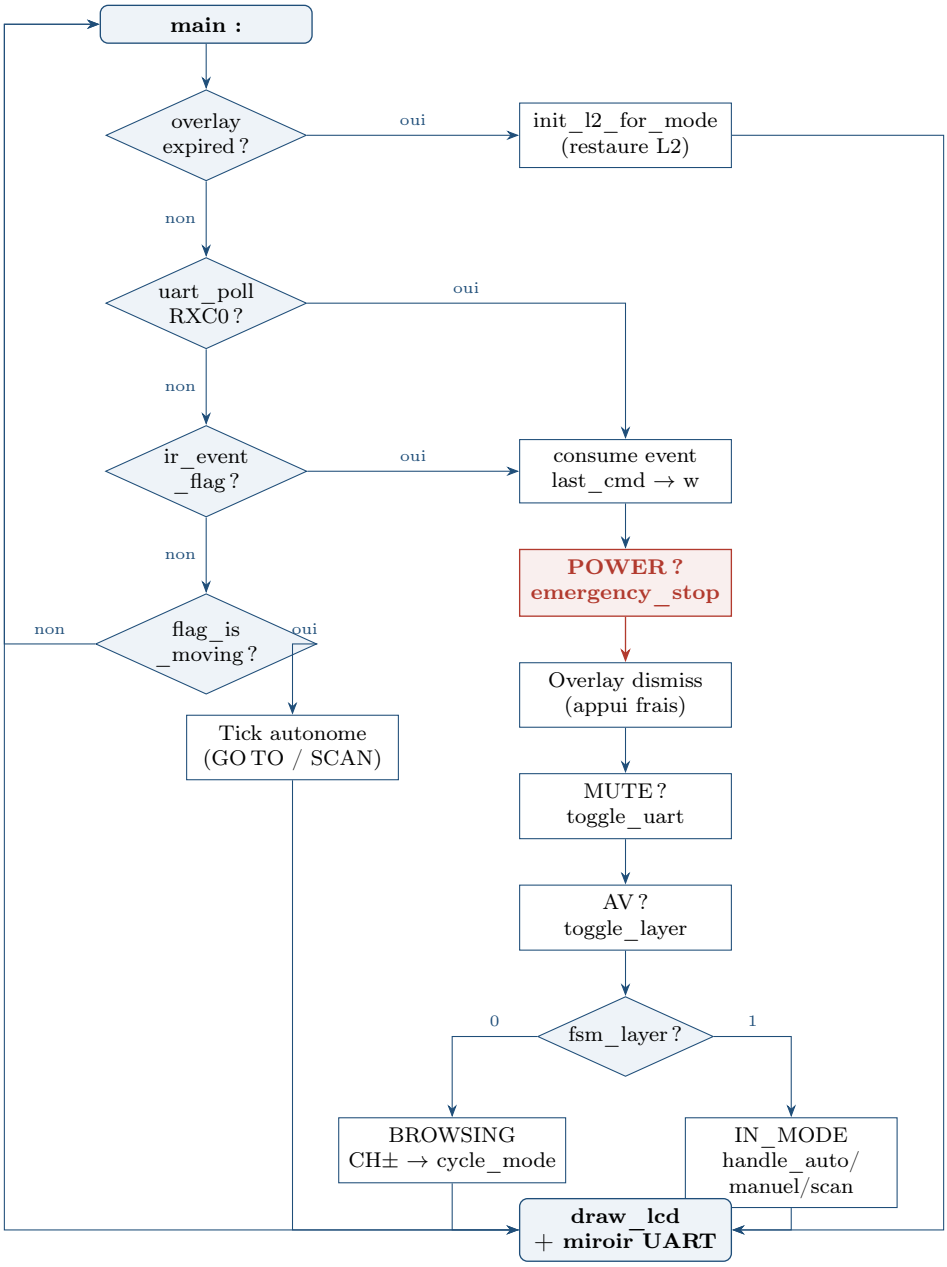
## A. Tableau récapitulatif des périphériques

| Périphérique | Port / Broche | Mode d'accès | Registres clés | Détails techniques |
|---|---|---|---|---|
| **LCD 2×16** | PORTA (données/adr. basses), PORTC (adr. hautes), PORTG (`/WR,/RD,ALE`) | Memory-mapped (`sts/lds`) | `MCUCR`(SRE, SRW10), addr. 0x8000 (IR), 0xC000 (DR) | Busy flag bit 7 obligatoire avant chaque écriture. Init : 4 commandes (0x01, 0x06, 0x0C, 0x38). Positionnement ligne 1 : 0x00–0x0F, ligne 2 : 0x40–0x4F. Miroir UART0 avec séquences ANSI. |
| **Servo Futaba S3003** | PB5 (OC1A), module M4 connecteur P8 | PWM matériel Timer1 (autonome) | `TCCR1A` (COM1A1, WGM11), `TCCR1B` (WGM13 :12, CS11), `ICR1`, `OCR1A` | Mode 14 Fast-PWM, prescaler/8 → 2 µs/tick. ICR1=10000 → 20 ms / 50 Hz. OCR1A : 500 (0°) à 900 (90°). Double-buffer anti-glitch. Arrêt : masque COM1A1, force PB5=0. |
| **Stepper JST X27** | PD0 (C1+), PD1 (C1-), PD2 (C2+), PD3 (C2-), module M1 | GPIO direct + LUT flash (`lpm`) | `PORTD`, `DDRD`, registre Z (pointeur LUT) | Table 6 motifs demi-pas en flash : {0x05, 0x01, 0x0B, 0x0A, 0x0E, 0x04}. 1080 pas/tour → 1/3°/pas. Délai 3 ms inter-pas. Position SRAM : `current_az_steps` 16 bits [0,1080). Accélération : 1°/5°/10° selon `held_count`. |
| **Récepteur IR RC5** | PE7 (INT7), module M2 | Interruption externe INT7 (front descendant) | `EICRB` (ISC71), `EIMSK` (INT7), `PINE` bit 7 | Protocole RC5 : 14 bits, encodage Manchester, $T_1$=1870 µs. ISR `EXT_INT7` : capture bloquante ∼25 ms, extraction cmd/toggle, lève `ir_event_flag`. Toggle bit pour détection tap/maintien. `held_count` sature à 0xFF. Répétition ∼9 Hz (113 ms/trame). |
| **Buzzer piézo** | PE2 (PORTE bit 2), module M2 | Bit-bang logiciel (`sbi/cbi` + `WAIT_US`) | `PORTE`, `DDRE` | Fréquences : 1500 Hz ($T_{1/2}$=333 µs, 45 cycles) puis 1000 Hz ($T_{1/2}$=500 µs, 30 cycles). Total ∼60 ms. Usages : transition couche (1×), arrêt urgence (2×), confirmation save (1×). |
| **Timer système (Timer2)** | Périphérique interne | Mode CTC + ISR (`OC2addr`) | `TCCR2` (WGM21, CS21 :20), `OCR2`=124, `TIMSK` (OCIE2) | Prescaler/32 → 125 kHz ; OCR2=124 → 125 ticks → 1 ms/tick. ISR `system_tick` : sauvegarde/restaure SREG. Décrémente `overlay_ms_left` (16 bits) ; lève `overlay_dirty` à 0. Écritures depuis main protégées par `cli/sei`. |

| Périphérique | Port / Broche | Mode d'accès | Registres clés | Détails techniques |
|---|---|---|---|---|
| **EEPROM interne** | Périphérique interne (4 Ko) | Registres I/O dédiés (accès direct) | `EEARH:EEARL` (adresse), `EEDR` (donnée), `EECR` (EEMWE, EEWE, EERE) | 32 slots de 16 octets. Adresse = `id×16` (4×`lsl/rol`). Magic byte 0x5A = valide. Écriture : `EEMWE`→`EEWE` en <4 cycles sous `cli`. Structure : magic(1), nom ASCII(10), azimut LE 16 bits(2), élévation 8 bits(1), padding(2). |
| **UART0** | PE0 (RXD0), PE1 (TXD0), coexiste avec M2 sur PORTE | Polling flags `UCSR0A` (non-bloquant) | `UBRR0H:L`=0 :12, `UCSR0B` (TXEN0, RXEN0), `UCSR0C` (UCSZ01 :00), `UDR0` | 19 200 Bd, 8-N-1. UBRR=12 → 19 231 Bd, erreur 0,16 %. Émission : polling `UDRE0`. Réception : polling `RXC0`. Miroir LCD via `uart_enabled`. Séquences ANSI : ESC[H (home), ESC[K (eff. ligne), ESC[J (eff. écran). |

## B. Organigramme de la boucle principale

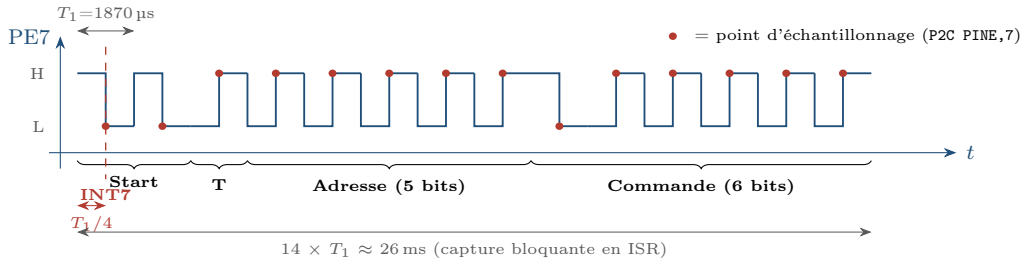
L'organigramme ci-dessous détaille le chemin d'exécution d'une itération de la boucle `main`. Les sources d'entrée (IR par interruption, UART par polling) convergent vers la même variable `last_cmd`, et le dispatch s'effectue par priorité décroissante. Le rendu LCD (`draw_lcd`) est exécuté une seule fois en fin de cycle.




**Légende.** Les losanges sont des tests conditionnels. Le bloc rouge POWER est prioritaire et inconditionnel (single-fire via `held_count`). Les blocs gris clair de la chaîne de dispatch s'exécutent en cascade : si la commande ne correspond pas, le test suivant est évalué. Le retour à `main:` après `draw_lcd` ferme la boucle événementielle.

## C. Chronogramme de capture RC5 (protocole Manchester)

Chaque trame RC5 contient 14 bits encodés en Manchester : un « 1 » logique correspond à une transition haut→bas au centre du bit, un « 0 » à bas→haut. La période d'un bit est $T_1 = 1870\,\mu s$ (calibrée sur le quartz RC interne 4 MHz). L'ISR `EXT_INT7` se déclenche sur le front descendant du start bit S1, attend $T_1/4$ pour se caler au centre du premier bit, puis échantillonne 14 fois à intervalles de $T_1$.



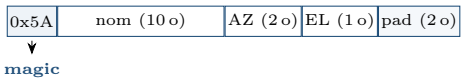

**Exemple illustré :** trame RC5 pour la touche CH+ (`CMD_CH_UP` = 0x20). Après inversion (`com b0`), les bits échantillonnés au niveau bas correspondent à des « 1 » logiques RC5. L'extraction des champs s'effectue par masquage : commande = `b0 & 0x3F`, adresse = bits croisés `b1:b0`, toggle = `b1[3]`.

## D. Satellites préprogrammés en EEPROM

La mémoire EEPROM interne (4 Ko) est organisée en 32 slots de 16 octets. Les 10 premiers slots sont préprogrammés via le fichier `.eep` (chargé par `avrdude`) avec des positions azimut/élévation réalistes pour démonstration. Les 22 slots restants sont libres pour les sauvegardes utilisateur (mode MANUEL → GUIDE).

| Slot | Nom | Azimut (°) | Élévation (°) | Type |
|---|---|---|---|---|
| 0 | **ISS** | 257 | 24 | Station spatiale (LEO) |
| 1 | **Hubble** | 142 | 56 | Télescope spatial (LEO) |
| 2 | **Tiangong** | 89 | 72 | Station spatiale chinoise (LEO) |
| 3 | **Starlink** | 310 | 45 | Constellation internet (LEO) |
| 4 | **Envisat** | 15 | 81 | Observation terrestre (LEO) |
| 5 | **NOAA-19** | 198 | 33 | Météorologie (LEO polaire) |
| 6 | **Iridium** | 275 | 65 | Communication (LEO) |
| 7 | **Terra** | 45 | 12 | Observation terrestre (LEO) |
| 8 | **Aqua** | 120 | 88 | Observation terrestre (LEO) |
| 9 | **Suomi NPP** | 82 | 41 | Météorologie (LEO polaire) |
| 10–31 | *Libres — sauvegardés sous le nom* ***SAT xx*** *(attribution auto.)* | | | |

**Structure d'un slot en mémoire :**




L'adresse EEPROM d'un slot est calculée par `id × 16` (4×`lsl/rol`). Un slot est considéré valide si son magic byte vaut `0x5A` ; la suppression consiste à écrire `0x00` à l'offset 0. L'azimut est stocké en little-endian sur 2 octets (plage 0–359°), l'élévation sur 1 octet (0–90°).

## E. Montage physique

La figure ci-dessous montre le montage complet sur la carte STK-300. Le stepper JST X27 (azimut) est connecté au module M1 via PORTD[0 :3]. Le servo Futaba S3003 (élévation) est connecté au module M4 (connecteur P8) piloté par PB5/OC1A. Le récepteur IR et le buzzer piézo partagent le module M2 sur PORTE (PE7 et PE2 respectivement). Le LCD HD44780 2×16 est accessible via le bus externe (PORTA + PORTC + PORTG).

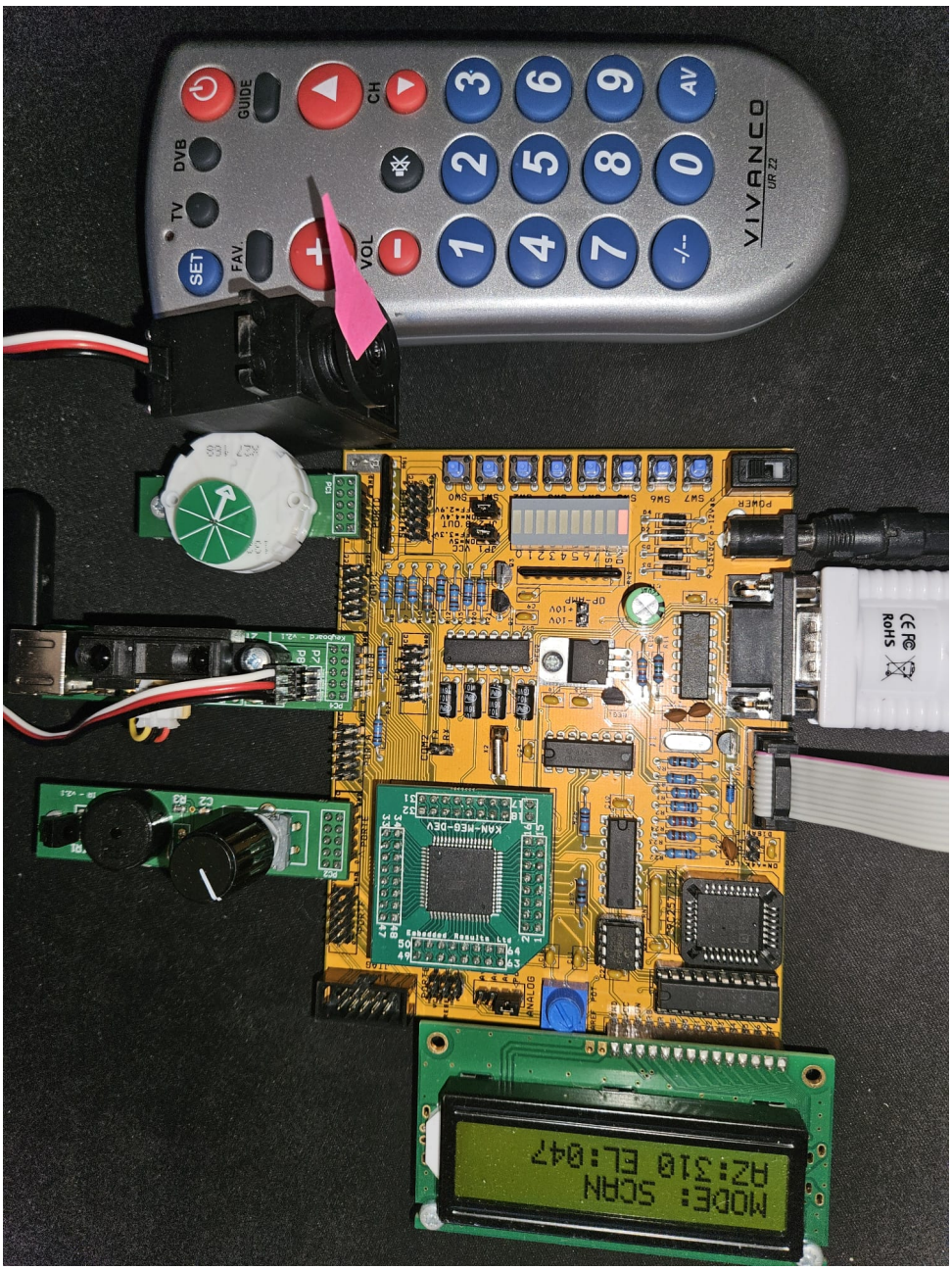



FIGURE 1 – Vue d'ensemble du montage sur la carte STK-300 avec les modules périphériques.

## F. Interface terminal UART (miroir VT100)

Lorsque le miroir UART est activé (`MUTE` sur la télécommande ou `U` au clavier), chaque cycle de rendu envoie l'état du LCD sur USART0 à 19 200 Bd (8-N-1). Le terminal émule un affichage 2 lignes via des séquences d'échappement ANSI/VT100 :

| Séquence | Effet |
|---|---|
| `ESC[H` | Curseur en position home (0,0) |
| `ESC[K` | Efface la ligne courante depuis le curseur |
| `ESC[J` | Efface l'écran depuis le curseur |
| `\r\n` | Passage à la ligne 2 |

Le terminal reproduit fidèlement les deux lignes du LCD, y compris les overlays transitoires (`SAVED`, `DELETED`, `UART ON/OFF`). Le contrôle bidirectionnel permet également d'envoyer des commandes au clavier, traduites en équivalents RC5 par `uart_poll` : les touches directionnelles (`W/S/A/D`), les actions (`Space`=GUIDE, `X`=DEL, `M`=AV, `P`=POWER), les accès numériques (`0-9`), et la commande `L` qui liste

tous les satellites valides de la base EEPROM.

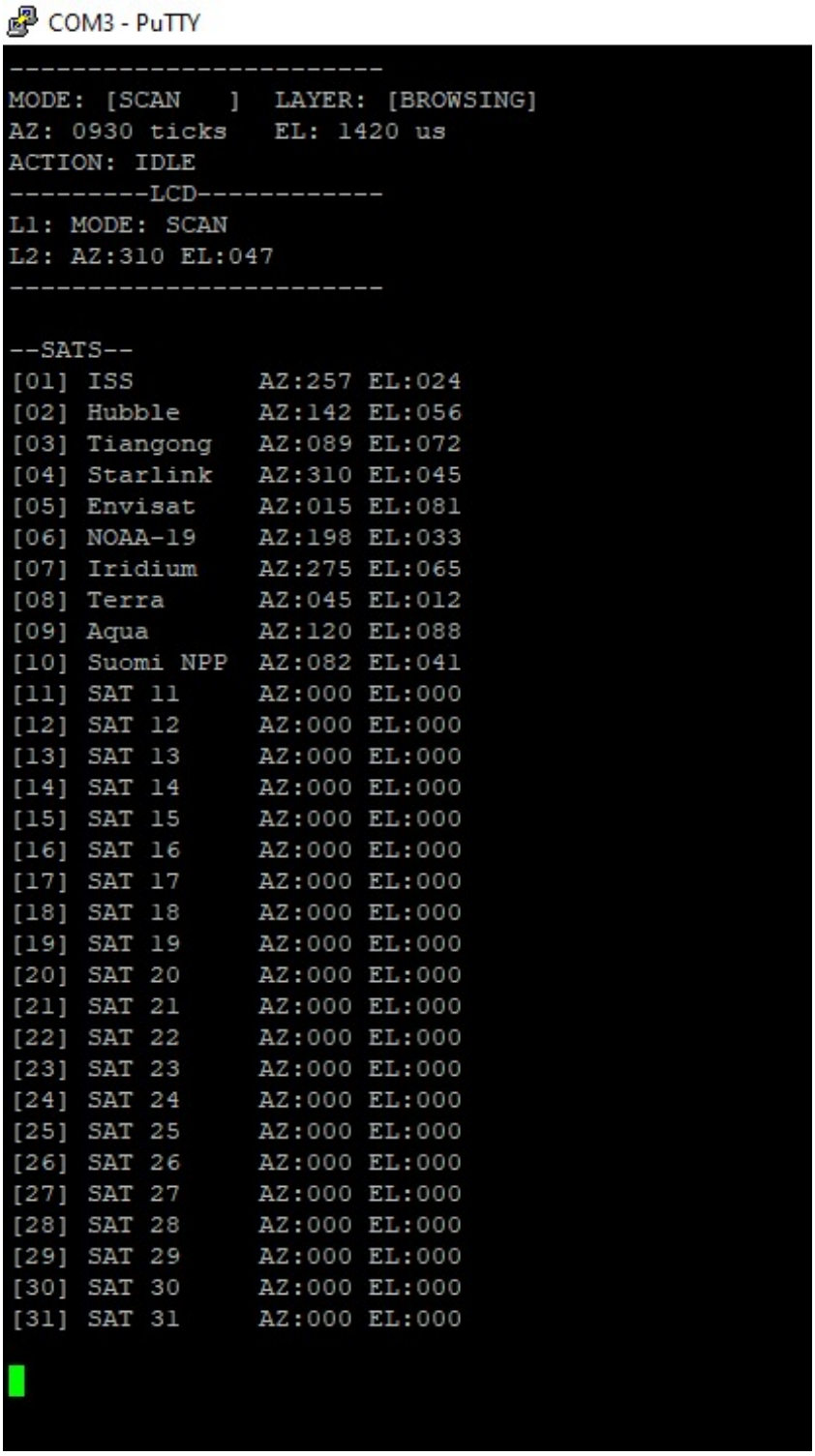



FIGURE 2 – Capture du terminal série affichant le miroir VT100 du LCD.

