Mikroprocesory

10. Správa napájení a návrh s nízkou spotřebou

Stanislav Vítek Katedra radioelektroniky České vysoké učení technické v Praze

V minulých přednáškách jsme viděli

1. MCU obsahuje jádro a různé periferie
2. Jednotlivé části MCU je možné nezávisle zapínat a vypínat
3. Jako zdroj hodinového taktu lze použít různě rychlé oscilátory
4. K MCU lze připojovat periferie
5. Existují techniky, jak flexibilně plánovat provádění úkolů

Obsah přednášky

1. Spotřeba MCU, měření a techniky snižování
2. Architektura napájecího systému STM32
3. Úsporné režimy
4. Návrhové vzory s nízkou spotřebou
5. FreeRTOS a jeho použití v low-power systémech

1. Spotřeba MCU a techniky jejího snižování

Spotřeba MCU

Spotřeba mikrokontroléru (MCU) se dělí na dvě základní složky:

1. Dynamická spotřeba (Dynamic Power Consumption)
   - Energie spotřebovaná při aktivní činnosti MCU.
   - Jsou to ztráty způsobené nabíjením a vybíjením kapacitních zátěží (převážně hradel tranzistorů) během přepínání logických stavů.
2. Statická spotřeba (Static Power Consumption)
   - Energie spotřebovaná, i když je MCU zcela nečinný (nebo v režimu spánku).
   - Je způsobena úniky (leakage) proudu.

Dynamická spotřeba - příčiny a optimalizace

Přepínací spotřeba: Hlavní složka. Vzniká při každé změně stavu na výstupech logických hradel. Energie je spotřebována k nabíjení a vybíjení parazitních kapacit C.

$$P_{switching} \propto C \times V_{DD}^2 \times f$$

Zkratový proud: Vzniká krátkým okamžikem, kdy se během přepínání PMOS i NMOS tranzistor v CMOS hradle nacházejí současně v částečně vodivém stavu, což vytvoří zkratovou cestu mezi napájením (V_{DD}) a zemí.

Optimalizace:

- DVFS (Dynamic Voltage and Frequency Scaling)
- Clock Gating - vypnutí hodin pro nepoužívané periferie
  - může uspořit 10–90 % celkové dynamické spotřeby

Co rozhoduje o velikosti dynamické spotřeby?

- zátěž na signálových vodičích, např. `I²C` pull-up 4.7k při 3.3V → 0.7mA
- hradlová kapacita (například u `GPIO` s velkou kapacitou kabelu)
- periferie v aktivním režimu
  - `PLL` → jednotky mA, `UART` → 100–300 µA, `ADC` → 100 µA – 1 mA
  - `Wi-Fi` → 80–200 mA, `BLE` adv. → 5–15 mA, `LoRa` → 40–120 mA

Parametry typických MCU

- STM32F4 @ 168 MHz: stovky µA/MHz
- STM32L4 @ 80 MHz: desítky µA/MHz
- MSP430: jednotky µA/MHz
- nRF52: ~55 µA/MHz

Statická spotřeba - příčiny a optimalizace

Únikový proud: V CMOS technologii jsou tranzistory (zejména ty menší v moderních procesech) stále v omezené míře vodivé, i když jsou v *vypnutém* stavu.

Tento proud $I_{leakage}$ protéká, i když se logické stavy nemění.

$$P_{static} = I_{leakage} \times V_{DD}$$

Vysoká teplota: Únikový proud se exponenciálně zvyšuje s rostoucí teplotou.

Optimalizace

- Režimy hlubokého spánku - odpojení nebo snížení napětí na velkých částech čipu

Exponenciální vliv teploty

- MCU může mít při –20 °C standby proud 300 nA
- ale při +85 °C to může být i 2–10 µA (!)

Proč je statickou spotřebu těžké minimalizovat

- určená výrobním procesem → roste s technologickou miniaturizací

Parametry typických MCU

- Low-power MCU (STM32L0/L4/L5/U5): 200–800 nA
- nRF52 Standby: ~300 nA
- MSP430 LPM3: 100–500 nA
- ESP32 (hluboký spánek): 5–20 µA (nemá extrémně nízký leakage)

Měření spotřeby MCU

Přesné měření spotřeby v µA a nA rozsahu je náročné. Běžné multimetry nejsou dostatečně přesné a mají příliš velký burden voltage.

Specializované nástroje

Nástroj	URL	Rozsah	Rozlišení	Poznámka
Nordic PPK2	link	200 nA – 1 A	100 nA	Velmi dobrý poměr cena/výkon
Otii Arc	link	1 µA – 5 A	100 nA	Pokročilá analýza profilů
Keysight N6705C	link	1 nA – 3 A	sub-nA	Profesionální, drahé
µCurrent Gold	link	1 nA – 1 A	nA	Pasivní bočník
Joulescope	link	1 µA – 3 A	µA	Open-source software

Praktické tipy pro měření

Příprava MCU

1. Odpojit debugger (SWD/JTAG) – může přidávat µA až mA
2. Vypnout všechny LED indikátory
3. Zkontrolovat floating piny → nastavit jako analog nebo s pull-up/down
4. Odpojit nepotřebné periferie na DPS

Měřicí setup

- Krátké vodiče mezi zdrojem a MCU, stínění proti rušení
- Stabilní napájecí napětí (použít baterii nebo LDO s nízkým šumem)

Typické chyby

- Měření s připojeným debuggerem
- Ignorování proudových špiček při probuzení
- Nedostatečná doba ustálení před měřením

2. Architektura napájecího systému STM32

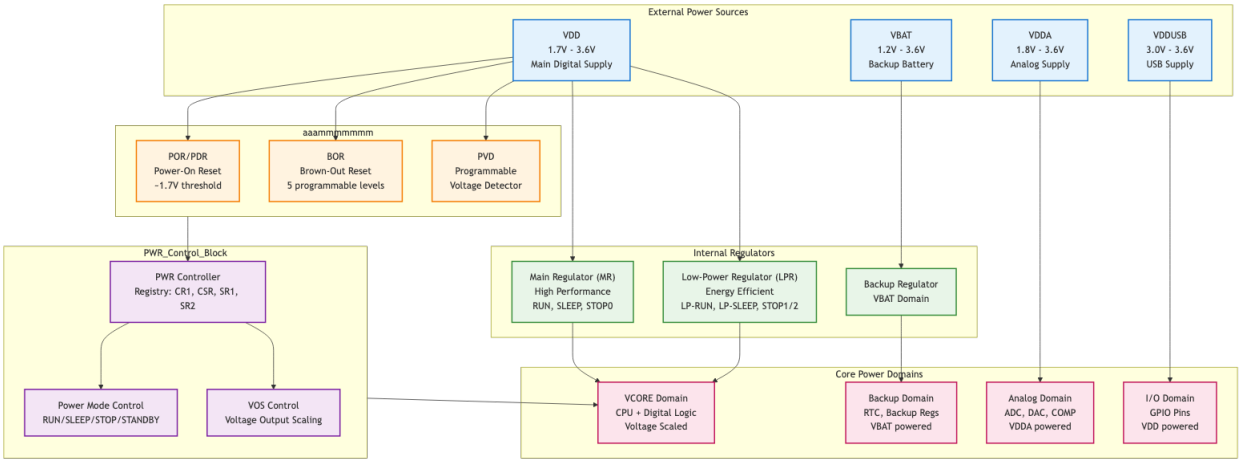

Figure 1: Zjednodušená architektura napájecího systému STM32

1. **Externí zdroje**: VDD (1.7–3.6V), VBAT (záloha), VDDA (analogová část), VDDUSB
2. **Supervize napětí**: POR/PDR reset, BOR (5 úrovní), PVD (programovatelný detektor)
3. **Regulátory**: Main Regulator (vysoký výkon), Low-Power Regulator (úsporný), Backup Regulator
4. **Napájecí domény**: VCORE (CPU + logika), Backup (RTC), Analog (ADC/DAC), I/O (GPIO)

2.1. Externí zdroje napájení

Pin	Rozsah	Účel	Poznámka
VDD	1.7–3.6V	Hlavní digitální napájení	Napájí regulátory, I/O, supervizi
VBAT	1.2–3.6V	Záložní baterie	RTC, backup registry, LSE
VDDA	1.8–3.6V	Analogové obvody	ADC, DAC, komparátory, reference
VDDUSB	3.0–3.6V	USB PHY	Pouze pokud je USB použito

Doporučení pro návrh DPS

- VDD a VDDA propojit přes feritovou perličku (filtr šumu)
- VBAT připojit i když nepoužíváte RTC (jinak floating = vyšší spotřeba)
- Blokovací kondenzátory co nejblíže pinům (100 nF + 10 µF)

2.2. Supervize napětí

Brown-out Reset (BOR)

Hardwarový mechanismus, který resetuje MCU při poklesu napájecího napětí pod kritickou mez.

