Mikroprocesory

11. Číslicové zpracování signálů na MCU

Stanislav Vítek Katedra radioelektroniky České vysoké učení technické v Praze

Obsah přednášky

1. Číslicové zpracování signálu na MCU
2. FPU - jednotka pro operace v plovoucí řádové čárce
3. Příklad 1 - generátor harmonického signálu
4. Příklad 2 - vzorkování signálu s antialiasingovým filtrem
5. Příklad 3 - FFT analýza pro prediktivní údržbu
6. Příklad 4 - 10pásmový audio EQ

1. Číslicové zpracování signálu na MCU

Proč DSP na MCU?

Moderní senzorické a řídicí aplikace generují velké množství dat.

Přenášet všechna data do vyšších vrstev (PC, server, cloud) bývá:

- drahé (energie → zvlášť u baterií a IoT),
- pomalé,
- často nemožné (komunikace, bezpečnost),
- zbytečné (většina dat je nezajímavá).

Proto se stále více zpracování dělá přímo na MCU:

MCU se stává „edge procesorem“, který z dat extrahuje informaci, ne pouze data.

Výhody DSP přímo na MCU

1. Reálný čas – determinismus

MCU (Cortex-M) je deterministický systém:

- žádné OS vrstvy (RTOS je deterministické),
- žádné cache miss penalty (M0/M3),
- predikovatelné latence přerušení.

DSP úlohy vyžadují konstantní latenci (typicky maximálně 10–200 µs).

- řízení motorů nebo celých robotů
- odhady polohy (IMU fusion),
- audio efekty,

2. Nízká spotřeba energie

DSP pipeliny běžící na MCU mají spotřebu:

- 2–30 mA u M4
- 30–80 mA u M7 → oproti tomu mobilní SoC (ARM-A53 apod.): stovky mA.

To umožňuje:

- nositelná elektronika (wearables, např. noise cancelling pro sluchátka),
- bateriové IoT senzory,
- prediktivní údržba v průmyslu (vibrace motorů).

Příklad Accelerometrické vibrace monitorované MCU s RTOS, úloha 2–3 ms každých 20 ms → roční provoz na jednu baterii CR2032

Reálné projekty s DSP na MCU

Sluchátka

- ANC → filtr se 128–256 koeficienty, běží i na M4F, EQ → biquad filtry

Drony

- stabilizace IMU → filtr 1–8 kHz, řízení BLDC motorů

IoT bezpečnostní senzory

- analýza zvuku (rozbití skla), vibrace v potrubí, detekce motorických anomálií

Průmyslové měření

- digitální filtry, FFT, korelace, detekce proudových špiček

Dilema: FPU ano nebo ne?

Dosud jsme FPU spíše nepoužívali, ostatně projekty na MCU bývají hodně binární :-)

Proč spíš ano?

- Nižší riziko přetečení
- Lepší přesnost v nízkých amplitudách
- FFT/filtry implementované v 32bitovém typu `float` jsou obvykle rychlejší než operace v pevné řádové čárce, pokud má MCU FPU
- Kompilátor lehce vektoruje kód (`VMLA.F32`, `VMUL.F32`)

Proč spíš ne?

- Vyšší spotřeba - FPU jednotka musí běžet, pipeline má více tranzistorů
  - instrukce často trvají déle než v datovém typu `int`
- Pokud ISR používá FPU, ukládá se velký kontext (`S0`–`S31`) → drahé!

Navíc ...

FPU není vždy dostupné

- Cortex-M0/M0+/M3 nemají FPU
- některé M4/M33 existují i bez FPU varianty
- levné čipy často FPU vypínají při low-power režimech

Pevná řádová čárka může být:

- rychlejší (SIMD MAC operace)
- úspornější (paměť, energie)
- determinističtější

Nejefektivnější DSP na MCU obvykle kombinuje obě aritmetiky

- filtry a DSP v pevné řádové čárce
- vysoká přesnost nebo AI části v plovoucí řádové čárce

Číselné formáty pro přesné výpočty - IEEE 754

`float` / `float32`

- 32 bitů → 1 bit znaménko + 8 bitů exponent (bias 127) + 23 bitů mantisa
- Nejběžnější formát s plovoucí řádovou čárkou, podpora v M4F, M7F, M33F

`double`

- 64 bitů → 1 bit znaménko + 11 bitů exponent (bias 1023) + 52 bitů mantisa
- Žádná HW podpora, velmi pomalé v SW emulaci

`half` / `float16`

- 16 bitů → 1 bit znaménko + 5 bit exponent + 10bit mantisa
- nativní podpora zatím jen v některých DSP akcelerátory v Cortex-M55/Helium

`bfloat16` - mimo IEEE 754, vyvinutý Google pro jejich TPU (Tensor Processor Unit)

- 16 bitů → 1 bit znaménko + 8 bit exponent + 7bit mantisa

Speciální float hodnoty

Strojové epsilon (ε)

