Mikroprocesory

4. CPU Cortex-M - mapa paměti, vnitřní periferie s sběrnice

Stanislav Vítek Katedra radioelektroniky České vysoké učení technické v Praze

Obsah přednášky

1. CPU Cortex-M4
2. Architektura sběrnic (AMBA)
3. Systém přerušení (NVIC)
4. GPIO a přerušení (EXTI)

1. CPU Cortex-M

Architektura Cortex-M4

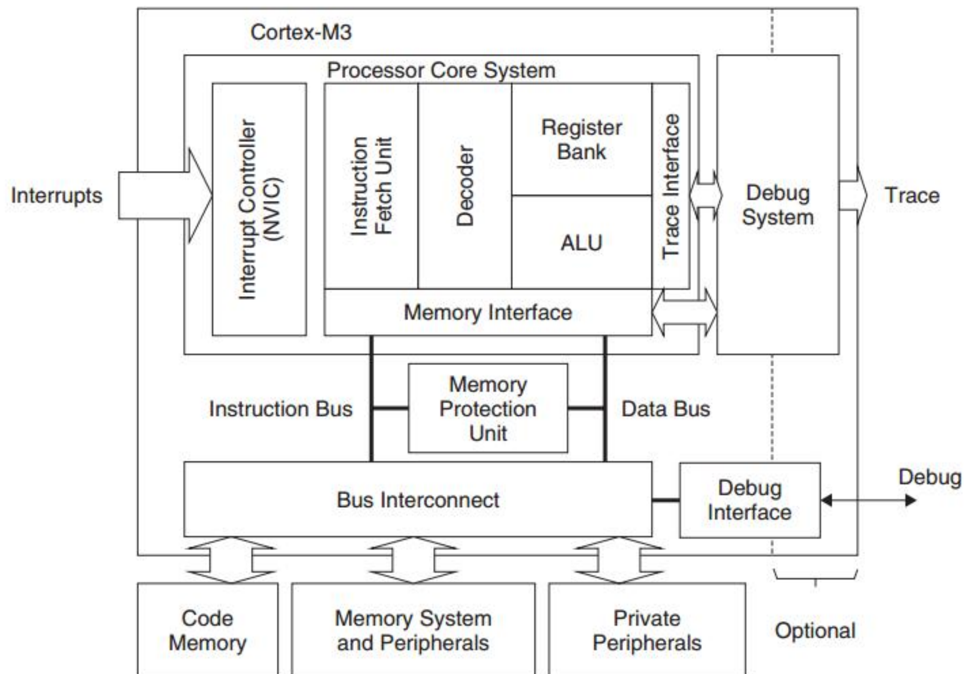


Figure 1: Zjednodušená architektura jádra Cortex-M

CPU Core periferie (Private Peripheral Bus):

- **NVIC** (Nested Vectored Interrupt Controller) – správa přerušení
- **SysTick** – 24-bit systémový časovač (RTOS tick)
- **MPU** (Memory Protection Unit) – ochrana paměti
- **FPU** (Floating Point Unit) – hardware podpora IEEE-754
- **DWT** (Data Watchpoint and Trace) – debugging
- **ITM** (Instrumentation Trace) – printf přes SWO

CPU periferie jsou mapovány v System oblasti (0xE000_0000)

Jak procesor startuje?

Po připojení napájení a uvolnění resetu:

1. **Inicializace hodin**
    - Obvykle startuje z interního RC oscilátoru
    - Později lze přepnout na HSE nebo PLL.

2. **Načtení Stack Pointeru (SP)**
   - Z adresy `0x0000_0000`
     - vektory jsou remapované podle boot módu
   - SP ukazuje na konec RAM (např. `0x2002_0000`)
3. **Načtení Reset Handler adresy**
   - Z adresy `0x0000_0004`
   - Skok na Reset_Handler
4. **Reset_Handler provede**
   - Kopírování `.data` z Flash do RAM,
   - vynulování `.bss`
   - inicializace systému (hodiny, FPU, cache)
   - volání `main()`.

Konfigurace CPU při startu a běhu, power management, diagnostika jsou ovlivněny registry sdruženými v **SCB (System Control Block)** na adrese `0xE000_ED00` (součást System Control Space).

Přesměrování paměti při startu (Boot modes)

Cortex-M4 může startovat z různých oblastí podle konfigurace pinů BOOT0 (B0) /BOOT1 (B1).

Remapping: Oblast 0x0000_0000 je alias podle boot módu (Flash/System/SRAM)

B1	B0	Adresa	Zdroj
x	0	0x0800_0000	**Main Flash** (běžný režim)
0	1	0x1FFF_0000	System memory (bootloader)
1	1	0x2000_0000	Embedded SRAM (debug)

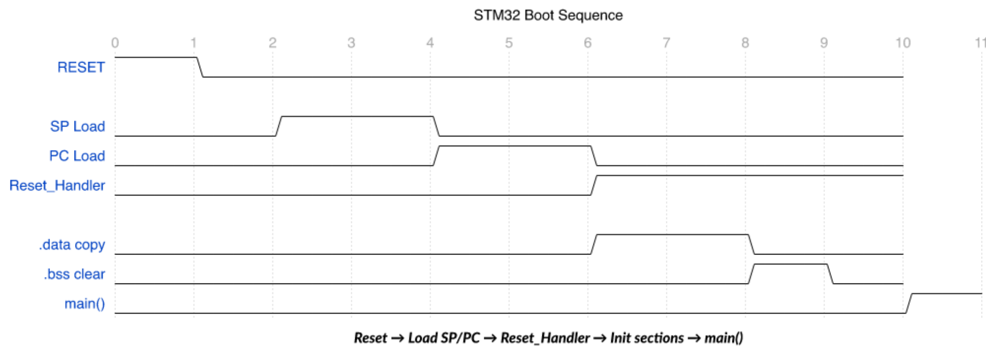


Figure 2: Průběh bootovací sekvence

Vektor tabulka přerušení

Vektor tabulka je **pole pointerů na ISR funkce**, indexované IRQ číslem z NVIC. (Např. `startup_stm32f4xx.s`.)

```
.section .isr_vector, "a"
.word _estack           // 0x00: Initial Stack Pointer
.word Reset_Handler     // 0x04: Reset (IRQ -15)
.word NMI_Handler       // 0x08: NMI (IRQ -14)
.word HardFault_Handler // 0x0C: Hard Fault (IRQ -13)
.word MemManage_Handler // 0x10: Memory Management
.word BusFault_Handler  // 0x14: Bus Fault
.word UsageFault_Handler // 0x18: Usage Fault
.word 0, 0, 0, 0        // 0x1C-0x28: Reserved
.word SVC_Handler       // 0x2C: SVCall (RTOS syscall)
.word DebugMon_Handler  // 0x30: Debug Monitor
.word 0                 // 0x34: Reserved
.word PendSV_Handler    // 0x38: PendSV (RTOS context switch)
.word SysTick_Handler   // 0x3C: SysTick (RTOS tick)
// External interrupts (IRQ 0+)
// ...
.word EXTI0_IRQHandler  // 0x58: IRQ6 - EXTI Line 0
.word EXTI1_IRQHandler  // 0x5C: IRQ7 - EXTI Line 1
// ... šdalí 80+ ůhandler
```

Klíč: Název v tabulce **MUSÍ odpovídat** názvu ISR funkce → linker je spojí!

SCB (System Control Block)

Registry ovlivňující start a běh:

Offset	Registr	Název	Použití při startu/běhu
0x00	**CPUID**	CPU ID Base	Detekce CPU typu (ARMv7-M, Cortex-M4)
0x08	**VTOR**	Vector Table Offset	Přesměrování tabulky vektorů (bootloader)
0x0C	**AIRCR**	App Interrupt/Reset Control	Software reset, priority grouping
0x10	**SCR**	System Control	Sleep modes (WFI/WFE), probuzení
0x14	**CCR**	Config and Control	Stack alignment, div-by-0 trap
0x04	**ICSR**	Interrupt Control/State	Pending/active přerušení
0x18-24	**SHPR[3]**	System Handler Priority	Priority SysTick, PendSV, faults
0x24	**SHCSR**	System Handler Control	Enable fault handlerů
0x28	**CFSR**	Fault Status	Diagnostika Memory/Bus/Usage faults

SCB - Příklad 1: Čtení CPUID

CPUID registr obsahuje informace o procesoru (read-only):

```
// Cteni CPUID registru (SCB->CPUID na 0xE000_ED00)
uint32_t cpuid = SCB->CPUID;

// Dekodovani bitu:
uint32_t implementer = (cpuid >> 24) & 0xFF; // Bits [31:24]: 0x41 = ARM
```

```
uint32_t variant = (cpuid >> 20) & 0x0F; // Bits [23:20]: Varianta
uint32_t arch    = (cpuid >> 16) & 0x0F; // Bits [19:16]: 0xC = ARMv7-M
uint32_t partno = (cpuid >> 4) & 0xFFF; // Bits [15:4]: 0xC24 = Cortex-M4
uint32_t revision = (cpuid >> 0) & 0x0F; // Bits [3:0]: Revize (r0p1)

// Priklad: Cortex-M4 r0p1
// CPUID = 0x410FC241
//  Implementer: 0x41 (ARM)
//  Variant: 0x0 (r0)
//  Architecture: 0xF (ARMv7-M)
//  PartNo: 0xC24 (Cortex-M4)
//  Revision: 0x1 (p1)
```

Použití: Runtime detekce CPU, diagnostika, bootloader rozhodování

SCB - Příklad 2: Software reset (AIRCR)

AIRCR registr umožňuje softwarově resetovat celý systém:

```
// Software reset celeho mikrokontroleru
void system_reset(void) {
   // AIRCR registr: 0xE000_ED0C
   // Bit [2]: SYSRESETREQ - Request system reset
   // Bits [31:16]: VECTKEY - Write key (0x05FA)

   SCB->AIRCR = (0x05FA << 16) | // VECTKEY povinny klic
               (SCB->AIRCR & 0x700) | // Zachovat PRIGROUP[10:8]
               (1 << 2);           // SYSRESETREQ

   // Reset probehne okamzite
   while(1); // Cekani na reset (safety)
}
```

```
// Watchdog timeout →restart
void watchdog_handler(void) {
   log_error("Watchdog timeout!");
   system_reset();
}
// Po firmware update
void firmware_update_done(void) {
   flash_complete();
   system_reset(); // Restart s novym FW
}
```

SCB - Příklad 3: Bootloader → Aplikace (VTOR)

VTOR registr přesměruje tabulku vektorů pro skok z bootloaderu do aplikace:

```
// Skok z bootloaderu do aplikace na jine adrese
void bootloader_jump_to_app(uint32_t app_address) {
   // Aplikace ma vlastni vektor tabulku na app_address
   // Napr. 0x0800_8000: bootloader 32kB, aplikace od 32kB

   // 1. Nacti Stack Pointer z aplikace (prvni slovo)
   uint32_t app_stack = *(volatile uint32_t*)app_address;
   // 2. Nacti Reset Handler z aplikace (druhe slovo)
   uint32_t app_entry = *(volatile uint32_t*)(app_address + 4);
   // 3. Vypni preruseni
```

```c
    __disable_irq();
    // 4. Presmeruj VTOR na novou vektor tabulku
    SCB->VTOR = app_address;
    // 5. Nastav Main Stack Pointer
    __set_MSP(app_stack);
    // 6. Skoc do Reset Handleru aplikace
    void (*app_reset)(void) = (void (*)(void))app_entry;
    app_reset(); // Uz se nevratime
}

// Pouziti: bootloader_jump_to_app(0x08008000);
```

Režimy procesoru Cortex-M4

Provozní režimy:

- **Thread mode** – běžné vykonávání programu (main, funkce)
- **Handler mode** – obsluha výjimek a přerušení (ISR)

Úrovně oprávnění:

- **Privileged** – plný přístup ke všem instrukcím a registrům
- **Unprivileged** – omezený přístup (pro RTOS user tasks)