- Chrání před nepředvídatelným chováním při nízkém napětí
- Typicky konfigurovatelné prahy: 1.7V, 2.0V, 2.2V, 2.4V (závisí na rodině)
- BOR je aktivní i v nejhlubších režimech spánku
- **Spotřeba:** ~1–5 µA (nelze vypnout u většiny MCU)

Power Voltage Detector (PVD)

Softwarově konfigurovatelný detektor napětí s přerušením.

- Umožňuje včasnou reakci před BOR resetem
- Lze nastavit práh vyšší než BOR → čas na uložení dat
- Generuje přerušení `PVD_IRQ`

Praktické použití PVD

```
// Konfigurace PVD na práh ~2.5V
PWR->CR |= PWR_CR_PLS_LEV5;  // Threshold level 5
PWR->CR |= PWR_CR_PVDE;      // Enable PVD

// Povolení přerušení
EXTI->IMR |= EXTI_IMR_MR16;   // PVD je na EXTI line 16
EXTI->RTSR |= EXTI_RTSR_TR16; // Rising edge (VDD klesá pod práh)
NVIC_EnableIRQ(PVD_IRQn);
```

PVD ISR — uložení kritických dat

```
void PVD_IRQHandler(void) {
    if (EXTI->PR & EXTI_PR_PR16) {
        EXTI->PR = EXTI_PR_PR16;  // Clear flag

        // Uložit kritická data do Backup SRAM nebo Flash
        SaveCriticalState();

        // Volitelně: přejít do Standby pro minimalizaci spotřeby
        EnterStandbyMode();
    }
}
```

BOR a spotřeba energie

Trade-off: bezpečnost vs spotřeba

- BOR nelze vypnout → konstantní spotřeba ~1–5 µA
- U některých MCU (STM32L) lze snížit BOR threshold pro nižší spotřebu
- Nižší práh = větší riziko nestabilního chování při pomalém poklesu napětí

Doporučení pro bateriové aplikace

1. Nastavit PVD práh nad BOR práh
2. Při PVD přerušení uložit stav a přejít do Standby
3. BOR pak provede čistý reset bez ztráty dat

Událost	Typický práh	Akce
PVD	2.5V	Uložit data, Standby
BOR	2.0V	Hardware reset
Kritické minimum	1.7V	MCU nefunkční

2.3. Interní regulátory

Main Regulator (MR)

- Napájí VCORE doménu v režimech **Run, Sleep, Stop0**
- Nejvyšší výkon, rychlý odezva na změny zátěže
- Spotřeba: desítky µA

Low-Power Regulator (LPR)

- Napájí VCORE v režimech **LP-Run, LP-Sleep, Stop1, Stop2**
- Nižší výkon, pomalejší reakce, ale výrazně nižší spotřeba
- Spotřeba: jednotky µA

Backup Regulator

- Napájí Backup doménu z VBAT
- Aktivní i při odpojeném VDD
- Umožňuje retenci dat v backup SRAM

PWR Control Registry

STM32F4 – `PWR->CR`

Bit	Název	Funkce
15:14	VOS	Voltage scaling (Scale 1/2/3)
9	DBP	Disable backup domain write protection
8	FPDS	Flash power-down in Stop
4	ADCDC1	ADC DC1
3	PDDS	Power-down deep sleep (0=Stop, 1=Standby)
2	CWUF	Clear wakeup flag
1	CSBF	Clear standby flag
0	LPDS	Low-power deep sleep (regulátor v Stop)

STM32L4 – `PWR->CR1`

Bit	Název	Funkce
14	LPR	Low-power run mode enable
10:9	VOS	Voltage scaling (Range 1/2)
8	DBP	Disable backup domain write protection
2:0	LPMS	Low-power mode selection (Stop0/1/2, Standby, Shutdown)

Přepínání regulátorů (STM32L4)

```
// Přechod do Low-Power Run režimu (používá LPR místo MR)
void enter_low_power_run(void) {
    // 1. Snížit frekvenci na max 2 MHz (požadavek pro LP Run)
    RCC->CR |= RCC_CR_MSION;
    while (!(RCC->CR & RCC_CR_MSIRDY));
    RCC->CFGR = (RCC->CFGR & ~RCC_CFGR_SW) | RCC_CFGR_SW_MSI;

    // 2. Přepnout na Low-Power Regulator
```

```c
    PWR->CR1 |= PWR_CR1_LPR;

    // 3. Počkat na přepnutí (REGLPF flag)
    while (!(PWR->SR2 & PWR_SR2_REGLPF));
}

// Návrat do Run režimu (Main Regulator)
void exit_low_power_run(void) {
    PWR->CR1 &= ~PWR_CR1_LPR;
    while (PWR->SR2 & PWR_SR2_REGLPF);  // Počkat na MR
    // Nyní lze zvýšit frekvenci
}
```

DVFS – Dynamické škálování napětí/frekvence

Voltage scaling umožňuje snížit VCORE napětí při nižších frekvencích → nižší spotřeba.

STM32L4/L5/U5 (Range Scaling)

Range	VCORE	Max frekvence	Použití
Range 1	~1.2V	80/120 MHz	High Performance
Range 2	~1.0V	26 MHz	Low Power

STM32F4 (VOS Scaling)

VOS	Bity	VCORE	Max frekvence
Scale 1	`11`	~1.2V	168 MHz
Scale 2	`10`	~1.0V	144 MHz
Scale 3	`01`	~0.9V	podle varianty

Klíčový rozdíl

- **L řady**: Změna VOS za běhu → HW automaticky upraví VCORE
- **F řady**: Nutno vypnout PLL → změnit VOS → znovu zapnout PLL

Voltage Scaling u STM32L4 (on-the-fly)

```c
// Přechod z Range 1 do Range 2 (snížení výkonu/spotřeby)
void switch_to_range2(void) {
    // 1. Snížit frekvenci pod 26 MHz (limit pro Range 2)
    // ... (např. přepnout na MSI)

    // 2. Přepnout voltage range
    PWR->CR1 = (PWR->CR1 & ~PWR_CR1_VOS) | PWR_CR1_VOS_1;  // Range 2

    // 3. Počkat na stabilizaci
    while (PWR->SR2 & PWR_SR2_VOSF);  // VOSF=0 když hotovo
}
```

- PLL může zůstat zapnutý (pokud splňuje frekvenční limit)
- Přechod trvá ~5 µs

Voltage Scaling u STM32F4 (nutný restart PLL)

```c
void switch_to_scale2(void) {
    // 1. Přepnout SYSCLK na HSI (pryč z PLL)
    RCC->CFGR = (RCC->CFGR & ~RCC_CFGR_SW) | RCC_CFGR_SW_HSI;
    while ((RCC->CFGR & RCC_CFGR_SWS) != RCC_CFGR_SWS_HSI);

    // 2. Vypnout PLL
    RCC->CR &= ~RCC_CR_PLLON;
    while (RCC->CR & RCC_CR_PLLRDY);

    // 3. Změnit VOS (možné jen při vypnutém PLL!)
    PWR->CR = (PWR->CR & ~PWR_CR_VOS) | PWR_CR_VOS_1;  // Scale 2

    // 4. Znovu zapnout PLL a přepnout SYSCLK
    RCC->CR |= RCC_CR_PLLON;
    while (!(RCC->CR & RCC_CR_PLLRDY));
    RCC->CFGR = (RCC->CFGR & ~RCC_CFGR_SW) | RCC_CFGR_SW_PLL;
}
```

- Přechod trvá stovky µs (restart PLL)

Správa PLL

- `PLL` má největší spotřebu ze všech oscilátorů a hodinových zdrojů
- Spuštění `PLL` trvá relativně dlouho, typicky 100–500 µs
  - rychlost ovlivňuje rychlost VCO (pomalejší VCO → rychlejší lock time)
  - s rostoucí teplotou se snižuje stabilita oscilátoru
  - během této doby nemůže systém přejít do **Run režimu**.

Návrh pro úsporu:

1. **Základní taktování:** MCU držet co nejvíce na nízkofrekvenčních a nízkospotřebových zdrojích, jako jsou `MSI` (Multi-Speed Internal), `LSI` nebo `LSE` (32 kHz).
2. **PLL na vyžádání:** `PLL` zapínat jen krátce a pouze v případě potřeby vysokého výkonu (např. pro rychlé výpočty, komunikaci s vysokou propustností).
3. **Přepnutí zpět na nízká takt:** Ihned po ukončení kritické úlohy přepnou zpět na základní taktování.