# Annexe G — Code source complet

`drivers/` (9 fichiers) | `libs/` (11 fichiers) | `main.asm`

### Fichier : main.asm

```
; file:    main.asm      target ATmega128L-4MHz-STK300
; purpose: antenna-pointer FSM. Owns reset, ISR vectors, main dispatch.
;          Sub-systems (RC5, LCD, servo, stepper, EEPROM, UART, overlay)
;          are pulled in via includes at the bottom of this file.
; IR:      Vivanco UR Z2 (RC5). Demod output on PE7 = INT7 (M2 module).

.include "m128def.inc"
.include "libs/macros.asm"
.include "libs/definitions.asm"
.include "libs/pinout.inc"
.include "libs/constants.inc"


; === SRAM state ===
; All FSM globals live here. Per-module ownership is listed in each
; module's header.
.dseg
.org    SRAM_START

; --- RC5 frame mirror (written by EXT_INT7, read by main_dispatch) ---
last_addr:      .byte 1     ; decoded RC5 address (== 0 for TV codes)
last_cmd:       .byte 1     ; decoded 6-bit RC5 command
last_tog:       .byte 1     ; last toggle bit; 0xFF = sentinel (no frame yet)
ir_event_flag:  .byte 1     ; 1 = ISR captured a frame, 0 = nothing pending

; --- Core FSM state ---
mode:           .byte 1     ; MODE_AUTO / MODE_MANUEL / MODE_SCAN
sub_state:      .byte 1     ; per-mode sub-FSM (cleared on layer change)
fsm_layer:      .byte 1     ; LAYER_BROWSING / LAYER_IN_MODE (AV toggles)
flag_is_moving: .byte 1     ; 1 = motor in autonomous transit, blocks AV
lcd_l2_buf:     .byte 16    ; LCD line-2 source buffer (16 chars, no NUL)

; --- Azimuth (X27 stepper) ---
current_az_steps: .byte 2   ; 0..1079 partial steps, wraps mod AZ_MAX_STEPS.
                            ; Degrees = current_az_steps / 3.

; --- Press-and-hold tracking (incremented by EXT_INT7) ---
held_count:     .byte 1     ; 0 = fresh press, saturates at 0xFF.

; --- Elevation (Futaba S3003, Timer1 OC1A) ---
target_ticks:   .byte 2     ; OCR1A target, range TICKS_MIN..TICKS_MAX

; --- AUTO mode (satellite database browser) ---
auto_substate:    .byte 1   ; ST_AUTO_BROWSING / MOVING_AZ / MOVING_EL / ...
target_sat_id:    .byte 1   ; slot index 0..31 (cursor)
target_az_auto:   .byte 2   ; target AZ in degrees
target_el_auto:   .byte 1   ; target EL in degrees
current_sat_name: .byte 11  ; 10 chars + null, from EEPROM via load_satellite

; --- SCAN slot-visibility filter (shared with AUTO) ---
; 0 = AUTO default (magic-byte test only).
; 1 = SCAN result set (also requires bit set in scan_results).
slot_filter_mode: .byte 1
scan_results:     .byte 4   ; 32-bit bitmap, bit i = slot i hit during sweep

; --- SCAN sweep working set ---
; Cache loaded once at sweep entry to avoid 46k EEPROM reads per sweep.
scan_cache_az:    .byte 64  ; 32 slots * 2 bytes (little-endian degrees)
scan_cache_el:    .byte 32  ; 32 slots * 1 byte degrees
scan_cache_valid: .byte 4   ; bitmap, bit i = slot i had magic byte 0x5A
scan_steps_done:  .byte 2   ; partials done this sweep (advances by 3 per deg)
scan_el_at_start: .byte 1   ; EL snapshot at sweep entry; match band centre


; === Interrupt vectors ===
.cseg
.org    0x0000
    rjmp    reset
.org    INT7addr            ; PE7 falling edge -> RC5 frame start
    rjmp    EXT_INT7
.org    OC2addr             ; Timer2 1 ms tick -> overlay countdown
    rjmp    system_tick
.org    INT_VECTORS_SIZE    ; skip past unused vectors


; ============================================================
; reset: power-on entry. Configure ports, init peripherals,
; zero SRAM state, paint initial LCD, enable interrupts.
; ============================================================
reset:
    LDSP    RAMEND                          ; stack at top of SRAM
```

```
    ; --- PORTB: servo on PB5 (= OC1A). All output, idle low. ---
    ; PORTB driven low BEFORE Timer1 takes over OC1A so we don't emit
    ; a stray high pulse on PB5 during the LCD init that follows.
    ldi     w, 0xFF
    out     DDRB, w
    clr     w
    out     PORTB, w

    ; --- PORTE: UART0 + buzzer + IR. One-shot DDR/PORT init. ---
    ; PE0 = RXD0 (in)         PE2 = buzzer (out)      PE7 = IR demod (in)
    ; PE1 = TXD0 (out)        PE3..PE6 unused
    ; UART0_init MUST NOT touch DDRE (course-provided uart.asm `out DDRE`
    ; was killing the buzzer bit -- left commented as a tombstone there).
    ldi     w, (1 << 1) | (1 << BUZZER_PIN) ; bit 1 = TXD0 (no symbol in m128def)
    out     DDRE, w
    clr     w
    out     PORTE, w

    ; --- Peripherals ---
    ; `call` (not rcall): lcd.asm is the last include and sits >2K words
    ; from reset. If you add anything that rcalls a far lib from reset:,
    ; convert it the same way.
    call    LCD_init
    call    LCD_clear
    call    uart_init_app
    rcall   servo_init                  ; Timer1 PWM, OCR1A = TICKS_MIN (0 deg)
    rcall   stepper_init                ; PORTD low nibble = coils, idle low

    ; --- SRAM init ---
    ; X27 has no end stop: logical AZ=0 == current physical orientation.
    ; User pre-aligns the antenna by hand before power-on.
    clr     w
    sts     current_az_steps, w
    sts     current_az_steps+1, w
    ; target_ticks already seeded to TICKS_MIN by servo_init -- don't touch.

    sts     mode, w                     ; MODE_AUTO
    sts     sub_state, w
    sts     fsm_layer, w                ; LAYER_BROWSING
    sts     last_addr, w
    sts     last_cmd, w
    sts     held_count, w
    sts     flag_is_moving, w
    sts     auto_substate, w
    sts     target_sat_id, w
    sts     ir_event_flag, w
    sts     slot_filter_mode, w
    sts     scan_results, w
    sts     scan_results+1, w
    sts     scan_results+2, w
    sts     scan_results+3, w

    ldi     w, 0xFF                     ; "no toggle observed yet"
    sts     last_tog, w

    rcall   draw_lcd                    ; initial paint

    ; --- Interrupts: Timer2 (overlay) + INT7 (RC5 edge) ---
    rcall   sys_time_init               ; Timer2 1 ms CTC
    ldi     w, (1 << ISC71)             ; INT7 falling edge
    out     EICRB, w
    in      w, EIMSK
    ori     w, (1 << INT7)
    out     EIMSK, w
    sei


; ============================================================
; main: dispatch loop.
;
; Each iteration:
;   1) Drain overlay-expiry: if Timer2 just cleared an overlay,
;      rebuild lcd_l2_buf from the live FSM state and redraw.
;   2) Poll UART for an injected RC5-equivalent command.
;   3) If RC5 ISR raised ir_event_flag, dispatch the frame.
;   4) Else if flag_is_moving, run one autonomous handler step
;      (AUTO transit / SCAN sweep).
;   5) Else idle: loop back to (1).
;
; RC5 capture itself lives in EXT_INT7 below -- this loop never
; spins on the IR pin.
; ============================================================
main:
    rcall   overlay_consume_dirty       ; Z=0 if just expired
    breq    main_no_dirty
```

```
    lds     w, fsm_layer
    cpi     w, LAYER_IN_MODE                ; only IN_MODE reads lcd_l2_buf
    breq    main_dirty_refresh
    rjmp    redraw                          ; brne would not reach redraw
main_dirty_refresh:
    rcall   init_l2_for_mode                ; repopulate buffer from FSM state
    rjmp    redraw
main_no_dirty:

    call    uart_poll                       ; Z=0 if UART injected a command
    breq    main_check_event
    rjmp    main_dispatch

main_check_event:
    lds     w, ir_event_flag
    tst     w
    brne    main_handle_ir
    lds     w, flag_is_moving
    tst     w
    brne    main_autonomous
    rjmp    main                            ; idle: keep polling

main_handle_ir:
    clr     w
    sts     ir_event_flag, w                ; consume event
    rjmp    main_dispatch

main_autonomous:
    ldi     w, 0xFF                         ; sentinel "no key pressed"
    sts     last_cmd, w
    rjmp    dispatch_in_mode


; ============================================================
; EXT_INT7: RC5 frame capture (falling edge on PE7).
;
; Blocks ~25 ms in rc5_capture_from_edge. While we're here Timer2
; overflows collapse into a single OCF2 (it's a flag, not a counter),
; so the overlay countdown slows during heavy IR use. Acceptable
; for the current 500 ms overlays; if anything time-critical ever
; hangs off Timer2, mask INT7 + `sei` early to allow nesting.
; ============================================================
EXT_INT7:
    push    _sreg
    in      _sreg, SREG
    push    u
    push    w
    push    _w
    push    r18
    push    r19
    push    b0
    push    b1
    push    b2

    rcall   rc5_capture_from_edge           ; b1:b0 = 14 raw bits

    ; Start-bit check: ~S1 = b1[5], ~S2 = b1[4]. Both must be 0.
    mov     w, b1
    andi    w, 0b00110000
    brne    ext_int7_done                   ; bad frame -> drop

    rcall   rc5_extract_fields              ; r16=cmd r17=addr r18=tog
    sts     last_cmd, r16
    sts     last_addr, r17
    mov     w, r18

    ; --- held_count tracking: same toggle = button held; new = fresh press ---
    lds     _w, last_tog
    cp      w, _w
    breq    ext_int7_same_tog
    sts     last_tog, w
    clr     _w
    sts     held_count, _w
    rjmp    ext_int7_flag
ext_int7_same_tog:
    lds     _w, held_count
    cpi     _w, 0xFF                        ; saturate, don't wrap
    breq    ext_int7_flag
    inc     _w
    sts     held_count, _w
ext_int7_flag:
    ldi     w, 1
    sts     ir_event_flag, w                ; tell main a frame is ready

ext_int7_done:
```

```
    pop     b2
    pop     b1
    pop     b0
    pop     r19
    pop     r18
    pop     _w
    pop     w
    pop     u
    out     SREG, _sreg
    pop     _sreg
    reti


; ============================================================
; main_dispatch: react to last_cmd. Priority order:
;   POWER (estop, always wins) -> overlay early-dismiss -> MUTE ->
;   AV (layer toggle) -> per-layer handlers.
; ============================================================
main_dispatch:
    lds     w, last_cmd

    ; --- POWER: emergency stop, single-fire ---
    ; Gated on held_count == 0 so a held POWER doesn't re-chirp.
    cpi     w, CMD_POWER
    brne    not_power
    lds     _w, held_count
    tst     _w
    breq    do_estop
    rjmp    redraw                          ; held -> skip
do_estop:
    rcall   emergency_stop
    rcall   overlay_force_clear             ; nuke any pending SAVED/DELETED
    clr     _w
    sts     fsm_layer, _w                   ; force BROWSING
    sts     sub_state, _w
    sts     flag_is_moving, _w              ; cancel autonomous transit
    sts     auto_substate, _w
    sts     slot_filter_mode, _w            ; drop SCAN filter on AUTO re-entry
    sts     scan_results, _w
    sts     scan_results+1, _w
    sts     scan_results+2, _w
    sts     scan_results+3, _w
    rjmp    redraw
not_power:

    ; --- Overlay early-dismiss on any fresh non-POWER press ---
    ; The press itself is NOT consumed -- falls through to dispatch
    ; below. Gated on held_count == 0 so RC5 repeats of the button
    ; that just FIRED the overlay don't nuke it within ~25 ms.
    lds     _w, held_count
    tst     _w
    brne    overlay_no_dismiss
    rcall   overlay_force_clear
overlay_no_dismiss:

    ; --- MUTE: toggle UART mirror, single-fire ---
    cpi     w, CMD_MUTE
    brne    not_mute
    lds     _w, held_count
    tst     _w
    brne    redraw
    call    toggle_uart
    rjmp    redraw
not_mute:

    ; --- AV: layer toggle, single-fire ---
    cpi     w, CMD_AV
    brne    not_av
    lds     _w, held_count
    tst     _w
    brne    redraw
    rcall   toggle_layer
    rjmp    redraw
not_av:

    ; --- Branch on layer ---
    lds     _w, fsm_layer
    cpi     _w, LAYER_IN_MODE
    breq    dispatch_in_mode

    ; --- BROWSING: CH+/- cycles modes (fresh press only) ---
    lds     _w, held_count
    tst     _w
    brne    redraw                          ; held -> ignore everything
    cpi     w, CMD_CH_UP
```

```
    brne    not_ch_up
    rcall   cycle_mode_next
    rjmp    redraw
not_ch_up:
    cpi     w, CMD_CH_DN
    brne    not_ch_dn
    rcall   cycle_mode_prev
    rjmp    redraw
not_ch_dn:
    rjmp    redraw                          ; unrecognised -> just refresh

; --- IN_MODE: every frame goes to the mode handler ---
; Handlers read held_count themselves to gate single-fire vs repeat.
; Overlay gate: while a transient message is up, freeze dispatch so
; handlers don't overwrite lcd_l2_buf with their normal content.
dispatch_in_mode:
    rcall   overlay_active
    brne    redraw                          ; overlay up -> just redraw
    lds     _w, mode
    cpi     _w, MODE_AUTO
    breq    do_auto
    cpi     _w, MODE_MANUEL
    breq    do_manuel
    rjmp    do_scan                         ; only SCAN left

do_auto:
    call    handle_auto
    rjmp    redraw
do_manuel:
    call    handle_manuel
    rjmp    redraw
do_scan:
    call    handle_scan
    ; fall through

redraw:
    call    draw_lcd
    rjmp    main


; === Module includes ===
; Order matters: included code sits AFTER reset/main, so the reset
; vector at 0x0000 cleanly jumps over it. drivers/lcd.asm is LAST
; because its .db tables must not collide with the reset vector --
; that's also why a couple of LCD_init calls in reset: use `call`
; rather than rcall (the labels sit beyond the +/-2K rcall range).
.include "drivers/timer_manager.asm"
.include "libs/overlay.asm"
.include "drivers/eeprom_layout.asm"
.include "drivers/rc5_logic.asm"
.include "drivers/uart_control.asm"
.include "libs/display.asm"
.include "drivers/math_revised.asm"
.include "drivers/logique_pointage.asm"
.include "drivers/0ec_motor_sequence.asm"
.include "drivers/buzzer.asm"
.include "libs/shared_handlers.asm"
.include "libs/mode_auto.asm"
.include "libs/mode_manuel.asm"
.include "libs/mode_scan.asm"
.include "drivers/lcd.asm"
```

**Fichier : drivers/0ec_motor_sequence.asm**

```
; file:    0ec_motor_sequence.asm   target ATmega128L-4MHz-STK300
; purpose: X27.168 stepper driver. 6 natural full-steps per rotor turn
;          (3 poles * 2 coils -> 60 deg rotor / step). With the 1:180
;          gearbox: 6 patterns = 2 deg at the output, so 3 patterns = 1 deg.
;
;          Pin mapping (drivers/pinout.inc): PD0 = C1+, PD1 = C1-,
;          PD2 = C2+, PD3 = C2-. No H-bridge -- coil terminals are
;          tied directly to AVR pins. "Idle" state is both terminals
;          at the same level (no differential voltage).


.dseg
motor_step_idx: .byte 1     ; cursor in motor_sequence_table, 0..5

.cseg
; === stepper_init ===
; PD0..PD3 as outputs, coils idle low, sequence cursor = 0.
stepper_init:
    in      r16, DDRD
    ori     r16, 0x0F
    out     DDRD, r16
    in      r16, PORTD
    andi    r16, 0xF0                   ; preserve PD4..PD7
    out     PORTD, r16
    clr     r16
    sts     motor_step_idx, r16
    ret


; === motor_sequence_table ===
; 6-pattern ring, one full electrical revolution per cycle.
; Decoded (PD3 PD2 PD1 PD0):
;   0x05 = +C1  +C2          0x0A = -C1  -C2
;   0x01 = +C1  idle         0x0E = -C1  idle
;   0x0B = idle -C2          0x04 = idle +C2
motor_sequence_table:
    .db 0x05, 0x01, 0x0B, 0x0A, 0x0E, 0x04


; === step_motor_forward ===
; Advance the cursor, wrap 5 -> 0, apply.
; Clobbers: r16, r17, Z
step_motor_forward:
    lds     r16, motor_step_idx
    inc     r16
    cpi     r16, 6
    brlo    smf_store
    clr     r16
smf_store:
    sts     motor_step_idx, r16
    rjmp    apply_motor_step


; === step_motor_backward ===
; Decrement the cursor, wrap 0 -> 5, apply.
step_motor_backward:
    lds     r16, motor_step_idx
    dec     r16
    brpl    smb_store
    ldi     r16, 5
smb_store:
    sts     motor_step_idx, r16
    rjmp    apply_motor_step


; === apply_motor_step ===
; Look up the current pattern in flash and write its nibble onto
; PORTD low (preserves PORTD high nibble). The 3 ms wait gives the
; rotor time to land between coil patterns (well under the X27's
; 250 deg/s start-stop limit).
; In:  r16 = current cursor (already updated by caller)
apply_motor_step:
    ldi     ZL, low(2*motor_sequence_table)
    ldi     ZH, high(2*motor_sequence_table)
    add     ZL, r16
    ldi     r17, 0
    adc     ZH, r17
    lpm     r16, Z                      ; r16 = pattern nibble

    in      r17, PORTD
    andi    r17, 0xF0                   ; keep PD4..PD7
    or      r17, r16
```

```
    out     PORTD, r17
    WAIT_MS 3                       ; rotor settle
    ret
```

### Fichier : drivers/buzzer.asm

```
; file:    buzzer.asm     target ATmega128L-4MHz-STK300
; purpose: piezo buzzer (M2 module, BUZZER_PIN of BUZZER_PORT = PE2).
;          DDR setup is done once in main.asm reset: (bundled into the
;          PORTE one-shot config); no buzzer_init needed here.


; === chirp ===
; Two-note "doot-doot" ~60 ms total: ~1.5 kHz for ~30 ms, then ~1 kHz
; for ~30 ms. Used as audible feedback for layer toggles, save/delete,
; ESTOP (called twice), SCAN hits, and "arrived" in AUTO.
;
; WAIT_US clobbers w and u (see libs/macros.asm), so we keep the
; outer counter in _w which WAIT_US leaves alone.
chirp:
    ; --- note 1: ~1.5 kHz, 45 * 666 us ~= 30 ms ---
    ldi     _w, 45
chirp_high_loop:
    sbi     BUZZER_PORT, BUZZER_PIN
    WAIT_US 333
    cbi     BUZZER_PORT, BUZZER_PIN
    WAIT_US 333
    dec     _w
    brne    chirp_high_loop

    ; --- note 2: ~1 kHz, 30 * 1000 us = 30 ms ---
    ldi     _w, 30
chirp_low_loop:
    sbi     BUZZER_PORT, BUZZER_PIN
    WAIT_US 500
    cbi     BUZZER_PORT, BUZZER_PIN
    WAIT_US 500
    dec     _w
    brne    chirp_low_loop

    cbi     BUZZER_PORT, BUZZER_PIN     ; leave the line low at idle
    ret
```

**Fichier : drivers/eeprom_layout.asm**

```
; file:    eeprom_layout.asm   target ATmega128L-4MHz-STK300
; purpose: satellite database in internal EEPROM.
;
;          Slot format (16 bytes per slot, 32 slots = 512 bytes):
;            offset 0      magic byte (0x5A = valid, anything else = empty)
;            offset 1..10  name (10 chars, "SAT xx    " for autogen)
;            offset 11..12 azimuth in degrees, little-endian
;            offset 13     elevation in degrees
;            offset 14..15 padding
;
;          API: EEPROM_read_byte / EEPROM_write_byte (raw)
;               find_free_slot / save_satellite / execute_save (write)
;               load_satellite / delete_satellite (read / wipe)

.equ EEPROM_SLOT_SIZE  = 16
.equ EEPROM_MAX_SLOTS  = 32
.equ EEPROM_START_ADDR = 0x0000


; === Initial DB (loaded by avrdude into the .eep image) ===
; Demo set: 10 satellites with realistic AZ/EL pairs.
.eseg
.org EEPROM_START_ADDR
    .db 0x5A, "ISS       ", low(257), high(257), 24, 0, 0
    .db 0x5A, "Hubble    ", low(142), high(142), 56, 0, 0
    .db 0x5A, "Tiangong  ", low(89),  high(89),  72, 0, 0
    .db 0x5A, "Starlink  ", low(310), high(310), 45, 0, 0
    .db 0x5A, "Envisat   ", low(15),  high(15),  81, 0, 0
    .db 0x5A, "NOAA-19   ", low(198), high(198), 33, 0, 0
    .db 0x5A, "Iridium   ", low(275), high(275), 65, 0, 0
    .db 0x5A, "Terra     ", low(45),  high(45),  12, 0, 0
    .db 0x5A, "Aqua      ", low(120), high(120), 88, 0, 0
    .db 0x5A, "Suomi NPP ", low(82),  high(82),  41, 0, 0


.cseg
; === EEPROM_read_byte ===
; Spin until any prior write finishes, then read one byte.
; In:  r17:r16 = address
; Out: r18 = data
EEPROM_read_byte:
    sbic    EECR, EEWE                  ; wait for prior write
    rjmp    EEPROM_read_byte
    out     EEARH, r17
    out     EEARL, r16
    sbi     EECR, EERE                  ; strobe read
    in      r18, EEDR
    ret


; === EEPROM_write_byte ===
; Write one byte. The EEMWE -> EEWE sequence is timing-critical and
; runs with interrupts disabled.
; In:  r17:r16 = address, r18 = data
EEPROM_write_byte:
    sbic    EECR, EEWE                  ; wait for prior write
    rjmp    EEPROM_write_byte
    out     EEARH, r17
    out     EEARL, r16
    out     EEDR, r18
    in      r19, SREG
    cli
    sbi     EECR, EEMWE                 ; master write enable
    sbi     EECR, EEWE                  ; commit (must be within 4 cycles)
    out     SREG, r19
    ret


; === find_free_slot ===
; Linear scan for a slot whose magic byte is NOT 0x5A.
; Out: r16 = slot index (0..31), or 0xFF if memory is full
find_free_slot:
    ldi     r20, 0
ffs_loop:
    cpi     r20, EEPROM_MAX_SLOTS
    brsh    ffs_full

    ; address = r20 * 16 -> swap nibbles
    mov     r16, r20
    swap    r16
    mov     r17, r16
    andi    r16, 0xF0
```

```
    andi    r17, 0x0F

    rcall   EEPROM_read_byte
    cpi     r18, 0x5A
    brne    ffs_found
    inc     r20
    rjmp    ffs_loop
ffs_found:
    mov     r16, r20
    ret
ffs_full:
    ldi     r16, 0xFF
    ret


; === save_satellite ===
; Write magic byte, "SAT xx    " name, AZ, EL into the slot.
; In:  r20 = slot index, r22:r21 = azimuth, r23 = elevation
; Uses Y for the running EEPROM address; uses lcd_l2_buf[0..1] as a
; scratch buffer for the two-digit ID conversion.
save_satellite:
    ; Y = slot base address = r20 * 16
    mov     YL, r20
    swap    YL
    mov     YH, YL
    andi    YL, 0xF0
    andi    YH, 0x0F

    ; magic byte
    mov     r16, YL
    mov     r17, YH
    ldi     r18, 0x5A
    rcall   EEPROM_write_byte

    ; --- "SAT " (4 chars) ---
    adiw    YL, 1
    mov     r16, YL
    mov     r17, YH
    ldi     r18, 'S'
    rcall   EEPROM_write_byte
    adiw    YL, 1
    mov     r16, YL
    mov     r17, YH
    ldi     r18, 'A'
    rcall   EEPROM_write_byte
    adiw    YL, 1
    mov     r16, YL
    mov     r17, YH
    ldi     r18, 'T'
    rcall   EEPROM_write_byte
    adiw    YL, 1
    mov     r16, YL
    mov     r17, YH
    ldi     r18, ' '
    rcall   EEPROM_write_byte

    ; --- ID as 2 ASCII digits -> use lcd_l2_buf as scratch ---
    mov     a0, r20
    inc     a0                          ; show 1..32 (not 0..31)
    clr     a1
    ldi     XL, low(lcd_l2_buf)
    ldi     XH, high(lcd_l2_buf)
    rcall   u16_to_dec2

    adiw    YL, 1
    mov     r16, YL
    mov     r17, YH
    lds     r18, lcd_l2_buf
    rcall   EEPROM_write_byte
    adiw    YL, 1
    mov     r16, YL
    mov     r17, YH
    lds     r18, lcd_l2_buf+1
    rcall   EEPROM_write_byte

    ; --- 4 trailing spaces to pad the name to 10 chars total ---
    ; (5 iterations was a historical bug -- wrote into the AZ low byte
    ;  and shifted every field by 1 on read-back.)
    ldi     r24, 4
pad_loop:
    adiw    YL, 1
    mov     r16, YL
    mov     r17, YH
    ldi     r18, ' '
    rcall   EEPROM_write_byte
```

```
    dec     r24
    brne    pad_loop

    ; --- Azimuth (2 bytes, little-endian) ---
    adiw    YL, 1
    mov     r16, YL
    mov     r17, YH
    mov     r18, r21
    rcall   EEPROM_write_byte
    adiw    YL, 1
    mov     r16, YL
    mov     r17, YH
    mov     r18, r22
    rcall   EEPROM_write_byte

    ; --- Elevation (1 byte) ---
    adiw    YL, 1
    mov     r16, YL
    mov     r17, YH
    mov     r18, r23
    rcall   EEPROM_write_byte
    ret


; === execute_save ===
; Capture current AZ + EL, find a free slot, write it.
; AZ = current_az_steps / 3 ; EL = (target_ticks - 500) * 9 / 40.
; UI feedback: chirp + "SAVED" overlay on success, "MEM FULL" on full.
execute_save:
    ; --- AZ degrees (push to stack pending free-slot lookup) ---
    lds     a0, current_az_steps
    lds     a1, current_az_steps+1
    ldi     b0, AZ_STEP_PARTIAL
    ldi     b1, 0
    rcall   div22
    push    c0
    push    c1

    ; --- EL degrees: (ticks - 500) * 9 / 40 ---
    lds     a0, target_ticks
    lds     a1, target_ticks+1
    ldi     b0, low(500)
    ldi     b1, high(500)
    sub     a0, b0
    sbc     a1, b1
    brcc    save_el_ok
    clr     a0
    clr     a1
save_el_ok:
    mov     c0, a0
    mov     c1, a1
    lsl     a0
    rol     a1
    lsl     a0
    rol     a1
    lsl     a0
    rol     a1
    add     a0, c0
    adc     a1, c1                      ; a = (ticks-500) * 9
    ldi     b0, 40
    ldi     b1, 0
    rcall   div22
    push    c0                          ; EL fits in one byte

    ; --- find a slot ---
    rcall   find_free_slot
    cpi     r16, 0xFF
    breq    save_full_pop

    ; --- pop angles + write slot ---
    pop     r23                         ; EL
    pop     r22                         ; AZ high
    pop     r21                         ; AZ low
    mov     r20, r16
    rcall   save_satellite

    rcall   chirp
    ldi     ZL, low(2*str_save_ok)
    ldi     ZH, high(2*str_save_ok)
    rcall   copy_flash_to_l2_buf
    rjmp    save_end

save_full_pop:
    pop     r23
    pop     r22
```

```
    pop     r21
    ldi     ZL, low(2*str_save_full)
    ldi     ZH, high(2*str_save_full)
    rcall   copy_flash_to_l2_buf

save_end:
    ret

str_save_ok:   .db "SAVED           ", 0, 0
str_save_full: .db "MEM FULL        ", 0, 0


; === load_satellite ===
; Read slot r16 into the SRAM mirror (current_sat_name + target_az_auto
; + target_el_auto). Sets Z=1 if the slot is valid (magic byte 0x5A),
; Z=0 otherwise.
; In:  r16 = slot index
; Out: Z flag; SRAM mirror updated only on valid
load_satellite:
    ; Y = slot base address = r16 * 16
    mov     YL, r16
    swap    YL
    mov     YH, YL
    andi    YL, 0xF0
    andi    YH, 0x0F

    ; magic byte
    mov     r16, YL
    mov     r17, YH
    rcall   EEPROM_read_byte
    cpi     r18, 0x5A
    brne    load_empty

    ; --- name (10 chars + NUL terminator we add ourselves) ---
    ldi     XL, low(current_sat_name)
    ldi     XH, high(current_sat_name)
    ldi     r24, 10
load_name_loop:
    adiw    YL, 1
    mov     r16, YL
    mov     r17, YH
    rcall   EEPROM_read_byte
    st      X+, r18
    dec     r24
    brne    load_name_loop
    clr     r18
    st      X, r18

    ; --- AZ (2 bytes) ---
    adiw    YL, 1
    mov     r16, YL
    mov     r17, YH
    rcall   EEPROM_read_byte
    sts     target_az_auto, r18
    adiw    YL, 1
    mov     r16, YL
    mov     r17, YH
    rcall   EEPROM_read_byte
    sts     target_az_auto+1, r18

    ; --- EL (1 byte) ---
    adiw    YL, 1
    mov     r16, YL
    mov     r17, YH
    rcall   EEPROM_read_byte
    sts     target_el_auto, r18

    sez                                 ; Z=1 -> valid
    ret

load_empty:
    clz                                 ; Z=0 -> invalid
    ret


; === delete_satellite ===
; Overwrite the slot's magic byte with 0x00 so subsequent loads / scans
; treat it as empty. The rest of the slot's bytes are left intact
; (find_free_slot will pick it up on the next save).
; In:  r16 = slot index
delete_satellite:
    mov     YL, r16
    swap    YL
    mov     YH, YL
    andi    YL, 0xF0
```

```
    andi    YH, 0x0F
    mov     r16, YL
    mov     r17, YH
    ldi     r18, 0x00
    rcall   EEPROM_write_byte
    ret
```