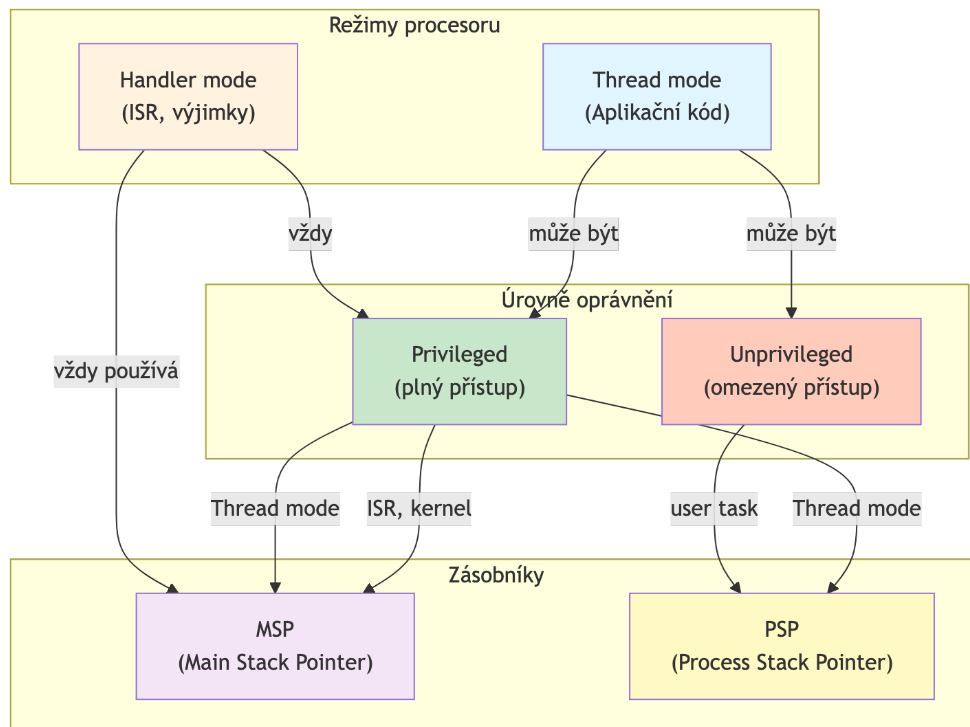


Figure 3: Režimy procesoru Cortex-M4

CONTROL registr

Volba režimu a zásobníku se provádí pomocí **CONTROL registru**:

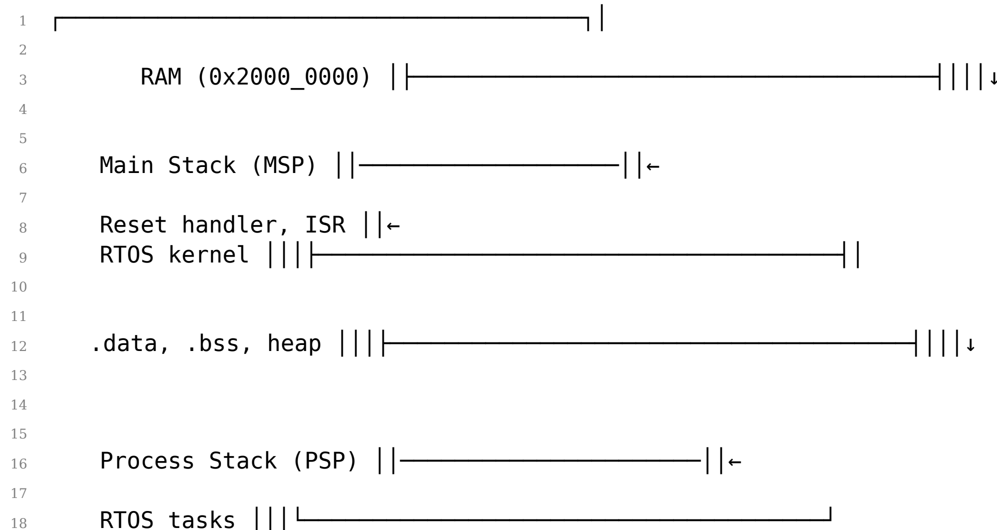
```
// CONTROL register (3 bity)
// Bit 0 (nPRIV): 0 = Privileged, 1 = Unprivileged
// Bit 1 (SPSEL): 0 = MSP, 1 = PSP (pouze v Thread mode)
// Bit 2 (FPCA): 0 = FPU neaktivni, 1 = FPU kontext aktivni

uint32_t control = __get_CONTROL();

// Prechod do unprivileged mode (nelze se vratit bez vyjimky!)
__set_CONTROL(control | 0x01);
__ISB(); // Instruction Synchronization Barrier

// Prepnuti na PSP (Process Stack Pointer)
__set_CONTROL(control | 0x02);
__ISB();
```

Použití:

- **RTOS** – kernel (privileged + MSP), user tasks (unprivileged + PSP)
- **Bezpečnost** – aplikace nemůže poškodit systémové registry, každý task má vlastní stack (PSP)

Main Stack Pointer (MSP) vs Process Stack Pointer (PSP)

Cortex-M4 má **dva zásobníky**:



Čtení/nastavení stack pointerů:

```
uint32_t msp = __get_MSP();
uint32_t psp = __get_PSP();

__set_MSP(0x20020000); // Nastavit MSP na konec RAM
__set_PSP(task_stack); // Nastavit PSP pro task
```

Co se ukládá na stack:

- Lokální proměnné funkcí
- Parametry funkcí (R0-R3 + navíc)
- Návratová adresa (LR)
- Kontext při přerušení

Přechody mezi režimy

- **Thread → Handler**:
  - Při přerušení (automaticky)
  - Hardware stacking na MSP
- **Handler → Thread**:
  - Návrat z ISR pomocí `BX LR`
  - `EXC_RETURN` hodnota v `LR`
- **Privileged → Unprivileged**:
  - Změna `CONTROL.nPRIV`
  - Nelze se vrátit bez exception!
- **Stack switch**:
  - `CONTROL.SPSEL` určuje MSP/PSP v Thread mode

Návrat z výjimky (EXC_RETURN)

EXC_RETURN je speciální hodnota, kterou hardware automaticky uloží do **Link Registru (LR)** při vstupu do přerušení/výjimky.

Jak to funguje:

1. **Vstup do ISR**: Hardware automaticky nastaví `LR = 0xFFFFFFFx` (podle kontextu)
2. **Konec ISR**: `BX LR` - návrat pomocí této speciální hodnoty
3. **Hardware dekóduje** EXC_RETURN a ví:
   - Kam se vrátit (Thread mode nebo Handler mode)
   - Který stack použít (MSP nebo PSP)
   - Jestli obnovit FPU kontext

EXC_RETURN	Návrat do	Stack	Použití
`0xFFFFFFF9`	Thread mode	MSP	Bare-metal aplikace
`0xFFFFFFFD`	Thread mode	PSP	RTOS task (vlastní PSP)
`0xFFFFFFF1`	Handler mode	MSP	Vnořené přerušení

Standardní rozložení paměti ARMv7-M

Cortex-M procesory mají pevně definovanou mapu paměti (4GB adresový prostor):

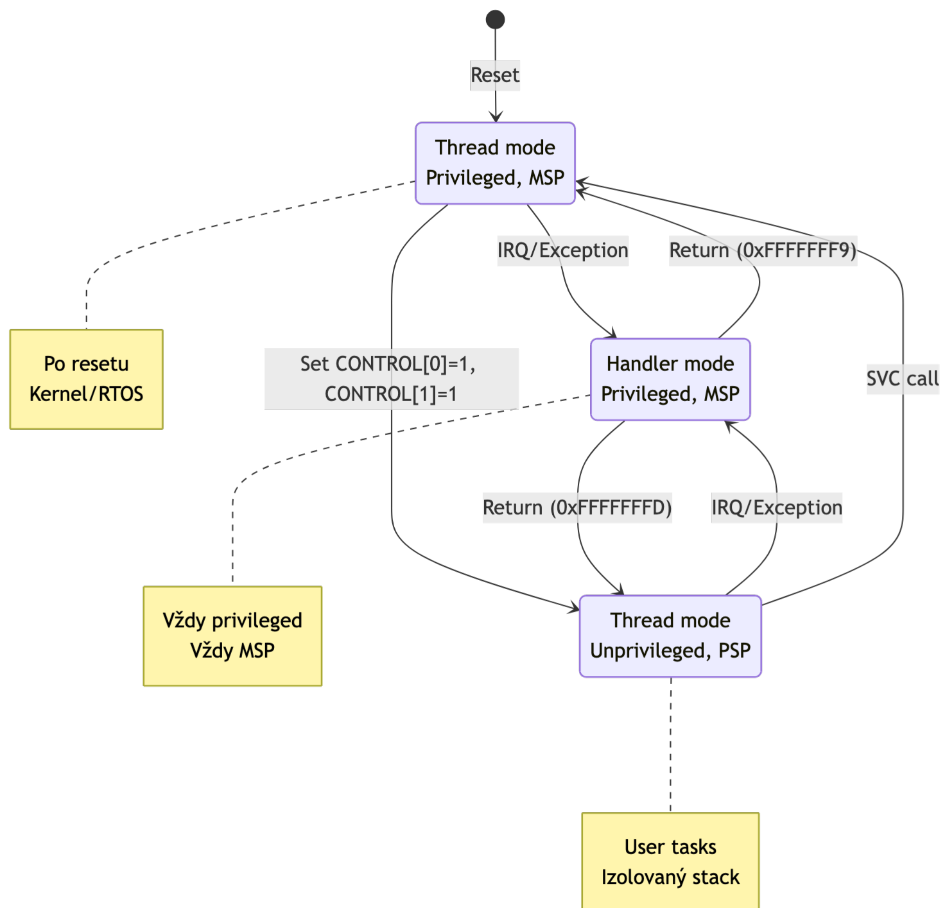
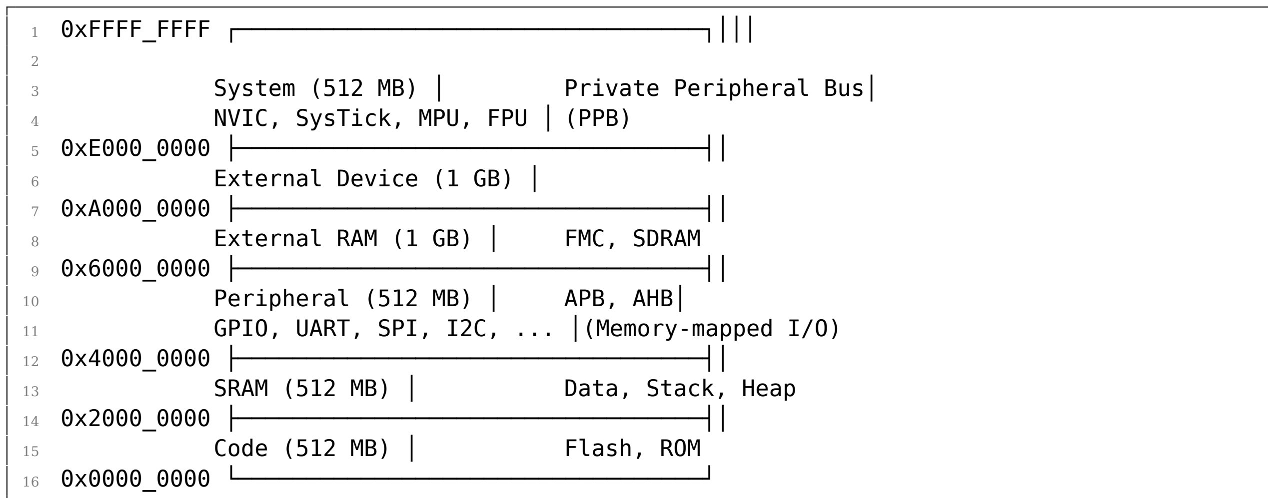