- nejmenší číslo, které přičtené k 1.0 ještě změní výsledek.

```
ε = 1.19209290e−7   (≈ 1.19 × 10⁻⁷)

1.0 + 1.19e−7   →  1.000000119
1.0 + 1e−8      →  stále 1.0     (už se nerozliší)
```

Zajímavá čísla

- ±0: Exponent = 0, Mantissa = 0
- ±∞: Exponent = 255, Mantissa = 0
- NaN: Exponent = 255, Mantissa ≠ 0

Krok mezi čísly

- kolem 1.0 ≈ 1.19e−7,
- kolem 1000.0 ≈ 0.000122,
- kolem 1e6 ≈ 0.125

Na co si dát pozor u práce s float32

Porovnávání čísel:

- `float32` čísla bývají nepřesná, běžné např.

```c
#include <math.h>

bool nearly_equal(float a, float b, float eps)
{
    return fabsf(a - b) < eps;
}
```

Operace s čísly s velmi rozdílným exponentem:

- při operaci se převádí operandy na stejné exponenty, může dojít ke ztrátě mantisy

`1 000 000.0f + 1.0f → stále 1 000 000.0f`

Kolik stojí float/double bez FPU?

- MCU bez FPU používají knihovnu libgcc pro emulaci IEEE754:

Operace	soft float	soft double)	Poznámka
sčítání	~80–120 cyklů	~130–200 cyklů	několik funkcí + normalizace
násobení	~120–200 cyklů	~200–350 cyklů	emulace mantisy + exponentu
dělení	300–800 cyklů	600–1500 cyklů	nejdražší – iterativní aproximace
sqrt	~600–1200 cyklů	ještě více	často >1 µs

Aritmetika s pevnou řádovou čárkou (fixed-point arithmetic)

MCU s jádry Cortex-M0/M3/M4/M7 se často spoléhají na celočíselnou aritmetiku pro rychlost a předvídatelnost

- absence složitých FPU – Floating-Point Unit, i když M4/M7 volitelně FPU mají
- způsob, jak reprezentovat reálná čísla pomocí standardní celočíselné aritmetiky.

Formát Q (Q Format)

Q formát je nejčastější způsob reprezentace reálných čísel s pevnou řádovou čárkou.

Q_n → celé číslo, kde n bitů je vyhrazeno pro zlomkovou část (za řádovou čárkou).

Definice příslušných datových typů součástí CMSIS Core

Formát Qm.n

m: počet bitů pro celočíselnou část (včetně znaménkového bitu, pokud se používá dvojkový doplněk).

n: počet bitů pro zlomkovou část.

Hodnota reprezentovaná v Qn:

$$x = \frac{X_{int}}{2^n}$$

Příklady:

- Q7 → rozsah –1 … +0.992, krok 2^-7

- Q15 → rozsah –1.0 … +0.99997, krok 2^-15
- Q31 → rozsah –1 … +0.999999999, používaný pro ARM CMSIS-DSP

  Z hlediska definice se jedná o typy Q1.7, Q1.15 a Q1.31 → znaménkový bit

Definice datových typů a struktura hlavičkových souborů

Hlavní include: `arm_math_types.h` ← zde jsou `q7_t` / `q15_t` / `q31_t`

```
CMSIS/
 └── DSP/
      └── Include/
           └── arm_math_types.h

arm_math.h
  ├── arm_math_types.h
  ├── arm_common_tables.h
  ├── dsp/*.h
  └── ...
```

Příklad definic:

```
typedef int8_t   q7_t;      // 7bitový fixed-point (Q0.7)
typedef int16_t  q15_t;     // 15bitový fixed-point (Q1.15)
typedef int32_t  q31_t;     // 31bitový fixed-point (Q1.31)
typedef int64_t  q63_t;     // pomocný akumulační typ

typedef float    float32_t;
typedef double   float64_t;
```

Převody mezi formáty

`float` → `Qn`

```
q15_t x_q15 = (int16_t)(x_float * 32768.0f);
```

Mezi dvěma Q formáty

Musíme posunout bitovou řádovou čárku:

$$Qm_1.n_1 \to Qm_2.n_2 : X_2 = X_1 \cdot 2^{(n_1 - n_2)}$$

Příklad:

- Q15 → Q31 = « 16
- Q31 → Q15 = » 16

Přetečení (Overflow)

Přetečení nastává, když výsledek aritmetické operace (zejména sčítání nebo násobení) překročí maximální rozsah daný počtem bitů.

Chování při přetečení (Wrap-Around):

- Standardní celočíselná aritmetika (jako v jazyce C) typicky řeší přetečení otočením (wrap-around).
- Např. pro 16bitové číslo:
$$\text{MAX} + 1 \rightarrow \text{MIN}$$

Toto vede k náhlé a obrovské chybě v signálu (např. kladné číslo se stane velkým záporným), což může způsobit nestabilitu filtru nebo chybné řízení.