### Fichier : drivers/lcd.asm

```
; file:    lcd.asm        target ATmega128L-4MHz-STK300
; purpose: HD44780 LCD driver (memory-mapped via external SRAM bus).
;          Wire-compatible with the course-provided lcd.asm; the only
;          additions in this project are the UART mirror hooks in
;          LCD_pos and LCD_wr_dr (both no-ops when uart_enabled = 0).

.equ    LCD_IR  = 0x8000        ; LCD instruction register
.equ    LCD_DR  = 0xc000        ; LCD data register


; === LCD_wr_ir ===
; Write w to the LCD instruction register, after the busy flag clears.
LCD_wr_ir:
    lds     u, LCD_IR           ; read IR (busy flag in bit 7)
    JB1     u, 7, LCD_wr_ir
    rcall   lcd_4us             ; DDRAM cursor increment time
    sts     LCD_IR, w
    ret


; === lcd_4us / lcd_2us ===
; Tiny calibrated waits (rcall + body + ret = 8 / 4 cycles @ 4 MHz).
lcd_4us:
    rcall   lcd_2us
lcd_2us:
    nop
    ret


; === LCD_putc / LCD_wr_dr ===
; Write a0 to the LCD data register, after busy clears. Two flow
; control codes are intercepted: CR -> LCD_cr, LF -> LCD_lf.
; When uart_enabled = 1, the character is also forwarded to UART0.
LCD:
LCD_putc:
    JK      a0, CR, LCD_cr
    JK      a0, LF, LCD_lf
LCD_wr_dr:
    push    a0
    lds     w, uart_enabled
    tst     w
    breq    lcd_wd_skip_uart
    call    UART0_putc          ; mirror char to serial
lcd_wd_skip_uart:
    pop     a0
    lds     w, LCD_IR
    JB1     w, 7, LCD_wr_dr
    rcall   lcd_4us
    sts     LCD_DR, a0
    ret


; === LCD_cr ===
; Carriage return: cursor to start of current line (keep DDRAM bit 6).
LCD_cr:
    lds     w, LCD_IR
    JB1     w, 7, LCD_cr
    andi    w, 0b01000000
    ori     w, 0b10000000       ; "set DDRAM address" opcode
    rcall   lcd_4us
    sts     LCD_IR, w
    ret


; === LCD_lf ===
; Line feed: jump cursor to start of line 2 (DDRAM 0x40).
LCD_lf:
    push    a0
    ldi     a0, 0x40
    rcall   LCD_pos
    pop     a0
    ret


; --- one-liners that wrap a fixed LCD command ---
LCD_clear:          JW  LCD_wr_ir, 0b00000001
LCD_home:           JW  LCD_wr_ir, 0b00000010
LCD_cursor_left:    JW  LCD_wr_ir, 0b00010000
LCD_cursor_right:   JW  LCD_wr_ir, 0b00010100
LCD_display_left:   JW  LCD_wr_ir, 0b00011000
LCD_display_right:  JW  LCD_wr_ir, 0b00011100
LCD_blink_on:       JW  LCD_wr_ir, 0b00001101
```

```
LCD_blink_off:      JW  LCD_wr_ir, 0b00001100
LCD_cursor_on:      JW  LCD_wr_ir, 0b00001110
LCD_cursor_off:     JW  LCD_wr_ir, 0b00001100


; === LCD_init ===
; Enable external SRAM interface, then send the HD44780 init sequence.
LCD_init:
    in      w, MCUCR
    sbr     w, (1<<SRE)+(1<<SRW10)
    out     MCUCR, w
    CW      LCD_wr_ir, 0b00000001       ; clear display
    CW      LCD_wr_ir, 0b00000110       ; entry mode: Inc=1, Shift=0
    CW      LCD_wr_ir, 0b00001100       ; Display=1, Cursor=0, Blink=0
    CW      LCD_wr_ir, 0b00111000       ; 8-bit, 2 lines, 5x8 font
    ret


; === LCD_pos ===
; Move the cursor to DDRAM address a0 (0x00..0x0F = line 1,
; 0x40..0x4F = line 2). Hooks into the UART mirror so the serial
; output gets a header before line 1 and a separator before line 2.
LCD_pos:
    push    a0
    cpi     a0, 0x00
    brne    lcd_pos_not_l1
    call    uart_lcd_header
    rjmp    lcd_pos_end
lcd_pos_not_l1:
    cpi     a0, 0x40
    brne    lcd_pos_end
    call    uart_lcd_newline
lcd_pos_end:
    pop     a0
    mov     w, a0
    ori     w, 0b10000000               ; "set DDRAM address" opcode
    rjmp    LCD_wr_ir
```

**Fichier : drivers/logique_pointage.asm**

```
; file:    logique_pointage.asm   target ATmega128L-4MHz-STK300
; purpose: pointing math + S3003 servo driver.
;           shortest_path_az    : wrap-aware signed delta on a 360 deg circle
;           angle_to_servo_pwm  : degrees -> OCR1A ticks
;           servo_init / off / on / nudge_up / nudge_down


.equ PWM_MIN     = 500          ; OCR1A at 0 deg
.equ PWM_MAX     = 900          ; OCR1A at 90 deg
.equ PWM_RANGE   = PWM_MAX - PWM_MIN


; === shortest_path_az ===
; Signed minimum-arc delta from (current) to (target) on a 360 deg circle.
;   delta in (180, 360]   -> subtract 360   (go CCW instead)
;   delta in [-360, -180) -> add 360        (go CW instead)
; In:  r19:r18 = target deg, r17:r16 = current deg
; Out: r17:r16 = signed delta in (-180, 180]
; Clobbers: r16, r17, r20
shortest_path_az:
    sub     r18, r16
    sbc     r19, r17
    mov     r16, r18
    mov     r17, r19

    cpi     r16, low(180)
    ldi     r20, high(180)
    cpc     r17, r20
    brge    path_subtract_360

    cpi     r16, low(-180)
    ldi     r20, high(-180)
    cpc     r17, r20
    brlt    path_add_360
    rjmp    shortest_path_end

path_subtract_360:
    subi    r16, low(360)
    sbci    r17, high(360)
    rjmp    shortest_path_end
path_add_360:
    subi    r16, low(-360)              ; +360
    sbci    r17, high(-360)
shortest_path_end:
    ret


; === angle_to_servo_pwm ===
; OCR1A = PWM_MIN + (angle * PWM_RANGE) / 90.
; In:  a1:a0 = elevation in deg (caller clamps to 0..90)
; Out: c1:c0 = OCR1A ticks
angle_to_servo_pwm:
    ldi     b0, low(PWM_RANGE)
    ldi     b1, high(PWM_RANGE)
    rcall   mul22                       ; c1:c0 = angle * PWM_RANGE

    mov     a0, c0
    mov     a1, c1
    ldi     b0, low(90)
    ldi     b1, high(90)
    rcall   div22                       ; c1:c0 = (angle * PWM_RANGE) / 90

    ldi     b0, low(PWM_MIN)
    ldi     b1, high(PWM_MIN)
    add     c0, b0
    adc     c1, b1
    ret


; === Servo hardware driver ===
; Timer1 Fast-PWM Mode 14, /8 prescaler -> 1 tick = 2 us.
; ICR1 = 10000 ticks = 20 ms period. OCR1A = pulse width.
; OC1A drives PB5 directly in hardware (set via COM1A1).

; === servo_init ===
; Configure Timer1 + park OCR1A at TICKS_MIN (0 deg).
servo_init:
    sbi     DDRB, 5                     ; PB5 = OC1A output
    ldi     r16, high(10000)
    out     ICR1H, r16
    ldi     r16, low(10000)
    out     ICR1L, r16
```

```
    ldi     r16, (1<<COM1A1) | (1<<WGM11)
    out     TCCR1A, r16
    ldi     r16, (1<<WGM13) | (1<<WGM12) | (1<<CS11)
    out     TCCR1B, r16
    ; Park at 0 deg: target_ticks = OCR1A = TICKS_MIN (= 1.0 ms pulse).
    ldi     r24, low(TICKS_MIN)
    ldi     r25, high(TICKS_MIN)
    sts     target_ticks, r24
    sts     target_ticks+1, r25
    out     OCR1AH, r25
    out     OCR1AL, r24
    ret


; === servo_off ===
; Disconnect OC1A from PB5 and drive the pin low. Servo goes limp
; after the S3003's analog hold cap discharges (~50 ms).
servo_off:
    in      r16, TCCR1A
    andi    r16, ~(1 << COM1A1)
    out     TCCR1A, r16
    cbi     PORTB, 5
    ret


; === servo_on ===
; Re-engage OC1A (idempotent: harmless if already engaged).
servo_on:
    in      r16, TCCR1A
    ori     r16, (1 << COM1A1)
    out     TCCR1A, r16
    ret


; === servo_nudge_up / servo_nudge_down ===
; target_ticks += / -= TICKS_STEP, clamp to [TICKS_MIN, TICKS_MAX],
; commit to OCR1A.
servo_nudge_up:
    lds     r24, target_ticks
    lds     r25, target_ticks+1
    adiw    r24, TICKS_STEP
    ldi     r16, low(TICKS_MAX)
    ldi     r17, high(TICKS_MAX)
    cp      r16, r24
    cpc     r17, r25
    brsh    servo_apply                 ; new <= MAX
    mov     r24, r16                    ; clamp to MAX
    mov     r25, r17
    rjmp    servo_apply

servo_nudge_down:
    lds     r24, target_ticks
    lds     r25, target_ticks+1
    sbiw    r24, TICKS_STEP
    ldi     r16, low(TICKS_MIN)
    ldi     r17, high(TICKS_MIN)
    cp      r24, r16
    cpc     r25, r17
    brsh    servo_apply                 ; new >= MIN
    mov     r24, r16                    ; clamp to MIN
    mov     r25, r17

servo_apply:
    sts     target_ticks, r24
    sts     target_ticks+1, r25
    out     OCR1AH, r25
    out     OCR1AL, r24
    ret
```

### Fichier : drivers/math_revised.asm

```
; file:    math_revised.asm   target ATmega128L-4MHz-STK300
; purpose: 16-bit math helpers used by the pointing logic.
;           mul22 / div22  : signed 16-bit multiply / divide
;           abs16          : |r17:r16|
;           mod360         : normalize angle to [0, 359]
;           clamp          : clamp angle to [0, MAX_ELEVATION]

.equ MAX_AZIMUTH   = 360
.equ MAX_ELEVATION = 90


; === mul22 ===
; c = a * b (16-bit unsigned -> 32-bit, but we use the low 16).
; In:  a1:a0, b1:b0
; Out: c1:c0 = low 16 of product (c3:c2 contain the high 16 internally)
mul22:
    CLR2    c3, c2                      ; high half of result
    MOV2    c1, c0, b1, b0              ; low half = b
    LSR2    c1, c0                      ; shift LSB into carry
    ldi     w, 16
_m22:
    brcc    PC+3                        ; carry clear -> no add
    ADD2    c3, c2, a1, a0
    ROR4    c3, c2, c1, c0
    DJNZ    w, _m22
    ret


; === div22 ===
; c = a / b ; d = a mod b (16-bit / 16-bit, non-restoring division).
; In:  a1:a0, b1:b0
; Out: c1:c0 = quotient, d1:d0 = remainder
div22:
    MOV2    c1, c0, a1, a0              ; c will hold the quotient
    CLR2    d1, d0                      ; d will hold the remainder
    ldi     w, 16
_d22:
    ROL4    d1, d0, c1, c0              ; shift carry into result
    SUB2    d1, d0, b1, b0
    brcc    PC+3                        ; restore if it went negative
    ADD2    d1, d0, b1, b0
    DJNZ    w, _d22
    ROL2    c1, c0                      ; final shift
    COM2    c1, c0                      ; complement
    ret


; === abs16 ===
; |r17:r16| in two's complement.
abs16:
    tst     r17
    brpl    abs16_end                   ; non-negative -> done
    com     r16
    com     r17
    subi    r16, low(-1)                ; +1 (two's complement)
    sbci    r17, high(-1)
abs16_end:
    ret


; === mod360 ===
; Normalize r17:r16 to [0, 359].
;   < 0   -> += 360 until non-negative
;   >=360 -> -= 360 until below 360
mod360:
mod360_check_neg:
    tst     r17
    brpl    mod360_check_pos
    subi    r16, low(-MAX_AZIMUTH)
    sbci    r17, high(-MAX_AZIMUTH)
    rjmp    mod360_check_neg
mod360_check_pos:
    cpi     r16, low(MAX_AZIMUTH)
    ldi     r18, high(MAX_AZIMUTH)
    cpc     r17, r18
    brlo    mod360_end
    subi    r16, low(MAX_AZIMUTH)
    sbci    r17, high(MAX_AZIMUTH)
    rjmp    mod360_check_pos
mod360_end:
    ret
```

```
; === clamp ===
; Clamp r17:r16 to [0, MAX_ELEVATION]. Used so the elevation servo
; can't be driven past its mechanical limit.
clamp:
    tst     r17
    brpl    clamp_check_upper
    clr     r16                         ; negative -> 0
    clr     r17
    rjmp    clamp_end
clamp_check_upper:
    cpi     r16, MAX_ELEVATION
    ldi     r18, 0
    cpc     r17, r18
    brlo    clamp_end                   ; < 90 -> ok
    breq    clamp_end                   ; == 90 -> ok
    ldi     r16, MAX_ELEVATION
    clr     r17
clamp_end:
    ret
```

### Fichier : drivers/rc5_logic.asm

```
; file:    rc5_logic.asm   target ATmega128L-4MHz-STK300
; purpose: Manchester capture + field extraction for RC5 frames.
;          Called from EXT_INT7 on each PE7 falling edge.


.equ RC5_ADDR_TV = 0x00

.cseg

; === rc5_capture_from_edge ===
; Sample the IR line every T1 microseconds for 14 bits, MSB first.
; Caller is responsible for entering at the start-bit falling edge
; (INT7 in this project). Blocks ~25 ms.
; Out:      b1:b0 = 14 raw bits
; Clobbers: b0, b1, b2, w, u
rc5_capture_from_edge:
    CLR2    b1, b0
    ldi     b2, 14
    WAIT_US (T1/4 - 2)                  ; align to bit centre (S1)
rc5_cap_loop:
    P2C     PINE, 7                     ; pin -> carry  (IR_PORT_IN, IR_PIN)
    ROL2    b1, b0                      ; shift bit into b1:b0
    WAIT_US (T1-4)                      ; next bit centre
    DJNZ    b2, rc5_cap_loop
    com     b0                          ; flip so 1 = "marked" half-bit
    ret


; === rc5_extract_fields ===
; Slice the 14 raw bits into the RC5 sub-fields.
;   bits 13-12 = start bits (S1, S2)   -- checked by caller
;   bit  11    = toggle
;   bits 10-6  = address (5 bits)
;   bits  5-0  = command (6 bits)
; In:  b1:b0 = raw frame
; Out: r16 = cmd, r17 = addr, r18 = toggle (masked, 0 or TOGGLE_MASK)
; Clobbers: r19
rc5_extract_fields:
    ; --- cmd = b0[5:0] ---
    mov     r16, b0
    andi    r16, 0x3F

    ; --- addr = b1[2:0] : b0[7:6] (5 bits) ---
    mov     r17, b1
    lsl     r17
    lsl     r17                         ; b1 bits to addr[4:2]
    mov     r19, b0
    swap    r19
    lsr     r19
    lsr     r19
    andi    r19, 0x03                   ; b0[7:6] -> addr[1:0]
    or      r17, r19
    andi    r17, 0x1F

    ; --- toggle = b1[3] (kept in TOGGLE_MASK bit position) ---
    mov     r18, b1
    andi    r18, 0x08
    ret
```

**Fichier : drivers/timer_manager.asm**

```
; ============================================================================
; Fichier      : timer_manager.asm
; Description  : Timer2 1 ms tick. The ISR drives the overlay countdown in
;                libs/overlay.asm (decrement + dirty flag).
;
;                A free-running sys_time counter + tick_elapsed_since helper
;                used to live here but nothing called them; they were removed
;                in the 2026-05-26 cleanup. If/when a future feature needs
;                wall-clock-ish timestamps, restore them from git history.
; Dépendances  : libs/overlay.asm declares overlay_ms_left / overlay_dirty
;                in SRAM; this ISR is the only writer of those bytes besides
;                main-context cli-guarded code in overlay.asm.
; ============================================================================

.cseg
; ============================================================================
; Fonction    : sys_time_init
; Description : Init Timer2. Mode CTC (WGM21=1), Prescaler /32. OCR2=124.
;               1ms tick @ 4MHz. Active l'interruption OCIE2.
; ============================================================================
sys_time_init:
    push r16
    ; Mode CTC, Prescaler /32
    ldi r16, (1<<WGM21) | (1<<CS21) | (1<<CS20)
    out TCCR2, r16
    ; OCR2 = 124
    ldi r16, 124
    out OCR2, r16
    ; Interrupt Enable
    in r16, TIMSK
    ori r16, (1<<OCIE2)
    out TIMSK, r16
    pop r16
    ret

; ============================================================================
; Fonction    : system_tick (ISR - Timer2 compare match)
; Description : 1 ms tick. Decrement overlay_ms_left (saturating at 0); when
;               the countdown crosses to zero, raise overlay_dirty so the
;               main loop knows to repaint line 2.
; ============================================================================
system_tick:
    ; Sauvegarde stricte du contexte
    push r16
    push r17
    in   r16, SREG
    push r16

    ; Si overlay_ms_left est déjà à 0, rien à faire.
    lds  r16, overlay_ms_left
    lds  r17, overlay_ms_left+1
    tst  r16
    brne sto_dec
    tst  r17
    breq system_tick_end

sto_dec:
    ; ms_left -= 1 (subi/sbci propage la borrow correctement)
    subi r16, 1
    sbci r17, 0
    sts  overlay_ms_left, r16
    sts  overlay_ms_left+1, r17

    ; Si on vient de passer à 0, set dirty = 1
    tst  r16
    brne system_tick_end
    tst  r17
    brne system_tick_end
    ldi  r16, 1
    sts  overlay_dirty, r16

system_tick_end:
    ; Restauration du contexte
    pop  r16
    out  SREG, r16
    pop  r17
    pop  r16
    reti
```