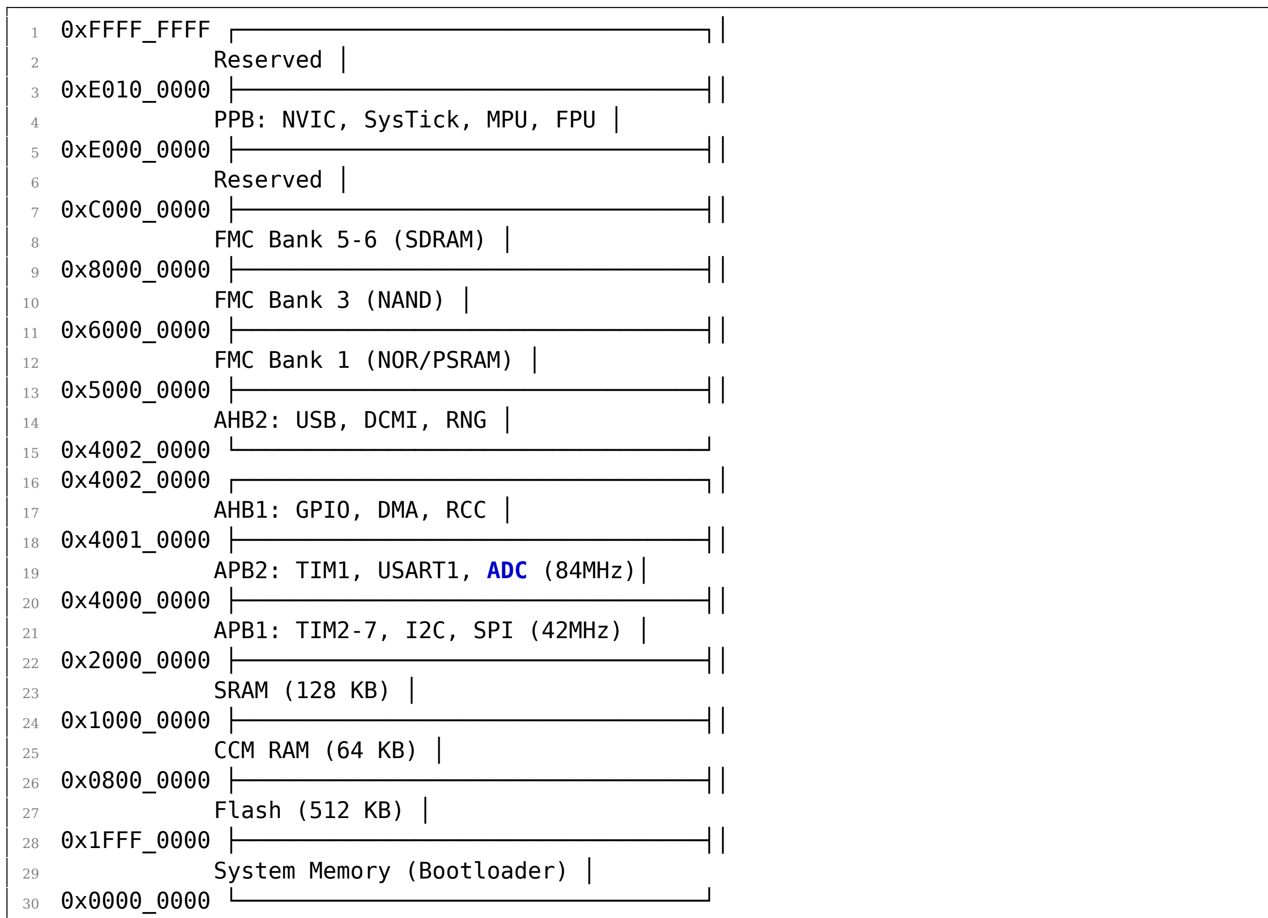
Figure 4: Stavový diagram přechodů mezi režimy jádra Cortex-M



```
0xFFFF_FFFF ┌──────────────────────────────────┐│││

            System (512 MB) │         Private Peripheral Bus│
            NVIC, SysTick, MPU, FPU │ (PPB)
0xE000_0000 ├──────────────────────────────────┤│
            External Device (1 GB) │
0xA000_0000 ├──────────────────────────────────┤│
            External RAM (1 GB) │     FMC, SDRAM
0x6000_0000 ├──────────────────────────────────┤│
            Peripheral (512 MB) │     APB, AHB│
            GPIO, UART, SPI, I2C, ... │(Memory-mapped I/O)
0x4000_0000 ├──────────────────────────────────┤│
            SRAM (512 MB) │           Data, Stack, Heap
0x2000_0000 ├──────────────────────────────────┤│
            Code (512 MB) │           Flash, ROM
0x0000_0000 └──────────────────────────────────┘
```

Výhody: Přenositelnost kódu, optimalizace kompilátoru, paralelní sběrnice

Mapa paměti STM32F4 (konkrétní příklad)



```
0xFFFF_FFFF ┌──────────────────────────────────┐│
            Reserved │
0xE010_0000 ├──────────────────────────────────┤│
            PPB: NVIC, SysTick, MPU, FPU │
0xE000_0000 ├──────────────────────────────────┤│
            Reserved │
0xC000_0000 ├──────────────────────────────────┤│
            FMC Bank 5-6 (SDRAM) │
0x8000_0000 ├──────────────────────────────────┤│
            FMC Bank 3 (NAND) │
0x6000_0000 ├──────────────────────────────────┤│
            FMC Bank 1 (NOR/PSRAM) │
0x5000_0000 ├──────────────────────────────────┤│
            AHB2: USB, DCMI, RNG │
0x4002_0000 └──────────────────────────────────┘
0x4002_0000 ┌──────────────────────────────────┐│
            AHB1: GPIO, DMA, RCC │
0x4001_0000 ├──────────────────────────────────┤│
            APB2: TIM1, USART1, ADC (84MHz)│
0x4000_0000 ├──────────────────────────────────┤│
            APB1: TIM2-7, I2C, SPI (42MHz) │
0x2000_0000 ├──────────────────────────────────┤│
            SRAM (128 KB) │
0x1000_0000 ├──────────────────────────────────┤│
            CCM RAM (64 KB) │
0x0800_0000 ├──────────────────────────────────┤│
            Flash (512 KB) │
0x1FFF_0000 ├──────────────────────────────────┤│
            System Memory (Bootloader) │
0x0000_0000 └──────────────────────────────────┘
```

Oblast Code (0x0000_0000 - 0x1FFF_FFFF)

Typické použití:

- **Flash paměť** (0x0800_0000) – program, konstanty, tabulky
- **System memory** (0x1FFF_xxxx) – bootloader od výrobce
- **ITCM RAM** (u některých MCU) – rychlá instrukční paměť

Charakteristika:

- **Executable** – lze spouštět kód (XN bit = 0)
- **Read-only** pro Flash (Write Protection možná)
- **Optimalizace** – I-Code bus (3-stage pipeline)
- **Cache-able** (pokud má MCU cache)

Příklad STM32F4:

```
#define FLASH_BASE 0x08000000UL
#define FLASH_SIZE (512 * 1024) // 512 KB
```

Runtime přemístění tabulky vektorů přerušení (VTOR)

Pro bootloadery nebo aplikace běžící z jiné adresy:

```
// VTOR: Vector Table Offset Register
#define SCB_VTOR (*(volatile uint32_t*)0xE000ED08)

void relocate_vector_table(uint32_t new_address) {
   // Nova adresa musi byt zarovnana na 512 bajtu (0x200)
   SCB_VTOR = new_address & 0xFFFFFE00;
}

// Priklad: Aplikace po bootloaderu zacina na 0x0800_8000
relocate_vector_table(0x08008000);

// Nyni NVIC nacita vektory z 0x0800_8000 misto 0x0000_0000
```

Default: `VTOR = 0x0000_0000` (Flash alias podle boot módu)

Oblast SRAM (0x2000_0000 - 0x3FFF_FFFF)

Typické použití:

- **Stack** – lokální proměnné, návratové adresy
- **Heap** – dynamická alokace (malloc)
- **Globální/statické proměnné** (.data, .bss sekce)
- **DMA buffery** – rychlý přístup pro periferie

Charakteristika:

- **Read/Write** – plný přístup
- **Non-executable** (lze změnit v MPU)
- **Optimalizace** – D-Code bus, System bus

- **Paralelní přístup** – současně s fetch z Flash

Příklad STM32F4:

```
#define SRAM_BASE 0x20000000UL
#define SRAM_SIZE (128 * 1024) // 128 KB
#define SRAM_END (SRAM_BASE + SRAM_SIZE)
```

Organizace SRAM – sekce paměti

```
0x2001_FFFF ┌──────────────────────────────┐←_estack│││

            Stack (MSP/PSP) │←    Roste ůdol ↓│────────────────│
                     Lokální ěpromnné││
                                   Return adresy├──────────────────────────────┤││

            (volné) ││├──────────────────────────────┤││

            Heap │←               Roste nahoru ↑│↑│
                                 malloc() alokace├──────────────────────────────┤││

            .bss │←               _sbss - _ebss│
            Neinicializované │    Vynulováno řpi startu│
            static int buf[256]; ││├──────────────────────────────┤││

            .data │←              _sdata - _edata│
            Inicializované │      Kopie z Flash (_sidata)│
            int x = 42; │
0x2000_0000 └──────────────────────────────┘
```

Linker script definuje symboly: `_sdata`, `_edata`, `_sbss`, `_ebss`, `_estack`

Startup kód – Reset Handler

```
void Reset_Handler(void) {
   uint32_t *src, *dst;

   // 1. Kopirovani .data z Flash →RAM
   src = &_sidata; // Load Address (Flash)
   dst = &_sdata; // Run Address (RAM)
   while (dst < &_edata) { *dst++ = *src++; }

   // 2. Vynulovani .bss
   dst = &_sbss;
   while (dst < &_ebss) { *dst++ = 0; }

   // 3. Inicializace systemu (hodiny, FPU, ...)
   SystemInit();

   // 4. Skok do main()
   main();
}
```

Proč? Flash je read-only → inicializované proměnné musí být v RAM

Oblast Peripheral (0x4000_0000 - 0x5FFF_FFFF)

Rozdělení podle sběrnice:

Sběrnice	Rozsah	Frekvence	Příklady periferií
APB1	0x4000_0000 - 0x4000_7FFF	42 MHz	TIM2-7, I2C1-3, SPI2-3, USART2-5
APB2	0x4001_0000 - 0x4001_5BFF	84 MHz	TIM1, TIM8-11, USART1, 6, ADC, SPI1
AHB1	0x4002_0000 - 0x4002_43FF	168 MHz	GPIO, DMA1-2, RCC, CRC
AHB2	0x5000_0000 - 0x5006_0BFF	168 MHz	USB OTG, DCMI, RNG

Charakteristika:

- **Memory-mapped I/O** – přístup jako k RAM (load/store)
- **Device memory type** – bez cache, strict ordering
- **Volatile access** – každý read/write jde na sběrnici

Oblast External RAM (0x6000_0000 - 0x9FFF_FFFF)

FMC (Flexible Memory Controller) – mapování:

Bank	Rozsah	Typ paměti
1	0x6000_0000 - 0x6FFF_FFFF	NOR/PSRAM/SRAM (256 MB)
3	0x8000_0000 - 0x8FFF_FFFF	NAND Flash
5-6	0xC000_0000 - 0xDFFF_FFFF	SDRAM (512 MB)

Konfigurace SDRAM (příklad):

```
void init_sdram(void) {
   // Zapnout FMC clock
   RCC->AHB3ENR |= RCC_AHB3ENR_FMCEN;

   // Konfigurace SDRAM timing
   FMC_Bank5_6->SDCR[0] =
      FMC_SDCR1_RBURST | // Read burst enable
      FMC_SDCR1_SDCLK_1 | // SDCLK = 2x HCLK
      (2 << 4) |        // CAS latency = 2
      (1 << 0);         // 8-bit column address

   // Load Mode Register command
   // ... dalsi inicializace
}
```

Oblast System (0xE000_0000 - 0xFFFF_FFFF)

Private Peripheral Bus (PPB):

Komponenta	Adresa	Účel
ITM	0xE000_0000	Instrumentation Trace Macrocell

Komponenta	Adresa	Účel
DWT	0xE000_1000	Data Watchpoint and Trace
FPB	0xE000_2000	Flash Patch and Breakpoint
SCS	0xE000_E000	System Control Space
NVIC	0xE000_E100	Nested Vectored Interrupt Controller
SysTick	0xE000_E010	System Timer
MPU	0xE000_ED90	Memory Protection Unit
FPU	0xE000_EF30	Floating Point Unit
Debug	0xE004_0000	Debug components (ETM, TPIU)

2. Architektura sběrnic (AMBA)

AMBA (Advanced Microcontroller Bus Architecture)

AMBA je otevřený standard od ARM pro propojení komponent v SoC:

Protokoly AMBA:

1. **AHB (Advanced High-performance Bus)** – vysoký výkon, pipeline
2. **APB (Advanced Peripheral Bus)** – nízká spotřeba, jednoduché periferie
3. **AXI (Advanced eXtensible Interface)** – nejnovější, multi-channel (Cortex-M7)

V Cortex-M4 typicky:

- **AHB-Lite** – zjednodušená verze AHB (single master)
- **APB** – pro pomalé periferie (UART, I2C, …)
- **Multi-layer AHB** – paralelní přístup k různým slave

Bus Matrix (AHB Crossbar)

Bus Matrix propojuje CPU rozhraní sběrnic s pamětí a periferiemi

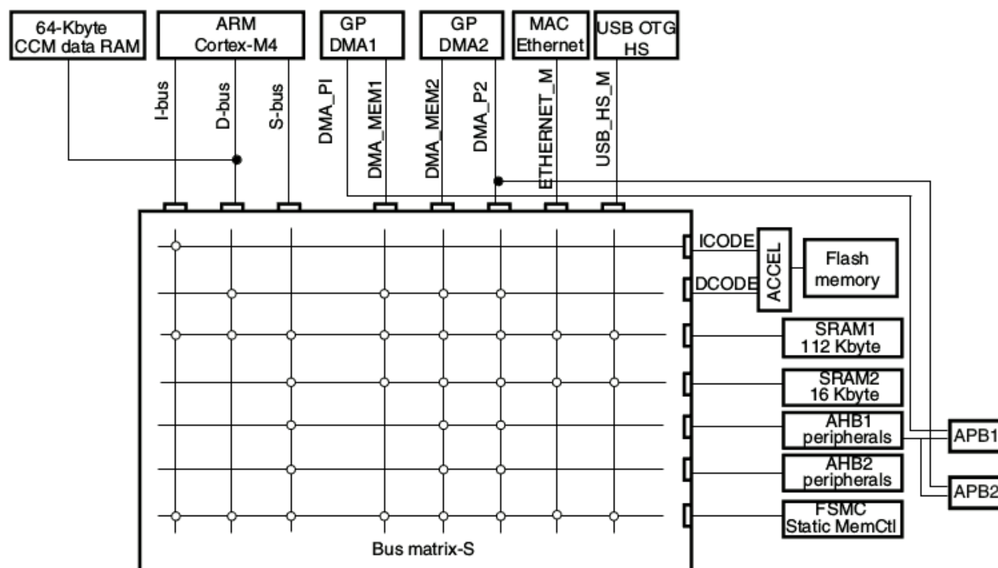


Figure 5: Matice sběrnic

Klíčové vlastnosti

- **Multi-master arbitrace** – CPU a DMA mohou současně přistupovat k různým slaves
- **Paralelní přístupy** – I-Code fetch + D-Code data access + DMA transfer současně
- **Priority-based** – CPU má vyšší prioritu než DMA při konfliktu