MSI (Multi-Speed Internal)

- Je dostupný jen u ultra-low-power řad:
  - STM32L0, L1, L4, L4+, L5, U5, WL, WB
- Velmi úsporný RC oscilátor:
  - frekvence ~100 kHz až ~48 MHz
  - frekvence rozděleny do 12 rozsahů (mohou se lišit podle řady)
- Startovací doba je extrémně nízká: ~10 µs podle frekvence
- Často se používá jako:
  - hlavní clock po probuzení ze Stop režimu
  - clock pro low-power periferie

- zdroj, ke kterému se MCU přepíná při DVFS

```c
// Hodnoty pro RCC_CFGR (Clock Configuration Register)
#define RCC_CFGR_SW_HSI     0x00      // System Clock Switch to HSI
#define RCC_CFGR_SW_PLL     (0x2U)    // System Clock Switch to PLL
#define RCC_CFGR_SWS_HSI    0x00      // System Clock Switch Status HSI
#define RCC_CFGR_SWS_PLL    (0x8U)    // System Clock Switch Status PLL

// Bity pro RCC_CR (Clock Control Register)
#define RCC_CR_HSION        (1U << 0) // HSI Enable
#define RCC_CR_HSIRDY       (1U << 1) // HSI Ready Flag
#define RCC_CR_PLLON        (1U << 24) // PLL Enable
#define RCC_CR_PLLRDY       (1U << 25) // PLL Ready Flag
```

```c
void Switch_To_Low_Power_Clock(void) {
    // 1. Povolení HSI (pokud již není zapnuto)
    RCC->CR |= RCC_CR_HSION;
    while (!(RCC->CR & RCC_CR_HSIRDY));

    // 2. Přepnutí SYSCLK na HSI
    RCC->CFGR &= ~RCC_CFGR_SW_PLL;
    RCC->CFGR |= RCC_CFGR_SW_HSI;
    while ((RCC->CFGR & 0xCU) != RCC_CFGR_SWS_HSI);

    // 3. Vypnutí PLL (po úspěšném přepnutí)
    RCC->CR &= ~RCC_CR_PLLON;
}
```

```c
void Switch_To_High_Performance_Clock(void) {
    // 1. Zapnutí PLL
    RCC->CR |= RCC_CR_PLLON;
    while (!(RCC->CR & RCC_CR_PLLRDY));

    // 2. Přepnutí SYSCLK na PLL
    RCC->CFGR &= ~RCC_CFGR_SW_HSI;
    RCC->CFGR |= RCC_CFGR_SW_PLL;

    while ((RCC->CFGR & 0xCU) != RCC_CFGR_SWS_PLL);
}
```

2.4. Napájecí domény

Moderní MCU nejsou jednolitý obvod napájený jedním V_{DD}.

Aby se dosáhlo nízké spotřeby, výrobci jej rozdělují na oddělené napájecí domény, které lze nezávisle:

- vypínat
- zpomalovat
- přepojovat na jiný clock
- udržovat v retention módu → většina je čipu vypnutá, ale vybrané části paměti nebo registrů zůstávají pod napětím, aby se zachoval stav mezi probuzeními.

Toto rozdělení je jedním z hlavních faktorů umožňujících µA a nA režimy.

Základní napájecí domény

1. CORE / VCORE – procesor, ALU, pipeline, cache

2. PERIPHERAL / VDD – GPIO, časovače, sběrnice, PWM, ADC, SPI, I²C …
3. MEMORY / SRAM banks – jednotlivé bloky RAM, někdy se dá každá banka vypnout zvlášť
4. ANALOG – ADC, DAC, komparátory, LDO pro analogovou část
5. ALWAYS-ON (AON) – RTC, wakeup logika, backup doména
6. BACKUP – BKPSRAM, RTC registry, LSE oscilátor
7. RF power domain – u BLE/WiFi SoC (nRF52, ESP32)

Každá doména má vlastní clock, napájecí regulátor, přístupy pro vypnutí/reset a retention možnosti

CORE doména (VCORE) – hlavní zdroj dynamické spotřeby

Co obsahuje:

- CPU (Cortex-M)
- Flash interface & prefetch
- u některých MCU cache a matematické jednotky (FPU, DSP)

V režimech spánku:

- Sleep: CORE off
- Stop/Standby: VCORE off nebo přepnutá na low-power LDO
- Shutdown: VCORE úplně vypnuto
- nízká leakage, ale nutnost reinitializace při probuzení (kromě retention RAM)

PERIPHERAL doména

Co zahrnuje:

- GPIO
- SPI, I²C, USART, DMA
- časovače (TIMx) a watchdog

V režimech spánku:

- Sleep: periferie běží
- Stop: většina periferií zastaví clock
- Standby/Shutdown: vše vypnuto kromě AON

MEMORY domény (SRAM banks)

Moderní STM32L4/L5/U5 mají až 4 nezávislé RAM banky:

- Normal mode – plná rychlost
- Low-power mode – nižší frekvence, nižší spotřeba
- Retention – RAM zapnutá, jádro vypnuté
- Off – RAM bank vypnutá

Praktický přínos:

- u IoT zařízení v deep-sleep režimu může být polovina RAM vypnutá
- tím se ušetří až 30–40% standby spotřeby.

ANALOG doména

Zahrnuje:

- ADC, DAC, analogové komparátory
- referenční napětí (Vref)
- analogové LDO

Spotřeba:

- typicky 150–800 µA (ADC)
- 50–200 µA (komparátor)
- Vref buffer ~150 µA

ALWAYS-ON doména (AON)

Obsahuje:

- RTC, LSE, , některé low-power časovače (LPTIM)
- wakeup controller
- zálohovací registry
- často i ULP coprocessor (u ESP32)

Spotřeba:

- 200–900 nA (STM32L4/L5)
- ~1 µA nRF52
- 5–20 µA u ESP32 (kvůli ULP + vlastní LDO)

BACKUP doména

Týká se hlavně STM32:

- LSE 32kHz krystal
- RTC registry
- BKPSRAM (4–8 kB podle řady)

Napájeno z VBAT.

- Vydrží běžet i při VDD = 0.
- Spotřeba typicky 200–500 nA
- přímá závislost na LSE typu (krystal vs RC)

RF doména (u BLE/Wi-Fi SoC)

Zcela oddělená power doména:

- vysílač, přijímač
- PLL pro RF, AES/crypto akcelerátor
- vypnuté úplně mimo RF aktivitu
- probouzené pouze krátce pro vysílání slotu nebo scan

Spotřeba:

- BLE TX/RX: 3–15 mA
- Wi-Fi TX: 100–250 mA

3. Úsporné režimy

Základní filozofie řízení spotřeby

MCU přechází mezi režimy podle:

- aktuální aktivity systému
- požadované odezvy
- dostupných zdrojů probuzení

Platí obecné pravidlo: > Čím hlubší spánek, tím nižší spotřeba – ale vyšší cena probuzení

(cena = latence, ztráta kontextu, nutnost reinicializace)

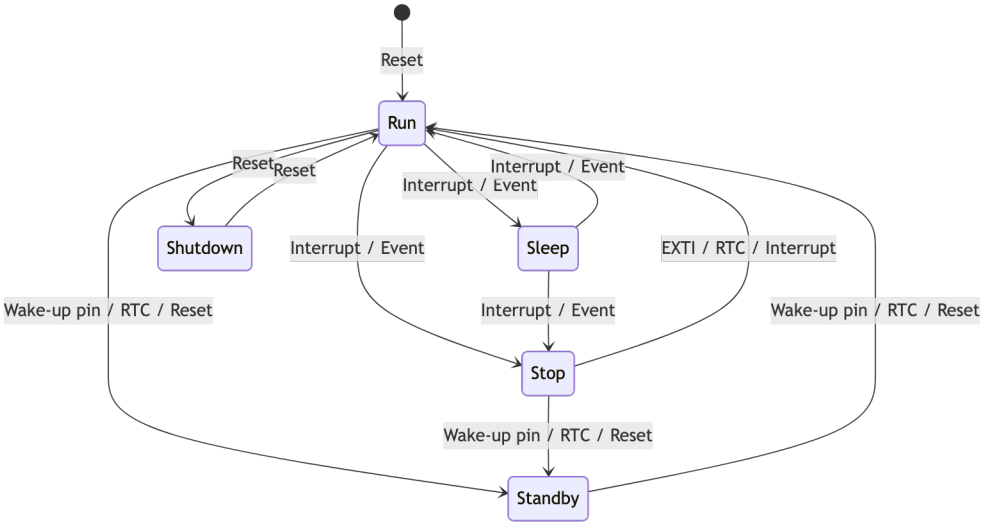

Figure 2: Režimy procesoru a možné přechody mezi nimi

Přehled registrů pro úsporné režimy (STM32)

SCB – řízení spánku jádra (Cortex-M)

System Control Block (SCB) → architektonický blok Cortex-M, stejný pro všechny STM32

Registr	Bit	Význam
`SCB->SCR`	`SLEEPDEEP`	Rozlišení Sleep vs. deep sleep (Stop / Standby / Shutdown)
`SCB->SCR`	`SEVONPEND`	Event při pending interrupt
`SCB->SCR`	`SLEEPONEXIT`	Automatický spánek po ISR

> Bez SLEEPDEEP nelze vstoupit do Stop / Standby / Shutdown.