### Fichier : drivers/uart_control.asm

```
; file:    uart_control.asm   target ATmega128L-4MHz-STK300
; purpose: UART mirror UI + serial command injector.
;
;          uart_poll reads any pending RX byte and maps it to an RC5
;          equivalent (last_cmd) so the FSM can be driven over USB.
;          uart_lcd_header / _newline / _footer paint a live ANSI
;          view of the LCD over UART (only when uart_enabled = 1).
;          toggle_uart flips the mirror on/off and shows an overlay.
;
;          The UART register init is in the course library:
;          libraries_cours/uart_et_strings/uart.asm (UART0_init).

.include "libraries_cours/uart_et_strings/uart.asm"

.dseg
uart_enabled: .byte 1       ; 0 = mute (no TX), 1 = mirror active
.cseg


; === uart_init_app ===
; Configure UART0 then start in disabled state. The user can MUTE
; (CMD_MUTE) on the remote to turn the mirror on.
uart_init_app:
    rcall   UART0_init
    ldi     w, 0
    sts     uart_enabled, w
    ret


; === uart_poll ===
; Called every main-loop iteration. If a byte is waiting in UDR0,
; translate it to an RC5 command and forge a fresh-press event so
; main_dispatch processes it.
; Out: Z = 0 (w = 1) if an event was injected, Z = 1 (w = 0) otherwise.
uart_poll:
    in      w, UCSR0A
    sbrs    w, RXC0                     ; data available?
    rjmp    up_none

    in      w, UDR0

    ; --- digits 0..9 (RC5 codes match 1:1) ---
    cpi     w, '0'
    brlo    up_not_num
    cpi     w, '9'+1
    brsh    up_not_num
    subi    w, '0'
    rjmp    up_set
up_not_num:

    cpi     w, ' '
    breq    up_space
    cpi     w, 'x'
    breq    up_x
    cpi     w, 'X'
    breq    up_x
    cpi     w, 'm'
    breq    up_m
    cpi     w, 'M'
    breq    up_m
    cpi     w, 'p'
    breq    up_p
    cpi     w, 'P'
    breq    up_p
    cpi     w, 'w'
    breq    up_w
    cpi     w, 'W'
    breq    up_w
    cpi     w, 's'
    breq    up_s
    cpi     w, 'S'
    breq    up_s
    cpi     w, 'a'
    breq    up_a
    cpi     w, 'A'
    breq    up_a
    cpi     w, 'd'
    breq    up_d
    cpi     w, 'D'
    breq    up_d
    cpi     w, 'l'
    breq    up_l
```

```
    cpi     w, 'L'
    breq    up_l
    cpi     w, 'u'
    breq    up_u
    cpi     w, 'U'
    breq    up_u
    rjmp    up_none

; --- key -> RC5 command map ---
up_u:     ldi w, CMD_MUTE
          rjmp up_set
up_p:     ldi w, CMD_POWER
          rjmp up_set
up_x:     ldi w, CMD_DEL
          rjmp up_set
up_m:     ldi w, CMD_AV
          rjmp up_set
up_space: ldi w, CMD_GUIDE
          rjmp up_set
up_w:     ldi w, CMD_VOL_UP
          rjmp up_set
up_s:     ldi w, CMD_VOL_DN
          rjmp up_set
up_a:     ldi w, CMD_CH_UP
          rjmp up_set
up_d:     ldi w, CMD_CH_DN
          rjmp up_set

up_l:
    rcall   uart_print_sat_list         ; not an FSM key -- print + bail
    rjmp    up_none


; --- up_set: forge a fresh-press event matching the IR ISR contract ---
up_set:
    sts     last_cmd, w
    clr     w
    sts     held_count, w               ; always treat UART press as fresh
    lds     w, last_tog
    com     w                           ; flip toggle so the main loop sees a fresh press
    sts     last_tog, w
    ldi     w, 1                        ; w = 1
    tst     w                           ; Z = 0 -> caller dispatches
    ret

up_none:
    clr     w                           ; Z = 1 -> caller goes back to polling
    ret


; === toggle_uart ===
; Flip uart_enabled, then show "UART ON"/"UART OFF" as a 500 ms overlay.
toggle_uart:
    lds     w, uart_enabled
    ldi     r25, 1
    eor     w, r25
    sts     uart_enabled, w

    tst     w
    brne    tu_on
    ldi     ZL, low(str_ui_u_off*2)
    ldi     ZH, high(str_ui_u_off*2)
    rjmp    tu_copy
tu_on:
    ldi     ZL, low(str_ui_u_on*2)
    ldi     ZH, high(str_ui_u_on*2)
tu_copy:
    rcall   copy_flash_to_l2_buf
    ldi     r24, low(500)
    ldi     r25, high(500)
    rcall   overlay_show
    ret


; === uart_print_sat_list ===
; Dump every valid EEPROM slot over UART as "[ID] NAME AZ:xxx EL:xxx".
; No-op when uart_enabled = 0 so 'l' in the disabled state is silent.
uart_print_sat_list:
    lds     w, uart_enabled
    tst     w
    brne    psl_run
    ret
psl_run:
    push    ZL
    push    ZH
```

```
    push    w
    ldi     ZL, low(str_ui_sat_hdr*2)
    ldi     ZH, high(str_ui_sat_hdr*2)
    call    uart_print_str_P
    pop     w
    pop     ZH
    pop     ZL

    ldi     r25, 0                      ; slot id
ups_loop:
    cpi     r25, 32
    brsh    ups_end

    mov     r16, r25
    rcall   load_satellite
    brne    ups_next                    ; empty -> skip

    ldi     a0, '['
    rcall   UART0_putc
    mov     a0, r25
    inc     a0                          ; display 1-based
    rcall   uart_print_dec2
    ldi     a0, ']'
    rcall   UART0_putc
    ldi     a0, ' '
    rcall   UART0_putc

    ldi     XL, low(current_sat_name)
    ldi     XH, high(current_sat_name)
ups_name_loop:
    ld      a0, X+
    tst     a0
    breq    ups_name_end
    rcall   UART0_putc
    rjmp    ups_name_loop
ups_name_end:

    ldi     a0, ' '
    rcall   UART0_putc
    ldi     a0, 'A'
    rcall   UART0_putc
    ldi     a0, 'Z'
    rcall   UART0_putc
    ldi     a0, ':'
    rcall   UART0_putc
    lds     a0, target_az_auto
    lds     a1, target_az_auto+1
    rcall   uart_print_dec3

    ldi     a0, ' '
    rcall   UART0_putc
    ldi     a0, 'E'
    rcall   UART0_putc
    ldi     a0, 'L'
    rcall   UART0_putc
    ldi     a0, ':'
    rcall   UART0_putc
    lds     a0, target_el_auto
    clr     a1
    rcall   uart_print_dec3

    ldi     a0, '\r'
    rcall   UART0_putc
    ldi     a0, '\n'
    rcall   UART0_putc

ups_next:
    inc     r25
    rjmp    ups_loop

ups_end:
    ldi     a0, '\r'
    rcall   UART0_putc
    ldi     a0, '\n'
    rcall   UART0_putc
    ret


; === uart_print_dec2 / dec3 / dec4_tmp ===
; Render a value as N ASCII digits over UART. They hijack scan_cache_az
; as scratch since SCAN's bitmap-walker isn't running concurrently with
; UART output. Caller's a0 (and a1 for dec3/4) holds the value.
uart_print_dec3:
    push    XL
    push    XH
```

```
    ldi     XL, low(scan_cache_az)
    ldi     XH, high(scan_cache_az)
    rcall   u16_to_dec3
    lds     a0, scan_cache_az
    rcall   UART0_putc
    lds     a0, scan_cache_az+1
    rcall   UART0_putc
    lds     a0, scan_cache_az+2
    rcall   UART0_putc
    pop     XH
    pop     XL
    ret

uart_print_dec2:
    push    XL
    push    XH
    ldi     XL, low(scan_cache_az)
    ldi     XH, high(scan_cache_az)
    rcall   u16_to_dec2
    lds     a0, scan_cache_az
    rcall   UART0_putc
    lds     a0, scan_cache_az+1
    rcall   UART0_putc
    pop     XH
    pop     XL
    ret

uart_print_dec4_tmp:
    push    XL
    push    XH
    ldi     XL, low(scan_cache_az)
    ldi     XH, high(scan_cache_az)
    call    u16_to_dec4
    lds     a0, scan_cache_az
    call    UART0_putc
    lds     a0, scan_cache_az+1
    call    UART0_putc
    lds     a0, scan_cache_az+2
    call    UART0_putc
    lds     a0, scan_cache_az+3
    call    UART0_putc
    pop     XH
    pop     XL
    ret


; === uart_print_str_P ===
; LPM-print a null-terminated flash string. Caller sets Z to the byte
; address (low(2*lbl)/high). Pushes w only.
uart_print_str_P:
    push    w
upsp_loop:
    lpm     a0, Z+
    tst     a0
    breq    upsp_end
    call    UART0_putc
    rjmp    upsp_loop
upsp_end:
    pop     w
    ret


; === LCD mirror over UART ===
; Called from drivers/lcd.asm's LCD_pos when the cursor moves to
; line 1 (0x00) or line 2 (0x40). header emits the status line +
; "L1: "; newline emits "L2: "; footer closes the block. Each is
; a no-op while uart_enabled = 0 (zero RX cost when disabled).
uart_lcd_header:
    lds     w, uart_enabled
    tst     w
    brne    ulh_run
    ret
ulh_run:
    push    ZL
    push    ZH
    push    w

    ldi     ZL, low(str_ui_sep*2)
    ldi     ZH, high(str_ui_sep*2)
    call    uart_print_str_P

    ldi     ZL, low(str_ui_mod*2)
    ldi     ZH, high(str_ui_mod*2)
    call    uart_print_str_P
```

```
    lds     w, mode
    cpi     w, MODE_AUTO
    brne    ulh_m1
    ldi     ZL, low(str_ui_auto*2)
    ldi     ZH, high(str_ui_auto*2)
    rjmp    ulh_md
ulh_m1:
    cpi     w, MODE_MANUEL
    brne    ulh_m2
    ldi     ZL, low(str_ui_manu*2)
    ldi     ZH, high(str_ui_manu*2)
    rjmp    ulh_md
ulh_m2:
    ldi     ZL, low(str_ui_scan*2)
    ldi     ZH, high(str_ui_scan*2)
ulh_md:
    call    uart_print_str_P

    ldi     ZL, low(str_ui_layer*2)
    ldi     ZH, high(str_ui_layer*2)
    call    uart_print_str_P

    lds     w, fsm_layer
    tst     w
    brne    ulh_l1
    ldi     ZL, low(str_ui_brws*2)
    ldi     ZH, high(str_ui_brws*2)
    rjmp    ulh_ld
ulh_l1:
    ldi     ZL, low(str_ui_in*2)
    ldi     ZH, high(str_ui_in*2)
ulh_ld:
    call    uart_print_str_P

    ; --- AZ ticks ---
    ldi     ZL, low(str_ui_az*2)
    ldi     ZH, high(str_ui_az*2)
    call    uart_print_str_P
    lds     a0, current_az_steps
    lds     a1, current_az_steps+1
    call    uart_print_dec4_tmp

    ; --- EL in microseconds (= ticks * 2) ---
    ldi     ZL, low(str_ui_el*2)
    ldi     ZH, high(str_ui_el*2)
    call    uart_print_str_P
    lds     a0, target_ticks
    lds     a1, target_ticks+1
    lsl     a0
    rol     a1
    call    uart_print_dec4_tmp

    ldi     ZL, low(str_ui_us*2)
    ldi     ZH, high(str_ui_us*2)
    call    uart_print_str_P

    ; --- ACTION: MOVING / IDLE ---
    ldi     ZL, low(str_ui_act*2)
    ldi     ZH, high(str_ui_act*2)
    call    uart_print_str_P
    lds     w, flag_is_moving
    tst     w
    breq    ulh_act_idl
    ldi     ZL, low(str_ui_mov*2)
    ldi     ZH, high(str_ui_mov*2)
    rjmp    ulh_act_d
ulh_act_idl:
    ldi     ZL, low(str_ui_idl*2)
    ldi     ZH, high(str_ui_idl*2)
ulh_act_d:
    call    uart_print_str_P

    ldi     ZL, low(str_ui_lcd*2)
    ldi     ZH, high(str_ui_lcd*2)
    call    uart_print_str_P

    ldi     ZL, low(str_ui_l1*2)
    ldi     ZH, high(str_ui_l1*2)
    call    uart_print_str_P

    pop     w
    pop     ZH
    pop     ZL
    ret
```

```
uart_lcd_newline:
    lds     w, uart_enabled
    tst     w
    brne    uln_run
    ret
uln_run:
    push    ZL
    push    ZH
    ldi     ZL, low(str_ui_l2hdr*2)
    ldi     ZH, high(str_ui_l2hdr*2)
    call    uart_print_str_P
    pop     ZH
    pop     ZL
    ret


uart_lcd_footer:
    lds     w, uart_enabled
    tst     w
    brne    ulf_run
    ret
ulf_run:
    push    ZL
    push    ZH
    ldi     ZL, low(str_ui_ftr*2)
    ldi     ZH, high(str_ui_ftr*2)
    call    uart_print_str_P
    pop     ZH
    pop     ZL
    ret


; === ANSI-decorated strings ===
; ESC[H = home, ESC[K = erase to EOL, ESC[J = erase to EOS.
str_ui_sep:     .db 27, "[H------------------------", 27, "[K", 13, 10, 0, 0
str_ui_mod:     .db "MODE: [", 0
str_ui_auto:    .db "AUTO   ]", 0, 0
str_ui_manu:    .db "MANUEL ]", 0, 0
str_ui_scan:    .db "SCAN   ]", 0, 0
str_ui_layer:   .db "  LAYER: [", 0, 0
str_ui_brws:    .db "BROWSING]", 27, "[K", 13, 10, 0, 0
str_ui_in:      .db "IN      ]", 27, "[K", 13, 10, 0, 0
str_ui_az:      .db "AZ: ", 0, 0
str_ui_el:      .db " ticks   EL: ", 0
str_ui_us:      .db " us", 27, "[K", 13, 10, 0, 0
str_ui_act:     .db "ACTION: ", 0, 0
str_ui_mov:     .db "MOVING", 27, "[K", 13, 10, 0
str_ui_idl:     .db "IDLE  ", 27, "[K", 13, 10, 0
str_ui_lcd:     .db "---------LCD------------", 27, "[K", 13, 10, 0
str_ui_l1:      .db "L1: ", 0, 0
str_ui_l2hdr:   .db 27, "[K", 13, 10, "L2: ", 0
str_ui_ftr:     .db 27, "[K", 13, 10, "------------------------", 27, "[K", 13, 10, 0, 0
str_ui_sat_hdr: .db 27, "[10;1H", 27, "[J--SATS--", 13, 10, 0, 0
str_ui_u_on:    .db "UART ON         ", 0, 0
str_ui_u_off:   .db "UART OFF        ", 0, 0
```

**Fichier : libs/constants.inc**

```
; file:    constants.inc   target ATmega128L-4MHz-STK300
; purpose: project-wide constants (FSM enums, RC5 codes, motion limits).
;          .equ-only -- one place to change a button mapping or a step
;          threshold. Inline literals elsewhere are forbidden.

.nolist

; === RC5 protocol ===
.equ  T1          = 1870            ; one bit period (us) @ 4 MHz internal RC
.equ  TOGGLE_MASK = 0b00001000      ; b1[3] after rc5_capture (un-shifted)

; === FSM modes ===
; SAVE was dropped: save is in MANUEL (GUIDE x2), delete is in AUTO (DEL x2).
.equ  MODE_AUTO   = 0               ; database browser + go-to
.equ  MODE_MANUEL = 1               ; jog + inline save
.equ  MODE_SCAN   = 2               ; 360 deg sweep + results browser
.equ  MODE_COUNT  = 3               ; for the mod-N wrap in cycle_mode_*

; === FSM layers (AV button toggles between them) ===
.equ  LAYER_BROWSING = 0            ; CH+/- cycles modes; menu view
.equ  LAYER_IN_MODE  = 1            ; mode handler owns the buttons

; === SCAN sub-states (display.asm reads ST_SCAN_RESULTS_BROWSE) ===
.equ  ST_SCAN_IDLE           = 0    ; waits for GUIDE
.equ  ST_SCAN_SWEEPING       = 1    ; one degree per main-loop iteration
.equ  ST_SCAN_NO_RESULTS     = 2    ; sweep done, zero hits
.equ  ST_SCAN_RESULTS_BROWSE = 3    ; sweep done, AUTO browser filtered

; === RC5 command codes (Vivanco UR Z2, TV map) ===
; "verified" tags are codes seen on the LCD debug line at C=xx during
; bench testing. Others are textbook RC5 TV defaults.
.equ  CMD_0       = 0x00
.equ  CMD_1       = 0x01
.equ  CMD_2       = 0x02
.equ  CMD_3       = 0x03
.equ  CMD_4       = 0x04
.equ  CMD_5       = 0x05
.equ  CMD_6       = 0x06
.equ  CMD_7       = 0x07
.equ  CMD_8       = 0x08
.equ  CMD_9       = 0x09
.equ  CMD_DEL     = 0x0A            ; destructive (verified)
.equ  CMD_MUTE    = 0x0D            ; toggle UART mirror
.equ  CMD_POWER   = 0x0C            ; emergency stop -- always wins
.equ  CMD_VOL_UP  = 0x10            ; MANUEL elevation +
.equ  CMD_VOL_DN  = 0x11            ; MANUEL elevation -
.equ  CMD_GUIDE   = 0x22            ; positive action (travel/save/start) (verified)
.equ  CMD_CH_UP   = 0x20            ; next mode / next slot / az +
.equ  CMD_CH_DN   = 0x21            ; prev mode / prev slot / az -
.equ  CMD_AV      = 0x38            ; layer toggle + chirp (verified)


; === Elevation servo (Futaba S3003, Timer1 OC1A) ===
; ICR1 = TIMER_TOP = 10000 ticks = 20 ms period (50 Hz), tick = 2 us.
; OCR1A = TICKS_MIN..TICKS_MAX = 500..900 (1.0..1.8 ms pulse).
; servo_init parks OCR1A at TICKS_MIN (0 deg).
.equ  TIMER_TOP   = 10000
.equ  TICKS_MIN   = 500             ; 1.0 ms pulse (0 deg)
.equ  TICKS_MAX   = 900             ; 1.8 ms pulse (90 deg)
.equ  TICKS_MID   = 750             ; 1.5 ms pulse (45 deg, unused at boot)
.equ  TICKS_STEP  = 10              ; 20 us per VOL+/- press

; === Azimuth stepper (X27.168, PORTD low nibble) ===
; Free-spinning, no end stop. 6 motor patterns per rotor turn, 1:180
; gearbox -> 1080 patterns per output revolution (1/3 deg per pattern).
.equ  AZ_MAX_STEPS    = 1080
.equ  AZ_STEP_PARTIAL = 3           ; partials per displayed degree
.equ  STEPPER_DELAY_MS = 3          ; coil settle between patterns

; === Hold-to-accelerate (MANUEL jog) ===
; held_count = consecutive RC5 frames with the same toggle. RC5
; auto-repeat is ~9 Hz, so each threshold maps to a hold duration:
;   held <  10 (~1 s)  -> 1 deg/press   (precise tap)
;   held <  30 (~3 s)  -> 5 deg/press   (~50 deg/s)
;   held >= 30         -> 10 deg/press  (~100 deg/s)
.equ  HOLD_THRESHOLD_MED  = 10
.equ  HOLD_THRESHOLD_FAST = 30
.equ  HOLD_STEP_SLOW      = 1
.equ  HOLD_STEP_MED       = 5
.equ  HOLD_STEP_FAST      = 10

.list
```

### Fichier : libs/definitions.asm

```
; file: definitions.asm   target ATmega128L-4MHz-STK300
; purpose library, definition of addresses and constants
; 20171114 A.S.

; === definitions  ===
.nolist         ; do not include in listing
.set    clock   = 4000000

.def    char    = r0    ; character (ASCII)
.def    _sreg   = r1    ; saves the status during interrupts
.def    _u  = r2    ; saves working reg u during interrupt
.def    u   = r3    ; scratch register (macros, routines)

.def    e0  = r4    ; temporary reg for PRINTF
.def    e1  = r5

.equ    c   = 8
.def    c0  = r8    ; 8-byte register c
.def    c1  = r9
.def    c2  = r10
.def    c3  = r11

.equ    d   = 12    ; 4-byte register d (overlapping with c)
.def    d0  = r12
.def    d1  = r13
.def    d2  = r14
.def    d3  = r15

.def    w   = r16   ; working register for macros
.def    _w  = r17   ; working register for interrupts

.equ    a   = 18
.def    a0  = r18   ; 4-byte register a
.def    a1  = r19
.def    a2  = r20
.def    a3  = r21

.equ    b   = 22
.def    b0  = r22   ; 4-byte register b
.def    b1  = r23
.def    b2  = r24
.def    b3  = r25

.equ    px  = 26    ; pointer x
.equ    py  = 28    ; pointer y
.equ    pz  = 30    ; pointer z

; === ASCII codes
.equ    BEL =0x07   ; bell
.equ    HT  =0x09   ; horizontal tab
.equ    TAB =0x09   ; tab
.equ    LF  =0x0a   ; line feed
.equ    VT  =0x0b   ; vertical tab
.equ    FF  =0x0c   ; form feed
.equ    CR  =0x0d   ; carriage return
.equ    SPACE   =0x20   ; space code
.equ    DEL =0x7f   ; delete
.equ    BS  =0x08   ; back space

; === STK-300 ===
.equ    LED = PORTB ; LEDs on STK-300
.equ    BUTTON  = PIND  ; buttons on the STK-300

; === module M2 (encoder/speaker/IR remote) ===
.equ    SPEAKER = 2 ; piezo speaker
.equ    ENCOD_A = 4 ; angular encoder A
.equ    ENCOD_B = 5 ; angular encoder B
.equ    ENCOD_I = 6 ; angular encoder button
.equ    IR  = 7 ; IR module for PCM remote control system

; === module M5 (I2C/1Wire) ===
.equ    SCL = 0 ; I2C serial clock
.equ    SDA = 1 ; I2C serial data
.equ    DQ  = 5 ; Dallas 1Wire
                ; master transmitter status codes, Table 88
.equ    I2CMT_START = 0x08      ; start
.equ    I2CMT_REPSTART = 0x10   ; repeated start
.equ    I2CMT_SLA_ACK= 0x18     ; slave ack
.equ    I2CMT_SLA_NOACK = 0x20  ; slave no ack
.equ    I2CMT_DATA_ACK = 0x28   ; data write, ack
.equ    I2CMT_DATA_NOACK = 0x30 ; data write, no ack
                ; master receiver status codes, Table 89
```

```
.equ    I2CMR_SLA_ACK   = 0x40  ; slave address ack
.equ    I2CMR_SLA_NACK  = 0x48  ; slave address no ack
.equ    I2CMR_DATA_ACK = 0x50   ; master data ack
.equ    I2CMR_DATA_NACK= 0x58   ; master data no ack

; === module M4 (Keyboard/Sharp/Servo) ===
.equ    KB_CLK  = 0 ; PC-AT keyboard clock line
.equ    KB_DAT  = 1 ; PC-AT keyboard data line
.equ    GP2_CLK = 2 ; Sharp GP2D02 distance measuring sensor
.equ    GP2_DAT = 3 ; Sharp GP2D02 distance measuring sensor
.equ    GP2_AVAL = 3; Shart GP2Y0A21 distance measuring sensor
.equ    SERVO1  = 4 ; Futaba position servo

; === module M3 (potentiometer/BNC) ===
.equ    POT = 0 ; potentiometer
.equ    BNC1    = 2 ; BNC input
.equ    BNC2    = 4 ; BNC input
.list
```

### Fichier : libs/display.asm

```
; file:    display.asm    target ATmega128L-4MHz-STK300
; purpose: LCD rendering. draw_lcd repaints both lines from FSM state;
;          format_current_az_el is the shared AZ/EL formatter used by
;          BROWSING and MANUEL. Decimal helpers u16_to_dec2/3/4 live here.
;          Port writes happen inside drivers/lcd.asm; we only call
;          LCD_pos / LCD_putc and the UART mirror.


; === draw_lcd ===
; Repaint both LCD lines, then emit the UART mirror trailer.
draw_lcd:
    rcall   draw_line1
    rcall   draw_line2
    call    uart_lcd_footer
    ret


; === draw_line1 ===
; Paint the title line. Layout depends on mode + layer + overlay:
;   overlay active        -> 16 blanks (modal popup centered on L2)
;   AUTO    BROWSING      -> "MODE: AUTO"
;   AUTO    IN_MODE       -> auto_draw_l1 ("Sat:" / "GO TO")
;   MANUEL  BROWSING      -> "MODE: MANUEL"
;   MANUEL  IN_MODE       -> "MANUEL MOVING"
;   SCAN    BROWSING      -> "MODE: SCAN"
;   SCAN    IN_MODE       -> "MODE: SCAN" unless RESULTS_BROWSE
;   SCAN    RESULTS_BROWSE-> reuses auto_draw_l1 (same sat-name view)
draw_line1:
    ldi     a0, 0x00                    ; DDRAM 0x00 = start of line 1
    rcall   LCD_pos

    ; Overlay path: blank line 1 so AUTO's stale "Sat: SAT 12" doesn't
    ; appear next to a "DELETED" L2 overlay (delete advances cursor
    ; BEFORE arming the overlay).
    rcall   overlay_active
    breq    dl1_normal                  ; Z=1 -> no overlay
    rjmp    dl1_blank

dl1_normal:
    lds     w, mode
    cpi     w, MODE_AUTO
    breq    dl1_auto
    cpi     w, MODE_MANUEL
    breq    dl1_manuel
    ; --- SCAN ---
    ; RESULTS_BROWSE reuses AUTO's line 1 so the post-sweep browser
    ; feels identical to AUTO.
    lds     w, fsm_layer
    cpi     w, LAYER_IN_MODE
    brne    dl1_scan_default
    lds     w, sub_state
    cpi     w, ST_SCAN_RESULTS_BROWSE
    brne    dl1_scan_default
    rjmp    auto_draw_l1
dl1_scan_default:
    ldi     ZL, low(2*str_scan)
    ldi     ZH, high(2*str_scan)
    rjmp    print_z_string

dl1_auto:
    lds     w, fsm_layer
    cpi     w, LAYER_IN_MODE
    breq    dl1_auto_custom
    ldi     ZL, low(2*str_auto)
    ldi     ZH, high(2*str_auto)
    rjmp    print_z_string
dl1_auto_custom:
    rjmp    auto_draw_l1

dl1_manuel:
    lds     w, fsm_layer
    cpi     w, LAYER_IN_MODE
    breq    dl1_manuel_in_mode
    ldi     ZL, low(2*str_manuel)
    ldi     ZH, high(2*str_manuel)
    rjmp    print_z_string
dl1_manuel_in_mode:
    ldi     ZL, low(2*str_manuel_moving)
    ldi     ZH, high(2*str_manuel_moving)
    rjmp    print_z_string

dl1_blank:
```

```
    ldi     _w, 16                      ; print 16 spaces
dl1_blank_loop:
    ldi     a0, ' '
    rcall   LCD_putc
    dec     _w
    brne    dl1_blank_loop
    ret


; === Flash strings (line 1) ===
; 16 chars exact + null + word-align pad.
str_auto:          .db "MODE: AUTO      ", 0, 0
str_manuel:        .db "MODE: MANUEL    ", 0, 0
str_manuel_moving: .db "MANUEL MOVING   ", 0, 0
str_scan:          .db "MODE: SCAN      ", 0, 0


; === print_z_string ===
; Print null-terminated flash string at Z (byte address: low(2*lbl)/high)
; until the null. Auto-increments Z. Clobbers: a0, Z.
print_z_string:
    lpm     a0, Z+
    tst     a0
    breq    pzs_done
    rcall   LCD_putc
    rjmp    print_z_string
pzs_done:
    ret


; === draw_line2 ===
; All paths funnel through lcd_l2_buf:
;   overlay active -> caller already wrote the message into the buffer
;   BROWSING       -> refresh buffer with live AZ/EL via format_current_az_el
;   IN_MODE        -> mode handler / init_l2_for_mode already filled it
; Then we print the 16 bytes literally (no NUL scan -- line is 16 chars).
draw_line2:
    ldi     a0, 0x40                    ; DDRAM 0x40 = start of line 2
    rcall   LCD_pos

    rcall   overlay_active
    brne    dl2_print                   ; overlay up -> print buffer as-is

    lds     w, fsm_layer
    cpi     w, LAYER_IN_MODE
    breq    dl2_print
    rcall   format_current_az_el        ; BROWSING: refresh from live state

dl2_print:
    ldi     XL, low(lcd_l2_buf)
    ldi     XH, high(lcd_l2_buf)
    ldi     _w, 16
dl2_print_loop:
    ld      a0, X+
    rcall   LCD_putc
    dec     _w
    brne    dl2_print_loop
    ret


; === format_current_az_el ===
; Render "AZ:xxx EL:xxx   " into lcd_l2_buf.
;   AZ digits at [3..5]   = current_az_steps / AZ_STEP_PARTIAL (/3)
;   EL digits at [10..12] = (target_ticks - TICKS_MIN) * 9 / 40
;                           (inverse of angle_to_servo_pwm)
; In:      SRAM current_az_steps, target_ticks
; Out:     SRAM lcd_l2_buf[0..15]
; Clobbers: w, _w, a, b, c, d0, X, Z
format_current_az_el:
    ; --- copy template into buffer ---
    ldi     ZL, low(2*str_az_el_template)
    ldi     ZH, high(2*str_az_el_template)
    ldi     XL, low(lcd_l2_buf)
    ldi     XH, high(lcd_l2_buf)
    ldi     _w, 16
faze_tmpl_loop:
    lpm     w, Z+
    st      X+, w
    dec     _w
    brne    faze_tmpl_loop

    ; --- AZ digits ---
    lds     a0, current_az_steps
    lds     a1, current_az_steps+1
    ldi     b0, AZ_STEP_PARTIAL
```

```
    ldi     b1, 0
    rcall   div22                       ; c1:c0 = steps / 3
    mov     a0, c0
    mov     a1, c1
    ldi     XL, low(lcd_l2_buf + 3)
    ldi     XH, high(lcd_l2_buf + 3)
    rcall   u16_to_dec3

    ; --- EL digits: (ticks - 500) * 9 / 40, clamp negative to 0 ---
    lds     a0, target_ticks
    lds     a1, target_ticks+1
    ldi     b0, low(TICKS_MIN)
    ldi     b1, high(TICKS_MIN)
    sub     a0, b0
    sbc     a1, b1
    brcc    faze_el_pos
    clr     a0
    clr     a1
faze_el_pos:
    mov     c0, a0                      ; save *1 copy
    mov     c1, a1
    lsl     a0
    rol     a1
    lsl     a0
    rol     a1
    lsl     a0
    rol     a1                          ; a = (ticks-500) * 8
    add     a0, c0
    adc     a1, c1                      ; a = (ticks-500) * 9
    ldi     b0, 40
    ldi     b1, 0
    rcall   div22                       ; c1:c0 = el_deg
    mov     a0, c0
    mov     a1, c1
    ldi     XL, low(lcd_l2_buf + 10)
    ldi     XH, high(lcd_l2_buf + 10)
    rcall   u16_to_dec3
    ret

str_az_el_template:  .db "AZ:000 EL:000   ", 0, 0


; === u16_to_dec3 ===
; 16-bit value -> 3 ASCII digits at X (leading zeros kept).
; Successive-subtraction (slow but small). Inputs >= 1000 corrupt the
; hundreds digit; callers stay in [0, 999] (az_steps/3 max = 359).
; In:      a1:a0 = value, X = dest
; Out:     3 bytes at *X..*(X+2), X += 3
; Clobbers: w, _w, a, b
u16_to_dec3:
    ; --- hundreds ---
    ldi     _w, '0' - 1                 ; pre-decrement, loop pre-increments
    ldi     b0, low(100)
    ldi     b1, high(100)
u16_h_loop:
    inc     _w
    sub     a0, b0
    sbc     a1, b1
    brcc    u16_h_loop                  ; still >= 0 -> another 100 fit
    add     a0, b0                      ; over-subtracted, restore one 100
    adc     a1, b1
    st      X+, _w

    ; --- tens (a1 is now 0; ignore high byte) ---
    ldi     _w, '0' - 1
u16_t_loop:
    inc     _w
    subi    a0, 10
    brcc    u16_t_loop
    subi    a0, -10                     ; add 10 back
    st      X+, _w

    ; --- ones ---
    subi    a0, -'0'                    ; +'0' (subi -k = add k)
    st      X+, a0
    ret


; === u16_to_dec2 ===
; 8-bit value -> 2 ASCII digits at X. Caller must keep a0 < 100.
; In:      a0 = 0..99, X = dest
; Out:     2 bytes at *X..*(X+1), X += 2
u16_to_dec2:
    ldi     _w, '0' - 1
u16_t_loop2:
```

```
    inc     _w
    subi    a0, 10
    brcc    u16_t_loop2
    subi    a0, -10
    st      X+, _w
    subi    a0, -'0'
    st      X+, a0
    ret


; === u16_to_dec4 ===
; 16-bit value -> 4 ASCII digits at X. Used by the UART mirror to print
; raw step counts / pulse-widths (uart_print_dec4_tmp).
; In:      a1:a0 = value (>9999 will overflow the leading digit), X = dest
; Out:     4 bytes at *X..*(X+3), X += 4
u16_to_dec4:
    ; --- thousands ---
    ldi     _w, '0' - 1
    ldi     b0, low(1000)
    ldi     b1, high(1000)
u16_k_loop:
    inc     _w
    sub     a0, b0
    sbc     a1, b1
    brcc    u16_k_loop
    add     a0, b0
    adc     a1, b1
    st      X+, _w

    ; --- hundreds ---
    ldi     _w, '0' - 1
    ldi     b0, low(100)
    ldi     b1, high(100)
u16_h_loop4:
    inc     _w
    sub     a0, b0
    sbc     a1, b1
    brcc    u16_h_loop4
    add     a0, b0
    adc     a1, b1
    st      X+, _w

    ; --- tens (a1 = 0 after the previous restores) ---
    ldi     _w, '0' - 1
u16_t_loop4:
    inc     _w
    subi    a0, 10
    brcc    u16_t_loop4
    subi    a0, -10
    st      X+, _w

    ; --- ones ---
    subi    a0, -'0'
    st      X+, a0
    ret
```