AHB (Advanced High-performance Bus)

Charakteristiky

- **Pipeline** – adresa v jednom cyklu, data v dalším
- **Burst transfer**
  - souvislé přenosy (4, 8, 16 beats)
- **Split transaction** – slave může pozastavit transakci
- **Single clock edge** – všechny signály na rising edge
- **Multi-master**
  - (v původním AHB, AHB-Lite = single master)

Signály (zjednodušeně):

```
HCLK     - Clock
HRESETn  - Reset (active low)
HADDR[31:0] - Address bus
HWDATA[31:0]- Write data
HRDATA[31:0]- Read data
HWRITE   - Write/Read direction
HSIZE[2:0] - Transfer size
HBURST[2:0] - Burst type
HTRANS[1:0] - Transfer type
HREADY   - Slave ready
HRESP    - Response
```

- Transfer size: byte, halfword, word
- Transfer type: IDLE, BUSY, NONSEQ, SEQ
- Response: OKAY, ERROR

AHB Transfer typy

HTRANS encoding:

HTRANS	Typ	Popis
00	IDLE	Žádný transfer
01	BUSY	Master není připraven (v burst)
10	NONSEQ	Single transfer nebo první v burst
11	SEQ	Následující transfer v burst

HBURST encoding:

HBURST	Typ	Délka
000	SINGLE	1 transfer
001	INCR	Undefined length burst
010	WRAP4	4-beat wrapping burst
011	INCR4	4-beat incrementing burst
100	WRAP8	8-beat wrapping burst
101	INCR8	8-beat incrementing burst
110	WRAP16	16-beat wrapping burst
111	INCR16	16-beat incrementing burst

AHB pipeline timing

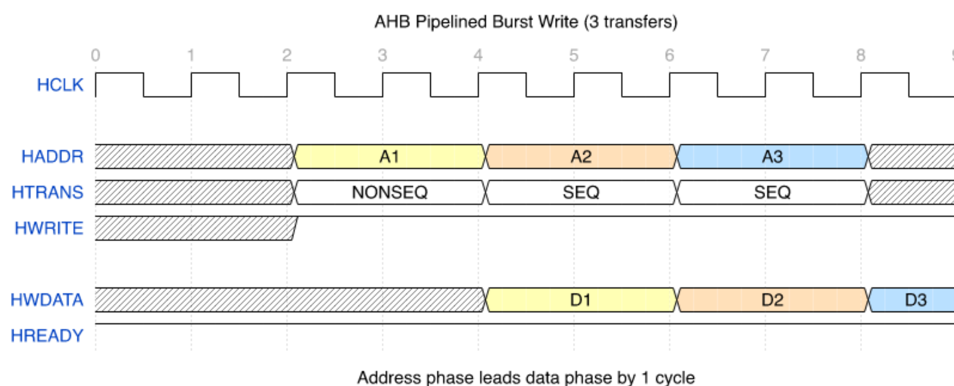


Figure 6: Komunikace na sběrnice AHB

Pipelining: Adresa A2 jde současně s daty D1 (address phase vede data phase o 1 cyklus).

Výhody:

- Vyšší throughput (přístup každý cyklus)
- Nižší latence pro burst transfery

APB (Advanced Peripheral Bus)

Charakteristiky:

- **Jednoduchý** – žádný pipeline, žádné burst
- **Nízká spotřeba** – minimální logika
- **Pomalý** – obvykle 1/2 nebo 1/4 rychlosti AHB
- **Non-pipelined** – adresa a data ve stejném cyklu

Signály:

```
PCLK      - Clock
PRESETn   - Reset
PADDR[31:0] - Address
PSEL      - Select (chip select)
PENABLE   - Enable strobe
PWRITE    - Write/Read
PWDATA[31:0]- Write data
PRDATA[31:0]- Read data
PREADY    - Slave ready (wait states)
PSLVERR   - Slave error
```

APB transfer timing

2-fázový protokol:

- **Setup phase**: PSEL=1, PENABLE=0 (adresa a data se stabilizují)

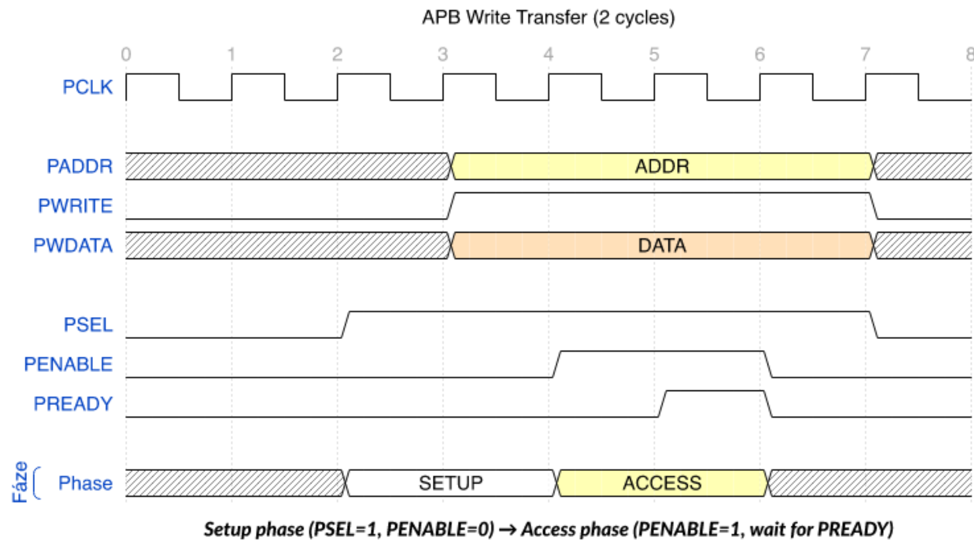


Figure 7: Komunikace na sběrnici APB

- **Access phase**: PENABLE=1, čeká na PREADY (slave potvrdí přijetí)

AHB to APB Bridge

APB bridge propojuje vysokorychlostní AHB s pomalým APB:

```
// Priklad mapovani v STM32F4:
// AHB1: 168 MHz (GPIO, DMA, RCC)
// APB1: 42 MHz (TIM2-7, I2C, SPI2-3, USART2-5)
// APB2: 84 MHz (TIM1, TIM8-11, USART1, ADC, SPI1)

// Pri pristupu z CPU (AHB) na USART1 (APB2):
// 1. AHB master (CPU) iniciuje write na 0x40011000 (USART1_DR)
// 2. Bus matrix routuje na APB2 bridge
// 3. APB2 bridge prevede AHB protocol →APB protocol
// 4. USART1 prijme data, potvrdi PREADY
// 5. Bridge vrati HREADY na AHB
```

Důsledky:

- Přístup k APB = několik extra cyklů (latence)
- Burst transfery se rozpadnou na single
- DMA k APB periferiím = pomalejší než k AHB

Bus arbitration (rozhodování)

Když více masterů (CPU, DMA1, DMA2, Ethernet, ...) chce přístup:

1. Fixní priority:

- Každý master má pevnou prioritu (0 = nejvyšší)
- Vyšší priorita = vždy vyhraje

- **Problém**: na nižší priority se nemusí vůbec dostat

2. Round-robin:

- Rotace mezi mastery (fair scheduling)
- Nikdo není zvýhodněn
- **Problém**: nedeterministické latence

3. Váhovaný round-robin:

- Kombinace priority a fair schedulingu
- Vysoká priorita = více slotů

Praktický příklad: DMA priorita

STM32F4 používá fixní priority s možností runtime změny DMA priority.

```
// DMA stream configuration
DMA_Stream_TypeDef *stream = DMA2_Stream0;

// Priority level (bits 17:16)
stream->CR &= ~DMA_SxCR_PL_Msk;
stream->CR |= DMA_SxCR_PL_1; // High priority

// Priority levels:
// 00: Low
// 01: Medium
// 10: High
// 11: Very High
```

Scénář:

- DMA1 Stream 0 (priority Low) čte z ADC
- DMA2 Stream 0 (priority High) čte z UART
- Když oba chtějí přístup do SRAM → DMA2 vyhraje

Memory-mapped I/O sémantika

Volatile access:

```
// Spravne - volatile zajisti pristup na sbernici
volatile uint32_t *gpio_odr = (volatile uint32_t*)0x40020014;
*gpio_odr = 0x0020; // Jde na AHB →APB2 →GPIOA

// Spatne - kompilator muze optimalizovat pryc
uint32_t *gpio = (uint32_t*)0x40020014;
*gpio = 0x0020;
*gpio = 0x0000; // Muze byt slouceno nebo odstraneno
```

Read-modify-write hazard:

```
// Nebezpecne - read-modify-write muze zpusobit race condition
GPIOA->ODR |= (1 << 5); // Read →Modify →Write

// Bezpecne - atomicky set/reset
```

```
GPIOA->BSRR = (1 << 5); // Pouze write, hardware handling
```

Bus stall a wait states

Flash wait states (STM32F4 při 168 MHz):

```
// Flash Access Control Register
FLASH->ACR = FLASH_ACR_LATENCY_5WS | // 5 wait states @ 168 MHz
             FLASH_ACR_ICEN |   // I-cache enable
             FLASH_ACR_DCEN |   // D-cache enable
             FLASH_ACR_PRFTEN;  // Prefetch enable
```

Proč wait states?

- Flash je pomalejší než CPU (50-60 MHz vs. 168 MHz)
- Při přístupu musí CPU čekat 5 cyklů na data
- **Cache a prefetch** dramaticky zrychlují (hit rate > 95%)

SRAM:

- Zero wait state (stejná rychlost jako CPU)
- Proto kritický kód v RAM běží rychleji

Protokol konverze (AHB master → APB slave)

1. **AHB NONSEQ write** na adresu 0x40000000 (APB1)
2. Bridge detekuje APB1 address range
3. Bridge převede:
   - `HADDR` → `PADDR`
   - `HWDATA` → `PWDATA`
   - `HWRITE` → `PWRITE`
4. Bridge vygeneruje `PSEL`, pak `PENABLE`
5. Čeká na `PREADY` od slave
6. Vrátí `HREADY` na AHB

Latence:

- AHB single transfer: 1 cyklus
- APB single transfer: 2 cykly (setup + access)
- **Total**: 2-3 AHB cykly (včetně bridging overhead)

AXI4 (Cortex-M7)

AXI (Advanced eXtensible Interface) je nejnovější AMBA protokol. Primárně v Cortex-M7, Cortex-M4 má AHB-Lite.

Výhody oproti AHB:

- **Multi-channel** – oddělené kanály pro read/write address, data, response
- **Out-of-order** ukončování transakcí (asynchronní)
- **Vyšší propustnost** – více transakcí současně
- **Podpora pro cache**

Kanály:

1. **Write Address** (AW)
2. **Write Data** (W)
3. **Write Response** (B)
4. **Read Address** (AR)
5. **Read Data** (R)

3. Systém přerušení (NVIC)

Přerušovací systém

Přerušovací systémem (Interrupt systém) podstatně zjednodušuje a zefektivňuje styk s vnitřními i vnějšími periferiemi.

Obsluha periferií bez přerušovacího systému

- Snížení výpočetního času pro hlavní program
- Program rozšířen o pravidelné testováním žádostí
- Obtížné zpracování v potřebných intervalech

Obsluha s využitím přerušovacího systému

- Periferie požádá o přerušení
- Procesor uloží žádost ke zpracování
- Testuje zda přerušení od dané periferie je povolené
- Vyhodnotí zda není jiná žádost s vyšší prioritou
- Přeruší ve vhodném okamžiku probíhající program (dokončí právě rozpracovanou instrukci nebo instrukce)
- Přejde ke zpracování obslužného programu

Přechod do přerušení

Žádost periferie o přerušení je vyvolána obvodově obvykle pomocí instrukce CALL adresa_přerušení.

CPU

- Uloží PC do zásobníkové paměti (adresu následující instrukce po poslední vykonané)
- Nastaví PC na adresu volaného přerušení (přepisem PC se ztrácí informace o místě, kde byl program přerušen).
- Je zablokována daná úroveň přerušovacího systému
- Procesor zahájí svoji činnost od nastavené adresy obslužného programu
- Používané registry v obslužném programu je nutné uložit včetně stavu příznakového registru
- Na konci přerušení musí být stav registrů a příznaků obnoven.
- Přerušení je ukončeno instrukcí RETI (povolení přerušení).
- Atypicky možnost návratu bez povolení přerušovacího systému.