PWR – řízení napájení a režimů

Power control (PWR) → klíčový blok pro všechny úsporné režimy

Registr	Typický obsah
`PWR_CRx`	Volba režimu (Stop / Standby / Shutdown)
`PWR_SR` / `PWR_SR1`	Wake-up flags
`PWR_WKUPCR`	Povolení wake-up pinů
`PWR_SCR`	Mazání wake-up příznaků

Typické bity:

- `PDDS` – Standby / Shutdown
- `LPDS` – Low-power regulator
- `CWUF` – clear wake-up flags

Aktivní režim (Run mode)

Defaultní režim po restartu MCU STM32

- CPU i periferie jsou plně aktivní
- Nejvyšší spotřeba energie
- Do úsporných režimů se přechází programem

Typické charakteristiky:

- běží aplikační kód
- aktivní přerušení (NVIC)
- zapnuté systémové hodiny (PLL, HCLK, PCLK)

Optimalizace i v Run módu:

- snížení frekvence
- vypínání nepoužívaných periferií
- přechod do Sleep / Stop při čekání

Ne každý reset znamená totéž:

- zapnutí napájení
- externí reset
- watchdog
- probuzení z hlubokého spánku

Reset flags umožňují softwaru zjistit příčinu resetu

- Typicky uloženy v registru `RCC_CSR` (nebo ekvivalent dle řady)

Režim spánku (Sleep Mode)

Nejméně hluboký režim spánku s nejvyšší spotřebou ze všech úsporných režimů.

> Použití pro krátké doby nečinnosti, kdy je potřeba, aby periferie zůstaly aktivní.

- Zastaví se hodiny CPU (jádro).
- Periferie a jejich hodiny zůstávají aktivní (pokud nejsou ručně deaktivovány).
- Zachování obsahu RAM a registrů.
- Probuzení je rychlé, program pokračuje od místa, kde se do režimu spánku vstoupilo.

U některých řad existuje i low-power Sleep mode, kde se vypne hlavní regulátor napětí.

Probuzení z režimu spánku

- Jakékoliv přerušení / událost z periferie (např. timer, USART, ADC) nebo externím přerušením (EXTI).

Režim zastavení (Stop Mode)

Nižší spotřeba než Sleep. > Vhodné pro periodické probouzení, delší nečinnost s požadavkem na rychlé probuzení.

- Zastaví se systémové hodiny (včetně většiny oscilátorů), tím se deaktivují všechny periferie s výjimkou těch, které mají nezávislý zdroj taktu (např. `LSI` / `LSE` pro `RTC`).
- Obsah SRAM a registrů je zachován.
- Regulátory napětí zůstávají aktivní, často v režimu nízké spotřeby.
- Probuzení externím přerušením (`EXTI`) nebo periferiemi s nezávislým zdrojem hodin.

Probuzení z režimu zastavení

- `RTC` časovač: hodiny reálného času, které běží na nezávislém oscilátoru (`LSE` nebo `LSI`).
- Některé periferie s nízkou spotřebou, u modernějších řad STM32 (např. STM32L) např.:
  - `LPUART` (Low-Power USART): Může detekovat startovní bit příchozího rámce a probudit systém.
  - `LPTIM` (Low-Power Timer): Může generovat přerušení.
- Méně časté je pro probuzení využití `IWDG` (Independent Watchdog).

Pohotovostní režim (Standby Mode)

U běžných řad MCU nejhlubší režim spánku s nejnižší spotřebou energie. > Maximální úspora energie, kde je možné tolerovat delší čas probuzení a ztrátu operačního stavu.

- Vypne se hlavní regulátor napětí a všechny hodiny a oscilátory.
- V chodu: Pouze záložní doména (Backup domain), včetně `RTC` a záložních registrů (pokud jsou povoleny).
- Obsah `RAM` a registrů je **ZTRACEN**, kromě záložní domény (Backup `SRAM` / `RTC` registry).
- Probouzení je dlouhé, ekvivalentní resetu (aplikace se musí znovu inicializovat).

Probuzení z pohotovostního režimu

- Wake-up Pin (`WKUP`): Dedikované externí piny (často označené `WKUP1` , `WKUP2` atd.) navázané přímo na záložní doménu. Jsou nejspolehlivějším způsobem, jak systém probudit externím signálem.
- `RTC` časovač: Stejně jako u Stop režimu, `RTC` zůstává napájeno a může generovat signál pro probuzení.
- `NRST` (Reset): Vyvolání vnějšího resetu pinu vždy probudí systém.
- `IWDG` (Independent Watchdog): Pokud dojde k vypršení Watchdog Timeru, dojde k resetu systému.

Retention mode

Retention není samostatný režim MCU

Je to vlastnost úsporného režimu. Určuje:

- které části systému zůstanou napájené
- jaký stav je zachován během spánku

Prakticky leží mezi:

- Stop (částečně zachovaný stav)
- Standby (žádný zachovaný stav → reset)

Umožňuje velmi nízkou spotřebu a návrat bez resetu

Retention mode dovolí uložit např.:

- malý blok `RAM` (např. 4–16 kB)
- `RTC` + jeho registry
- obsah zálohovaných domén (Backup registers)
- stav čítačů `LPTIM` / `LPUART` konfigurací
- hodnoty `GPIO` konfigurace
- kontext RTOS (stack + `TCB`) — u platformy, která to podporuje

STM32 (např. L4/L5, U5)

- Vypnuta většina domén, ale SRAM2 nebo „Backup RAM“ zůstává napájená.
- RTC běží dál.
- Probuzení proběhne přes wake-up pin/RTC/LPTIM, ale dochází k resetu, takže firmware si musí vzít data z retained RAM.

Nordic nRF52/nRF53

- Lze si vybrat které bloky RAM zůstanou pod napětím.
- MCU se probudí do resetu → stav lze vzít z retained RAM.

Silicon Labs EFM32

- Lze uchovat část nebo celou RAM.

ESP32

- Retenční paměť (RTC FAST/RTC SLOW) = malé RAM bloky běžící mimo hlavní napájecí doménu.
- Probuzení ≠ reset logiky, ale CPU začíná z ROM bootloaderu a přečte retained data.

Shutdown režim

Nejhlubší spací režim (STM32L0/L1/L4/L4+/L5, WL), extrémně nízká spotřeba ~20–50 nA.

Shutdown mode minimalizuje spotřebu za cenu úplné ztráty stavu a návratu přes reset.

- Co funguje
  - RTC (pokud má samostatnou Vbat), LSE jen pokud je v Backup doméně povolen

  - Wake-up piny (WKUP), IWDG (podle řady), záložní registry (Backup registers)
  - V některých řadách retention RAM (L4, L4+, L5, U5 → volitelné bloky)
- Co nefunguje
  - CPU a veškerá main SRAM, všechny periferie
  - Všechny napájecí domény kromě Backup/Vbat

Probuzení: jako reset, ale indikace je v `PWR_SR2.SHDUF`

Předhled některých zdrojů probuzení

RTC Wake-Up

- běží na LSE (32.768 kHz) nebo LSI (~32 kHz),
- dostupný ve všech hlubokých režimech včetně Standby/Shutdown,
- typická spotřeba: 300–600 nA + LSE oscilátor.

LPTIM (Low-Power Timer)

- běží i ve Stop režimu (z LSE nebo LSI), velmi nízká spotřeba (desítky nA).
- generuje události, přerušení, DMA požadavky,
- fungovat jako one-shot wake-up timer (lepší než RTC pokud není třeba datum/čas).

EXTI z GPIO

- náběžná nebo sestupná hrana,
- u hlubokých režimů jen úroveň signálu,
- pin musí být v režimu analog nebo pull-up/down → eliminace digitálních bufferů kvůli úspoře.

Analogové wake-up

Umí jen některé MCU (STM32L4+, STM32U5):

- Analog watchdog (ADC) může probouzet při překročení prahů,
- velmi úsporný režim — ADC běží na minimálním proudu.

UART wake-up

- Detekce start bitu → probuzení při příjmu znaku,
- Address-mark wake-up (pro multi-drop RS485).

I²C wake-up

- Umí zejména řady L0, L4, U5.
- Wake-up při shodě adresy nebo STOP condition.

Comparator wake-up

- Komparátor může běžet v ultra-low-power
- Typické použití: threshold detection, wake-up při překročení prahu světla, zvuku, vibrací atd.

Praktické okolnosti probouzení

Problémy a úskalí při probouzení

Chybné *zámky hodin* (clock gating)

- Některé periferie vyžadují zapnuté určité hodiny i pro wake-up detektor (např. UART RX → APB musí zůstat napájen).

Digitální piny v nevhodném režimu

- Vstup je *floating* → velký I/O leak → spotřeba o několik µA vyšší.

Race-conditions po probuzení

- Po probuzení běží ISR, ale hodiny periferií nejsou ještě připravené.