**Fichier : libs/macros.asm**

```
; file: macros.asm   target ATmega128L-4MHz-STK300
; purpose library, general-purpose macros
; author (c) R.Holzer (adapted MICRO210/EE208 A.Schmid)
; v2019.01 20180820 AxS

; ==============
;   pointers
; ==============

; --- loading an immediate into a pointer XYZ,SP ---
.macro  LDIX    ; sram
    ldi xl, low(@0)
    ldi xh,high(@0)
    .endmacro
.macro  LDIY    ; sram
    ldi yl, low(@0)
    ldi yh,high(@0)
    .endmacro
.macro  LDIZ    ; sram
    ldi zl, low(@0)
    ldi zh,high(@0)

    .endmacro
.macro  LDZD    ; sram, reg ; sram+reg -> Z
    mov zl,@1
    clr zh
    subi    zl, low(-@0)
    sbci    zh,high(-@0)
    .endmacro
.macro  LDSP    ; sram
    ldi r16, low(@0)
    out spl,r16
    ldi r16,high(@0)
    out sph,r16
    .endmacro

; --- load/store SRAM addr into pointer XYZ ---
.macro  LDSX    ; sram
    lds xl,@0
    lds xh,@0+1
    .endmacro
.macro  LDSY    ; sram
    lds yl,@0
    lds yh,@0+1
    .endmacro
.macro  LDSZ    ; sram
    lds zl,@0
    lds zh,@0+1
    .endmacro
.macro  STSX    ; sram
    sts @0,  xl
    sts @0+1,xh
    .endmacro
.macro  STSY    ; sram
    sts @0,  yl
    sts @0+1,yh
    .endmacro
.macro  STSZ    ; sram
    sts @0,  zl
    sts @0+1,zh
    .endmacro

; --- push/pop pointer XYZ ---
.macro  PUSHX           ; push X
    push    xl
    push    xh
    .endmacro
.macro  POPX            ; pop X
    pop xh
    pop xl
    .endmacro

.macro  PUSHY           ; push Y
    push    yl
    push    yh
    .endmacro
.macro  POPY            ; pop Y
    pop yh
    pop yl
    .endmacro

.macro  PUSHZ           ; push Z
```

```
    push    zl
    push    zh
    .endmacro
.macro  POPZ            ; pop Z
    pop zh
    pop zl
    .endmacro

; --- multiply/divide Z ---
.macro  MUL2Z           ; multiply Z by 2
    lsl zl
    rol zh
    .endmacro
.macro  DIV2Z           ; divide Z by 2
    lsr zh
    ror zl
    .endmacro

; --- add register to pointer XYZ ---
.macro  ADDX    ;reg        ; x <- y+reg
    add xl,@0
    brcc    PC+2
    subi    xh,-1       ; add carry
    .endmacro
.macro  ADDY    ;reg        ; y <- y+reg
    add yl,@0
    brcc    PC+2
    subi    yh,-1       ; add carry
    .endmacro
.macro  ADDZ    ;reg        ; z <- z+reg
    add zl,@0
    brcc    PC+2
    subi    zh,-1       ; add carry
    .endmacro

; ====================
;   miscellaneous
; ====================

; --- output/store (regular I/O space) immediate value ---
.macro  OUTI    ; port,k    output immediate value to port
    ldi w,@1
    out @0,w
    .endmacro

; --- output/store (extended I/O space) immediate value ---
.macro OUTEI    ; port,k    output immediate value to port
    ldi w,@1
    sts @0,w
    .endmacro

; --- add immediate value ---
.macro  ADDI
    subi    @0,-@1
    .endmacro
.macro  ADCI
    sbci    @0,-@1
    .endmacro

; --- inc/dec with range limitation ---
.macro  INC_LIM ; reg,limit
    cpi @0,@1
    brlo    PC+3
    ldi @0,@1
    rjmp    PC+2
    inc @0
    .endmacro

.macro  DEC_LIM ; reg,limit
    cpi @0,@1
    breq    PC+5
    brlo    PC+3
    dec @0
    rjmp    PC+2
    ldi @0,@1
    .endmacro

; --- inc/dec with cyclic range ---
.macro  INC_CYC ; reg,low,high
    cpi @0,@2
    brsh    _low    ; reg>=high then reg=low
    cpi @0,@1
    brlo    _low    ; reg< low  then reg=low
    inc @0
    rjmp    _done
```

```
_low:   ldi @0,@1
_done:
    .endmacro

.macro  DEC_CYC ; reg,low,high
    cpi @0,@1
    breq    _high   ; reg=low then reg=high
    brlo    _high   ; reg<low then reg=high
    dec @0
    cpi @0,@2
    brsh    _high   ; reg>=high then high
    rjmp    _done
_high:  ldi @0,@2
_done:
    .endmacro

.macro  INCDEC  ;port,b1,b2,reg,low,high
    sbic    @0,@1
    rjmp    PC+6

    cpi @3,@5
    brlo    PC+3
    ldi @3,@4
    rjmp    PC+2
    inc @3

    sbic    @0,@2
    rjmp    PC+7

    cpi @3,@4
    breq    PC+5
    brlo    PC+3
    dec @3
    rjmp    PC+2
    ldi @3,@5
    .endmacro

; --- wait loops ---
; wait 10...196608 cycles
.macro  WAIT_C  ; k
    ldi w,  low((@0-7)/3)
    mov u,w         ; u=LSB
    ldi w,high((@0-7)/3)+1  ; w=MSB
    dec u
    brne    PC-1
    dec u
    dec w
    brne    PC-4
    .endmacro

; wait micro-seconds (us)
; us = x*3*1000'000/clock)  ==> x=us*clock/3000'000
.macro  WAIT_US ; k
    ldi w, low((clock/1000*@0/3000)-1)
    mov u,w
    ldi w,high((clock/1000*@0/3000)-1)+1 ; set up: 3 cyles
    dec u
    brne    PC-1        ; inner loop: 3 cycles
    dec u       ; adjustment for outer loop
    dec w
    brne    PC-4
    .endmacro

; wait mili-seconds (ms)
.macro  WAIT_MS ; k
    ldi w, low(@0)
    mov u,w     ; u = LSB
    ldi w,high(@0)+1    ; w = MSB
wait_ms:
    push    w       ; wait 1000 usec
    push    u
    ldi w, low((clock/3000)-5)
    mov u,w
    ldi w,high((clock/3000)-5)+1
    dec u
    brne    PC-1        ; inner loop: 3 cycles
    dec u       ; adjustment for outer loop
    dec w
    brne    PC-4
    pop u
    pop w

    dec u
    brne    wait_ms
    dec w
```

```
    brne    wait_ms
    .endmacro

; --- conditional jumps/calls ---
.macro  JC0         ; jump if carry=0
    brcs    PC+2
    rjmp    @0
    .endmacro
.macro  JC1         ; jump if carry=1
    brcc    PC+2
    rjmp    @0
    .endmacro

.macro  JK  ; reg,k,addr    ; jump if reg=k
    cpi @0,@1
    breq    @2
    .endmacro
.macro  _JK ; reg,k,addr    ; jump if reg=k
    cpi @0,@1
    brne    PC+2
    rjmp    @2
    .endmacro
.macro  JNK ; reg,k,addr    ; jump if not(reg=k)
    cpi @0,@1
    brne    @2
    .endmacro

.macro  CK  ; reg,k,addr    ; call if reg=k
    cpi @0,@1
    brne    PC+2
    rcall   @2
    .endmacro
.macro  CNK ; reg,k,addr    ; call if not(reg=k)
    cpi @0,@1
    breq    PC+2
    rcall   @2
    .endmacro

.macro  JSK ; sram,k,addr   ; jump if sram=k
    lds w,@0
    cpi w,@1
    breq    @2
    .endmacro
.macro  JSNK    ; sram,k,addr   ; jump if not(sram=k)
    lds w,@0
    cpi w,@1
    brne    @2
    .endmacro

; --- loops ---
.macro  DJNZ    ; reg,addr  ; decr and jump if not zero
    dec @0
    brne    @1
    .endmacro
.macro  DJNK    ; reg,k,addr    ; decr and jump if not k
    dec @0
    cpi @0,@1
    brne    @2
    .endmacro

.macro  IJNZ    ; reg,addr  ; inc and jump if not zero
    inc @0
    brne    @1
    .endmacro
.macro  IJNK    ; reg,k,addr    ; inc and jump if not k
    inc @0
    cpi @0,@1
    brne    @2
    .endmacro
.macro  _IJNK   ; reg,k,addr    ; inc and jump if not k
    inc @0
    ldi w,@1
    cp  @0,w
    brne    @2
    .endmacro

.macro  ISJNK   ; sram,k,addr   ; inc sram and jump if not k
    lds w,@0
    inc w
    sts @0,w
    cpi w,@1
    brne    @2
    .endmacro
.macro  _ISJNK  ; sram,k,addr   ; inc sram and jump if not k
    lds w,@0
```

```
    inc w
    sts @0,w
    cpi w,@1
    breq    PC+2
    rjmp    @2
    .endmacro

.macro  DSJNK   ; sram,k,addr   ; dec sram and jump if not k
    lds w,@0
    dec w
    sts @0,w
    cpi w,@1
    brne    @2
    .endmacro

; --- table lookup ---
.macro  LOOKUP  ;reg, index,tbl
    push    ZL
    push    ZH
    mov zl,@1       ; move index into z
    clr zh
    subi    zl, low(-2*@2)  ; add base address of table
    sbci    zh,high(-2*@2)
    lpm         ; load program memory (into r0)
    mov @0,r0
    pop ZH
    pop ZL
    .endmacro

.macro  LOOKUP2 ;r1,r0, index,tbl
    mov zl,@2       ; move index into z
    clr zh
    lsl zl      ; multiply by 2
    rol zh
    subi    zl, low(-2*@3)  ; add base address of table
    sbci    zh,high(-2*@3)
    lpm         ; get LSB byte
    mov w,r0        ; temporary store LSB in w
    adiw    zl,1        ; increment Z
    lpm         ; get MSB byte
    mov @0,r0       ; mov MSB to res1
    mov @1,w        ; mov LSB to res0
    .endmacro

.macro  LOOKUP4 ;r3,r2,r1,r0, index,tbl
    mov zl,@4       ; move index into z
    clr zh
    lsl zl      ; multiply by 2
    rol zh
    lsl zl      ; multiply by 2
    rol zh
    subi    zl, low(-2*@5)  ; add base address of table
    sbci    zh,high(-2*@5)
    lpm
    mov @1,r0       ; load high word LSB
    adiw    zl,1
    lpm
    mov @0,r0       ; load high word MSB
    adiw    zl,1
    lpm
    mov @3,r0       ; load low word LSB
    adiw    zl,1
    lpm
    mov @2,r0       ; load low word MSB
    .endmacro

.macro  LOOKDOWN ;reg,index,tbl
    ldi ZL, low(2*@2)   ; load table address
    ldi ZH,high(2*@2)
    clr @1
loop:   lpm
    cp  r0,@0
    breq    found
    inc @1
    adiw    ZL,1
    tst r0
    breq    notfound
    rjmp    loop
notfound:
    ldi @1,-1
found:
    .endmacro

; --- branch table ---
.macro  C_TBL   ; reg,tbl
```

```
    ldi ZL, low(2*@1)
    ldi ZH,high(2*@1)
    lsl @0
    add ZL,@0
    brcc    PC+2
    inc ZH
    lpm
    push    r0
    lpm
    mov zh,r0
    pop zl
    icall
    .endmacro
.macro  J_TBL   ; reg,tbl
    ldi ZL, low(2*@1)
    ldi ZH,high(2*@1)
    lsl @0
    add ZL,@0
    brcc    PC+2
    inc ZH
    lpm
    push    r0
    lpm
    mov zh,r0
    pop zl
    ijmp
    .endmacro

.macro  BRANCH  ; reg       ; branching using the stack
    ldi w, low(tbl)
    add w,@0
    push    w
    ldi w,high(tbl)
    brcc    PC+2
    inc w
    push    w
    ret
tbl:
    .endmacro

; --- multiply/division ---
.macro  DIV2    ; reg
    lsr @0
    .endmacro
.macro  DIV4    ; reg
    lsr @0
    lsr @0
    .endmacro
.macro  DIV8    ; reg
    lsr @0
    lsr @0
    lsr @0
    .endmacro

.macro  MUL2    ; reg
    lsl @0
    .endmacro
.macro  MUL4    ; reg
    lsl @0
    lsl @0
    .endmacro
.macro  MUL8    ; reg
    lsl @0
    lsl @0
    lsl @0
    .endmacro

; ====================================
;   extending existing instructios
; ====================================

; --- immediate ops with r0..r15 ---
.macro  _ADDI
    ldi w,@1
    add @0,w
    .endmacro
.macro  _ADCI
    ldi w,@1
    adc @0,w
    .endmacro
.macro  _SUBI
    ldi w,@1
    sub @0,w
    .endmacro
.macro  _SBCI
```

```
    ldi w,@1
    sbc @0,w
    .endmacro
.macro  _ANDI
    ldi w,@1
    and @0,w
    .endmacro
.macro  _ORI
    ldi w,@1
    or  @0,w
    .endmacro
.macro  _EORI
    ldi w,@1
    eor @0,w
    .endmacro
.macro  _SBR
    ldi w,@1
    or  @0,w
    .endmacro
.macro  _CBR
    ldi w,~@1
    and @0,w
    .endmacro
.macro  _CPI
    ldi w,@1
    cp  @0,w
    .endmacro
.macro  _LDI
    ldi w,@1
    mov @0,w
    .endmacro

; --- bit access for port p32..p63 ---
.macro  _SBI
    in  w,@0
    ori w,1<<@1
    out @0,w
    .endmacro
.macro  _CBI
    in  w,@0
    andi    w,~(1<<@1)
    out @0,w
    .endmacro

; --- extending branch distance to +/-2k ---
.macro  _BREQ
    brne    PC+2
    rjmp    @0
    .endmacro
.macro  _BRNE
    breq    PC+2
    rjmp    @0
    .endmacro
.macro  _BRCS
    brcc    PC+2
    rjmp    @0
    .endmacro
.macro  _BRCC
    brcs    PC+2
    rjmp    @0
    .endmacro
.macro  _BRSH
    brlo    PC+2
    rjmp    @0
    .endmacro
.macro  _BRLO
    brsh    PC+2
    rjmp    @0
    .endmacro
.macro  _BRMI
    brpl    PC+2
    rjmp    @0
    .endmacro
.macro  _BRPL
    brmi    PC+2
    rjmp    @0
    .endmacro
.macro  _BRGE
    brlt    PC+2
    rjmp    @0
    .endmacro
.macro  _BRLT
    brge    PC+2
    rjmp    @0
    .endmacro
```

```
.macro  _BRHS
    brhc    PC+2
    rjmp    @0
    .endmacro
.macro  _BRHC
    brhs    PC+2
    rjmp    @0
    .endmacro
.macro  _BRTS
    brtc    PC+2
    rjmp    @0
    .endmacro
.macro  _BRTC
    brts    PC+2
    rjmp    @0
    .endmacro
.macro  _BRVS
    brvc    PC+2
    rjmp    @0
    .endmacro
.macro  _BRVC
    brvs    PC+2
    rjmp    @0
    .endmacro
.macro  _BRIE
    brid    PC+2
    rjmp    @0
    .endmacro
.macro  _BRID
    brie    PC+2
    rjmp    @0
    .endmacro

; ====================
;   bit operations
; ====================

; --- moving bits ---
.macro  MOVB    ; reg1,b1, reg2,b2  ; reg1,bit1 <- reg2,bit2
    bst @2,@3
    bld @0,@1
    .endmacro
.macro  OUTB    ; port1,b1, reg2,b2 ; port1,bit1 <- reg2,bit2
    sbrs    @2,@3
    cbi @0,@1
    sbrc    @2,@3
    sbi @0,@1
    .endmacro
.macro  INB ; reg1,b1, port2,b2 ; reg1,bit1 <- port2,bit2
    sbis    @2,@3
    cbr @0,1<<@1
    sbic    @2,@3
    sbr @0,1<<@1
    .endmacro

.macro  Z2C             ; zero to carry
    sec
    breq    PC+2    ; (Z=1)
    clc
    .endmacro
.macro  Z2INVC              ; zero to inverse carry
    sec
    brne    PC+2    ; (Z=0)
    clc
    .endmacro

.macro  C2Z             ; carry to zero
    sez
    brcs    PC+2    ; (C=1)
    clz
    .endmacro

.macro  B2C ; reg,b         ; bit to carry
    sbrc    @0,@1
    sec
    sbrs    @0,@1
    clc
    .endmacro
.macro  C2B ; reg,b         ; carry to bit
    brcc    PC+2
    sbr @0,(1<<@1)
    brcs    PC+2
    cbr @0,(1<<@1)
    .endmacro
.macro  P2C ; port,b        ; port to carry
```

```
    sbic    @0,@1
    sec
    sbis    @0,@1
    clc
    .endmacro
.macro  C2P ; port,b        ; carry to port
    brcc    PC+2
    sbi @0,@1
    brcs    PC+2
    cbi @0,@1
    .endmacro

; --- inverting bits ---
.macro  INVB    ; reg,bit       ; inverse reg,bit
    ldi w,(1<<@1)
    eor @0,w
    .endmacro
.macro  INVP    ; port,bit      ; inverse port,bit
    sbis    @0,@1
    rjmp    PC+3
    cbi @0,@1
    rjmp    PC+2
    sbi @0,@1
    .endmacro
.macro  INVC                ; inverse carry
    brcs    PC+3
    sec
    rjmp    PC+2
    clc
    .endmacro

; --- setting a single bit ---
.macro  SETBIT  ; reg(0..7)
; in    reg (0..7)
; out   reg with bit (0..7) set to 1.
; 0=00000001
; 1=00000010
; ...
; 7=10000000
    mov w,@0
    clr @0
    inc @0
    andi    w,0b111
    breq    PC+4
    lsl @0
    dec w
    brne    PC-2
    .endmacro

; --- logical operations with masks ---
.macro  MOVMSK  ; reg1,reg2,mask    ; reg1 <- reg2 (mask)
    ldi w,~@2
    and @0,w
    ldi w,@2
    and @1,w
    or  @0,@1
    .endmacro
.macro  ANDMSK  ; reg1,reg2,mask    ; reg1 <- ret 1 AND reg2 (mask)
    mov w,@1
    ori w,~@2
    and @0,w
    .endmacro
.macro  ORMSK   ; reg1,reg2,mask    ; reg1 <- ret 1 AND reg2 (mask)
    mov w,@1
    andi    w,@2
    or  @0,w
    .endmacro

; --- logical operations on bits ---
.macro  ANDB    ; r1,b1, r2,b2, r3,b3   ; reg1,b1 <- reg2,b2 AND reg3,b3
    set
    sbrs    @4,@5
    clt
    sbrs    @2,@3
    clt
    bld @0,@1
    .endmacro
.macro  ORB ; r1,b1, r2,b2, r3,b3   ; reg1.b1 <- reg2.b2 OR reg3.b3
    clt
    sbrc    @4,@5
    set
    sbrc    @2,@3
    set
    bld @0,@1
    .endmacro
```

```
.macro  EORB    ; r1,b1, r2,b2, r3,b3   ; reg1.b1 <- reg2.b2 XOR reg3.b3
    sbrc    @4,@5
    rjmp    f1
f0: bst @2,@3
    rjmp    PC+4
f1: set
    sbrc    @0,@1
    clt
    bld @0,@0
    .endmacro

; --- operations based on register bits ---
.macro  FB0 ; reg,bit       ; bit=0
    cbr @0,1<<@1
    .endmacro
.macro  FB1 ; reg,bit       ; bit=1
    sbr @0,1<<@1
    .endmacro
.macro  _FB0    ; reg,bit       ; bit=0
    ldi w,~(1<<@1)
    and @0,w
    .endmacro
.macro  _FB1    ; reg,bit       ; bit=1
    ldi w,1<<@1
    or  @0,w
    .endmacro
.macro  SB0 ; reg,bit,addr      ; skip if bit=0
    sbrc    @0,@1
    .endmacro
.macro  SB1 ; reg,bit,addr      ; skip if bit=1
    sbrs    @0,@1
    .endmacro
.macro  JB0 ; reg,bit,addr      ; jump if bit=0
    sbrs    @0,@1
    rjmp    @2
    .endmacro
.macro  JB1 ; reg,bit,addr      ; jump if bit=1
    sbrc    @0,@1
    rjmp    @2
    .endmacro
.macro  CB0 ; reg,bit,addr      ; call if bit=0
    sbrs    @0,@1
    rcall   @2
    .endmacro
.macro  CB1 ; reg,bit,addr      ; call if bit=1
    sbrc    @0,@1
    rcall   @2
    .endmacro
.macro  WB0 ; reg,bit       ; wait if bit=0
    sbrs    @0,@1
    rjmp    PC-1
    .endmacro
.macro  WB1 ; reg,bit       ; wait if bit=1
    sbrc    @0,@1
    rjmp    PC-1
    .endmacro
.macro  RB0 ; reg,bit       ; return if bit=0
    sbrs    @0,@1
    ret
    .endmacro
.macro  RB1 ; reg,bit       ; return if bit=1
    sbrc    @0,@1
    ret
    .endmacro

; wait if bit=0 with timeout
; if timeout (in units of 5 cyc) then jump to addr
.macro  WB0T    ; reg,bit,timeout,addr
    ldi w,@2+1
    dec w   ; 1 cyc
    breq    @3  ; 1 cyc
    sbrs    @0,@1   ; 1 cyc
    rjmp    PC-3    ; 2 cyc = 5 cycles
    .endmacro

; wait if bit=1 with timeout
; if timeout (in units of 5 cyc) then jump to addr
.macro  WB1T    ; reg,bit,timeout,addr
    ldi w,@2+1
    dec w   ; 1 cyc
    breq    @3  ; 1 cyc
    sbrc    @0,@1   ; 1 cyc
    rjmp    PC-3    ; 2 cyc = 5 cycles
    .endmacro
```

```
; --- operations based on port bits ---
.macro  P0  ; port,bit      ; port=0
    cbi @0,@1
    .endmacro
.macro  P1  ; port,bit      ; port=1
    sbi @0,@1
    .endmacro
.macro  SP0 ; port,bit      ; skip if port=0
    sbic    @0,@1
    .endmacro
.macro  SP1 ; port,bit      ; skip if port=1
    sbis    @0,@1
    .endmacro
.macro  JP0 ; port,bit,addr     ; jump if port=0
    sbis    @0,@1
    rjmp    @2
    .endmacro
.macro  JP1 ; port,bit,addr     ; jump if port=1
    sbic    @0,@1
    rjmp    @2
    .endmacro
.macro  CP0 ; port,bit,addr     ; call if port=0
    sbis    @0,@1
    rcall   @2
    .endmacro
.macro  CP1 ; port,bit,addr     ; call if port=1
    sbic    @0,@1
    rcall   @2
    .endmacro
.macro  WP0 ; port,bit      ; wait if port=0
    sbis    @0,@1
    rjmp    PC-1
    .endmacro
.macro  WP1 ; port,bit      ; wait if port=1
    sbic    @0,@1
    rjmp    PC-1
    .endmacro
.macro  RP0 ; port,bit      ; return if port=0
    sbis    @0,@1
    ret
    .endmacro
.macro  RP1 ; port,bit      ; return if port=1
    sbic    @0,@1
    ret
    .endmacro

; wait if port=0 with timeout
; if timeout (in units of 5 cyc) then jump to addr
.macro  WP0T    ; port,bit,timeout,addr
    ldi w,@2+1
    dec w   ; 1 cyc
    breq    @3  ; 1 cyc
    sbis    @0,@1   ; 1 cyc
    rjmp    PC-3    ; 2 cyc = 5 cycles
    .endmacro

; wait if port=1 with timeout
; if timeout (in units of 5 cyc) then jump to addr
.macro  WP1T    ; port,bit,timeout,addr
    ldi w,@2+1
    dec w   ; 1 cyc
    breq    @3  ; 1 cyc
    sbic    @0,@1   ; 1 cyc
    rjmp    PC-3    ; 2 cyc = 5 cycles
    .endmacro

; ============================
;   multi-byte operations
; ============================

.macro  SWAP4           ; swap 2 variables
    mov w ,@0
    mov @0,@4
    mov @4,w
    mov w ,@1
    mov @1,@5
    mov @5,w
    mov w ,@2
    mov @2,@6
    mov @6,w
    mov w ,@3
    mov @3,@7
    mov @7,w
    .endmacro
.macro  SWAP3
```

```
    mov w ,@0
    mov @0,@3
    mov @3,w
    mov w ,@1
    mov @1,@4
    mov @4,w
    mov w ,@2
    mov @2,@5
    mov @5,w
    .endmacro
.macro  SWAP2
    mov w ,@0
    mov @0,@2
    mov @2,w
    mov w ,@1
    mov @1,@3
    mov @3,w
    .endmacro
.macro  SWAP1
    mov w ,@0
    mov @0,@1
    mov @1,w
    .endmacro

.macro  LDX4    ;r..r0      ; load from (x+)
    ld  @3,x+
    ld  @2,x+
    ld  @1,x+
    ld  @0,x+
    .endmacro
.macro  LDX3    ;r..r0
    ld  @2,x+
    ld  @1,x+
    ld  @0,x+
    .endmacro
.macro  LDX2    ;r..r0
    ld  @1,x+
    ld  @0,x+
    .endmacro

.macro  LDY4    ;r..r0      ; load from (y+)
    ld  @3,y+
    ld  @2,y+
    ld  @1,y+
    ld  @0,y+
    .endmacro
.macro  LDY3    ;r..r0
    ld  @2,y+
    ld  @1,y+
    ld  @0,y+
    .endmacro
.macro  LDY2    ;r..r0
    ld  @1,y+
    ld  @0,y+
    .endmacro

.macro  LDZ4    ;r..r0      ; load from (z+)
    ld  @3,z+
    ld  @2,z+
    ld  @1,z+
    ld  @0,z+
    .endmacro
.macro  LDZ3    ;r..r0
    ld  @2,z+
    ld  @1,z+
    ld  @0,z+
    .endmacro
.macro  LDZ2    ;r..r0
    ld  @1,z+
    ld  @0,z+
    .endmacro

.macro  STX4    ;r..r0      ; store to (x+)
    st  x+,@3
    st  x+,@2
    st  x+,@1
    st  x+,@0
    .endmacro
.macro  STX3    ;r..r0
    st  x+,@2
    st  x+,@1
    st  x+,@0
    .endmacro
.macro  STX2    ;r..r0
    st  x+,@1
```

```
    st  x+,@0
    .endmacro

.macro  STY4    ;r..r0      ; store to (y+)
    st  y+,@3
    st  y+,@2
    st  y+,@1
    st  y+,@0
    .endmacro
.macro  STY3    ;r..r0
    st  y+,@2
    st  y+,@1
    st  y+,@0
    .endmacro
.macro  STY2    ;r..r0
    st  y+,@1
    st  y+,@0
    .endmacro

.macro  STZ4    ;r..r0      ; store to (z+)
    st  z+,@3
    st  z+,@2
    st  z+,@1
    st  z+,@0
    .endmacro
.macro  STZ3    ;r..r0
    st  z+,@2
    st  z+,@1
    st  z+,@0
    .endmacro
.macro  STZ2    ;r..r0
    st  z+,@1
    st  z+,@0
    .endmacro

.macro  STI4    ;addr,k     ; store immediate
    ldi w,  low(@1)
    sts @0+0,w
    ldi w, high(@1)
    sts @0+1,w
    ldi w,byte3(@1)
    sts @0+2,w
    ldi w,byte4(@1)
    sts @0+3,w
    .endmacro
.macro  STI3    ;addr,k
    ldi w,  low(@1)
    sts @0+0,w
    ldi w, high(@1)
    sts @0+1,w
    ldi w,byte3(@1)
    sts @0+2,w
    .endmacro
.macro  STI2    ;addr,k
    ldi w,  low(@1)
    sts @0+0,w
    ldi w, high(@1)
    sts @0+1,w
    .endmacro
.macro  STI ;addr,k
    ldi w,@1
    sts @0,w
    .endmacro

.macro  INC4            ; increment
    ldi w,0xff
    sub @3,w
    sbc @2,w
    sbc @1,w
    sbc @0,w
    .endmacro
.macro  INC3
    ldi w,0xff
    sub @2,w
    sbc @1,w
    sbc @0,w
    .endmacro
.macro  INC2
    ldi w,0xff
    sub @1,w
    sbc @0,w
    .endmacro

.macro  DEC4            ; decrement
    ldi w,0xff
```

```
    add @3,w
    adc @2,w
    adc @1,w
    adc @0,w
    .endmacro
.macro  DEC3
    ldi w,0xff
    add @2,w
    adc @1,w
    adc @0,w
    .endmacro
.macro  DEC2
    ldi w,0xff
    add @1,w
    adc @0,w
    .endmacro

.macro  CLR9            ; clear (also clears the carry)
    sub @0,@0
    clr @1
    clr @2
    clr @3
    clr @4
    clr @5
    clr @6
    clr @7
    clr @8
    .endmacro
.macro  CLR8
    sub @0,@0
    clr @1
    clr @2
    clr @3
    clr @4
    clr @5
    clr @6
    clr @7
    .endmacro
.macro  CLR7
    sub @0,@0
    clr @1
    clr @2
    clr @3
    clr @4
    clr @5
    clr @6
    .endmacro
.macro  CLR6
    sub @0,@0
    clr @1
    clr @2
    clr @3
    clr @4
    clr @5
    .endmacro
.macro  CLR5
    sub @0,@0
    clr @1
    clr @2
    clr @3
    clr @4
    .endmacro
.macro  CLR4
    sub @0,@0
    clr @1
    clr @2
    clr @3
    .endmacro
.macro  CLR3
    sub @0,@0
    clr @1
    clr @2
    .endmacro
.macro  CLR2
    sub @0,@0
    clr @1
    .endmacro

.macro  COM4            ; one's complement
    com @0
    com @1
    com @2
    com @3
    .endmacro
.macro  COM3
```

```
    com @0
    com @1
    com @2
    .endmacro
.macro  COM2
    com @0
    com @1
    .endmacro

.macro  NEG4            ; negation (two's complement)
    com @0
    com @1
    com @2
    com @3
    ldi w,0xff
    sub @3,w
    sbc @2,w
    sbc @1,w
    sbc @0,w
    .endmacro
.macro  NEG3
    com @0
    com @1
    com @2
    ldi w,0xff
    sub @2,w
    sbc @1,w
    sbc @0,w
    .endmacro
.macro  NEG2
    com @0
    com @1
    ldi w,0xff
    sub @1,w
    sbc @0,w
    .endmacro

.macro  LDI4    ; r..r0, k  ; load immediate
    ldi @3,  low(@4)
    ldi @2, high(@4)
    ldi @1,byte3(@4)
    ldi @0,byte4(@4)
    .endmacro
.macro  LDI3
    ldi @2,  low(@3)
    ldi @1, high(@3)
    ldi @0,byte3(@3)
    .endmacro
.macro  LDI2
    ldi @1,  low(@2)
    ldi @0, high(@2)
    .endmacro

.macro  LDS4            ; load direct from SRAM
    lds @3,@4
    lds @2,@4+1
    lds @1,@4+2
    lds @0,@4+3
    .endmacro
.macro  LDS3
    lds @2,@3
    lds @1,@3+1
    lds @0,@3+2
    .endmacro
.macro  LDS2
    lds @1,@2
    lds @0,@2+1
    .endmacro

.macro  STS4            ; store direct to SRAM
    sts @0+0,@4
    sts @0+1,@3
    sts @0+2,@2
    sts @0+3,@1
    .endmacro
.macro  STS3
    sts @0+0,@3
    sts @0+1,@2
    sts @0+2,@1
    .endmacro
.macro  STS2
    sts @0+0,@2
    sts @0+1,@1
    .endmacro
```

```
.macro  STDZ4   ; d, r3,r2,r1,r0
    std z+@0+0,@4
    std z+@0+1,@3
    std z+@0+2,@2
    std z+@0+3,@1
    .endmacro
.macro  STDZ3   ; d, r2,r1,r0
    std z+@0+0,@3
    std z+@0+1,@2
    std z+@0+2,@1
    .endmacro
.macro  STDZ2   ; d, r1,r0
    std z+@0+0,@2
    std z+@0+1,@1
    .endmacro

.macro  LPM4            ; load program memory
    lpm
    mov @3,r0
    adiw    zl,1
    lpm
    mov @2,r0
    adiw    zl,1
    lpm
    mov @1,r0
    adiw    zl,1
    lpm
    mov @0,r0
    adiw    zl,1
    .endmacro
.macro  LPM3
    lpm
    mov @2,r0
    adiw    zl,1
    lpm
    mov @1,r0
    adiw    zl,1
    lpm
    mov @0,r0
    adiw    zl,1
    .endmacro
.macro  LPM2
    lpm
    mov @1,r0
    adiw    zl,1
    lpm
    mov @0,r0
    adiw    zl,1
    .endmacro
.macro  LPM1
    lpm
    mov @0,r0
    adiw    zl,1
    .endmacro

.macro  MOV4            ; move between registers
    mov @3,@7
    mov @2,@6
    mov @1,@5
    mov @0,@4
    .endmacro
.macro  MOV3
    mov @2,@5
    mov @1,@4
    mov @0,@3
    .endmacro
.macro  MOV2
    mov @1,@3
    mov @0,@2
    .endmacro

.macro  ADD4            ; add
    add @3,@7
    adc @2,@6
    adc @1,@5
    adc @0,@4
    .endmacro
.macro  ADD3
    add @2,@5
    adc @1,@4
    adc @0,@3
    .endmacro
.macro  ADD2
    add @1,@3
    adc @0,@2
```

```
    .endmacro

.macro  SUB4            ; subtract
    sub @3,@7
    sbc @2,@6
    sbc @1,@5
    sbc @0,@4
    .endmacro
.macro  SUB3
    sub @2,@5
    sbc @1,@4
    sbc @0,@3
    .endmacro
.macro  SUB2
    sub @1,@3
    sbc @0,@2
    .endmacro

.macro  CP4         ; compare
    cp  @3,@7
    cpc @2,@6
    cpc @1,@5
    cpc @0,@4
    .endmacro
.macro  CP3
    cp  @2,@5
    cpc @1,@4
    cpc @0,@3
    .endmacro
.macro  CP2
    cp  @1,@3
    cpc @0,@2
    .endmacro

.macro  TST4            ; test
    clr w
    cp  @3,w
    cpc @2,w
    cpc @1,w
    cpc @0,w
    .endmacro
.macro  TST3
    clr w
    cp  @2,w
    cpc @1,w
    cpc @0,w
    .endmacro
.macro  TST2
    clr w
    cp  @1,w
    cpc @0,w
    .endmacro

.macro  ADDI4           ; add immediate
    subi    @3,  low(-@4)
    sbci    @2, high(-@4)
    sbci    @1,byte3(-@4)
    sbci    @0,byte4(-@4)
    .endmacro
.macro  ADDI3
    subi    @2,  low(-@3)
    sbci    @1, high(-@3)
    sbci    @0,byte3(-@3)
    .endmacro
.macro  ADDI2
    subi    @1,  low(-@2)
    sbci    @0, high(-@2)
    .endmacro

.macro  SUBI4           ; subtract immediate
    subi    @3,  low(@4)
    sbci    @2, high(@4)
    sbci    @1,byte3(@4)
    sbci    @0,byte4(@4)
    .endmacro
.macro  SUBI3
    subi    @2,  low(@3)
    sbci    @1, high(@3)
    sbci    @0,byte3(@3)
    .endmacro
.macro  SUBI2
    subi    @1,  low(@2)
    sbci    @0, high(@2)
    .endmacro
```

```
.macro  LSL5            ; logical shift left
    lsl @4
    rol @3
    rol @2
    rol @1
    rol @0
    .endmacro
.macro  LSL4
    lsl @3
    rol @2
    rol @1
    rol @0
    .endmacro
.macro  LSL3
    lsl @2
    rol @1
    rol @0
    .endmacro
.macro  LSL2
    lsl @1
    rol @0
    .endmacro

.macro  LSR4            ; logical shift right
    lsr @0
    ror @1
    ror @2
    ror @3
    .endmacro
.macro  LSR3
    lsr @0
    ror @1
    ror @2
    .endmacro
.macro  LSR2
    lsr @0
    ror @1
    .endmacro

.macro  ASR4            ; arithmetic shift right
    asr @0
    ror @1
    ror @2
    ror @3
    .endmacro
.macro  ASR3
    asr @0
    ror @1
    ror @2
    .endmacro
.macro  ASR2
    asr @0
    ror @1
    .endmacro

.macro  ROL8            ; rotate left through carry
    rol @7
    rol @6
    rol @5
    rol @4
    rol @3
    rol @2
    rol @1
    rol @0
    .endmacro
.macro  ROL7
    rol @6
    rol @5
    rol @4
    rol @3
    rol @2
    rol @1
    rol @0
    .endmacro
.macro  ROL6
    rol @5
    rol @4
    rol @3
    rol @2
    rol @1
    rol @0
    .endmacro
.macro  ROL5
    rol @4
    rol @3
```

```
    rol @2
    rol @1
    rol @0
    .endmacro
.macro  ROL4
    rol @3
    rol @2
    rol @1
    rol @0
    .endmacro
.macro  ROL3
    rol @2
    rol @1
    rol @0
    .endmacro
.macro  ROL2
    rol @1
    rol @0
    .endmacro

.macro  ROR8            ; rotate right through carry
    ror @0
    ror @1
    ror @2
    ror @3
    ror @4
    ror @5
    ror @6
    ror @7
    .endmacro
.macro  ROR7
    ror @0
    ror @1
    ror @2
    ror @3
    ror @4
    ror @5
    ror @6
    .endmacro
.macro  ROR6
    ror @0
    ror @1
    ror @2
    ror @3
    ror @4
    ror @5
    .endmacro
.macro  ROR5
    ror @0
    ror @1
    ror @2
    ror @3
    ror @4
    .endmacro
.macro  ROR4
    ror @0
    ror @1
    ror @2
    ror @3
    .endmacro
.macro  ROR3
    ror @0
    ror @1
    ror @2
    .endmacro
.macro  ROR2
    ror @0
    ror @1
    .endmacro

.macro  PUSH2
    push    @0
    push    @1
    .endmacro
.macro  POP2
    pop @1
    pop @0
    .endmacro

.macro  PUSH3
    push    @0
    push    @1
    push    @2
    .endmacro
.macro  POP3
```

```
    pop @2
    pop @1
    pop @0
    .endmacro

.macro  PUSH4
    push    @0
    push    @1
    push    @2
    push    @3
    .endmacro
.macro  POP4
    pop @3
    pop @2
    pop @1
    pop @0
    .endmacro

.macro  PUSH5
    push    @0
    push    @1
    push    @2
    push    @3
    push    @4
    .endmacro
.macro  POP5
    pop @4
    pop @3
    pop @2
    pop @1
    pop @0
    .endmacro

; --- SRAM operations ---
.macro  INCS4   ; sram      ; increment SRAM 4-byte variable
    lds w,@0
    inc w
    sts @0,w
    brne    end
    lds w,@0+1
    inc w
    sts @0+1,w
    brne    end
    lds w,@0+2
    inc w
    sts @0+2,w
    brne    end
    lds w,@0+3
    inc w
    sts @0+3,w
end:
    .endmacro

.macro  INCS3   ; sram      ; increment SRAM 3-byte variable
    lds w,@0
    inc w
    sts @0,w
    brne    end
    lds w,@0+1
    inc w
    sts @0+1,w
    brne    end
    lds w,@0+2
    inc w
    sts @0+2,w
end:
    .endmacro

.macro  INCS2   ; sram      ; increment SRAM 2-byte variable
    lds w,@0
    inc w
    sts @0,w
    brne    end
    lds w,@0+1
    inc w
    sts @0+1,w
end:
    .endmacro

.macro  INCS    ; sram      ; increment SRAM 1-byte variable
    lds w,@0
    inc w
    sts @0,w
    .endmacro
```

```
.macro  DECS4   ; sram      ; decrement SRAM 4-byte variable
    ldi w,1
    lds u,@0
    sub u,w
    sts @0,u
    clr w
    lds u,@0+1
    sbc u,w
    sts @0+1,u
    lds u,@0+2
    sbc u,w
    sts @0+2,u
    lds u,@0+3
    sbc u,w
    sts @0+3,u
    .endmacro
.macro  DECS3   ; sram      ; decrement SRAM 3-byte variable
    ldi w,1
    lds u,@0
    sub u,w
    sts @0,u
    clr w
    lds u,@0+1
    sbc u,w
    sts @0+1,u
    lds u,@0+2
    sbc u,w
    sts @0+2,u
    .endmacro
.macro  DECS2   ; sram      ; decrement SRAM 2-byte variable
    ldi w,1
    lds u,@0
    sub u,w
    sts @0,u
    clr w
    lds u,@0+1
    sbc u,w
    sts @0+1,u
    .endmacro
.macro  DECS    ; sram      ; decrement
    lds w,@0
    dec w
    sts @0,w
    .endmacro

.macro  MOVS4   ; addr0,addr1   ; [addr0] <-- [addr1]
    lds w,@1
    sts @0,w
    lds w,@1+1
    sts @0+1,w
    lds w,@1+2
    sts @0+2,w
    lds w,@3+1
    sts @0+3,w
    .endmacro
.macro  MOVS3   ; addr0,addr1   ; [addr0] <-- [addr1]
    lds w,@1
    sts @0,w
    lds w,@1+1
    sts @0+1,w
    lds w,@1+2
    sts @0+2,w
    .endmacro
.macro  MOVS2   ; addr0,addr1   ; [addr0] <-- [addr1]
    lds w,@1
    sts @0,w
    lds w,@1+1
    sts @0+1,w
    .endmacro
.macro  MOVS    ; addr0,addr1   ; [addr0] <-- [addr1]
    lds w,@1
    sts @0,w
    .endmacro

.macro  SEXT    ; reg1,reg0 ; sign extend
    clr @0
    sbrc    @1,7
    dec @0
    .endmacro

; =======================================
;   Jump/Call with constant arguments
; =======================================

; --- calls with arguments a,b,XYZ ---
```

```
.macro  CX  ; subroutine,x
    ldi xl, low(@1)
    ldi xh,high(@1)
    rcall   @0
    .endmacro
.macro  CXY ; subroutine,x,y
    ldi xl, low(@1)
    ldi xh,high(@1)
    ldi yl, low(@2)
    ldi yh,high(@2)
    rcall   @0
    .endmacro
.macro  CXZ ; subroutine,x,z
    ldi xl, low(@1)
    ldi xh,high(@1)
    ldi zl, low(@2)
    ldi zh,high(@2)
    rcall   @0
    .endmacro
.macro  CXYZ    ; subroutine,x,y,z
    ldi xl, low(@1)
    ldi xh,high(@1)
    ldi yl, low(@2)
    ldi yh,high(@2)
    ldi zl, low(@3)
    ldi zh,high(@3)
    rcall   @0
    .endmacro
.macro  CW  ; subroutine,w
    ldi w, @1
    rcall   @0
    .endmacro
.macro  CA  ; subroutine,a
    ldi a0, @1
    rcall   @0
    .endmacro
.macro  CAB ; subroutine,a,b
    ldi a0, @1
    ldi b0, @2
    rcall   @0
    .endmacro

; --- jump with arguments w,a,b ---
.macro  JW  ; subroutine,w
    ldi w, @1
    rjmp    @0
    .endmacro
.macro  JA  ; subroutine,a
    ldi a0, @1
    rjmp    @0
    .endmacro
.macro  JAB ; subroutine,a,b
    ldi a0, @1
    ldi b0, @2
    rjmp    @0
    .endmacro
.list
```