Rozdíl mezi voláním funkce a ISR (Cortex M)

Volání funkce (běžný podprogram)

- Synchronní - program explicitně volá funkci pomocí BL nebo BLX
- Kontext: Používá stejný stack (MSP nebo PSP)
- Registry: Caller musí uložit R0-R3, R12 (caller-saved), callee ukládá R4-R11 (callee-saved)
- Návrat: BX LR - normální návratová adresa
- Přepnutí režimu: NE - zůstává v Thread nebo Handler mode
- Overhead: Minimální - jen uložení potřebných registrů

```c
void main(void) {
   my_function(); // Explicitni volani
}

void my_function(void) {
   // R0-R3, R12 caller-saved
   // R4-R11 callee-saved (pokud pouzite)
   return; // BX LR
}
```

Volání ISR (Interrupt Service Routine)

- Asynchronní - hardware automaticky při události (IRQ)
- Kontext: Automatický přechod do Handler mode + použití MSP
- Hardware stacking: CPU automaticky ukládá 8 registrů (R0-R3, R12, LR, PC, xPSR) - 12 cyklů
- Návrat: BX LR s EXC_RETURN (0xFFFFFFFx) - dekóduje hardware
- Přepnutí režimu: ANO - Thread → Handler mode
- Overhead: Větší - hardware stacking/unstacking + tail-chaining optimalizace

```c
// Hardware automaticky pri IRQ:
// 1. Ulozi R0-R3, R12, LR, PC, xPSR na stack (12 cyklu)
// 2. Nastavi LR = EXC_RETURN (0xFFFFFFF9)
// 3. Prepne do Handler mode
// 4. Skoci na ISR vector

void EXTI0_IRQHandler(void) { // Jsme v Handler mode, MSP stack
   EXTI->PR = (1 << 0); // Clear flag
   // Hardware pri BX LR:
   // 1. Dekoduje EXC_RETURN
   // 2. Obnovi R0-R3, R12, LR, PC, xPSR (10 cyklu)
   // 3. Vrati se do Thread mode
}
```

Výjimka (Exception) vs. Přerušení (Interrupt)

Terminologie v ARM Cortex-M:

Exception (výjimka) – obecný pojem pro libovolnou událost, která přeruší běžící kód:

- **Interní výjimky**: Reset, NMI, HardFault, SVC, PendSV, SysTick
- **Externí výjimky**: přerušení z periferií (EXTI, UART, Timer, …)

Interrupt (přerušení) – podmnožina exceptions:

- Pouze **externí** zdroje (periferie, GPIO, …)
- V ARM dokumentaci: IRQ = Interrupt Request

V praxi se termíny často **zaměňují** (oba spouští ISR)

- U ARMu je **Exception** nadřazený pojem.

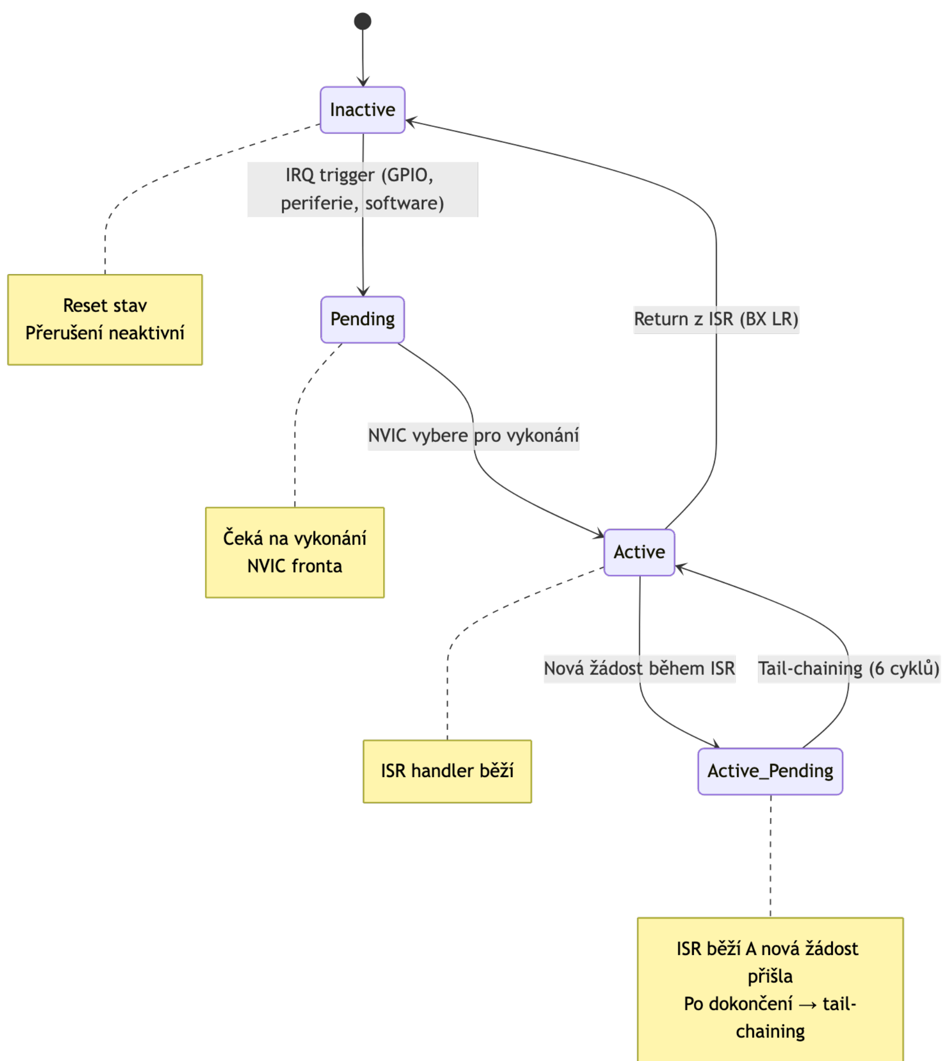


Figure 8: Stavy výjimek

Problémy většího počtu přerušení

Je-li v aplikaci více zdrojů přerušení, mohou nastat problémy s dobou přístupu do zpracování jednotlivých obsluh. Volba procesoru pak ovlivňuje:

- Počet úrovní přerušovacího sytému.
- Možnost změny priority daného přerušení.
- Možnost přerušení v přerušení

Jednoúrovňový systém s pevně danými prioritami zdrojů

- Každé přerušení má svoji adresu a po vstupu do obsluhy přerušení maže procesor jeho příznak. Změna priorit jednotlivých přerušení je obtížná (AVR).
- Více přerušení má jednu adresu – příznak musí být mazán programově. Programová změna priorit přerušení – je možná (TMS320C15, některé 8051, ARM)
- Nested interrupt – Způsob jak vyhovět časově kritickému přerušení při zpracovávání jiného déle trvajícího přerušení. Podpořeno více úrovněmi nebo definovaným postupem.

Přerušovací systém s volitelnou úrovní priority

Více úrovňový systém s pevně danými prioritami

- Přerušení s jakoukoliv prioritou lze přesunout do vyšší úrovně priority, z které přeruší i přerušení s vyšší prioritou umístěné v nižší úrovni priorit – tzv. přerušení v přerušení.

Žádosti o přerušení se testují v definovaném okamžiku strojového cyklu procesoru a posléze se vyhodnocují.

Vyhodnocení – je-li přerušení povolené, není další žádost přerušení s vyšší prioritou a je povolený přerušovací systém.

Vyvolání přerušení – instrukcí LCALL adresa přerušení nebo přečtením adresy. Dojde k uložení návratové adresy, přenesení adresy přerušení do PC, zakázání přerušovacího systému, případně uložení některých registrů).

Zpracováním přerušení nesmí být ovlivněny hodnoty registrů a příznaků před přechodem do přerušení.

NVIC (Nested Vectored Interrupt Controller)

NVIC je hardwarový řadič přerušení integrovaný v Cortex-M jádře:

Vlastnosti:

- **Nested** – přerušení mohou přerušovat jiná přerušení
- **Vectored** – každé přerušení má svůj handler v tabulce vektorů
- **Prioritní** – 0-255 úrovní priority (u STM32F4: 16 úrovní, 4 bity)
- **Deterministický** – fixní latence (12 cyklů na Cortex-M4)
- **Tail-chaining** – rychlé přepínání mezi ISR (6 cyklů)
- **Late-arriving** – optimalizace pro vyšší prioritu během entry

Komponenty:

- **NVIC registers** – Enable, Pending, Priority, Active
- **SCB registers** – System Control Block (ICSR, AIRCR, …) - viz sekce CPU
- **Vector table** – tabulka ukazatelů na handlery

Maskovatelné vs. Nemaskovatelné přerušení

Maskovatelné přerušení (Maskable Interrupt):

- Lze **povolit/zakázat** pomocí NVIC (ISER/ICER registry)
- Lze **blokovat globálně** pomocí `CPSID i` (disable IRQ)
- Lze nastavit **prioritu** (0-15 na STM32F4)
- **Většina přerušení** patří do této kategorie

```
// Zakazat EXTI0
NVIC_DisableIRQ(EXTI0_IRQn);

// Zakazat vsechna IRQ (CPSID i)
__disable_irq();
```

Nemaskovatelné přerušení (NMI):

- **Nelze zakázat** – vždy aktivní
- **Druhá nejvyšší priorita** (po Reset)
- Použití: **kritické události** (watchdog, power failure, safety)

```
void NMI_Handler(void) {
    // Kriticka obsluha -nelze vypnout!
}
```

NVIC Registry

Hlavní registry (32-bit pro každých 32 přerušení):

```
#define NVIC_BASE 0xE000E100UL

typedef struct {
    volatile uint32_t ISER[8]; // Interrupt Set Enable (0xE000E100)
    uint32_t RESERVED0[24];
    volatile uint32_t ICER[8]; // Interrupt Clear Enable (0xE000E180)
    uint32_t RESERVED1[24];
    volatile uint32_t ISPR[8]; // Interrupt Set Pending (0xE000E200)
    uint32_t RESERVED2[24];
    volatile uint32_t ICPR[8]; // Interrupt Clear Pending (0xE000E280)
    uint32_t RESERVED3[24];
    volatile uint32_t IABR[8]; // Interrupt Active Bit (0xE000E300, RO)
    uint32_t RESERVED4[56];
    volatile uint8_t IP[240]; // Interrupt Priority (0xE000E400)
    uint32_t RESERVED5[644];
    volatile uint32_t STIR; // Software Trigger Interrupt (0xE000EF00)
} NVIC_Type;

#define NVIC ((NVIC_Type*)NVIC_BASE)
```

NVIC Priority levels

STM32F4 má 4-bitové priority → 16 úrovní (0-15):

- **0** = nejvyšší priorita
- **15** = nejnižší priorita

Priority grouping (PRIGROUP):

Dělí 4 bity na **preempt** priority a **sub** priority:

PRIGROUP	Preempt bits	Sub bits	Preempt levels	Sub levels
0	0	4	1	16
1	1	3	2	8
2	2	2	4	4
3	3	1	8	2
4	4	0	16	1

Preempt priority:

- Určuje, zda přerušení může přerušit jiné.

Sub priority:

- Určuje pořadí, když více přerušení čeká.