Saturace (Saturation)

Metoda řízení přetečení, která je klíčová pro DSP → místo otočení se výsledek operace omezí (saturuje) na maximální (nebo minimální) hodnotu daného formátu

- Pokud je výsledek větší než X_{max}, nastaví se na X_{max}.
- Pokud je výsledek menší než X_{min}, nastaví se na X_{min}.

Výhoda: Saturace zaručuje, že chyba v signálu je omezená a předvídatelná.

Instrukce na Cortex-M

Jádra Cortex-M4/M7 obsahují dedikované instrukce pro saturující sčítání/odčítání (např. SSAT, QADD, QDADD) a násobení (součást SIMD/DSP instrukcí)

```
int32_t y = __SSAT(x, 16); // saturace na 16bit rozsah
```

Sčítání/Odčítání:

- Přímé celočíselné sčítání/odčítání (pokud mají obě čísla stejný n).

`__QADD16` – SIMD instrukce

- Udělá dvě saturující 16bitová sčítání v jednom taktu
- Pracuje vždy pouze nad jedním 32bit slovem, kde jsou dvě hodnoty Q15

`| high16: q15_t | low16: q15_t |`

```
int32_t A = (a1 << 16) | (uint16_t)a0;
int32_t B = (b1 << 16) | (uint16_t)b0;

// C obsahuje [a1+b1 | a0+b0] se saturací
int32_t C = __QADD16(A, B);
```

`arm_add_q15` – knihovní DSP funkce

- Nachází se v CMSIS-DSP
- Implementuje vektorové sčítání dvou `Q15` polí
- Funguje pro libovolnou délku signálu
- Pokud je dostupná SIMD operace, použuje ji

Příklad:

```
q15_t a[3] = {30000, 20000, -32000};
q15_t b[3] = {10000, 20000, -20000};
q15_t r[3];

arm_add_q15(a, b, r, 3);
// r = {32767, 32767, -32768}   ← saturováno
```

Násobení

- Násobení dvou Q_n čísel dává Q_{2n} (bitově dvojnásobnou) přesnost.
- Pro zachování původní přesnosti Q_n je nutné výsledek posunout doprava o n bitů (ekvivalent dělení 2^n):

$$\text{Výsledek} = (\text{Op1} \cdot \text{Op2}) \gg n$$

Cortex-M4/M7/M33 mají:

- `SMLAD` / `__SMLAD` – Dual 16-bit multiply + accumulate (bez saturace)
- `SMLSD` – mix + saturace do 32 bitů
- `SMUAD` / `__SMUAD` – Dual 16-bit multiply (bez akumulace)

Jak `__SMLAD` funguje

Operátory jsou 32bit slova, v nich jsou dve Q15 čísla:

```
a = [ a_hi | a_lo ]   // dvě int16_t čísla
b = [ b_hi | b_lo ]
```

Instrukce provede v 32 bitech, bez saturace.

```
result = a_lo * b_lo + a_hi * b_hi + acc;
```

Cortex-M4/M7 mají DSP rozšíření založené na MAC operacích (Multiply-Accumulate) pro akceleraci filtrů.

CMSIS-DSP většinou dělá:

- SIMD multiply + accumulate (dot product)
- Scalar saturating multiply (arm_mult_q15) pro jednotlivé prvky

Příklad

Spočítej $(x1 \times y1 + x2 \times y2)$ v jednom cyklu

```
q15_t x1 = 10000;   // 0.305 v Q15
q15_t x2 = -5000;   // -0.152 v Q15

q15_t y1 = 12000;   // 0.366
q15_t y2 = 3000;    // 0.091

// Zabal dva Q15 prvky do 32-bit slova:
uint32_t A = __PKHBT(x1, x2, 16);   // x1 = low, x2 = high
```

```c
uint32_t B = __PKHBT(y1, y2, 16);

int32_t acc = 0;

// SIMD: dvě násobení + sčítání, sum je v Q30 (protože Q15 × Q15)
int32_t sum = __SMLAD(A, B, acc);

// výsledek jako Q15:
int32_t q15res = sum >> 15;
```

Jak vypadá násobení v CMSIS-DSP?

```c
void arm_mult_q15(
    const q15_t * pSrcA,
    const q15_t * pSrcB,
    q15_t * pDst,
    uint32_t blockSize)
{
    for (i=0; i<blockSize; i++) {
        // Q15 × Q15 = Q30 → >>15 → Q15
        q31_t product = ((q31_t)pSrcA[i] * pSrcB[i]) >> 15;
        pDst[i] = __SSAT(product, 16);
    }
}
```

K žádné SIMD instrukci zde nedochází — je to skalární implementace

2. FPU - jednotka pro operace v pohyblivé řádové čárce

FPU na STM32 platformách

Cortex-M4F: FPv4-SP (Single Precision)

- 32x 32-bit S-registry (`S0` - `S31`)
- Také přístupné jako 16x 64-bit D-registry (`D0` - `D15`)
- `FPSCR` - Floating Point Status and Control Register

Cortex-M7F: FPv5 (Single + Double Precision)

- Stejné registry + podpora double-precision
- Lepší výkon díky pipeline optimalizaci