Latence probuzení

Typické hodnoty (pro STM32):

Režim	Latence	Poznámky
Sleep	~1–5 µs	žádná změna clocku
Stop (MSI)	50–200 µs	jen soft-start MSI
Stop (PLL/HSE)	0.5–2 ms	nutnost lock PLL
Standby	1–5 ms	cold boot, inicializace RAM
Shutdown	10+ ms	absolutní minimum funkcí

Wake-up time vs spotřeba energie

Základní trade-off

Hlubší režim spánku = nižší spotřeba, ale delší doba probuzení a vyšší energie na probuzení.

$$E_{total} = E_{sleep} + E_{wakeup} + E_{active}$$

Kdy se vyplatí hlubší spánek?

Pro výpočet break-even bodu:

$$T_{break-even} = \frac{E_{wakeup,deep} - E_{wakeup,shallow}}{P_{shallow} - P_{deep}}$$

Příklad: Stop vs Standby

- Stop: 1 µA, wake-up 200 µs @ 5 mA
- Standby: 100 nA, wake-up 3 ms @ 10 mA

→ Pro periody kratší než ~33 s je Stop efektivnější

Parametr	Stop	Standby
Sleep power	1 µA	100 nA
Wake energy	1 µJ	30 µJ
Break-even	—	~33 s

Optimalizace wake-up cyklu

Race-to-sleep strategie

1. Probudit se na nejvyšší frekvenci (minimalizace active time)
2. Rychle zpracovat úlohu
3. Okamžitě přejít do nejhlubšího možného režimu

Volba hodinového zdroje po probuzení

- **MSI** (~4 MHz): rychlý start (~10 µs), dostatečný pro většinu úloh
- **HSI** (16 MHz): střední rychlost (~2 µs), vyšší výkon
- **PLL**: pomalý start (~200 µs–2 ms), maximální výkon

Debugging low-power systémů

Problém: debugger vs spotřeba

- SWD/JTAG interface spotřebuje 100 µA – 10 mA
- Připojený debugger může blokovat vstup do deep sleep
- ST-Link/J-Link drží MCU v aktivním stavu

Řešení

1. Odpojení debuggeru pro měření

- Měřit spotřebu bez připojeného debuggeru
- Použít UART/RTT pro logging místo breakpointů

2. Low-power debug mode (některé MCU)

```
// STM32: povolení debug v Stop režimu
DBGMCU->CR |= DBGMCU_CR_DBG_STOP;
// Pozor: zvyšuje spotřebu o ~100 µA
```

3. GPIO toggling pro timing

```
// Měření doby probuzení pomocí GPIO
void WakeupHandler(void) {
    GPIO_SET(DEBUG_PIN);   // Oscilloscope trigger
    DoWork();
    GPIO_CLEAR(DEBUG_PIN);
    EnterSleep();
}
```

Typické problémy a řešení

Symptom	Příčina	Řešení
Spotřeba v sleep příliš vysoká	Floating GPIO	Nastavit jako analog/pull
MCU se neprobouzí	Špatná EXTI konfigurace	Zkontrolovat edge/level
Spotřeba kolísá	Periferní clock běží	Vypnout nepoužívané clocks
Vysoká špička při wake-up	PLL startup	Použít MSI místo PLL

Práce s úspornými režimy

- **WFI (Wait For Interrupt):** Jádro přejde do režimu spánku a probudí se jakýmkoliv aktivním přerušením.
- **WFE (Wait For Event):** Jádro přejde do režimu spánku a probudí se buď přerušením, nebo událostí (některé periferie, jako `DMA`).

Před použitím těchto instrukcí je třeba nastavit řídicí registr `SCR` (System Control Register)

SCR	Funkce	Režim
SLEEPONEXIT	Po návratu z ISR se nevracet do hlavního kódu, ale okamžitě znovu uspat	Sleep/Stop
SLEEPDEEP	0 = Použít instrukci jako Sleep Mode.	Sleep
SLEEPDEEP	1 = Použít instrukci jako Stop/Standby Mode (závisí na PWR registrech).	Stop/Standby

Režim spánku

Nejmělčí režim. Stačí vynulovat bit SLEEPDEEP a použít instrukci WFI.

```c
// Nastaví bit SLEEPDEEP v registru SCR (System Control Register)
#define SET_SLEEPDEEP()     (SCB->SCR |= SCB_SCR_SLEEPDEEP_Msk)
#define CLEAR_SLEEPDEEP()   (SCB->SCR &= ~SCB_SCR_SLEEPDEEP_Msk)

void Enter_Sleep_Mode(void) {
    // 1. Zajištění, že SLEEPDEEP bit je vynulován pro Sleep Mode
    CLEAR_SLEEPDEEP();

    // 2. Vstup do Sleep Mode (čekání na jakékoli přerušení)
    // Periferie zůstávají aktivní.
    __WFI();

    // Po probuzení pokračuje kód zde...
}
```

Režim zastavení

Stop Mode vyžaduje nastavení bitu `SLEEPDEEP` a konfiguraci periferie `PWR` (Power Control), konkrétně:

- Nastavit bit `SLEEPDEEP` v registru `SCR`.
- Vymazat bit `PDDS` (Power Down Deep Sleep) v registru `PWR_CR`.
- Konfigurace regulátoru napětí pro nízkou spotřebu (např. Stop Mode regulátor).

```c
// Konkrétní bitové masky pro PWR se mohou lišit!
#define PWR_CR_PDDS_Pos     (3U)
#define PWR_CR_PDDS_Msk     (0x1U << PWR_CR_PDDS_Pos)
// Low-power deep sleep (nebo ekvivalentní pro regulátor)
#define PWR_CR_LPDS_Msk     (0x2U)
// 1. Povolení hodin periferie PWR (typicky v registru RCC->APB1ENR)
RCC->APB1ENR |= RCC_APB1ENR_PWREN;

// 2. Konfigurace PWR_CR pro Stop Mode:
// a) Vynulování bitu PDDS (Power Down Deep Sleep) -> říkáme, že NEchceme Standby
PWR->CR &= ~PWR_CR_PDDS_Msk;
// b) Nastavení napěťového regulátoru do Low-power režimu (podle rodiny)
```

```
PWR->CR |= PWR_CR_LPDS_Msk;

// 3. Nastavení bitu SLEEPDEEP v SCR pro aktivaci Deep Sleep
SET_SLEEPDEEP();

// 4. Vstup do Stop Mode (čekání na přerušení, např. z EXTI nebo RTC)
__WFI();

// 5. Po probuzení: Vynulování SLEEPDEEP (pro jistotu) a rekonfigurace hodin.
CLEAR_SLEEPDEEP();

// Po Stop režimu restart jádra (PLL + reinit RCC)
```

Pohotovostní režim

Nejhlubší režim, probuzení znamená hardwarový reset.

Vyžaduje:

- nastavení bitu `SLEEPDEEP`
- nastavení bitu `PDDS` (Power Down Deep Sleep) v `PWR_CR`.

```
// Konkrétní bitové masky pro PWR se mohou lišit!
#define PWR_CR_PDDS_Pos     (3U)
#define PWR_CR_PDDS_Msk     (0x1U << PWR_CR_PDDS_Pos)
#define PWR_CSR_WUF_Msk     (0x1U) // Wakeup Flag
```

Pokud je dostupný shutdown režim:

- `PDDS=0`, `LPSDSR = 1` (Low-Power Shutdown)

```
// 1. Povolení hodin periferie PWR
RCC->APB1ENR |= RCC_APB1ENR_PWREN;

// 2. Vyčištění Wakeup Flagu (WUF) – nutné pro některé řady před vstupem!
// PWR->CR |= PWR_CR_CWUF; // Clear Wakeup Flag

// 3. Konfigurace PWR_CR pro Standby Mode:
// a) Nastavení bitu PDDS (Power Down Deep Sleep) -> říkáme, že chceme Standby
PWR->CR |= PWR_CR_PDDS_Msk;

// b) Nastavení bitu SLEEPDEEP v SCR
SET_SLEEPDEEP();

// 4. Vstup do Standby Mode (čekání na externí WKUP pin nebo RTC)
// Instrukce WFI vypne hlavní regulátor.
__WFI();

// 5. Kód ZDE NEPOKRAČUJE!
// Probuzení způsobí Reset, provádění se vrátí na začátek main()
```

Shutdown režim

Dostupný jen u některých low-power variant

```
// Uvedení do módu
PWR->CR1 |= PWR_CR1_LPSDSR; // Shutdown
```

```
PWR->CR1 |= PWR_CR1_ULP;    // Ultra-low-power
SCB->SCR |= SCB_SCR_SLEEPDEEP_Msk;
__WFI();

// Ověření pro probuzení
uint32_t csr = PWR->CSR1;
if (csr & PWR_CSR1_WUF1) { /* Wake-up from WKUP1 pin */ }
PWR->SCR |= PWR_SCR_CWUF;  // Clear flags

// Zjištění resetu
if (RCC->CSR & RCC_CSR_SBF) { /* Probuzení ze Standby/Shutdown */ }
RCC->CSR |= RCC_CSR_RMVF;  // Clear reset flags
```

4. Návrhové vzory s nízkou spotřebou

1. Event-driven (událostmi řízené) programování

Princip: `CPU` je většinu času ve spánku. Probouzí ho pouze konkrétní událost (`EXTI` , `LPTIM` , `ADC` event, `DMA` complete). Veškerá logika je napsaná jako stavový automat: událost → přechod → akce → návrat do spánku.