Nastavení priority grouping

```c
// AIRCR: Application Interrupt and Reset Control Register
#define SCB_AIRCR (*(volatile uint32_t*)0xE000ED0C)

void NVIC_SetPriorityGrouping(uint32_t group) {
   uint32_t reg = SCB_AIRCR;
   reg &= ~(0xFFFF0000 | (7 << 8)); // Clear key and PRIGROUP
   reg |= (0x05FA << 16) |       // VECTKEY
         ((group & 7) << 8);      // PRIGROUP
   SCB_AIRCR = reg;
}

// Priklad: 4 preempt levels, 4 sub levels
NVIC_SetPriorityGrouping(2);
```

CMSIS funkce:

```c
NVIC_SetPriorityGrouping(NVIC_PRIORITYGROUP_2);
```

Povolení/zakázání přerušení

Enable interrupt:

```c
// Pomoci CMSIS
NVIC_EnableIRQ(USART1_IRQn); // IRQn = 37 pro STM32F4
// Primo do registru
NVIC->ISER[37 / 32] = (1 << (37 % 32));
```

Disable interrupt:

```c
NVIC_DisableIRQ(USART1_IRQn);
// Primo
NVIC->ICER[37 / 32] = (1 << (37 % 32));
```

Set priority:

```
NVIC_SetPriority(USART1_IRQn, 5); // Priority 5 (0-15)
// Primo (STM32 pouziva horni 4 bity)
NVIC->IP[37] = (5 << 4);
```

Interrupt Service Routine (ISR)

Definice handleru:

```
void USART1_IRQHandler(void) {
   // 1. Zjistit zdroj preruseni
   if (USART1->SR & USART_SR_RXNE) {
      // Prijat byte
      uint8_t data = USART1->DR; // Cteni vymaze flag
      handle_rx(data);
   }

   if (USART1->SR & USART_SR_TXE) {
      // Transmit buffer empty
      if (tx_buffer_available()) {
         USART1->DR = get_next_byte();
      } else {
         USART1->CR1 &= ~USART_CR1_TXEIE; // Disable TX interrupt
      }
   }

   // Poznamka: Vetsina flagu se maze automaticky (ctenim DR)
   // nebo explicitne (SCB->ICSR)
}
```

Kontext switch při přerušení

```
Higher addresses
+------------------+
|     xPSR     | ← Program Status Register
+------------------+
|      PC      | ← Return address
+------------------+
|      LR      | ← Link Register (pre-IRQ value)
+------------------+
|     R12      |
+------------------+
|      R3      |
+------------------+
|      R2      |
+------------------+
|      R1      |
+------------------+
|      R0      | ← SP after stacking
+------------------+
Lower addresses
```

Celkem: 8 registrů × 4 bajty = 32 bajtů

Pokud FPU: Navíc S0-S15, FPSCR (68 bajtů celkem)

Latence přerušení (Interrupt Latency)

Cortex-M4 (bez FPU):

12 cyklů – minimální latence od signálu IRQ po první instrukci ISR

Breakdown:

1. **1 cyklus** – detekce přerušení
2. **1 cyklus** – vector fetch (načtení adresy z tabulky)
3. **8 cyklů** – stacking (R0-R3, R12, LR, PC, xPSR)
4. **2 cykly** – pipeline flush a fetch první instrukce ISR

10 cyklů na návrat z ISR (context pop)

S FPU (lazy stacking):

- **12 cyklů** (FPU kontext se odkládá jen pokud ISR používá FPU)
- **+17 cyklů** při prvním FPU přístupu v ISR

Tail-chaining (zřetězení obsluhy)

Pokud další přerušení čeká ve frontě (Pending) během běhu ISR:

- **Přeskočí** context push/pop mezi ISR
- Pouze **6 cyklů** na přechod mezi handlery
- Výrazná **úspora času** při vysoké frekvenci přerušení

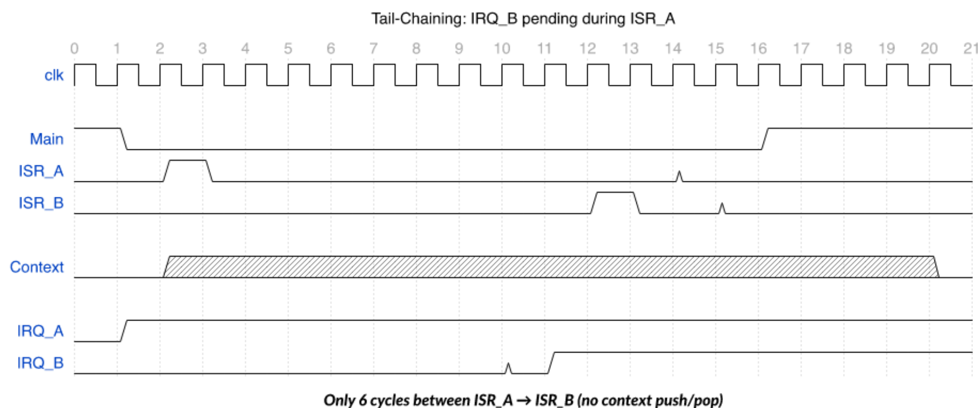


Figure 9: Zřetězení obsluhy přerušení

Praktický dopad: DMA dokončení + UART RX ve stejný okamžik → rychlé přepnutí

Late-arrival optimalizace

Pokud přerušení s vyšší prioritou přijde **během vstupu** do nižší:

- NVIC **zruší** vstup do nižší priority
- Okamžitě **vstoupí** do vyšší priority ISR

- Ušetří ~12 cyklů (nevejde do nesprávné ISR)

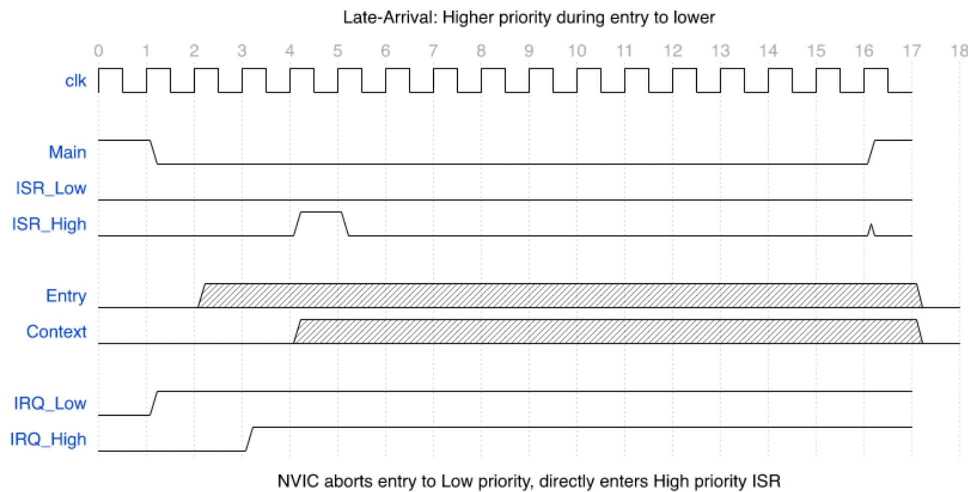


Figure 10: Late arrival optimalizace

Preempce (Přednostní přerušení)

Preempce = vyšší priorita může přerušit obsluhu nižší priority

Pravidla:

1. **Vyšší priorita** (nižší číslo) může přerušit běžící ISR
2. **Stejná nebo nižší** priorita musí čekat (Pending stav)
3. **Sub-priorita** určuje pořadí při stejné Preempt prioritě

Vnořená přerušení: hloubka vnoření teoreticky neomezená (prakticky omezena stackem)

Příklad (PRIGROUP=2):

```
NVIC_SetPriority(TIM2_IRQn, (2<<4)|1); // P=2, S=1
NVIC_SetPriority(USART1_IRQn, (1<<4)|3); // P=1, S=3

// TIM2 bezi →USART1 prijde →PREEMPTS (1 < 2)
```

Kritické sekce – zakázání přerušení

Globální disable:

```
// CMSIS
__disable_irq(); // CPSID i (set PRIMASK)
// kriticka operace
__enable_irq(); // CPSIE i (clear PRIMASK)
```

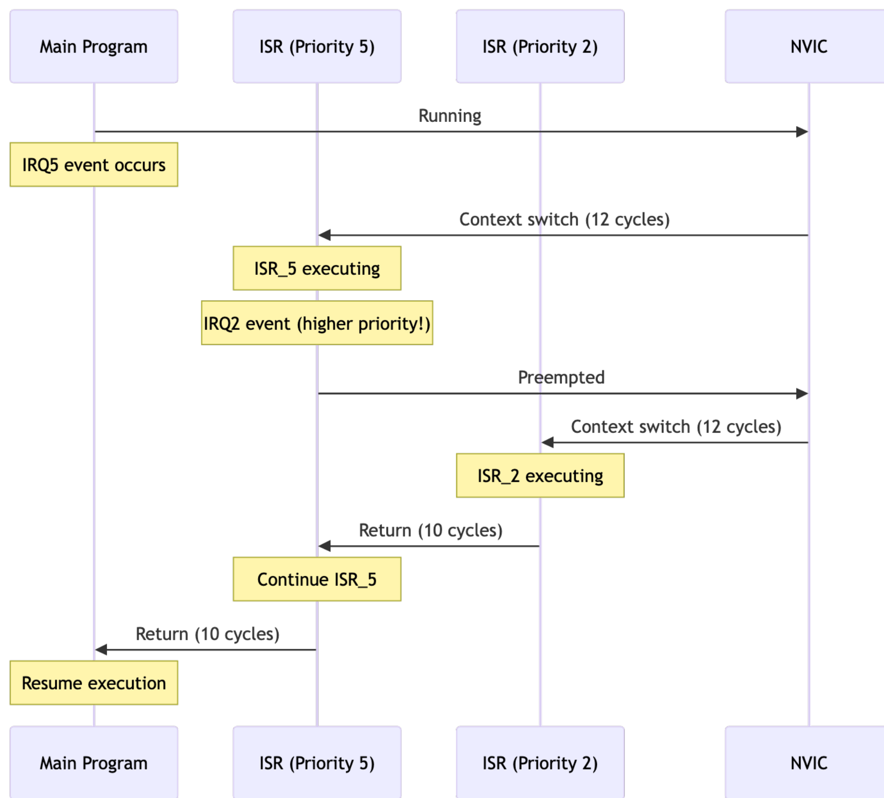


Figure 11: Přednostní přerušení

Disable s prioritním prahem (BASEPRI):

```c
// Zakaze preruseni s prioritou >= 5 (nizsi dulezitost)
__set_BASEPRI(5 << 4); // Horni 4 bity
// kriticka operace
__set_BASEPRI(0); // Enable all
```

Použití:

- **PRIMASK**: Rychlé, ale blokuje vše (včetně HardFault!)
- **BASEPRI**: Selektivní, lze povolit high-priority interrupts

SysTick timer

SysTick je 24-bitový down-counter integrovaný v Cortex-M:

Použití: delay funkce, timeouty, RTOS tick (typicky 1 ms)

Konfigurace:

```c
void SysTick_Config(uint32_t ticks) {

   SysTick->LOAD = ticks - 1;  // Reload value
   SysTick->VAL = 0;           // Clear current value
   SysTick->CTRL = SysTick_CTRL_CLKSOURCE_Msk | // Core clock
               SysTick_CTRL_TICKINT_Msk | // Enable interrupt
               SysTick_CTRL_ENABLE_Msk; // Enable counter

   NVIC_SetPriority(SysTick_IRQn, 15); // Lowest priority
}

// 1 ms tick @ 168 MHz
SysTick_Config(168000000 / 1000);
```

SysTick handler

```c
volatile uint32_t system_ticks = 0;

void SysTick_Handler(void) {
   system_ticks++;

   // Pro FreeRTOS:
   // xPortSysTickHandler();
}

void delay_ms(uint32_t ms) {
   uint32_t start = system_ticks;
   while ((system_ticks - start) < ms) {
      __WFI(); // Wait for interrupt (sleep)
   }
}

uint32_t millis(void) {
   return system_ticks;
}
```

Praktický příklad: UART RX s circular buffer

```c
#define RX_BUFFER_SIZE 256
volatile uint8_t rx_buffer[RX_BUFFER_SIZE];
volatile uint16_t rx_head = 0;
volatile uint16_t rx_tail = 0;

void USART1_Init(void) {
   // ... GPIO, clock config

   USART1->CR1 = USART_CR1_UE | // USART enable
              USART_CR1_RE | // Receiver enable
              USART_CR1_RXNEIE; // RX interrupt enable

   NVIC_SetPriority(USART1_IRQn, 3);
   NVIC_EnableIRQ(USART1_IRQn);
}

void USART1_IRQHandler(void) {
   if (USART1->SR & USART_SR_RXNE) {
      uint8_t data = USART1->DR;

      uint16_t next = (rx_head + 1) % RX_BUFFER_SIZE;
      if (next != rx_tail) { // Buffer not full
         rx_buffer[rx_head] = data;
         rx_head = next;
      }
      // else: overflow, discard
   }
}
```

Optimalizace latence přerušení

1. Minimalizace délky trvání ISR

```c
// Spatne - dlouha ISR
void TIM2_IRQHandler(void) {
   TIM2->SR &= ~TIM_SR_UIF;
   process_sensor_data(); // 100+ µs!
}

// Dobre - rychla signalizace (nebo pouzit frontu)
volatile bool sensor_ready = false;

void TIM2_IRQHandler(void) {
   TIM2->SR &= ~TIM_SR_UIF;
   sensor_ready = true;  // 1-2 cykly
}

void main_loop(void) {
   while (1) {
      if (sensor_ready) {
         sensor_ready = false;
         process_sensor_data();
      }
   }
}
```

2. Použití inline funkcí

```
static inline void handle_uart_rx(uint8_t data) __attribute__((always_inline));

void USART1_IRQHandler(void) {
    if (USART1->SR & USART_SR_RXNE) {
        handle_uart_rx(USART1->DR); // Inline →zadny call overhead
    }
}
```

3. Vynechat operace v plovoucí řádové čátŕce

```
// Spatne
void ADC_IRQHandler(void) {
    float voltage = ADC1->DR * 3.3f / 4096.0f; // Pomale!
}

// Dobre
void ADC_IRQHandler(void) {
    uint16_t raw = ADC1->DR;
    // Konverze v main loop nebo pouzit fixed-point
}
```