Flagy pro kompilaci

- bez správných flagů při kompilaci bude použita SW emulace, i když je FPU k dispozici

```
-mfloat-abi=hard          # Hardware FPU ABI
-mfpu=fpv4-sp-d16         # M4F FPU type
-mfpu=fpv5-sp-d16         # M7F single precision
-mfpu=fpv5-d16            # M7F double precision
```

Ověření správné kompilace a zapojení FPU

```
void test_fpu_usage(void) {
    float a = 1.0f, b = 2.0f, c;
    c = a + b;  // Mělo by generovat VADD.F32
}
```

Assembler by měl obsahovat funkci `vadd.f32`

```
vadd.f32 s2, s0, s1
```

Pokud obsahuje skok na emulovanou funkci, FPU se nepoužije

```
bl __aeabi_fadd
```

FPU registry

Cortex-M4F/M7F/M33F obsahují FPU typu FPv4-SP-D16 nebo FPv5

Floating-point registry S0-S31

- 32 `float32` registrů `S0` - `S31`, lze párovat do `D0` – `D15` (64bit)

Rozdělení:

- S0–S15: low 16 registrů (FPv4-SP-D16)
- S16–S31: high 16 registrů (ne každé MCU je má – závislé na implementaci)

FPSCR – Floating Point Status and Control Register

- flagy výjimek (Invalid, ZeroDiv, Overflow, Underflow, Inexact)
- RMode – rounding mode (default: Round-to-Nearest)

FPCCR – Floating-Point Context Control Register

`ASPEN` – Automatic State Preservation Enable → HW automaticky ukládá/obnovuje FPU registry při ISR

`LSPEN` – Lazy State Preservation Enable → registr FPU se ukládají „líně“ = až když ISR/funkce skutečně použije FPU

`UFRDY` – FP ready status

MVFR0/MVFR1 – Media and Floating Point Feature Registers

Určují, co FPU vlastně umí:

- single-precision support
- double-precision support (u M4 nikdy, u M7/M33 občas)
- SIMD extensions (u M4 nejsou v FPU – jen integer SIMD)

Použití FPU během ISR

A) Lazy stacking ON (default)

- MCU odloží uložení FPU registrů, dokud nejsou použity
- první FPU instrukce v ISR vyvolá uložení `S0` – `S15`
- overhead cca 20–40 cyklů navíc, podle MCU

B) Lazy stacking OFF

- při vstupu do ISR se vždy uloží `S0` – `S15`
- overhead ~100+ cyklů (!)

Ukládá se minimálně `S0` – `S15` a `FPSCR` → 17×4B = 68 bajtů zásobníku

3. Příklad 1 - generátor harmonického signálu

Cíl

Generovat harmonický signál na MCU bez FPU a nebo s FPU

Požadavky

- Výstup na R2R DAC (GPIO->ODR), nejméně ve standardní audio kvalitě
- Cílem je maximálně flexibilita - možná změna frekvence

Obsah

1. Realizace generátoru pomocí LUT
2. Analýza ARM assembleru (MCU bez FPU)
3. Výpočetní náročnost se SW emulací bez FPU
4. Implementace pomocí pevné řádové čárky
5. Generátor harmonického signálu na MCU s FPU
6. Porovnání plovoucí vs. pevná řádová čárka

Realizace generátoru pomocí LUT

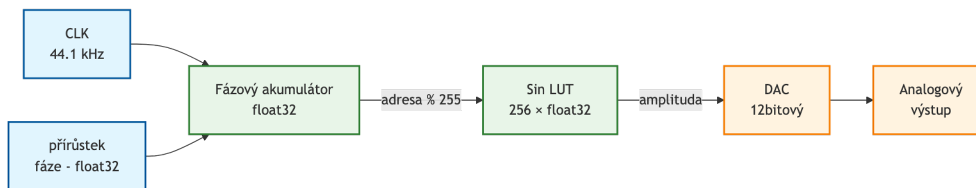

Figure 1: Blokové schéma generátoru

Jak to funguje?

- v `SRAM` budou připraveny předpočítané hodnoty `sin`
- čítač bude v `ISR` volat funkci `generate_sample` pro výpočet dalšího vzorku
- vypočtený `float` vzorek se převede na 12bitové `uint32_t` číslo a pošle na `GPIO`