### Fichier : libs/mode_auto.asm

```
; file:    mode_auto.asm   target ATmega128L-4MHz-STK300
; purpose: AUTO mode = "Go-To" satellite database browser.
;          CH+/-  cycle through visible slots, GUIDE travels to the
;          selected slot, CMD_DEL twice deletes it. Drives the X27
;          on AZ and the servo on EL once GUIDE is pressed; the main
;          loop keeps re-entering us via flag_is_moving until both
;          axes are within tolerance.
;
; sub-states (auto_substate):
;   ST_AUTO_BROWSING     scrolling the database
;   ST_AUTO_MOVING_AZ    transit: step toward target azimuth
;   ST_AUTO_MOVING_EL    transit: nudge servo toward target EL
;   ST_AUTO_ARRIVED      reached the slot (display-only)
;   ST_AUTO_CONFIRM_DEL  CMD_DEL prompt visible, awaiting second press

.equ ST_AUTO_BROWSING    = 0
.equ ST_AUTO_MOVING_AZ   = 1
.equ ST_AUTO_MOVING_EL   = 2
.equ ST_AUTO_ARRIVED     = 3
.equ ST_AUTO_CONFIRM_DEL = 4


; === handle_auto ===
; Dispatch by auto_substate.
handle_auto:
    lds     w, auto_substate
    cpi     w, ST_AUTO_BROWSING
    brne    ha_check_az
    rjmp    auto_browsing
ha_check_az:
    cpi     w, ST_AUTO_MOVING_AZ
    brne    ha_check_el
    rjmp    auto_moving_az
ha_check_el:
    cpi     w, ST_AUTO_MOVING_EL
    brne    ha_check_del
    rjmp    auto_moving_el
ha_check_del:
    cpi     w, ST_AUTO_CONFIRM_DEL
    brne    ha_end
    rjmp    auto_confirm_del
ha_end:
    ret


; === auto_browsing ===
; React only to FRESH presses (held -> ignore, UI feel).
auto_browsing:
    lds     w, last_cmd
    lds     _w, held_count
    tst     _w
    breq    ab_check_btns
    rjmp    ab_end
ab_check_btns:
    cpi     w, CMD_CH_UP
    breq    ab_next
    cpi     w, CMD_CH_DN
    breq    ab_prev
    cpi     w, CMD_GUIDE
    breq    ab_go_bridge
    cpi     w, CMD_DEL
    breq    ab_del_bridge
    rjmp    ab_check_0

ab_go_bridge:  rjmp    ab_go
ab_del_bridge: rjmp    ab_del

ab_check_0:
    ; --- Digit shortcut: 1..9 -> slot 0..8, 0 -> slot 9 ---
    cpi     w, CMD_0
    brsh    ab_check_9
    rjmp    ab_end
ab_check_9:
    cpi     w, CMD_9+1
    brlo    ab_is_digit
    rjmp    ab_end
ab_is_digit:
    cpi     w, CMD_0
    breq    ab_digit_0
    mov     r16, w
    dec     r16
    rjmp    ab_load
```

```
ab_digit_0:
    ldi     r16, 9
    rjmp    ab_load


; --- ab_next / ab_prev: skip empty + filtered-out slots ---
; Safety counter MUST live in a register load_satellite doesn't touch.
; load_satellite clobbers r16, r17, r18, r24, X, Y -> use r25.
ab_next:
    lds     r16, target_sat_id
    ldi     r25, 32                     ; tries left before giving up
ab_n_loop:
    inc     r16
    cpi     r16, 32
    brlo    ab_n_check
    clr     r16                         ; wrap 31 -> 0
ab_n_check:
    sts     target_sat_id, r16          ; is_slot_visible reads it from SRAM
    rcall   is_slot_visible
    brne    ab_n_cont
    rjmp    ab_refresh                  ; visible slot found
ab_n_cont:
    lds     r16, target_sat_id
    dec     r25
    breq    ab_n_fail
    rjmp    ab_n_loop
ab_n_fail:
    rjmp    ab_empty

ab_prev:
    lds     r16, target_sat_id
    ldi     r25, 32
ab_p_loop:
    tst     r16
    breq    ab_p_wrap
    dec     r16
    rjmp    ab_p_check
ab_p_wrap:
    ldi     r16, 31                     ; wrap 0 -> 31
ab_p_check:
    sts     target_sat_id, r16
    rcall   is_slot_visible
    brne    ab_p_cont
    rjmp    ab_refresh
ab_p_cont:
    lds     r16, target_sat_id
    dec     r25
    breq    ab_p_fail
    rjmp    ab_p_loop
ab_p_fail:
    rjmp    ab_empty


; --- ab_load: digit-jump target, with revert on rejection ---
; r25 keeps the PREVIOUS cursor so we can roll back on empty/filtered.
ab_load:
    lds     r25, target_sat_id          ; r25 = previous, for revert
    sts     target_sat_id, r16          ; tentative new candidate
    rcall   is_slot_visible
    brne    ab_load_fail
    rjmp    ab_refresh

ab_load_fail:
    ; Revert + re-populate SRAM mirror so LCD lines match the old slot.
    sts     target_sat_id, r25
    mov     r16, r25
    rcall   load_satellite
    rjmp    ab_refresh


ab_empty:
    ; Database empty: show "EMPTY     " and zero the targets.
    ldi     ZL, low(2*str_auto_empty)
    ldi     ZH, high(2*str_auto_empty)
    ldi     XL, low(current_sat_name)
    ldi     XH, high(current_sat_name)
    ldi     _w, 11                      ; 10 chars + NUL
abe_loop:
    lpm     w, Z+
    st      X+, w
    dec     _w
    brne    abe_loop
    clr     w
    sts     target_az_auto, w
    sts     target_az_auto+1, w
```

```
    sts     target_el_auto, w
    rjmp    ab_refresh


ab_go:
    ; Validate slot before locking flag_is_moving.
    lds     r16, target_sat_id
    rcall   load_satellite
    brne    ab_end                      ; invalid -> ignore
    ldi     w, ST_AUTO_MOVING_AZ
    sts     auto_substate, w
    ldi     w, 1
    sts     flag_is_moving, w
    rjmp    ab_refresh


ab_end:
    rjmp    ab_refresh

ab_refresh:
    rcall   format_auto_l2
    ret


ab_del:
    ; First DEL press: arm the prompt. Second press (handled by
    ; auto_confirm_del) commits.
    lds     r16, target_sat_id
    rcall   load_satellite
    brne    ab_end                      ; can't delete an empty slot

    ldi     w, ST_AUTO_CONFIRM_DEL
    sts     auto_substate, w
    ldi     ZL, low(2*str_auto_del_prompt)
    ldi     ZH, high(2*str_auto_del_prompt)
    rcall   copy_flash_to_l2_buf
    ret                                 ; no ab_refresh -- keep the prompt


; === auto_confirm_del ===
; Inside the DEL prompt:
;   second DEL  -> wipe slot, chirp, advance cursor, overlay "DELETED"
;   any other   -> cancel, re-dispatch the press as a normal browsing key
auto_confirm_del:
    lds     w, last_cmd
    cpi     w, CMD_DEL
    brne    acd_cancel
    lds     _w, held_count
    tst     _w
    brne    acd_keep_prompt             ; held repeats -> ignore

    lds     r16, target_sat_id
    rcall   delete_satellite
    rcall   chirp
    ldi     w, ST_AUTO_BROWSING
    sts     auto_substate, w

    ; Advance to the next visible slot NOW. ab_next mutates target_sat_id
    ; and re-fills lcd_l2_buf via format_auto_l2; we then overwrite the
    ; buffer with "DELETED" for the overlay. When the 500 ms overlay
    ; expires, init_l2_for_mode -> format_auto_l2 paints the new slot.
    rcall   ab_next

    ldi     ZL, low(2*str_auto_deleted)
    ldi     ZH, high(2*str_auto_deleted)
    rcall   copy_flash_to_l2_buf
    ldi     r24, low(500)
    ldi     r25, high(500)
    rcall   overlay_show
    ret

acd_cancel:
    ldi     w, ST_AUTO_BROWSING
    sts     auto_substate, w
    rjmp    handle_auto                 ; re-evaluate the new button

acd_keep_prompt:
    ret


; === auto_moving_az ===
; Per dispatch: step exactly 1 deg toward target_az_auto on the
; shortest-path arc, until current_az_deg == target_az_auto exactly.
; The X27 is precise enough (SCAN lands on exact degrees, same here)
; so we don't need a tolerance band on AZ. GUIDE (fresh press) cancels.
```

```
auto_moving_az:
    lds     w, last_cmd
    cpi     w, CMD_GUIDE
    brne    am_az_cont
    lds     _w, held_count
    tst     _w                          ; lds doesn't touch flags -- must tst
    brne    am_az_cont                  ; held repeats -> keep moving
    rjmp    am_cancel                   ; fresh GUIDE press -> cancel transit
am_az_cont:
    ; current_az_deg = current_az_steps / 3
    lds     a0, current_az_steps
    lds     a1, current_az_steps+1
    ldi     b0, 3
    ldi     b1, 0
    rcall   div22
    mov     r16, c0
    mov     r17, c1

    lds     r18, target_az_auto
    lds     r19, target_az_auto+1
    rcall   shortest_path_az            ; r17:r16 = signed delta

    ; Check if Delta == 0 (perfect match)
    mov     w, r16
    or      w, r17
    breq    az_done

    ; --- step exactly 1 deg in delta's direction ---
    ldi     a0, AZ_STEP_PARTIAL         ; 3 partials = 1 deg
    tst     r17                         ; high byte negative? -> CCW
    brpl    az_cw
    rcall   az_step_ccw_n
    rjmp    am_az_end
az_cw:
    rcall   az_step_cw_n
    rjmp    am_az_end

az_done:
    ldi     w, ST_AUTO_MOVING_EL
    sts     auto_substate, w

am_az_end:
    rcall   format_auto_l2
    ret


; === auto_moving_el ===
; Per dispatch: nudge OCR1A by one TICKS_STEP toward the target EL.
; Margin = 15 ticks (~3 deg). GUIDE cancels.
auto_moving_el:
    lds     w, last_cmd
    cpi     w, CMD_GUIDE
    brne    am_el_cont
    lds     _w, held_count
    tst     _w                          ; lds doesn't touch flags -- must tst
    brne    am_el_cont                  ; held repeats -> keep moving
    rjmp    am_cancel                   ; fresh GUIDE press -> cancel transit
am_el_cont:
    lds     a0, target_el_auto          ; target deg -> OCR1A target
    clr     a1
    rcall   angle_to_servo_pwm          ; c1:c0 = target PWM

    lds     r16, target_ticks
    lds     r17, target_ticks+1

    ; delta = target (c1:c0) - current (r17:r16)
    mov     a0, c0
    mov     a1, c1
    sub     a0, r16
    sbc     a1, r17

    ; |delta|
    tst     a1
    brpl    el_abs_ok
    com     a0
    com     a1
    subi    a0, low(-1)
    sbci    a1, high(-1)
el_abs_ok:
    tst     a1
    brne    el_not_in_margin
    cpi     a0, 16                      ; within ~15 ticks?
    brlo    el_done
el_not_in_margin:
    cp      c0, r16                     ; target > current -> up
```

```
    cpc     c1, r17
    brsh    el_up
    rcall   servo_nudge_down
    rjmp    am_el_end
el_up:
    rcall   servo_nudge_up
    rjmp    am_el_end

el_done:
    ldi     w, ST_AUTO_BROWSING
    sts     auto_substate, w
    clr     w
    sts     flag_is_moving, w
    rcall   chirp                       ; "arrived"

am_el_end:
    rcall   format_auto_l2
    ret


; === am_cancel ===
; GUIDE-during-transit: drop back to BROWSING, release motion flag.
am_cancel:
    ldi     w, ST_AUTO_BROWSING
    sts     auto_substate, w
    clr     w
    sts     flag_is_moving, w
    rcall   format_auto_l2
    ret


; === format_auto_l2 ===
; Render "AZ:xxx EL:xxx   " using the slot TARGET (not current position).
; AZ at [3..5], EL at [10..12].
format_auto_l2:
    ldi     ZL, low(2*str_auto_l2_template)
    ldi     ZH, high(2*str_auto_l2_template)
    ldi     XL, low(lcd_l2_buf)
    ldi     XH, high(lcd_l2_buf)
    ldi     _w, 16
fa_loop:
    lpm     w, Z+
    st      X+, w
    dec     _w
    brne    fa_loop

    lds     a0, target_az_auto
    lds     a1, target_az_auto+1
    ldi     XL, low(lcd_l2_buf + 3)
    ldi     XH, high(lcd_l2_buf + 3)
    rcall   u16_to_dec3

    lds     a0, target_el_auto
    clr     a1
    ldi     XL, low(lcd_l2_buf + 10)
    ldi     XH, high(lcd_l2_buf + 10)
    rcall   u16_to_dec3
    ret


; === auto_draw_l1 ===
; AUTO IN_MODE line 1. Layout:
;   MOVING_AZ / MOVING_EL  -> "GO TO <name>"
;   else                   -> "Sat: <name>"
; Name is from current_sat_name (10 chars max + NUL); we space-pad
; to fill the remaining columns.
auto_draw_l1:
    ldi     a0, 0x00
    rcall   LCD_pos

    lds     w, auto_substate
    cpi     w, ST_AUTO_MOVING_AZ
    breq    adl1_moving
    cpi     w, ST_AUTO_MOVING_EL
    breq    adl1_moving

    ldi     ZL, low(2*str_auto_sat)     ; "Sat: " (5 chars) + name slot
    ldi     ZH, high(2*str_auto_sat)
    rcall   print_z_string
    ldi     _w, 11                      ; 16 - 5 columns left for name
    rjmp    adl1_name

adl1_moving:
    ldi     ZL, low(2*str_auto_go)      ; "GO TO " (6 chars)
    ldi     ZH, high(2*str_auto_go)
```

```
    rcall   print_z_string
    ldi     _w, 9                       ; name budget (leaves col 15 untouched)

adl1_name:
    ldi     XL, low(current_sat_name)
    ldi     XH, high(current_sat_name)
adl1_nloop:
    ld      a0, X+
    tst     a0
    breq    adl1_fill
    rcall   LCD_putc
    dec     _w
    brne    adl1_nloop
    rjmp    adl1_end

adl1_fill:
    tst     _w
    breq    adl1_end
    ldi     a0, ' '
    rcall   LCD_putc
    dec     _w
    rjmp    adl1_fill

adl1_end:
    ret


; === Flash strings ===
str_auto_l2_template: .db "AZ:000 EL:000   ", 0, 0
str_auto_sat:         .db "Sat: ", 0
str_auto_go:          .db "GO TO ", 0, 0
str_auto_empty:       .db "EMPTY     ", 0, 0
str_auto_del_prompt:  .db "DEL? PUSH DEL   ", 0, 0
str_auto_deleted:     .db "DELETED         ", 0, 0
```