ICSR (Interrupt Control and State Register)

ICSR (0xE000ED04) poskytuje informace o stavu výjimek a umožňuje softwarovou kontrolu:

```
#define SCB_ICSR (*(volatile uint32_t*)0xE000ED04)

// Bits [8:0] - VECTACTIVE: Aktualne aktivni exception number
// Bits [21:12] - VECTPENDING: Nejvyssi pending exception number
// Bit [22] - ISRPENDING: Indikuje pending interrupt (bez NMI/Faults)
// Bit [23] - ISRPREEMPT: Pending exception preempts current
// Bit [25] - PENDSTCLR: Clear SysTick pending (write 1)
// Bit [26] - PENDSTSET: Set SysTick pending (write 1)
// Bit [27] - PENDSVCLR: Clear PendSV pending (write 1)
// Bit [28] - PENDSVSET: Set PendSV pending (write 1)
// Bit [31] - NMIPENDSET: Set NMI pending (write 1)
```

ICSR - Praktické příklady

Zjištění aktuálního IRQ:

```
uint32_t icsr = SCB->ICSR;
uint32_t active = icsr & 0x1FF;

if (active == 0) {
    printf("Thread mode\n");
} else if (active < 16) {
    printf("System exception: %lu\n", active);
} else {
    printf("External IRQ: %lu\n", active - 16);
}
```

Kontrola pending interrupt:

```
bool has_pending_interrupt(void) {
   return (SCB->ICSR & SCB_ICSR_ISRPENDING_Msk) != 0;
}
```

ITM (Instrumentation Trace Macrocell)

ITM umožňuje non-intrusive debug output přes SWO (Serial Wire Output):

Výhody:

- **Bez UART** – nevyžaduje extra pin, jde přes debug interface
- **Rychlé** – minimální overhead (několik cyklů)
- **Printf-style** – lze použít pro trace, timing, events
- **32 portů** – oddělené kanály pro různé subsystémy

Konfigurace (nutné pro ITM):

```
// 1. Enable TRCENA (Trace enable) v DWT
CoreDebug->DEMCR |= CoreDebug_DEMCR_TRCENA_Msk;
// 2. Unlock ITM
ITM->LAR = 0xC5ACCE55;
// 3. Enable ITM, set ATBID
ITM->TCR = (1 << ITM_TCR_ITMENA_Pos) | // Enable ITM
           (1 << ITM_TCR_SYNCENA_Pos); // Enable sync packets
// 4. Enable stimulus port 0
ITM->TER = 1; // Enable port 0
```

ITM - Praktické použití

Printf přes ITM:

```
void ITM_SendChar(char ch) {
   // Wait until ready
   while (!(ITM->PORT[0].u32 & 1));
   ITM->PORT[0].u8 = ch;
}

int _write(int file, char *ptr, int len) {
   for (int i = 0; i < len; i++) {
      ITM_SendChar(ptr[i]);
   }
   return len;
}

// Nyni printf() jde pres SWO
printf("IRQ latency: %lu cycles\n", cycles);
```

Trace vstupu/výstupu z ISR:

```
void EXTI0_IRQHandler(void) {
    // Entry marker (port 1)
   ITM->PORT[1].u32 = 1;

```

```c
    // Obsluha...

    // Exit marker
    ITM->PORT[1].u32 = 0;
}
```

ITM - Měření latence ISR pomocí DWT

```c
volatile uint32_t isr_entry_time;
volatile uint32_t isr_duration;

void EXTI0_IRQHandler(void) {
    uint32_t entry = DWT->CYCCNT; // Read cycle counter

    EXTI->PR = (1 << 0);
    handle_button_press();

    uint32_t exit = DWT->CYCCNT;
    isr_duration = exit - entry;

    ITM->PORT[2].u32 = isr_duration; // Output pres ITM
}

// DWT cycle counter konfigurace:
CoreDebug->DEMCR |= CoreDebug_DEMCR_TRCENA_Msk;
DWT->CTRL |= DWT_CTRL_CYCCNTENA_Msk; // Enable counter

// V OpenOCD/GDB:
// monitor tpiu config internal - uart off 168000000
// monitor itm port 0 on
```

4. GPIO a přerušení (EXTI)

Vlastnosti GPIO na STM32F4

Základní parametry:

- **16 pinů na port** (PA0-PA15, PB0-PB15, ...), **až 9 portů** (GPIOA-GPIOI) = 144 pinů
- **4 režimy**: Input, Output, Alternate Function, Analog
- **Programovatelná rychlost**: Low (8 MHz), Medium (50 MHz), High (100 MHz), Very High (180 MHz)
- **Pull-up/pull-down** rezistory (typicky 40 kΩ)

Elektrické parametry:

- **Napěťové úrovně** (při VDD = 3.3V):
  - VIL (Input Low): max 0.99V (0.3×VDD)
  - VIH (Input High): min 2.31V (0.7×VDD)
  - VOL (Output Low): max 0.4V @ 8mA
  - VOH (Output High): min VDD - 0.4V @ 8mA

Proudové limity:

- **Max proud per pin**: 25mA (Very High speed)
- **Max součet proudů všech GPIO**: 120mA
- **Typické použití**: 2-8mA (Low/Medium speed)
- **Pull-up/pull-down**: ~80µA @ 3.3V (R ≈ 40kΩ)

Další vlastnosti GPIO

5V tolerantní piny (FT):

- Mohou snést **5V na vstupu** i když MCU běží na 3.3V
- Užitečné pro komunikaci s 5V logikou (Arduino, staré periferie)
- **Pouze některé piny jsou FT** (viz datasheet, značeno "FT")
- **Pozor**: Výstup je stále 3.3V (není 5V output!)

Ochranné obvody:

- **ESD ochrana** (Human Body Model: 2kV)
- **Vstupní diody** k VDD a GND (clamping)
- **Schmitt trigger** na vstupech (hystereze proti šumu)

Clock enable:

- GPIO vyžaduje **zapnutí hodin** přes RCC (AHB1ENR)
- **Po resetu jsou hodiny vypnuté** → GPIO nefunguje!

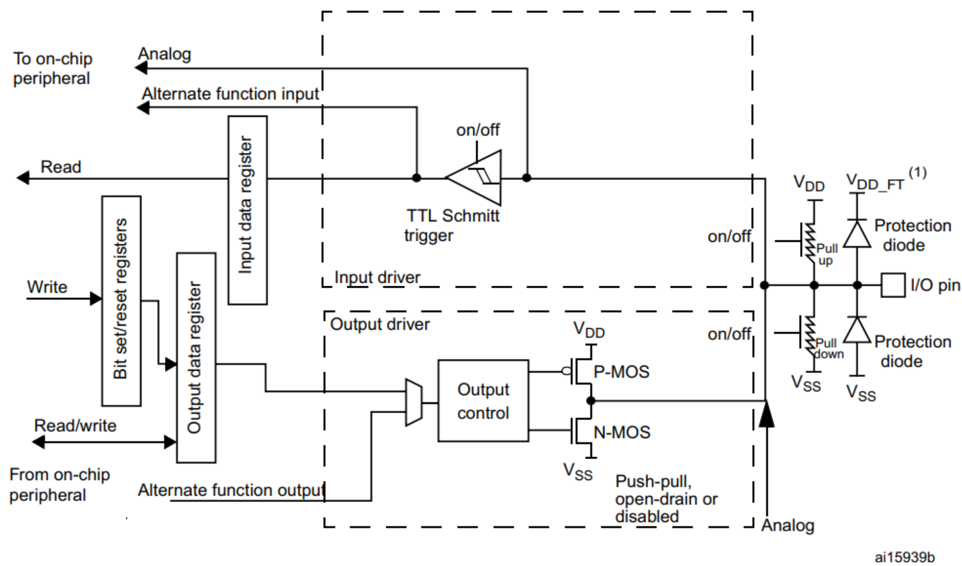


Figure 12: Blokové schéma GPIO procesorů STM32

Blokové schéma GPIO periferie

GPIO Registry (CMSIS struktura)

```c
typedef struct {
   volatile uint32_t MODER; // Mode register (offset 0x00)
   volatile uint32_t OTYPER; // Output type (offset 0x04)
   volatile uint32_t OSPEEDR; // Output speed (offset 0x08)
   volatile uint32_t PUPDR; // Pull-up/pull-down (offset 0x0C)
   volatile uint32_t IDR; // Input data (RO) (offset 0x10)
   volatile uint32_t ODR; // Output data   (offset 0x14)
   volatile uint32_t BSRR; // Bit set/reset (WO) (offset 0x18)
   volatile uint32_t LCKR; // Configuration lock (offset 0x1C)
   volatile uint32_t AFR[2]; // Alternate function (offset 0x20-0x24)
} GPIO_TypeDef;

// Bazove adresy (AHB1 bus, 0x400 offset mezi porty)
#define GPIOA_BASE 0x40020000UL
#define GPIOB_BASE 0x40020400UL
#define GPIOC_BASE 0x40020800UL
// ...
#define GPIOA ((GPIO_TypeDef*)GPIOA_BASE)
```

MODER (Mode Register) – režimy pinu

Každý pin má **2 bity** v MODER (celkem 32 bitů pro 16 pinů):

MODER[2i+1:2i]	Režim	Popis
00	Input	Výchozí po resetu, high impedance
01	Output	Push-pull nebo open-drain
10	Alternate	Funkce periferie (UART, SPI, ...)

MODER[2i+1:2i]	Režim	Popis
11	Analog	Připojeno na ADC/DAC, digitální část vypnuta

```
// PA5 jako Output
GPIOA->MODER &= ~(0x3 << (5 * 2)); // Clear bits [11:10]
GPIOA->MODER |= (0x1 << (5 * 2)); // Set to 01 (Output)

// PA9 jako Alternate Function (USART1_TX)
GPIOA->MODER &= ~(0x3 << (9 * 2));
GPIOA->MODER |= (0x2 << (9 * 2)); // Set to 10 (AF)
```

OTYPER (Output Type Register)

Určuje typ výstupu (1 bit na pin):

OTYPER[i]	Typ	Zapojení	Použití
0	Push-Pull	Active high + low	LED, CMOS logic
1	Open-Drain	Pouze pull-down	I2C, 5V tolerantní

```
// PA5 Push-Pull
GPIOA->OTYPER &= ~(1 << 5); // 0 = Push-Pull

// PB6 Open-Drain (pro I2C SCL)
GPIOB->OTYPER |= (1 << 6); // 1 = Open-Drain
```

OSPEEDR (Output Speed Register)

Rychlost slew rate (2 bity na pin):

OSPEEDR[2i+1:2i]	Rychlost	f_max	I_max	Použití
00	Low	8 MHz	2 mA	GPIO obecné
01	Medium	50 MHz	8 mA	SPI, I2C
10	High	100 MHz	25 mA	SDIO, Ethernet
11	Very High	180 MHz	25 mA	High-speed interfaces

Vyšší rychlost = vyšší spotřeba + více EMI (elektromagnetické rušení)

```
// PA5 Medium speed (pro LED - staci)
GPIOA->OSPEEDR &= ~(0x3 << (5 * 2));
GPIOA->OSPEEDR |= (0x1 << (5 * 2)); // 01 = Medium

// PA12 Very High (pro USB_DP)
GPIOA->OSPEEDR |= (0x3 << (12 * 2)); // 11 = Very High
```

PUPDR (Pull-Up/Pull-Down Register)

Interní rezistory (2 bity na pin, typicky 40 kΩ):

PUPDR[2i+1:2i]	Konfigurace	Použití
00	No pull-up/down	Floating (default)
01	Pull-up	Tlačítko active-low, I2C (+ ext)
10	Pull-down	Tlačítko active-high
11	Reserved	Nepoužívat

```
// PA0 s pull-down (tlacitko na VDD)
GPIOA->PUPDR &= ~(0x3 << (0 * 2));
GPIOA->PUPDR |= (0x2 << (0 * 2)); // 10 = Pull-down

// PA13 s pull-up (SWDIO debug)
GPIOA->PUPDR &= ~(0x3 << (13 * 2));
GPIOA->PUPDR |= (0x1 << (13 * 2)); // 01 = Pull-up
```

IDR a ODR (Input/Output Data Registers)

IDR (Input Data Register) – read-only:

```
// Cteni stavu pinu PA0
if (GPIOA->IDR & (1 << 0)) {
   // Pin je HIGH
}

// Cteni celeho portu (16 bitu)
uint16_t port_state = GPIOA->IDR & 0xFFFF;
```

ODR (Output Data Register) – read/write:

```
// Nastavit PA5 na HIGH
GPIOA->ODR |= (1 << 5);

// Nastavit PA5 na LOW
GPIOA->ODR &= ~(1 << 5);

// Toggle PA5 (POZOR: neni atomicke!)
GPIOA->ODR ^= (1 << 5);
```

Problém s ODR: Read-Modify-Write → race condition s ISR/DMA

BSRR (Bit Set/Reset Register) – atomické operace

BSRR je 32-bitový write-only registr:

- **Bity [15:0]** → Set (nastaví pin na 1)
- **Bity [31:16]** → Reset (nastaví pin na 0)

```
// Set PA5 (atomicky)
GPIOA->BSRR = (1 << 5);

// Reset PA5 (atomicky)
GPIOA->BSRR = (1 << (5 + 16));
```

Výhoda oproti ODR:

- **Hardwarově atomické** – žádné race conditions, **rychlejší**, **bezpečné v ISR**

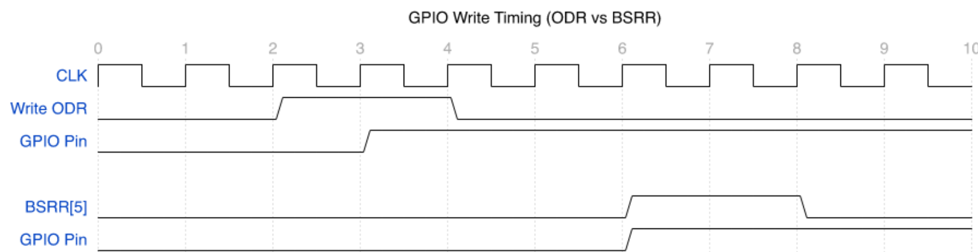


Figure 13: Časový průběh čtení a nastavení bitů GPIO

AFR (Alternate Function Register)

Pro připojení pinu k periferii (UART, SPI, TIM, ...):

- **AFR[0]** (AFRL) → piny 0-7 (4 bity na pin)
- **AFR[1]** (AFRH) → piny 8-15 (4 bity na pin)

AF číslo	Funkce (příklad PA9/PA10)
AF0	System (SWDIO, SWD)
AF1	TIM1, TIM2
AF4	I2C1, I2C2, I2C3
AF5	SPI1, SPI2
AF7	USART1, USART2, USART3
AF10	USB OTG FS
AF12	SDIO

```
// PA9 jako USART1_TX (AF7)

// Alternate function
GPIOA->MODER |= (0x2 << (9 * 2));

// Clear AF bits
GPIOA->AFR[1] &= ~(0xF << ((9-8)*4));

// AF7 = USART1
GPIOA->AFR[1] |= (0x7 << ((9-8)*4));
```

Kompletní příklad: Inicializace LED

```
void init_led_pa5(void) {
   // 1. Zapnout hodiny pro GPIOA (RCC - AHB1)
   RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN; // Bit 0

   // 2. Mode: Output
   GPIOA->MODER &= ~(0x3 << (5 * 2)); // Clear
   GPIOA->MODER |= (0x1 << (5 * 2)); // Output (01)

   // 3. Type: Push-Pull
   GPIOA->OTYPER &= ~(1 << 5);  // Push-Pull (0)

   // 4. Speed: Medium (dostatecne pro LED)
   GPIOA->OSPEEDR &= ~(0x3 << (5 * 2));
```

```
14     GPIOA->OSPEEDR |= (0x1 << (5 * 2)); // Medium (01)
15
16     // 5. Pull: No pull-up/down (LED ma vlastni rezistor)
17     GPIOA->PUPDR &= ~(0x3 << (5 * 2)); // No pull (00)
18
19     // 6. Vychozi stav: LOW
20     GPIOA->BSRR = (1 << (5 + 16)); // Reset bit
21 }
22
23 void led_on(void) { GPIOA->BSRR = (1 << 5); }
24 void led_off(void) { GPIOA->BSRR = (1 << (5 + 16)); }
```

GPIO přerušení (EXTI)

GPIO piny mohou generovat **externí přerušení** pomocí **EXTI** (External Interrupt/Event Controller):

Princip:

- **16 EXTI linií** (EXTI0-EXTI15) pro GPIO piny
- Každá linie může být připojena k jednomu pinu z různých portů
  - EXTI0 → PA0, PB0, PC0, … (vybere se jeden)
  - EXTI1 → PA1, PB1, PC1, … (vybere se jeden)
  - atd.

Konfigurace:

1. **SYSCFG_EXTICR** – výběr portu pro EXTI linii
2. **EXTI_IMR** – povolení přerušení (Interrupt Mask Register)
3. **EXTI_RTSR/FTSR** – trigger na rising/falling edge
4. **NVIC** – povolení IRQ handleru

EXTI mapování na NVIC

EXTI linie jsou mapovány na **IRQ čísla** v NVIC:

EXTI linie	IRQ Handler	IRQn	Popis
EXTI0	EXTI0_IRQHandler	6	Pin x0 (PA0, PB0, …)
EXTI1	EXTI1_IRQHandler	7	Pin x1 (PA1, PB1, …)
EXTI2	EXTI2_IRQHandler	8	Pin x2
EXTI3	EXTI3_IRQHandler	9	Pin x3
EXTI4	EXTI4_IRQHandler	10	Pin x4
EXTI5-9	EXTI9_5_IRQHandler	23	Piny x5-x9 (sdílený)
EXTI10-15	EXTI15_10_IRQHandler	40	Piny x10-x15 (sdílený)

Vektor tabulka obsahuje adresy těchto handlerů → při přerušení CPU automaticky skočí na správný handler.

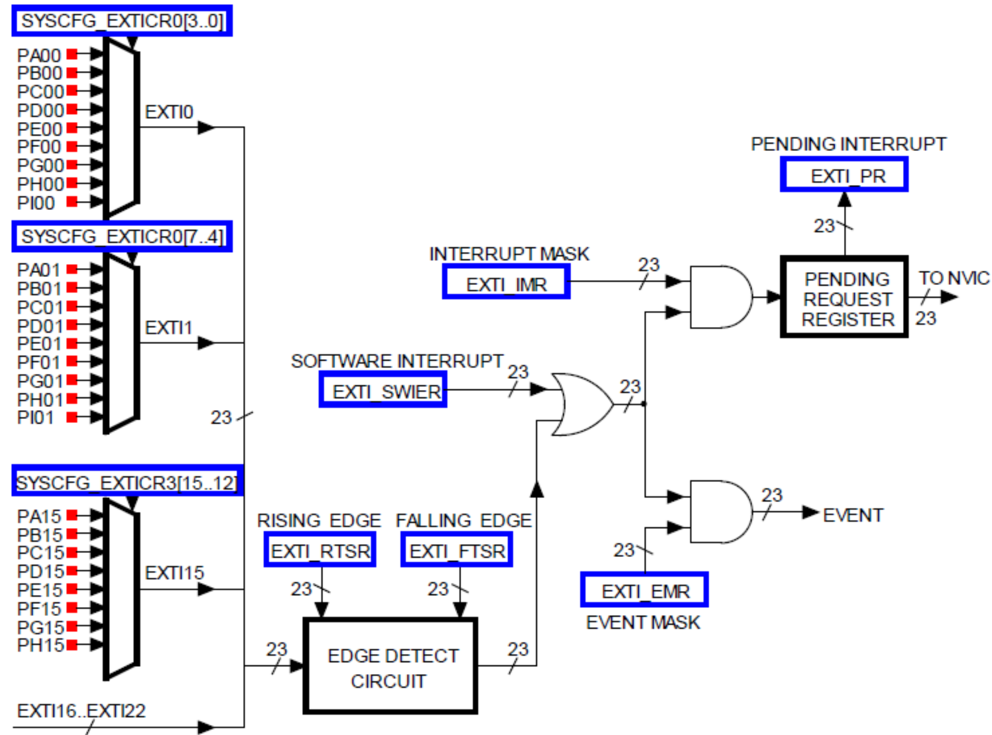


Figure 14: Blokové schéma systému externího přerušení EXTI

Propojení: NVIC → Vektor tabulka → ISR

```
GPIO Pin PA0 stisknut↓

EXTI0 detekuje rising edge↓

EXTI0 nastaví pending flag v NVIC↓

NVIC najde IRQ číslo: 6↓

NVIC řčpete vektor na adrese: VTOR + (6+16) * 4 = 0x0000_0058↓

Najde pointer: EXTI0_IRQHandler↓

CPU čskoí na adresu EXTI0_IRQHandler funkce↓

Vykoná se švá kód:
   void EXTI0_IRQHandler(void) {
      if (EXTI->PR & (1<<0)) {
         EXTI->PR = (1<<0); // Clear flag
         toggle_led();
      }
   }
```

IRQ číslo z NVIC → index do tabulky → **adresa ISR**

EXTI Registry (STM32F4)

```c
typedef struct {
    volatile uint32_t IMR; // Interrupt Mask Register
    volatile uint32_t EMR; // Event Mask Register
    volatile uint32_t RTSR; // Rising Trigger Selection
    volatile uint32_t FTSR; // Falling Trigger Selection
    volatile uint32_t SWIER; // Software Interrupt Event Register
    volatile uint32_t PR; // Pending Register
} EXTI_TypeDef;

#define EXTI ((EXTI_TypeDef*)0x40013C00)
```

Vlastnosti:

- **20 edge detektorů** (EXTI0-15 pro GPIO, 16-19 pro PVD, RTC, USB, atd.)
- **Nezávislý trigger** pro každou linii (rising, falling, obojí)
- **Individuální maskování** (IMR - lze zakázat jednotlivé linie)
- **Pending bit** pro každou linii (PR - indikuje čekající přerušení)
- **Software trigger** (SWIER - lze vyvolat přerušení softwarově)

EXTI Software Trigger

EXTI umožňuje **softwarově vyvolat přerušení** bez fyzického GPIO signálu:

```c
// Vyvolat EXTI0 preruseni softwarove
EXTI->SWIER |= (1 << 0); // Set software interrupt bit

// Hardware automaticky:
// 1. Nastavi EXTI->PR bit 0 (pending)
// 2. Vyvola EXTI0_IRQHandler (pokud je povoleno v IMR a NVIC)

// V ISR musite smazat pending flag normalne:
void EXTI0_IRQHandler(void) {
    if (EXTI->PR & (1 << 0)) {
        EXTI->PR = (1 << 0); // Clear pending
        // Obsluha...
    }
}
```

Použití: Testování ISR, simulace událostí, RTOS signalizace

Příklad: Tlačítko na PA0 s přerušením

```c
void init_button_interrupt(void) {
    // 1. Zapnout hodiny
    RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN;
    RCC->APB2ENR |= RCC_APB2ENR_SYSCFGEN;

    // 2. Konfigurace PA0 jako input s pull-down
    GPIOA->MODER &= ~(0x3 << 0); // Input mode
    GPIOA->PUPDR &= ~(0x3 << 0);
    GPIOA->PUPDR |= (0x2 << 0); // Pull-down

    // 3. EXTI0 pripojit na port A (PA0)
    SYSCFG->EXTICR[0] &= ~0xF; // Clear EXTI0
    SYSCFG->EXTICR[0] |= 0x0; // 0 = Port A
```

```c

    // 4. Povolit EXTI0, trigger na rising edge
    EXTI->IMR |= (1 << 0); // Unmask EXTI0
    EXTI->RTSR |= (1 << 0); // Rising edge
    EXTI->FTSR &= ~(1 << 0); // No falling edge

    // 5. Povolit EXTI0 v NVIC
    NVIC_SetPriority(EXTI0_IRQn, 5);
    NVIC_EnableIRQ(EXTI0_IRQn);
}
```

```c
void EXTI0_IRQHandler(void) {
    // Zkontrolovat pending flag
    if (EXTI->PR & (1 << 0)) {
        // Vymazat flag zapisem 1
        EXTI->PR = (1 << 0);

        // Obsluha tlacitka
        toggle_led();
    }
}
```

Příklad: Tlačítko s debouncing

Problém: Mechanické tlačítko generuje **zákmity** (bounce) při stisku/uvolnění → několik přerušení místo jednoho.

Řešení: Software debouncing pomocí časovače.

```c
#define DEBOUNCE_TIME_MS 50

volatile uint32_t last_interrupt_time = 0;

void init_button_with_debounce(void) {
    // GPIO + EXTI konfigurace (jako predchozi slide)
    init_button_interrupt();

    // SysTick pro casovani (1ms tick)
    SysTick_Config(SystemCoreClock / 1000);
}

volatile uint32_t millis = 0;

void SysTick_Handler(void) {
    millis++;
}
```

```c
void EXTI0_IRQHandler(void) {
    if (EXTI->PR & (1 << 0)) {
        EXTI->PR = (1 << 0); // Clear flag

        uint32_t current_time = millis;

        // Ignorovat interrupt pokud je prilis brzy
        if ((current_time - last_interrupt_time)
            > DEBOUNCE_TIME_MS) {

            last_interrupt_time = current_time;
```

```
        // Zpracovani tlacitka
        handle_button_press();
      }
    }
}

void handle_button_press(void) {
  static uint8_t led_state = 0;
  led_state = !led_state;

  if (led_state) {
    GPIOA->BSRR = (1 << 5); // LED ON
  } else {
    GPIOA->BSRR = (1 << (5+16)); // LED OFF
  }
}
```