1. C implementace

Jak velkou tabulku zvolit? Podle harmonického zkreslení.

```
64 vzorků   → THD ≈ -45 dB  (256B paměti)
256 vzorků  → THD ≈ -65 dB  (1 KB paměti) - přijatelné zkreslení (60dB ~ 10bit DAC)
1024 vzorků → THD ≈ -85 dB  (4 KB paměti) - dobrá kvalita (80dB ~ 13bit DAC)
static float sin_table[256];          // Lookup table v SRAM
static float phase = 0.0f;            // Aktuální fáze oscilátoru
static float phase_increment = 0.0f;  // Frekvence step
```

```
// Inicializace lookup table
void oscillator_init(float frequency, float sample_rate) {
    // Naplnění sin tabulky
    for (int i = 0; i < 256; i++) {
        sin_table[i] = sinf(2.0f * M_PI * i / 256.0f);
    }

    // Výpočet přírůstku fáze
    phase_increment = (2.0f * M_PI * frequency) / sample_rate;
}
void generate_sample(void) {
    // 1. Normalizace fáze na index tabulky
    uint32_t index = (uint32_t)((phase / (2*M_PI)) * 256);

    // 2. Lookup sin hodnoty
    float sample = sin_table[index & 255];

    // 3. Škálování pro 12-bit DAC (0-4095)
    uint32_t dac_value = (uint32_t)((sample + 1.0f) * 2048);

    // 4. Výstup na GPIO (R2R DAC)
    GPIOA->ODR = dac_value;

    // 5. Aktualizace fáze pro další vzorek
    phase += phase_increment;

    // 6. Limit fáze (zabránění přetečení)
    if (phase >= 2*M_PI) {
        phase -= 2*M_PI;
    }
}
```

X **Float operace bez FPU** = SW emulace X **Každá float operace** = library call (~20-50 cycles)

X **6× float operace** na vzorek = ~200+ taktů X **Limit fáze** = podmíněné větvení (pipeline stall)

2. Analýza ARM assembleru (MCU bez FPU)

STM32F103/F401/F042 - Cortex M3/M4 bez FPU

```
generate_sample:
    push    {r4, lr}                ; Function prologue: 2 cycles

    ; --- 1. PHASE NORMALIZATION: phase / (2*PI) ---
    ldr     r0, =phase              ; Load phase address: 1 cycle
    ldr     r0, [r0]                ; Load phase value: 1 cycle
    ldr     r1, =0x40C90FDB         ; Load 2*PI constant: 1 cycle
    bl      __aeabi_fdiv            ; SOFTWARE FLOAT DIVISION: 42 cycles!

    ; __aeabi_fdiv:                       ; 32-bit float division
    ; Exponent extraction: 8 cycles
    ; Mantissa alignment: 12 cycles
    ; Division algorithm: 18 cycles
    ; Result normalization: 4 cycles
    ; TOTAL: ~42 cycles

    ; --- 2. MULTIPLY BY 256 ---
    ldr     r1, =0x43800000         ; Load 256.0f: 1 cycle
```

```
    bl      __aeabi_fmul            ; SOFTWARE FLOAT MULTIPLY: 28 cycles!
    ; __aeabi_fmul:                       ; 32-bit float multiply
    ; Mantissa multiply: 16 cycles
    ; Exponent addition: 6 cycles
    ; Normalization: 6 cycles
    ; TOTAL: ~28 cycles

    ; --- 3. FLOAT TO INTEGER CONVERSION ---
    bl      __aeabi_f2uiz           ; SOFTWARE FLOAT→UINT: 24 cycles!

    ; __aeabi_f2uiz:                      ; Float to unsigned int
    ; Extract exponent: 4 cycles
    ; Shift mantissa: 8-16 cycles (variable)
    ; Range check: 4 cycles
    ; TOTAL: ~24 cycles (average)

    ; --- 4. ARRAY INDEX & LOOKUP ---
    and     r0, r0, #255            ; Mask to 8 bits: 1 cycle
    ldr     r1, =sin_table          ; Load table address: 1 cycle
    ldr     r0, [r1, r0, lsl #2]    ; sin_table[index]: 1 cycle (SRAM!)

    ; --- 5. DAC SCALING: (sample + 1.0) * 2048 ---
    ldr     r1, =0x3F800000         ; Load 1.0f: 1 cycle
    bl      __aeabi_fadd            ; SOFTWARE FLOAT ADD: 18 cycles!


    ; __aeabi_fadd:                       ; Float addition
    ; Exponent alignment: 8 cycles
    ; Mantissa add: 4 cycles
    ; Normalization: 6 cycles
    ; TOTAL: ~18 cycles

    ldr     r1, =0x45000000         ; Load 2048.0f: 1 cycle
    bl      __aeabi_fmul            ; SOFTWARE FLOAT MULTIPLY: 28 cycles!

    ; --- 6. FINAL FLOAT→INT ---
    bl      __aeabi_f2uiz           ; SOFTWARE FLOAT→UINT: 24 cycles!

    ; --- 7. GPIO WRITE ---
    ldr     r1, =0x4002000C         ; GPIOA_ODR address: 1 cycle
    str     r0, [r1]               ; Write to GPIO: 1 cycle

    ; --- 8. PHASE INCREMENT ---
    ldr     r0, =phase              ; Load phase address: 1 cycle
    ldr     r2, [r0]                ; Load current phase: 1 cycle
    ldr     r1, =phase_increment    ; Load increment address: 1 cycle
    ldr     r1, [r1]                ; Load increment value: 1 cycle
    mov     r0, r2                  ; Move phase to r0: 1 cycle
    bl      __aeabi_fadd            ; SOFTWARE FLOAT ADD: 18 cycles!
    ; --- 9. PHASE WRAP CHECK ---
    ldr     r1, =0x40C90FDB         ; Load 2*PI: 1 cycle
    bl      __aeabi_fcmpge          ; SOFTWARE FLOAT COMPARE: 12 cycles!
    cmp     r0, #0                  ; Check result: 1 cycle
    beq     .L_no_wrap              ; Branch if no wrap: 1 cycle

    ; Phase wrap subtract
    ldr     r0, =phase              ; 1 cycle
    ldr     r2, [r0]                ; 1 cycle
    ldr     r1, =0x40C90FDB         ; 1 cycle
    mov     r0, r2                  ; 1 cycle
```

```
    bl      __aeabi_fsub            ; SOFTWARE FLOAT SUBTRACT: 18 cycles!

.L_no_wrap:
    ldr     r1, =phase              ; Store result back: 1 cycle
    str     r0, [r1]                ; 1 cycle

    pop     {r4, pc}                ; Function epilogue: 2 cycles
```