### Fichier : libs/mode_manuel.asm

```
; file:    mode_manuel.asm   target ATmega128L-4MHz-STK300
; purpose: MANUEL mode handler. CH+/- jog azimuth (X27 stepper);
;          VOL+/- jog elevation (Timer1 PWM on OC1A); GUIDE arms a
;          two-tap save to EEPROM. Step size grows with hold time
;          (pick_step_size). LCD line 2 = shared AZ/EL formatter.
;
; sub_state: 0 = JOGGING (default), 1 = CONFIRM SAVE
;
; Button bindings in IN_MODE:
;   CH+      az_step_cw       (HOLD_STEP_*: 1/5/10 deg per repeat)
;   CH-      az_step_ccw
;   VOL+     servo_nudge_up   (target_ticks += TICKS_STEP, clamp MAX)
;   VOL-     servo_nudge_down (target_ticks -= TICKS_STEP, clamp MIN)
;   GUIDE    arm "SAVE:PRESS GUIDE" prompt; second tap commits
;   anything else: ignored


; === handle_manuel ===
; Entered when fsm_layer = IN_MODE and mode = MANUEL. POWER and AV
; were already filtered out by main. We always end by repainting
; lcd_l2_buf -- cheap and keeps the readout fresh.
handle_manuel:
    lds     w, last_cmd

    lds     _w, held_count
    tst     _w
    brne    m_check_repeatable

    ; --- Single-fire branch (fresh press) ---
    lds     _w, sub_state
    cpi     _w, 1
    breq    m_handle_confirm
    cpi     w, CMD_GUIDE
    breq    m_prompt_save
    rjmp    m_check_repeatable

m_handle_confirm:
    ; Any single-fire press in CONFIRM cancels the prompt. A second
    ; GUIDE inside the same fresh-press window commits the save.
    clr     _w
    sts     sub_state, _w               ; revert to JOGGING
    cpi     w, CMD_GUIDE
    breq    m_do_save
    rjmp    m_refresh_lcd

m_prompt_save:
    ldi     _w, 1
    sts     sub_state, _w
    ldi     ZL, low(2*str_manuel_save_prompt)
    ldi     ZH, high(2*str_manuel_save_prompt)
    rcall   copy_flash_to_l2_buf
    ret

m_do_save:
    ; execute_save chirps + writes "SAVED" / "MEM FULL" into lcd_l2_buf.
    ; Arm a 500 ms overlay so the message clears itself; meanwhile
    ; dispatch_in_mode in main.asm freezes our handler so the next
    ; format_manuel_l2 can't overwrite the message.
    rcall   execute_save
    ldi     r24, low(500)
    ldi     r25, high(500)
    rcall   overlay_show
    ret

m_check_repeatable:
    ; In CONFIRM, ignore repeat presses AND don't repaint -- otherwise
    ; format_manuel_l2 would wipe the "SAVE:PRESS GUIDE" prompt.
    lds     _w, sub_state
    cpi     _w, 1
    breq    m_ret

    cpi     w, CMD_CH_UP
    breq    m_az_up
    cpi     w, CMD_CH_DN
    breq    m_az_dn
    cpi     w, CMD_VOL_UP
    breq    m_el_up
    cpi     w, CMD_VOL_DN
    breq    m_el_dn
    rjmp    m_refresh_lcd

m_ret:
```

```
    ret

m_az_up:
    rcall   az_step_cw
    rjmp    m_refresh_lcd
m_az_dn:
    rcall   az_step_ccw
    rjmp    m_refresh_lcd
m_el_up:
    rcall   servo_nudge_up
    rjmp    m_refresh_lcd
m_el_dn:
    rcall   servo_nudge_down
m_refresh_lcd:
    rcall   format_manuel_l2
    ret


; === pick_step_size ===
; Map held_count to degrees per repeat. RC5 auto-repeat is ~9 Hz.
;   held <  10  -> 1 deg/press  (~10 deg/s if held)
;   held <  30  -> 5 deg/press  (~50 deg/s)
;   held >= 30  -> 10 deg/press (~100 deg/s, under X27's 250 deg/s limit)
; Out:      a1 = degrees per repeat
; Clobbers: w, a1
pick_step_size:
    lds     w, held_count
    cpi     w, HOLD_THRESHOLD_FAST
    brsh    pss_fast
    cpi     w, HOLD_THRESHOLD_MED
    brsh    pss_med
    ldi     a1, HOLD_STEP_SLOW
    ret
pss_med:
    ldi     a1, HOLD_STEP_MED
    ret
pss_fast:
    ldi     a1, HOLD_STEP_FAST
    ret


; === az_step_cw / az_step_cw_n ===
; az_step_cw    : MANUEL jog. Step size from pick_step_size (1/5/10 deg).
; az_step_cw_n  : caller-supplied step count. a0 = partials to do CW.
;                 Used by AUTO transit for exact 1-deg increments.
;
; Both share the same body: update current_az_steps (mod-1080 wrap),
; pulse step_motor_forward a0 times. step_motor_forward preserves a0.
az_step_cw:
    rcall   pick_step_size              ; a1 = step degrees
    mov     a0, a1
    add     a0, a1
    add     a0, a1                      ; a0 = 3 * a1 (max 30, no overflow)
az_step_cw_n:
    lds     YL, current_az_steps
    lds     YH, current_az_steps+1
    clr     w
    add     YL, a0
    adc     YH, w
    cpi     YL, low(AZ_MAX_STEPS)
    ldi     w,  high(AZ_MAX_STEPS)
    cpc     YH, w
    brlo    az_cw_store
    subi    YL, low(AZ_MAX_STEPS)       ; wrap: subtract one full turn
    sbci    YH, high(AZ_MAX_STEPS)
az_cw_store:
    sts     current_az_steps, YL
    sts     current_az_steps+1, YH
azcw_pulse_loop:
    rcall   step_motor_forward
    dec     a0
    brne    azcw_pulse_loop
    ret


; === az_step_ccw / az_step_ccw_n ===
; Mirror of az_step_cw / az_step_cw_n: subtract, wrap by +AZ_MAX_STEPS on
; underflow, pulse step_motor_backward.
az_step_ccw:
    rcall   pick_step_size
    mov     a0, a1
    add     a0, a1
    add     a0, a1                      ; a0 = 3 * step degrees
az_step_ccw_n:
    lds     YL, current_az_steps
```

```
    lds     YH, current_az_steps+1
    clr     w
    sub     YL, a0
    sbc     YH, w
    brcc    az_ccw_store                ; no borrow -> still positive
    subi    YL, low(-AZ_MAX_STEPS)      ; underflowed: Y += AZ_MAX_STEPS
    sbci    YH, high(-AZ_MAX_STEPS)
az_ccw_store:
    sts     current_az_steps, YL
    sts     current_az_steps+1, YH
azccw_pulse_loop:
    rcall   step_motor_backward
    dec     a0
    brne    azccw_pulse_loop
    ret


; === format_manuel_l2 ===
; Alias: MANUEL line 2 is the shared "AZ:xxx EL:xxx   " readout.
format_manuel_l2:
    rjmp    format_current_az_el


str_manuel_save_prompt: .db "SAVE:PRESS GUIDE", 0, 0
```