Implementace — kroky

1. Identifikuj události, které skutečně vyžadují `CPU` (ne všechno musí).
2. Používej `EXTI` / `LPTIM` / `RTC` pro wake-up místo pollingových smyček.
3. Minimalizuj práci v `ISR` (jen signalizuj stav a přesuň práci do úlohy).
4. Po dokončení akce použij race-to-sleep: dokonči práci co nejrychleji a vrať se do nejhlubšího možného režimu.
5. V RTOS: implementuj idle hook s `WFI` / `WFE` nebo tickless režim.

Pseudokód

```
while (1) {
  enter_deep_sleep(); // WFI / WFE
  // probudí EXTI / LPTIM
  flag = read_wakeup_source();
  switch(flag) {
    case SENSOR_TRIGGER:
      sample_or_enable_peripheral();
      process_sample();
      disable_peripheral();
      break;
    case RTC_WAKE:
      sync_time();
      break;
  }
}
```

2. DMA offload (přesun práce na DMA)

Princip: Zdroj dat (`ADC` , `UART RX` , `SPI`) zpracovává `DMA` místo `CPU` .

- `CPU` se probudí pouze na významnou událost (např. naplnění bufferu), nebo vůbec (přímé zpracování v periferii).

Implementace — kroky

1. Konfiguruj periférii `ADC` / `SPI` / `UART` v režimu `DMA` (circular nebo double-buffer).
2. Nastav `DMA` na přenos přímo do `RAM` (aligned buffer).
3. Použij half-transfer (`HT`) a transfer-complete (`TC`) přerušení jen pokud potřebuješ `CPU` . Jinak zpracuj data přenosem `DMA` → periferie.
4. Po dokončení přenosu procesuj data rychle a vrať se do spánku.

ADC + DMA (double buffer)

- `ADC` → `DMA` (circular, double buffer) → `RAM` bank A/B
- `DMA` `TC` / `HT` event → set flag (ne vždy probouzet)
- Hlavní smyčka: zpracuj blok → možná transmit → sleep

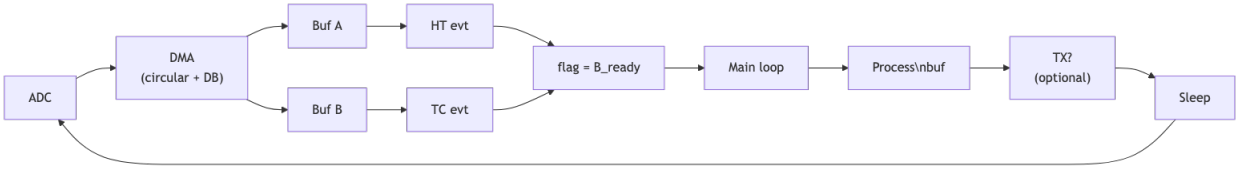

Figure 3: Ideový návrh aplikace

Pseudokód (ADC sampling bez častého buzení)

```
configure_ADC_DMA_circular(bufA, bufB);
start_ADC_DMA();
enter_low_power();
on_DMA_half_transfer() {
  // jen označit buffer k zpracování bez těžké práce
  set_flag(half);
}
on_DMA_full_transfer() {
  set_flag(full);
}
main_loop() {
  if(flag) {
    process_buffer();
    clear_flag();
  }
  enter_low_power();
}
```

Výhody

- `CPU` může být v úsporném režimu i při vysoké frekvenci měření.
- Menší jitter, víc deterministické vstupy.

Úskalí

- `DMA` buffer alignment / cache coherence
  - na `MCU` s D-cache je třeba invalidovat/čistit cache.
- `DMA` adresování a bezpečné sdílení bufferu mezi `ISR` a hlavní smyčkou.
- Některé periferie neumějí `DMA` v deep stop (ověřit v datasheetu).

3. Power-gating senzorů a externí logika

Princip: Senzory a externí moduly napájej odděleně a zapni je jen pro měření / `TX`.

Implementace — kroky

1. Použij `P-FET` / `N-MOS` s `EN` pinem řízeným `GPIO`
2. Pro citlivé senzory vyřeš wake-up: senzor vyžaduje stabilizaci typicky 1–10 ms
    - čas stabilizace se často podceňuje → zbytečné opakované měření.

Tipy

- pokud senzor vyžaduje pull-up (I^2C) → místo trvalých pull-up použít vnitřní.
- používat tranzistory s ultra-low únikem pro `VBAT` oblasti.

Příklad návrhu low-power senzoru

Úloha

- Periodicky (např. každých 15 minut) číst data ze senzoru,
- lokálně filtrovat/rozhodovat, případně poslat data → jen když je to nutné,
- jinak co nejnižší průměrná spotřeba.

Požadavky

- Životnost na baterii ≥ 1 rok (CR2032) nebo zcela autonomní provoz s malým solárním panelem + superkapacitou.

Omezení

- Malé rozměry, nízká cena, spolehlivost v teplotním rozsahu.

Architektura systému

Energetický subsystém

- primární napájení: CR2032 (≈ 200–240 mAh při nízkém odběru) nebo Li-ion (např. 300–1000 mAh)
- volitelně: solární panel + boost + supercap / malý Li-ion charger + MPPT / load switch
- power-gating (load switch) pro senzory a RF

Regulace napětí

- buck/boost nebo LDO podle baterie a cílového VCORE (např. 1.8–3.3 V)
- PMIC pokud potřebné (Power Management Integrated Circuit)

MCU

- low-power MCU s RAM retention, LPTIM/RTC, DMA, event interconnect (PPI/EXTI)
- příklady: STM32L4/L5/U5, nRF52/53, MCU s ULP copro (ESP32-ULP / Ambiq Apollo)

Senzory

- analogové (ADC) nebo digitální (I^2C/SPI) senzory s low-power sleep mode
- pokud senzory nemají low-power standby → power-gate

Komunikace

- nízkoenergetické rádiové rozhraní (BLE, LoRaWAN, NB-IoT) s power-gate a řízeným TX duty-cycle

Další rozšíření: flash pro záznamy (log) a proudový sense pin pro profilování.

Režimy provozu a stavový automat

1. **Deep sleep** — většina systému vypnuta, `RTC` / `LPTIM` běží, AON retention.
2. **Wake** – inicializace senzoru, vyčkáí na stabilizaci.
3. **Sample** — `ADC` + `DMA` + auton. periferie sbírají vzorky.
4. **Process** — rychlé zpracování (filter, threshold) — race-to-sleep.
5. **TX (volitelné)** — pokud podmínky splněny, zapnout RF, vyslat a vypnout.
6. **Maintenance** — občasná údržba (OTA check, RTC sync, flash GC).

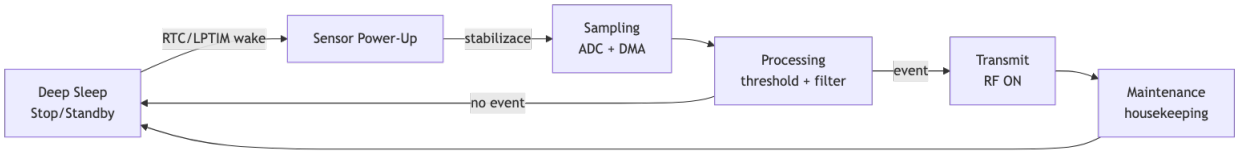

Figure 4: Stavový automat

Klíčové HW vzory a komponenty

1. Power gating
   - load switch (P-MOS nebo dedikovaný integrovaný obvod) řízený GPIO; velmi nízký únik ve vypnutém stavu.
2. LDO regulátor s nízkým klidovým proudem
   - pokud LDO, vybírej s nízkým IQ (<1 µA) nebo buck-boost měnič s ultra nízkým klidovým proudem.
3. MPPT / energy harvester front-end
   - pro solární: jednoduchý MPPT nebo boost charger; pro malé aplikace i diode + supercap + shunt-based MPPT jednoduché řešení.
4. Supercap pro buffering
   - pokud harvesting → supercap (např. 0.1–1 F) na stabilizaci krátkých TX (špičky).

SW architektura a návrhové vzory

1. Event-driven firmware / tickless RTOS
   - Idle → `WFI` / `WFE`; `LPTIM` / `RTC` jako systémový timer;
   - používá se tickless FreeRTOS nebo no-RTOS konečný automat.
2. Race-to-sleep
   - vykonej práci co nejrychleji na vyšším výkonu → vrať se do deep sleep.
3. `DMA` & autonomous peripherals
   - `ADC` → `DMA` s kruhovým bufferem;

- `ADC` sampling bez `CPU`;
- `DMA` `TC` / `HT` flagy jen pro blokové zpracování.