Časová analýza funkce pro generování vzorků

```
Nejhorší případ:      215 hodinových taktů/vzorek
Nelepší případ:       180 hodinových taktů/vzorek
Realistický odhad:   ~195 hodinových taktů/vzorek
```

3. Výpočetní náročnost se SW emulací bez FPU

STM32F103 @ 72 MHz

```
CPU frekcence:            72 MHz
Hodinový takt/vzorek:     195 (průměr)
Max vzorkovací fr.:       72,000,000 / 195 = 369 kSPS

Sample_rate_8kHz:         195 cycles @ 8 kHz   = 2.2% CPU OK
Sample_rate_16kHz:        195 cycles @ 16 kHz  = 4.3% CPU OK
Sample_rate_22kHz:        195 cycles @ 22 kHz  = 5.9% CPU OK
Sample_rate_44kHz:        195 cycles @ 44 kHz  = 11.8% CPU OK
Sample_rate_48kHz:        195 cycles @ 48 kHz  = 13.0% CPU OK
```

MCU STM32F401 @ 84 MHz

```
CPU frekvence             84 MHz
Max vzorkovací fr.:       84,000,000 / 195 = 431 kSPS

Sample_rate_44kHz:        195 cycles @ 44 kHz  = 10.1% CPU OK
Sample_rate_96kHz:        195 cycles @ 96 kHz  = 21.7% CPU OK
Sample_rate_192kHz:       195 cycles @ 192 kHz = 43.4% CPU ⚠
```

4. Implementace pomocí pevné řádové čárky

```
typedef int16_t q15_t;
typedef int32_t q31_t;

// Lookup table v Q15 format
static q15_t sin_table_q15[256];       // Každý element 2 bytes
static q31_t phase_q31 = 0;            // 32-bit phase accumulator
static q31_t phase_increment_q31 = 0;  // Q31 phase step

// Inicializace Q15 tabulky
void oscillator_init_q15(uint32_t frequency, uint32_t sample_rate) {
    // Naplnění sin tabulky v Q15
    for (int i = 0; i < 256; i++) {
        float sin_val = sinf(2.0f * M_PI * i / 256.0f);
        sin_table_q15[i] = (q15_t)(sin_val * 32767.0f);
```

```
    }

    // Phase increment v Q31 (32-bit precision)
    // phase_increment = (2^31 * frequency) / sample_rate
    uint64_t temp = ((uint64_t)frequency << 31) / sample_rate;
    phase_increment_q31 = (q31_t)temp;
}
```

Funkce na generování vzorku

- pouze celočíselné operace

```
void generate_sample_q15(void) {
    // 1. Extract table index from upper 8 bits
    uint32_t index = (phase_q31 >> 24) & 0xFF;

    // 2. Lookup sin value (Q15)
    q15_t sample_q15 = sin_table_q15[index];

    // 3. Convert Q15 to 12-bit DAC (0-4095)
    // sample_q15 range: [-32768, +32767]
    // DAC range: [0, 4095]
    uint32_t dac_value = ((int32_t)sample_q15 + 32768) >> 4;

    // 4. GPIO output
    GPIOA->ODR = dac_value;

    // 5. Phase increment (32-bit addition!)
    phase_q31 += phase_increment_q31;

    // Automatic wrap at 2^32 - no explicit check needed!
}
```

```
generate_sample_q15:
    ; --- 1. EXTRACT TABLE INDEX ---
    ldr     r0, =phase_q31          ; Load phase address: 1 cycle
    ldr     r0, [r0]                ; Load phase value: 1 cycle
    lsr     r1, r0, #24             ; Shift right 24 bits: 1 cycle
    and     r1, r1, #255            ; Mask to 8 bits: 1 cycle

    ; --- 2. TABLE LOOKUP ---
    ldr     r2, =sin_table_q15      ; Load table address: 1 cycle
    ldrsh   r2, [r2, r1, lsl #1]    ; Load Q15 value: 1 cycle

    ; --- 3. Q15 TO DAC CONVERSION ---
    add     r2, r2, #32768          ; Add offset: 1 cycle
    lsr     r2, r2, #4              ; Divide by 16: 1 cycle

    ; --- 4. GPIO WRITE ---
    ldr     r1, =0x4002000C         ; GPIOA_ODR: 1 cycle
    str     r2, [r1]                ; Write: 1 cycle

    ; --- 5. PHASE INCREMENT ---
    ldr     r1, =phase_q31          ; Phase address: 1 cycle
    ldr     r2, =phase_increment_q31 ; Increment address: 1 cycle
    ldr     r2, [r2]                ; Load increment: 1 cycle
    add     r0, r0, r2              ; Add (32-bit): 1 cycle
    str     r0, [r1]                ; Store new phase: 1 cycle
```

```
    bx      lr                      ; Return: 1 cycle
```