### Fichier : libs/mode_scan.asm

```
; file:    mode_scan.asm   target ATmega128L-4MHz-STK300
; purpose: SCAN mode = 360 deg raster sweep + results browser.
;          On GUIDE we cache the EEPROM DB into SRAM (scan_load_cache),
;          snapshot the current EL as the belt centre, then advance
;          AZ by 1 deg per main-loop tick. Per degree we walk the
;          cache and set bits in scan_results for slots whose AZ ==
;          current_az_deg AND |sat.el - belt_centre| <= 10. After a
;          full revolution we either drop into NO_RESULTS or hand off
;          to the AUTO browser filtered through the bitmap.
;
; sub_state values live in libs/constants.inc as ST_SCAN_*.


; === scan_load_cache ===
; One-shot at sweep entry: mirror all 32 EEPROM slots into
; scan_cache_az/el and set the validity bitmap. Avoids ~46k EEPROM
; reads/sweep. Restores target_sat_id and the LCD source mirror on
; exit so AV-back-to-AUTO sees the same cursor it had.
;
; r25 = slot loop counter (load_satellite doesn't touch r25, no other
; caller is alive concurrently, so we can use it across the rcall).
scan_load_cache:
    lds     r16, target_sat_id          ; save cursor on stack
    push    r16

    clr     r16
    sts     scan_cache_valid, r16
    sts     scan_cache_valid+1, r16
    sts     scan_cache_valid+2, r16
    sts     scan_cache_valid+3, r16

    ldi     r25, 0
slc_loop:
    sts     target_sat_id, r25          ; mirror so load_satellite uses r25
    mov     r16, r25
    call    load_satellite              ; Z=1 if valid; fills target_az/el_auto
    brne    slc_invalid

    ; --- valid: copy AZ (2 B) and EL (1 B) into the cache ---
    ; X = &scan_cache_az[r25 * 2]
    mov     r17, r25
    lsl     r17
    ldi     XL, low(scan_cache_az)
    ldi     XH, high(scan_cache_az)
    add     XL, r17
    clr     r17
    adc     XH, r17
    lds     r18, target_az_auto
    st      X+, r18
    lds     r18, target_az_auto+1
    st      X, r18

    ; X = &scan_cache_el[r25]
    ldi     XL, low(scan_cache_el)
    ldi     XH, high(scan_cache_el)
    add     XL, r25
    clr     r17
    adc     XH, r17
    lds     r18, target_el_auto
    st      X, r18

    ; --- set bit r25 in scan_cache_valid ---
    mov     r17, r25
    lsr     r17
    lsr     r17
    lsr     r17                         ; r17 = byte offset (0..3)
    mov     r16, r25
    andi    r16, 0x07                   ; r16 = bit position (0..7)
    ldi     XL, low(scan_cache_valid)
    ldi     XH, high(scan_cache_valid)
    add     XL, r17
    clr     r17
    adc     XH, r17
    ld      r18, X
    push    ZH
    push    ZL
    ldi     ZL, low(2*bitmask_lut)
    ldi     ZH, high(2*bitmask_lut)
    add     ZL, r16
    clr     r17
    adc     ZH, r17
    lpm     r17, Z
```

```
    pop     ZL
    pop     ZH
    or      r18, r17
    st      X, r18
    rjmp    slc_advance

slc_invalid:
    ; --- empty slot: zero its cache entries (validity bit stays 0) ---
    mov     r17, r25
    lsl     r17
    ldi     XL, low(scan_cache_az)
    ldi     XH, high(scan_cache_az)
    add     XL, r17
    clr     r17
    adc     XH, r17
    clr     r18
    st      X+, r18
    st      X, r18

    ldi     XL, low(scan_cache_el)
    ldi     XH, high(scan_cache_el)
    add     XL, r25
    clr     r17
    adc     XH, r17
    clr     r18
    st      X, r18

slc_advance:
    inc     r25
    cpi     r25, 32
    brsh    slc_done                    ; r25 >= 32 -> exit
    rjmp    slc_loop                    ; rjmp keeps us in range

slc_done:
    ; --- restore cursor + LCD mirror ---
    pop     r16
    sts     target_sat_id, r16
    call    load_satellite
    ret


; === handle_scan ===
; Dispatch by sub_state. POWER + AV already handled in main.
handle_scan:
    lds     w, sub_state
    cpi     w, ST_SCAN_SWEEPING
    brne    hs_disp_not_sweeping
    rjmp    handle_scan_sweeping
hs_disp_not_sweeping:
    cpi     w, ST_SCAN_NO_RESULTS
    brne    hs_disp_not_no_results
    rjmp    handle_scan_no_results
hs_disp_not_no_results:
    cpi     w, ST_SCAN_RESULTS_BROWSE
    brne    hs_disp_not_browse
    rjmp    handle_auto                 ; delegate: AUTO browser w/ bitmap filter
hs_disp_not_browse:
    ; fall through to IDLE handler


; === handle_scan_idle ===
; Default state. GUIDE (fresh press) starts the sweep:
;   1. cache the DB                (scan_load_cache)
;   2. snapshot current EL         (scan_el_at_start)
;   3. zero scan_results + counter
;   4. sub_state = SWEEPING, flag_is_moving = 1
;   5. paint initial "SCANNING AZ:xxx" so the user doesn't see a stale frame
handle_scan_idle:
    lds     w, last_cmd
    cpi     w, CMD_GUIDE
    brne    hs_idle_done
    lds     _w, held_count
    tst     _w
    brne    hs_idle_done                ; ignore RC5 repeats

    rcall   scan_load_cache

    ; --- belt centre = (target_ticks - 500) * 9 / 40 (clamp <0 to 0) ---
    lds     a0, target_ticks
    lds     a1, target_ticks+1
    ldi     b0, low(500)
    ldi     b1, high(500)
    sub     a0, b0
    sbc     a1, b1
    brcc    hs_el_snap_ok
```

```
    clr     a0
    clr     a1
hs_el_snap_ok:
    mov     c0, a0
    mov     c1, a1
    lsl     a0
    rol     a1
    lsl     a0
    rol     a1
    lsl     a0
    rol     a1                          ; a = (ticks-500) * 8
    add     a0, c0
    adc     a1, c1                      ; a = (ticks-500) * 9
    ldi     b0, 40
    ldi     b1, 0
    rcall   div22
    sts     scan_el_at_start, c0

    ; --- zero the bitmap + progress counter ---
    clr     w
    sts     scan_steps_done, w
    sts     scan_steps_done+1, w
    sts     scan_results, w
    sts     scan_results+1, w
    sts     scan_results+2, w
    sts     scan_results+3, w

    ldi     w, ST_SCAN_SWEEPING
    sts     sub_state, w
    ldi     w, 1
    sts     flag_is_moving, w           ; engage main_autonomous
    rjmp    hs_render_sweeping_lcd
hs_idle_done:
    ret


; === handle_scan_sweeping ===
; Per dispatch:
;   1. 3 partials CW (= 1 deg). The 3 ms WAIT_MS inside apply_motor_step
;      is the pacing; main_autonomous re-enters as fast as that allows.
;   2. current_az_steps += 3 (mod AZ_MAX_STEPS)
;   3. scan_steps_done += 3 (== full turn when == AZ_MAX_STEPS)
;   4. hs_match_check -> r24 = hits at this degree
;   5. On any hit: render banner, chirp N times (cap 3), pause 1 s
;   6. Termination check; either NO_RESULTS or hand off to RESULTS_BROWSE
;   7. Otherwise repaint "SCANNING AZ:xxx"
handle_scan_sweeping:
    rcall   step_motor_forward
    rcall   step_motor_forward
    rcall   step_motor_forward

    ; current_az_steps += 3, wrap mod AZ_MAX_STEPS
    lds     YL, current_az_steps
    lds     YH, current_az_steps+1
    adiw    YL, 3
    cpi     YL, low(AZ_MAX_STEPS)
    ldi     w, high(AZ_MAX_STEPS)
    cpc     YH, w
    brlo    hs_az_no_wrap
    subi    YL, low(AZ_MAX_STEPS)
    sbci    YH, high(AZ_MAX_STEPS)
hs_az_no_wrap:
    sts     current_az_steps, YL
    sts     current_az_steps+1, YH

    ; scan_steps_done += 3
    lds     YL, scan_steps_done
    lds     YH, scan_steps_done+1
    adiw    YL, 3
    sts     scan_steps_done, YL
    sts     scan_steps_done+1, YH

    rcall   hs_match_check              ; r24 = hits this degree
    tst     r24
    breq    hs_check_done

    ; --- hit feedback: banner + N chirps (cap 3) + 1 s pause ---
    rcall   hs_render_hit_lcd           ; preserves r24
    cpi     r24, 4
    brlo    hs_chirp_count_ok
    ldi     r24, 3
hs_chirp_count_ok:
hs_chirp_loop:
    tst     r24
    breq    hs_chirp_done
```

```
    rcall   chirp
    dec     r24
    rjmp    hs_chirp_loop
hs_chirp_done:

    ldi     r17, 10                     ; 10 * 100 ms = 1 s
hs_hit_pause_loop:
    WAIT_MS 100
    dec     r17
    breq    hs_hit_pause_done
    rjmp    hs_hit_pause_loop           ; rjmp keeps backward jump in range
hs_hit_pause_done:

hs_check_done:
    ; Done? scan_steps_done >= AZ_MAX_STEPS (= one full turn)
    lds     YL, scan_steps_done
    lds     YH, scan_steps_done+1
    cpi     YL, low(AZ_MAX_STEPS)
    ldi     w, high(AZ_MAX_STEPS)
    cpc     YH, w
    brsh    hs_sweep_done
    rjmp    hs_render_sweeping_lcd

hs_sweep_done:
    ; Sweep over: double chirp, drop motion flag, branch on hit count.
    rcall   chirp
    rcall   chirp
    clr     w
    sts     flag_is_moving, w

    ; OR all 4 bytes of scan_results: zero iff zero hits.
    lds     w, scan_results
    lds     _w, scan_results+1
    or      w, _w
    lds     _w, scan_results+2
    or      w, _w
    lds     _w, scan_results+3
    or      w, _w
    tst     w
    brne    hs_sd_hits

    ; --- no hits ---
    ldi     w, ST_SCAN_NO_RESULTS
    sts     sub_state, w
    ldi     ZL, low(2*str_scan_no_results)
    ldi     ZH, high(2*str_scan_no_results)
    rjmp    copy_flash_to_l2_buf

hs_sd_hits:
    ; --- hits: enable bitmap filter, seed cursor, paint AUTO line 2 ---
    ldi     w, ST_SCAN_RESULTS_BROWSE
    sts     sub_state, w
    ldi     w, 1
    sts     slot_filter_mode, w
    clr     w
    sts     auto_substate, w

    rcall   find_first_visible_slot     ; r24 = lowest hit slot id
    sts     target_sat_id, r24
    mov     r16, r24
    call    load_satellite              ; refresh current_sat_name etc.
    rjmp    format_auto_l2


; === handle_scan_no_results ===
; Terminal screen after a zero-hit sweep. flag_is_moving is already 0
; so AV exits cleanly. We just sit here.
handle_scan_no_results:
    ret


; === hs_render_sweeping_lcd ===
; Paint "SCANNING AZ:xxx " into lcd_l2_buf; AZ digits at [12..14].
hs_render_sweeping_lcd:
    ldi     ZL, low(2*str_scan_sweeping)
    ldi     ZH, high(2*str_scan_sweeping)
    rcall   copy_flash_to_l2_buf

    lds     a0, current_az_steps
    lds     a1, current_az_steps+1
    ldi     b0, AZ_STEP_PARTIAL
    ldi     b1, 0
    rcall   div22                       ; c1:c0 = az_steps / 3

    mov     a0, c0
```

```
    mov     a1, c1
    ldi     XL, low(lcd_l2_buf + 12)
    ldi     XH, high(lcd_l2_buf + 12)
    rcall   u16_to_dec3
    ret


; === hs_match_check ===
; Walk scan_cache, OR each match's bit into scan_results, count hits.
; Slot is a hit iff:
;   sat.az == current_az_deg               (wrap-aware, exact match)
;   |sat.el - scan_el_at_start| <= 10       (belt in EL)
;
; AZ tolerance is 0 because we step exactly 1 deg per iteration -- each
; sat's AZ is sampled exactly once per sweep (one beep, no smearing).
;
; Out:      r24 = hit count this call. Bits in scan_results updated.
; Clobbers: extensive (r16-r23, r25, w, _w, a-d, X, Y, Z).
hs_match_check:
    ; current_az_deg = current_az_steps / 3 (held in Y for the loop)
    lds     a0, current_az_steps
    lds     a1, current_az_steps+1
    ldi     b0, AZ_STEP_PARTIAL
    ldi     b1, 0
    rcall   div22
    mov     YL, c0
    mov     YH, c1

    clr     r24                         ; hit_count
    clr     r25                         ; slot_id

hmc_loop:
    ; --- bit r25 of scan_cache_valid set? ---
    mov     r17, r25
    lsr     r17
    lsr     r17
    lsr     r17
    mov     r16, r25
    andi    r16, 0x07
    ldi     ZL, low(scan_cache_valid)
    ldi     ZH, high(scan_cache_valid)
    add     ZL, r17
    clr     r17
    adc     ZH, r17
    ld      r18, Z
    push    ZH
    push    ZL
    ldi     ZL, low(2*bitmask_lut)
    ldi     ZH, high(2*bitmask_lut)
    add     ZL, r16
    clr     r17
    adc     ZH, r17
    lpm     r17, Z
    pop     ZL
    pop     ZH
    and     r18, r17
    brne    hmc_v_passed
    rjmp    hmc_next

hmc_v_passed:
    ; --- AZ check: shortest_path(sat.az, cur_az) == 0 ---
    mov     r17, r25
    lsl     r17
    ldi     ZL, low(scan_cache_az)
    ldi     ZH, high(scan_cache_az)
    add     ZL, r17
    clr     r17
    adc     ZH, r17
    ld      r18, Z+                     ; sat.az low
    ld      r19, Z                      ; sat.az high

    mov     r16, YL                     ; shortest_path_az: cur in r17:r16
    mov     r17, YH
    rcall   shortest_path_az            ; r17:r16 = signed delta
    rcall   abs16                       ; r17:r16 = |delta|

    or      r17, r16                    ; nonzero iff |delta| != 0
    breq    hmc_az_in_band
    rjmp    hmc_next

hmc_az_in_band:
    ; --- EL check: |sat.el - belt_centre| <= 10 ---
    ldi     ZL, low(scan_cache_el)
    ldi     ZH, high(scan_cache_el)
    add     ZL, r25
```

```
    clr     r17
    adc     ZH, r17
    ld      r18, Z                      ; sat.el

    lds     r17, scan_el_at_start
    cp      r18, r17
    brsh    hmc_el_sat_ge
    sub     r17, r18                    ; sat.el < cur -> diff = cur - sat
    mov     r18, r17
    rjmp    hmc_el_have_diff
hmc_el_sat_ge:
    sub     r18, r17                    ; sat.el >= cur -> diff = sat - cur
hmc_el_have_diff:
    cpi     r18, 11
    brlo    hmc_hit
    rjmp    hmc_next

hmc_hit:
    ; --- OR bit r25 into scan_results, bump hit count ---
    mov     r17, r25
    lsr     r17
    lsr     r17
    lsr     r17
    mov     r16, r25
    andi    r16, 0x07
    ldi     ZL, low(scan_results)
    ldi     ZH, high(scan_results)
    add     ZL, r17
    clr     r17
    adc     ZH, r17
    ld      r18, Z
    push    ZH
    push    ZL
    ldi     ZL, low(2*bitmask_lut)
    ldi     ZH, high(2*bitmask_lut)
    add     ZL, r16
    clr     r17
    adc     ZH, r17
    lpm     r17, Z
    pop     ZL
    pop     ZH
    or      r18, r17
    st      Z, r18
    inc     r24

hmc_next:
    inc     r25
    cpi     r25, 32
    brsh    hmc_done
    rjmp    hmc_loop
hmc_done:
    ret


; === find_first_visible_slot ===
; Find the lowest slot id whose scan_results bit is set. Caller must
; have checked that at least one bit is set; the "none" fallback returns
; 0 just to keep the routine total.
; Out: r24 = slot id
find_first_visible_slot:
    clr     r24
ffvs_loop:
    mov     r17, r24
    lsr     r17
    lsr     r17
    lsr     r17                         ; byte offset
    mov     r16, r24
    andi    r16, 0x07                   ; bit position
    ldi     ZL, low(scan_results)
    ldi     ZH, high(scan_results)
    add     ZL, r17
    clr     r17
    adc     ZH, r17
    ld      r18, Z

    push    ZH
    push    ZL
    ldi     ZL, low(2*bitmask_lut)
    ldi     ZH, high(2*bitmask_lut)
    add     ZL, r16
    clr     r17
    adc     ZH, r17
    lpm     r17, Z
    pop     ZL
    pop     ZH
```

```
    and     r18, r17
    brne    ffvs_done

    inc     r24
    cpi     r24, 32
    brsh    ffvs_none
    rjmp    ffvs_loop
ffvs_none:
    clr     r24                         ; defensive: shouldn't be reached
ffvs_done:
    ret


; === hs_render_hit_lcd ===
; Paint "N SATS DETECTED " into lcd_l2_buf, with N (1..9, capped at 9
; for display) at column 0. r24 is PRESERVED so the caller's chirp loop
; sees the real (uncapped) hit count.
hs_render_hit_lcd:
    mov     r18, r24
    cpi     r18, 10
    brlo    hs_rhl_no_cap
    ldi     r18, 9
hs_rhl_no_cap:
    subi    r18, -'0'                   ; -> ASCII '1'..'9'

    ldi     ZL, low(2*str_scan_hit)
    ldi     ZH, high(2*str_scan_hit)
    rcall   copy_flash_to_l2_buf
    ldi     XL, low(lcd_l2_buf)
    ldi     XH, high(lcd_l2_buf)
    st      X, r18
    ret


; === Flash strings ===
; init_l2_for_mode (shared_handlers.asm) falls through to str_scan_l2
; for the SCAN AV-enter prompt -- symbol name kept for that caller.
str_scan_l2:           .db "SCAN: PUSH GUIDE", 0, 0
str_scan_sweeping:     .db "SCANNING AZ:000 ", 0, 0
str_scan_no_results:   .db "NO SATS FOUND   ", 0, 0
str_scan_hit:          .db "N SATS DETECTED ", 0, 0
str_scan_hits_done:    .db "HITS DETECTED   ", 0, 0

bitmask_lut: .db 1, 2, 4, 8, 16, 32, 64, 128
```

### Fichier : libs/overlay.asm

```
; file:    overlay.asm    target ATmega128L-4MHz-STK300
; purpose: timed transient message on LCD line 2 (SAVED, DELETED,
;          UART ON/OFF, ...). Owns timing only -- callers fill
;          lcd_l2_buf before calling overlay_show. Timer2 1 ms ISR
;          decrements the countdown and raises overlay_dirty on
;          expiry; main loop polls overlay_consume_dirty to repaint.

.dseg
overlay_ms_left: .byte 2    ; 16-bit countdown, 0 = idle
overlay_dirty:   .byte 1    ; ISR sets to 1 when countdown reaches 0
.cseg


; === overlay_show ===
; Arm the countdown. Caller must have filled lcd_l2_buf already.
; In:  r25:r24 = duration in ms (e.g. ldi r24,low(500) / r25,high(500))
overlay_show:
    ; All call sites are main-context (sei has already run). Preserve r16
    ; because main_dispatch holds last_cmd in w (=r16) across the chain;
    ; clobbering it makes every cpi-against-CMD_x miss.
    push    r16
    cli                             ; atomic 16-bit write
    sts     overlay_ms_left, r24
    sts     overlay_ms_left+1, r25
    clr     r16
    sts     overlay_dirty, r16      ; clear stale "just-expired" flag
    sei
    pop     r16
    ret


; === overlay_active ===
; Is an overlay currently displayed?
; Out: Z=1 if idle, Z=0 if showing. Caller does `brne <render overlay>`.
; Preserves r16/r17 (and the Z flag across the pops).
overlay_active:
    push    r16
    push    r17
    lds     r16, overlay_ms_left
    lds     r17, overlay_ms_left+1
    or      r16, r17                ; Z = (ms_left == 0)
    in      r0, SREG                ; save Z across the pops
    pop     r17
    pop     r16
    out     SREG, r0
    ret


; === overlay_consume_dirty ===
; Poll-and-clear for the main loop. Returns Z=0 if the ISR just
; expired the overlay (and clears the flag); Z=1 otherwise.
; Preserves r16 + the Z flag.
overlay_consume_dirty:
    push    r16
    lds     r16, overlay_dirty
    tst     r16
    breq    ocd_clean
    clr     r16
    sts     overlay_dirty, r16
    ldi     r16, 1                  ; force Z=0 (was dirty)
    tst     r16
    in      r0, SREG
    pop     r16
    out     SREG, r0
    ret
ocd_clean:
    in      r0, SREG                ; Z=1 already from the tst above
    pop     r16
    out     SREG, r0
    ret


; === overlay_force_clear ===
; Cancel any active overlay immediately. Used by main_dispatch's
; "any-key dismisses overlay" path and by emergency_stop. Does NOT
; raise overlay_dirty (caller will redraw anyway).
overlay_force_clear:
    push    r16
    cli
    clr     r16
    sts     overlay_ms_left, r16
    sts     overlay_ms_left+1, r16
```

```
    sts     overlay_dirty, r16
    sei
    pop     r16
    ret
```

**Fichier : libs/pinout.inc**

```
; file:    pinout.inc   target ATmega128L-4MHz-STK300
; purpose: hardware pin/port allocation, single source of truth.
;          .equ-only -- choose where each Mx daughter-board lands and
;          re-assemble. No other file should hard-code PORT / PIN.
;
; Current allocation:
;   PORTA  LCD data bus (memory-mapped XMEM)
;   PORTB  M4: S3003 elevation servo on PB5 = OC1A (Timer1 Fast-PWM)
;   PORTC  LCD address high byte (memory-mapped XMEM)
;   PORTD  M1: X27 azimuth stepper (low nibble = 4 coil terminals)
;   PORTE  M2: IR demod on PE7, buzzer on PE2, UART0 on PE0/PE1
;   PORTF  free (future ADC / spare)
;   PORTG  LCD control: PG0=/WR, PG1=/RD, PG2=ALE

.nolist

; === M2 module (IR + buzzer) on PORTE ===
; IR demodulator: idle HIGH, drives LOW for each marked half-bit.
.equ  IR_PORT_IN  = PINE            ; read input level here
.equ  IR_PIN      = IR              ; bit 7 (from definitions.asm)

; Piezo buzzer: toggled by chirp.
.equ  BUZZER_PORT = PORTE
.equ  BUZZER_DDR  = DDRE
.equ  BUZZER_PIN  = SPEAKER         ; bit 2 (from definitions.asm)

; === M4 module (S3003 elevation servo) on PORTB ===
; M4's P8 connector routes to PB5 = OC1A = Timer1 16-bit hardware PWM.
; (Earlier hardware revision used M4 P7 -> PB4 = OC0; that's 8-bit and
; can't reach 50 Hz with usable resolution. Moving the M4 jumper from
; P7 to P8 is the fix.)
.equ  SERVO_PORT    = PORTB
.equ  SERVO_DDR     = DDRB
.equ  SERVO_EL_PIN  = 5             ; PB5 = OC1A

; === M1 module (X27 azimuth stepper) on PORTD ===
; Bipolar 2-coil, 4 GPIO lines. Driven directly from AVR pins (~20 mA
; per coil at 5 V, no H-bridge needed for the X27).
;   PD0 = Coil1+ (motor pin 1)       PD2 = Coil2+ (motor pin 4)
;   PD1 = Coil1- (motor pin 2)       PD3 = Coil2- (motor pin 3)
.equ  STEPPER_PORT     = PORTD
.equ  STEPPER_DDR      = DDRD
.equ  STEPPER_MASK     = 0x0F       ; PD0..PD3 = coil outputs
.equ  STEPPER_C1P_PIN  = 0
.equ  STEPPER_C1N_PIN  = 1
.equ  STEPPER_C2P_PIN  = 2
.equ  STEPPER_C2N_PIN  = 3

; === LCD (memory-mapped HD44780 over external SRAM bus) ===
; Already hard-coded in drivers/lcd.asm as 0x8000 (instruction) and
; 0xC000 (data). Listed here for completeness.
;   PORTA = AD0..AD7 (multiplexed)
;   PORTC = A8..A15
;   PG0 = /WR, PG1 = /RD, PG2 = ALE

.list
```

### Fichier : libs/shared_handlers.asm

```
; file:    shared_handlers.asm   target ATmega128L-4MHz-STK300
; purpose: FSM glue dispatched from main BEFORE per-mode handlers:
;            - can_transition  : motion-in-progress gate
;            - cycle_mode_*    : CH+/- in BROWSING
;            - toggle_layer    : AV-button BROWSING <-> IN_MODE
;            - emergency_stop  : POWER reaction
;            - init_l2_for_mode: paint mode-specific line 2 on AV-enter
;            - is_slot_visible : AUTO/SCAN slot filter
;            - copy_flash_to_l2_buf : LPM copy helper


; === can_transition ===
; Block mode/layer changes while a motor transit is running.
; Out: r16 = 1 (ok) or 0 (blocked)
can_transition:
    lds     r16, flag_is_moving
    tst     r16
    breq    transition_ok
    clr     r16
    ret
transition_ok:
    ldi     r16, 1
    ret


; === emergency_stop ===
; POWER button: kill all motion and chirp twice. Caller (main) owns
; clearing fsm_layer / sub_state / flag_is_moving afterwards.
;   1. servo_off  -> OC1A disconnected, PB5 driven low (S3003 goes limp)
;   2. zero PORTD coil bits, leaving upper nibble untouched
;   3. two chirps so the user hears that POWER landed
emergency_stop:
    rcall   servo_off
    in      w, STEPPER_PORT
    andi    w, ~STEPPER_MASK
    out     STEPPER_PORT, w
    rcall   chirp
    rcall   chirp
    ret


; === cycle_mode_next ===
; mode = (mode + 1) mod MODE_COUNT. No-op if motion is in progress.
cycle_mode_next:
    rcall   can_transition
    tst     r16
    breq    cmn_end
    lds     w, mode
    inc     w
    cpi     w, MODE_COUNT
    brlo    cmn_store
    clr     w                           ; wrap MODE_COUNT -> 0
cmn_store:
    sts     mode, w
cmn_end:
    ret


; === cycle_mode_prev ===
; mode = (mode - 1) mod MODE_COUNT. No-op while moving.
cycle_mode_prev:
    rcall   can_transition
    tst     r16
    breq    cmp_end
    lds     w, mode
    dec     w
    brpl    cmp_store                   ; still >= 0 -> keep
    ldi     w, MODE_COUNT-1             ; underflow -> wrap to last
cmp_store:
    sts     mode, w
cmp_end:
    ret


; === toggle_layer ===
; AV button: flip fsm_layer, reset sub_state, chirp. On IN_MODE entry
; paint line 2 from the mode's formatter. On BROWSING exit also wipe
; SCAN visibility state so AUTO isn't left filtered.
toggle_layer:
    rcall   can_transition
    tst     r16
    breq    tl_end
```

```
    lds     w, fsm_layer
    cpi     w, LAYER_BROWSING
    breq    tl_to_in_mode

    ; --- was IN_MODE, going to BROWSING ---
    ldi     w, LAYER_BROWSING
    sts     fsm_layer, w
    clr     w
    sts     slot_filter_mode, w
    sts     scan_results, w
    sts     scan_results+1, w
    sts     scan_results+2, w
    sts     scan_results+3, w
    rjmp    tl_finish

tl_to_in_mode:
    ldi     w, LAYER_IN_MODE
    sts     fsm_layer, w
    rcall   init_l2_for_mode

tl_finish:
    rcall   servo_on                    ; re-engage OC1A (no-op if not estopped)
    clr     w
    sts     sub_state, w
    rcall   chirp
tl_end:
    ret


; === init_l2_for_mode ===
; Paint lcd_l2_buf for AV-enter, depending on current mode.
;   AUTO   -> format_auto_l2          (target AZ/EL for the slot)
;   MANUEL -> format_current_az_el    (live AZ/EL)
;   SCAN   -> static "PUSH GUIDE" stub
init_l2_for_mode:
    lds     w, mode
    cpi     w, MODE_AUTO
    breq    il2_auto
    cpi     w, MODE_MANUEL
    breq    il2_manuel
    ldi     ZL, low(2*str_scan_l2)      ; fall-through = SCAN
    ldi     ZH, high(2*str_scan_l2)
    rjmp    copy_flash_to_l2_buf
il2_auto:
    rjmp    format_auto_l2
il2_manuel:
    rjmp    format_manuel_l2            ; = format_current_az_el (alias)


; === is_slot_visible ===
; Decide whether the AUTO/SCAN browser should land on a slot.
;   slot_filter_mode = 0 -> visible iff load_satellite says valid (0x5A)
;   slot_filter_mode = 1 -> also requires bit (slot_id) set in scan_results
;
; In:       target_sat_id SRAM = candidate slot (caller stores it first)
; Out:      Z = 1 visible, Z = 0 skip   (matches load_satellite semantics)
; Clobbers: same set as load_satellite (r16, r17, r18, r24, X, Y).
;           Does NOT touch r19 (=a1) or any state AUTO holds across the call.
is_slot_visible:
    lds     r16, target_sat_id
    rcall   load_satellite              ; Z=1 if magic byte 0x5A
    brne    isv_invisible

    lds     r17, slot_filter_mode
    tst     r17
    breq    isv_visible                 ; AUTO default -> short-circuit

    ; --- SCAN filter: bit (slot_id) of scan_results must be set ---
    ; load_satellite trashed r16; reload the slot id from SRAM.
    lds     r16, target_sat_id
    mov     r17, r16
    lsr     r17
    lsr     r17
    lsr     r17                         ; r17 = byte offset (0..3)
    andi    r16, 0x07                   ; r16 = bit position (0..7)

    ldi     XL, low(scan_results)
    ldi     XH, high(scan_results)
    add     XL, r17
    clr     r17
    adc     XH, r17
    ld      r18, X

    ; r17 = 1 << r16 (one-hot mask)
```

```
    ldi     r17, 1
isv_shift:
    tst     r16
    breq    isv_check
    lsl     r17
    dec     r16
    rjmp    isv_shift
isv_check:
    and     r18, r17
    brne    isv_visible

isv_invisible:
    clz                                 ; Z=0 -> "skip"
    ret
isv_visible:
    sez                                 ; Z=1 -> "land here"
    ret


; === copy_flash_to_l2_buf ===
; LPM-copy 16 bytes from Z into lcd_l2_buf.
; In:       Z = source flash byte address (pre-doubled via low(2*lbl)/high)
; Out:      lcd_l2_buf[0..15] filled
; Clobbers: w, _w, X, Z
copy_flash_to_l2_buf:
    ldi     XL, low(lcd_l2_buf)
    ldi     XH, high(lcd_l2_buf)
    ldi     _w, 16
cflb_loop:
    lpm     w, Z+
    st      X+, w
    dec     _w
    brne    cflb_loop
    ret
```