4. Power-aware ovladače
   - ovladač umí *suspend* / *resume* a minimalizovat režii na inicializaci.
5. DVFS
   - pokud `MCU` podporuje `DVFS`, lze v případě potřeby krátce zvýšit `VCORE` a pak snížit po dokončení operace
6. Watchdog a fail-safe
   - `IWDG` s vhodným timeoutem → při kritické chybě reset a safe mode.

Detailní energetický rozpočet

Perioda měření: $T = 15\,\text{minut} = 900\,\text{s}$.

Časy a proudy během cyklu:

- start senzoru: $0.05\,\text{s}$ při $5.0\,\text{mA}$ (power-gate + stabilizace)
- vzorkování: $1.00\,\text{s}$ při $10.0\,\text{mA}$ (ADC + MCU active)
- zpracování: $0.10\,\text{s}$ s při $5.0\,\text{mA}$ (compute)
- přenos (pokud potřeba):
  - $0.05\,\text{s}$ při $10.0\,\text{mA}$ — uvažujeme, že TX se děje jen pokud threshold splněn;
  - pro jednoduchost: 10% cyklů vedou k TX
- režim spánku: zbytek periody při $1.0\,\mu\text{A}$

Krok 1 — součet aktivních intervalů (bez TX)

Aktivní intervaly (bez TX):

- startup: $0.05\,\text{s} \times 5.0\,\text{mA} = 0.25\,\text{mA·s}$
- sample: $1.00\,\text{s} \times 10.0\,\text{mA} = 10.00\,\text{mA·s}$
- process: $0.10\,\text{s} \times 5.0\,\text{mA} = 0.50\,\text{mA·s}$

Celkový proud v aktivním itervalu → $10.75\,\text{mA·s}$

Aktivní čas → $1.15\,\text{s}$, čas spánku → $900 - 1.15 = 898.85\,\text{s}$

Proud v úsporném režimu → $1.0\,\mu\text{A} \times 898.85\,\text{s}$

Celkový mA·s na cyklus (bez TX) = $10.75 + 0.89885 = 11.64885\,\text{mA·s}$

Průměrný proud: $I_{avg} = \frac{11.64885}{900}\,\text{mA} = 12.943166\,\mu\text{A}$

Krok 2 — přidání TX v 10 % případů

TX energie: $0.05\,\text{s} \times 10.0\,\text{mA} = 0.50\,\text{mA·s}$ → z toho 10%

Celkový proud s příspěvkem TX: $I_{avg} \approx 13.00\,\mu\text{A}$

Krok 3 — životnost na CR2032 (při 220 mAh)

Efektivní kapacita CR2032 je ≈ 220 mAh.

Životnost baterie (v hodinách)

$$t = \frac{C_{mAh}}{I_{mA}} = \frac{220}{0.012998722} \approx 16924.9\,\text{h}$$

Závěr:

- při uvedených parametrech ≈ ~1.9 roku na jednu CR2032 (220 mAh).
- výsledné číslo závisí na teplotě a skutečné kapacitě baterie.

Energy harvesting scénář (solární + supercap)

Cílové I_{avg} z předchozího příkladu ~ 13 µA při 3.3 V.

Spotřeba:

$$P_{avg} = 13\mu A \times 3.3V = 42.9\mu W$$

- Indoor solar typicky dává ~1–50 µW/cm² (silně závisí na osvětlení).
- Outdoor (slunečno) může generovat stovky až tisíce µW/cm².

Praktický návrh:

- malý solární panel 4–10 cm² za dobrého osvětlení může dodávat stovky µW až několik mW — dostatečné pro udržení 43 µW systému.
- supercap (např. 0.1–1 F) slouží jako buffer pro TX peaky (cca 10–100 mA krátkych ms).

Energie superkapacitoru:

$$E = C \times V^2$$

$$V = 3.3\,\text{V}, C = 0.1\,\text{F} \rightarrow E = 0.5445\,\text{J}$$

TX spotřebuje např. 10 mA × 0.05 s = 0.0005 A·s = 0.00165 J při 3.3 V

→ supercap 0.1 F má dostatečnou energii na stovky takových vysílání.

Závěr:

- S panely o 1 mW průměrného výkonu je harvesting možný (1 mW » 43 µW spotřeba)
- Indoor harvesting je náročnější, ale stále dosažitelný se správným panelem a solárním regulátorem (MPPT).

5. FreeRTOS a jeho použití v low-power systémech

Co je FreeRTOS (a co není)

- minimální RTOS jádro pro malé MCU (Cortex-M, RISC-V, AVR, ...)
- deterministický scheduler (prioritní, preemptivní)
- sada synchronizačních mechanismů: semafory, mutexy, fronty, notifikace
- extrémně modulární – může běžet už od kilobajtů RAM

Není to:

- technologie „všechno v jedné krabici" jako Zephyr nebo ThreadX
- framework nebo HAL
- systém s vlastními drivery (vše si píšeš nebo používáš HAL od výrobce `MCU`)

Pro low-power aplikace je FreeRTOS velmi vhodný → kontrola nad idle hookem a má tickless režim.

Základní prvky FreeRTOS

1. Tasky (úlohy)

```
void vSensorTask(void *arg)
{
    for(;;) {
        ReadSensor();
        vTaskDelay(pdMS_TO_TICKS(500));
    }
}

xTaskCreate(vSensorTask, "sens", 256, NULL, 2, NULL);
```

→ Úloha běží s danou prioritou a může se blokovat (`vTaskDelay`) → usnadňuje low-power řízení cyklů.

2. Direct-To-Task Notifications

Nejrychlejší mechanismus v FreeRTOSu (rychlejší než semafor / fronta).

Perfektní pro low-power: minimální overhead, budí jen konkrétní úlohu.

ISR:

```
vTaskNotifyGiveFromISR(taskHandle, &xHPW);
```

Task:

```
ulTaskNotifyTake(pdTRUE, portMAX_DELAY);
```

3. Fronty (queues)

Používají se, když se předávají data nebo eventy více spotřebitelům.

```
struct Packet pkt;
xQueueSend(q, &pkt, portMAX_DELAY);
```

Fronty mají větší overhead než notifikace → méně vhodné pro extrémní low-power.

4. Semafory

Hodí se pro synchronizaci s `ISR` (např. signály z `DMA`, `GPIO`).

```
xSemaphoreTake(xSem, portMAX_DELAY);
```

Opět: větší overhead než notifikace, ale jasnější sémantika.

Kompletní příklad - Plánování úloh probouzených z UART ISR

ISR přijme byte z UARTu

- uloží byte do bufferu
- dá notifikaci úloze
- provede `portYIELD_FROM_ISR` pokud byla odblokovaná úloha vyšší priority

Úloha zpracuje data a zase se zablokuje

→ ideální příklad event-driven architektury

UART ISR → notify → Task

```
void USART2_IRQHandler(void)
{
    uint8_t byte;
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    /* Přečteme data z DR přímo nebo použijeme HAL */
    if (LL_USART_IsActiveFlag_RXNE(USART2)) {
        byte = LL_USART_ReceiveData8(USART2);
        rxBuffer[rxWriteIndex++] = byte;

        /* Notifikace úloze, že jsou data k dispozici */
        vTaskNotifyGiveFromISR(uartTaskHandle, &xHigherPriorityTaskWoken);
    }

    /* Pokud se probudila úloha s vyšší prioritou → přepnout! */
    portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}
```

Úloha: zpracování UART dat

```
void vUartTask(void *pvParameters)
{
    for (;;)
    {
        /* Úloha čeká, dokud ISR neposílá notify */
        ulTaskNotifyTake(pdTRUE, portMAX_DELAY);
        /* pdTRUE = automaticky resetuje počítadlo notifikací */

        /* Nyní má CPU. Zpracujeme přijatá data */
        while (rxReadIndex != rxWriteIndex)
        {
            uint8_t b = rxBuffer[rxReadIndex++];
            processByte(b);
        }
```

```
15
16          /* Po dokončení se vrátíme do blokovaného stavu a scheduler pustí jiné úlohy */
17      }
18  }
19  xTaskCreate(
20      vUartTask,
21      "UART",
22      256,
23      NULL,
24      3,              // vysoká priorita – reakce na data z IRQ
25      &uartTaskHandle
26  );
27
28  xTaskCreate(
29      vBackgroundTask,
30      "BG",
31      256,
32      NULL,
33      1,              // nižší priorita
34      NULL
35  );
36
37  vTaskStartScheduler();
```

Co tenhle příklad demonstruje

1. Plánování podle priority
   - Jakmile přijde byte → `ISR` probudí `UART` Task → ten má vysokou prioritu → scheduler ho okamžitě přepne.
2. Zablokování úlohy
   - Úloha nebere `CPU`, pokud nemá data → úplně minimální spotřeba `CPU` i energie.
3. Ideální pattern pro low-power
   - Když nic nepřichází na `UART`, `MCU` může usnout v tickless režimu.
4. Event-driven architektura
   - Žádný polling, žádné busy-waiting.