Porovnání výkonnosti implementací

Implementation	Cycles/Sample	Max Sample Rate @ 72 MHz	Audio Performance
Float (no FPU)	195 cycles	369 kSPS	44kHz = 11.8% CPU
Q15 Fixed-Point	15 cycles	4.8 MSPS	44kHz = 0.9% CPU
Performance Gain	**13× faster**	**13× higher**	**13× lower CPU**

5. Generátor harmonického signálu na MCU s FPU

Cíl

- Implementace generátoru harmonického signálu s FPU
- Implementace v C zůstává stejná, pouze se zapojí FPU
- Běh `float` operací má na dedikované periferii významně větší efektivitu

Testované platformy

- **STM32F407** @ 168 MHz (Cortex-M4F s single-precision FPU)
- **STM32F746** @ 216 MHz (Cortex-M7F s single/double-precision FPU)
- **STM32H743** @ 480 MHz (Cortex-M7F s enhanced FPU)

Analýza ARM assembleru (MCU s FPU)

```
generate_sample:
    ; Phase normalization: phase / (2π)
    vldr.32 s0, [phase_addr]        ; 2 cycles - load phase
    vldr.32 s1, [two_pi_addr]       ; 2 cycles - load 2π constant
    vdiv.f32 s2, s0, s1            ; 14 cycles - hardware division!

    ; Table index: result * 256
    vldr.32 s3, [const_256_addr]    ; 2 cycles - load 256.0f
    vmul.f32 s4, s2, s3            ; 1 cycle - hardware multiply!

    ; Float to int conversion
    vcvt.u32.f32 s5, s4            ; 1 cycle - hardware conversion!
    vmov r0, s5                     ; 1 cycle - move to ARM register

    ; Table lookup (SRAM access)
    and r0, r0, #255                ; 1 cycle - mask to table size
    ldr r1, =sin_table              ; 1 cycle - table base address
    ldr s6, [r1, r0, lsl #2]        ; 2 cycles - load sin_table[index]

    ; DAC scaling: sample * 2048 + 2048
    vldr.32 s7, [const_2048_addr]   ; 2 cycles - load 2048.0f
    vmul.f32 s8, s6, s7            ; 1 cycle - hardware multiply!
    vadd.f32 s9, s8, s7            ; 1 cycle - hardware addition!
    ; Float to int for DAC output
    vcvt.u32.f32 s10, s9           ; 1 cycle - hardware conversion!
    vmov r2, s10                    ; 1 cycle - move to ARM register
```

```
    ; GPIO write
    ldr r3, =GPIOA_ODR              ; 1 cycle - GPIO base address
    str r2, [r3]                    ; 1 cycle - write to GPIO->ODR

    ; Phase increment
    vldr.32 s11, [phase_increment_addr] ; 2 cycles - load increment
    vadd.f32 s0, s0, s11            ; 1 cycle - hardware addition!

    ; Phase wrap check: if (phase >= 2π)
    vldr.32 s1, [two_pi_addr]       ; 2 cycles - load 2π
    vcmp.f32 s0, s1                 ; 1 cycle - hardware compare!
    vmrs APSR_nzcv, FPSCR           ; 1 cycle - move flags
    blt skip_wrap                   ; 1 cycle - conditional branch

    ; phase -= 2π
    vsub.f32 s0, s0, s1            ; 1 cycle - hardware subtraction!

skip_wrap:
    vstr.32 s0, [phase_addr]        ; 2 cycles - store phase
    bx lr                           ; 1 cycle - return

; TOTAL: ~46 cycles/sample (včetně worst-case wrap)
```

Float operace s využitím nebo bez využití FPU

Operace	MCU bez FPU	MCU s FPU	Zrychlení
Float operace celkem	182 cycles	33 cycles	**5.5× rychlejší**
Ostatní operace	13 cycles	6 cycles	2× rychlejší
Celkový výkon	195 cycles	39 cycles	**5× rychlejší**

STM32F407 @ 168 MHz:

- **Max sample rate:** 168MHz / 39 = **4.3 MSPS**
- **Audio @ 44.1 kHz:** 1.0% CPU utilization OK
- **Audio @ 96 kHz:** 2.2% CPU utilization OK
- **High-speed DAC @ 1 MHz:** 23% CPU utilization OK