Základní architektura low-power aplikací

```
+-----------------------------+
| High Priority Tasks         |
|  • měření, DMA, zpracování  |
+-----------------------------+
| Mid Priority Tasks          |
|  • komunikace, logování     |
+-----------------------------+
| Low Priority Tasks          |
|  • údržbové operace         |
+-----------------------------+
| Idle Task                   |
|  • vstup do sleep/deep-sleep|
+-----------------------------+
```

Když jsou všechny tasky blokované → Idle → `MCU` spánek.

Příklady úloh vhodných pro low-power aplikace

1. Periodická úloha

```c
void vBlinkTask(void *arg)
{
    for(;;) {
        ToggleLED();
        vTaskDelay(pdMS_TO_TICKS(1000));
    }
}
```

- `vTaskDelay()` blokuje task → scheduler může jít spát

2. Čtení senzoru každé 2 sekundy (RTC event → task)

Lepší než `vTaskDelay()`, protože umožňuje deep-sleep:

```c
void vSensorTask(void *arg)
{
    for(;;) {
        ulTaskNotifyTake(pdTRUE, portMAX_DELAY);
        float t = ReadTemperature();
        Send(t);
    }
}
```

ISR:

```c
// RTC alarm → notify sensor task
vTaskNotifyGiveFromISR(sensorTaskHandle, ...);
// sensorTaskHandle - ID úlohy, který přiřadí scheduler
```

Co se děje?

1. Task čeká (blokuje se) na notifikaci, která přijde z `ISR`.
2. Scheduler vidí, že nikdo nic nechce dělat → přepne se do Idle tasku.
3. Idle task vstoupí do `WFI()` → `MCU` je v hlubokém sleepu.
4. Po chvíli `RTC` alarm → probudí `MCU` → vyvolá `ISR`.
5. `ISR` pošle direct-to-task notification do `vSensorTask`.
6. FreeRTOS probudí task a ten:
   - přečte senzor,
   - pošle data,
   - a opět se blokuje.

3. Synchronizace více úloh (Queue) - sběr dat z více zdrojů

```c
typedef struct {
    uint8_t src;
    uint16_t value;
} Sample;
```

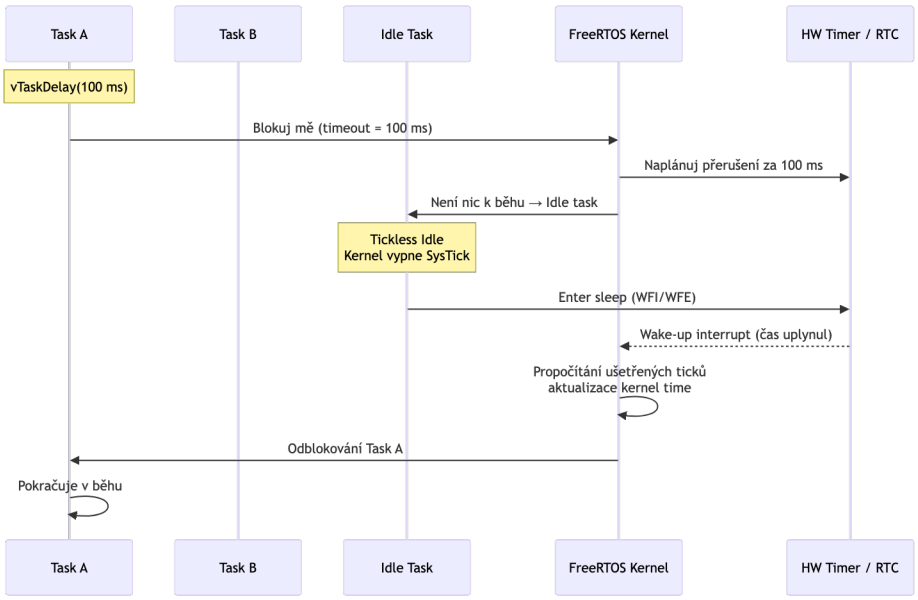

Figure 5: Low-power aplikace ve FreeRTOS

```
5
6  void vProducerTask(void *arg) {
7      Sample s = { .src = 1, .value = ReadSensor() };
8      xQueueSend(xQueue, &s, 0);
9  }
10
11 void vConsumerTask(void *arg) {
12     Sample s;
13     for(;;) {
14         xQueueReceive(xQueue, &s, portMAX_DELAY);
15         ProcessSample(&s);
16     }
17 }
```

Aktivace tickless režimu ve FreeRTOS

V FreeRTOSConfig.h stačí:

```
1  #define configUSE_TICKLESS_IDLE        1
2  #define configEXPECTED_IDLE_TIME_BEFORE_SLEEP  5  // minimální počet ticků
```

Tím RTOS přepne do volání hooku:

```
1  void vPortSuppressTicksAndSleep(TickType_t xExpectedIdleTime);
```

Tuto funkci lze použít pro:

- přípravu `MCU` do low-power režimu,
- nastavení wake-up timer (`RTC` / `LPTIM`),
- uspání jádra a po probuzení obnovení `SysTick` a času RTOS.

Přechod do STOP módu v tickless režimu

Idle hook:

```
void vApplicationIdleHook(void)
{
    __WFI();   // pokud nechceš deep sleep, jen jednoduchý WFI
}
```

Pokročilejší deep-sleep integrace (STM32 příklad):

```
void vPortSuppressTicksAndSleep(TickType_t xExpectedIdleTime)
{
    /* OK Zastavit SysTick */
    portNVIC_SYSTICK_CTRL_REG &= ~portNVIC_SYSTICK_ENABLE_BIT;

    /* OK Nastavit RTC wake-up za (xExpectedIdleTime * tick) */
    Setup_RTC_Wakeup(xExpectedIdleTime);
    /* OK Příprava na STOP */
    PreSleepProcessing();

     /* OK Režim STOP – probudí jen RTC/EXTI */
    HAL_PWR_EnterSTOPMode(PWR_MAINREGULATOR_ON, PWR_STOPENTRY_WFI);

    /* OK Po probuzení: obnovit takty a SysTick */
    PostSleepProcessing();

    portNVIC_SYSTICK_CTRL_REG |= portNVIC_SYSTICK_ENABLE_BIT;
}
```

Co je potřeba doplnit:

- `PreSleepProcessing()` = vypnout nepotřebné periferie / ztlumit I/O
- `PostSleepProcessing()` = znovu zapnout hodiny, `PLL`, periferie

FreeRTOS sám spočítá, na kolik hodinových taktů bude uspán.

FreeRTOS + Watchdog (IWDG) jako periodický wake-up

Watchdog `ISR` → notify tasku:

```
void IWDG_IRQHandler(void)
{
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;
    vTaskNotifyGiveFromISR(hwdTaskHandle, &xHigherPriorityTaskWoken);
    portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}
```

WD-task v RTOS:

```
void vWatchdogTask(void *arg)
{
    for(;;)
    {
        ulTaskNotifyTake(pdTRUE, portMAX_DELAY);  // čeká na WD interrupt
        /* Refresh watchdog */
        IWDG->KR = 0xAAAA;

```

```
        /* Krátký “housekeeping“ */
        CheckBatteryVoltage();
        CheckFlags();

        /* pak rovnou zpět do spánku */
    }
}
```

V tickless režimu se tento task probudí jen když `WD` expiruje.

FreeRTOS + DMA+ADC (double-buffer) → low-power pipeline

Toto je velmi běžný low-power pattern:

- `ADC` běží v low-power režimu (např. oversampling nebo `LPADC`).
- `DMA` posílá půlku / celý buffer do `RAM`.
- RTOS task se budí jen při `HT` / `TC` události.
- Po zpracování bloků se `CPU` opět uspí.

DMA ISR:

```
void DMA1_Channel1_IRQHandler(void)
{
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;
    if (DMA1->ISR & DMA_ISR_HTIF1) {
        DMA1->IFCR = DMA_IFCR_CHTIF1;
        vTaskNotifyGiveFromISR(adcTaskHandle, &xHigherPriorityTaskWoken);
    }

    if (DMA1->ISR & DMA_ISR_TCIF1) {
        DMA1->IFCR = DMA_IFCR_CTCIF1;
        vTaskNotifyGiveFromISR(adcTaskHandle, &xHigherPriorityTaskWoken);
    }

    portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}
```

ADC-processing task:

```
void vAdcTask(void *arg)
{
    for(;;)
    {
        ulTaskNotifyTake(pdTRUE, portMAX_DELAY);

        uint16_t *block = (dma_half == 0) ? adcBufferA : adcBufferB;

        ProcessAdcBlock(block, BLOCK_SIZE);

        /* Po zpracování se scheduler vrátí do idle → deep sleep */
    }
}
```

Efekt:

- žádné pravidelné tick-based probouzení

- `CPU` zapíná jen při přenosu `DMA` → `RAM`, což je extrémně efektivní