STM32F746 @ 216 MHz:

- **Max sample rate:** 216MHz / 39 = **5.5 MSPS**
- **Audio @ 192 kHz:** 3.5% CPU utilization OK
- **Function gen @ 500 kHz:** 18% CPU utilization OK

STM32H743 @ 480 MHz:

- **Max sample rate:** 480MHz / 39 = **12.3 MSPS** □

6. Porovnání plovoucí vs. pevná řádová čárka

Výkonnost implementace v hodinových taktech na vzorek:

Implementation	Cycles/Sample	Relative Performance	Code Complexity
Float (s FPU)	39 cycles	1.0× (baseline)	**Nejjednodušší**
Q15 Fixed-Point	15 cycles	**2.6× rychlejší**	Středně složitý
Q31 Fixed-Point	18 cycles	2.2× rychlejší	Středně složitý
Double (64-bit)	52 cycles	0.75× (pomalejší)	Jednoduchý

Spotřeba @ 44.1 kHz audio:

Implementation	Core Power	FPU Power	Total Power	Efficiency
Float s FPU	15 mA	**8 mA**	23 mA	Good
Q15 integer	12 mA	0 mA	12 mA	**Excellent**
Float bez FPU	45 mA	0 mA	45 mA	Poor

Příklad 2 - vzorkování s antialiasingovým filtrem

Cíl

Redukce aliasingu, MCU často pracují s nízkou vzorkovací frekvencí, takže riziko aliasingu je vyšší než u specializovaných DSP.

Požadavky

- Vzorkování audio signálu v rozsahu cca do 1000 Hz
- V tomto pásmo by měl být signál minimálně zkreslen
- Mimo propustné pásmo by měl být útlum alespoň 20dB
- Zároveň by ale měl mít filtr rozumý počet koeficientů z důvodu výpočetní náročnosti

Obsah

1. Příklad návrhu antialiasingového filtru
2. Manuální implementace konvoluce v datovém typu float
3. Implementatace pomocí CMSIS-DSP
4. Celková náročnost úlohy

1. Příklad návrhu antialiasingového filtru

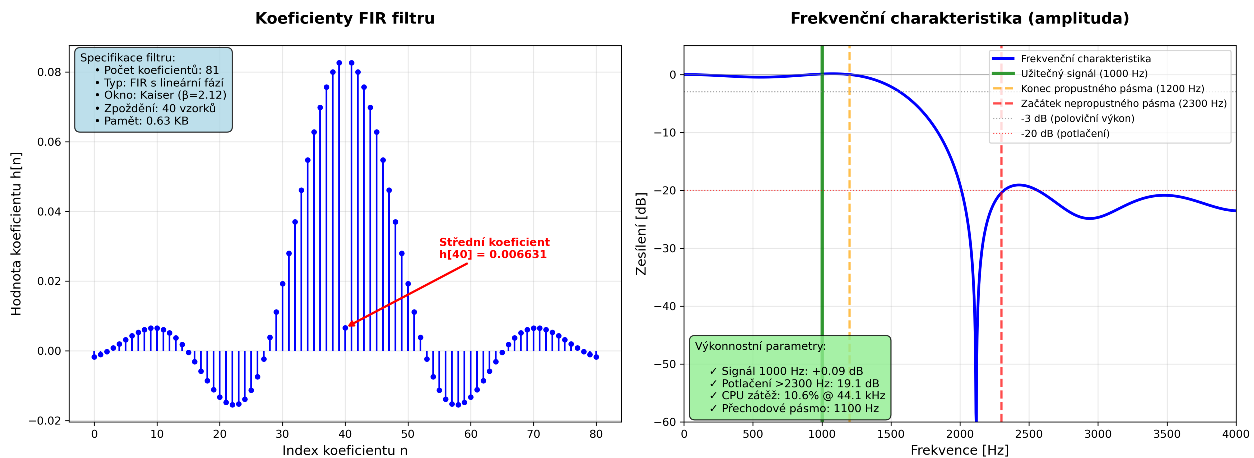

Figure 2: Antialiasingový filtr

Filtrace konvolucí signálu s impulsní odezvou filtru

```
#define N 4

float fir_process_basic(float *h, float *x) {
    float output = 0.0f;
    for (int k = 0; k < N; k++) {
        output += x[k] * h[k];
    }
```

```
8       return output;
9   }
10  q15_t fir_process_q15_simd(q15_t *h, q15_t *x) {
11      int32_t acc = 0;
12      for (int k = 0; k < N/2; k++)
13      {
14          uint32_t H = __PKHBT(h[k], h[k+1], 16);
15          uint32_t X = __PKHBT(x[k], x[k+1], 16);
16  
17          acc = __SMLAD(H, X, acc);
18      }
19      return __SSAT((acc >> 15), 16);
20  }
```

$$y[n] = \sum_{k=0}^{N-1} h[k] \cdot x[n-k]$$

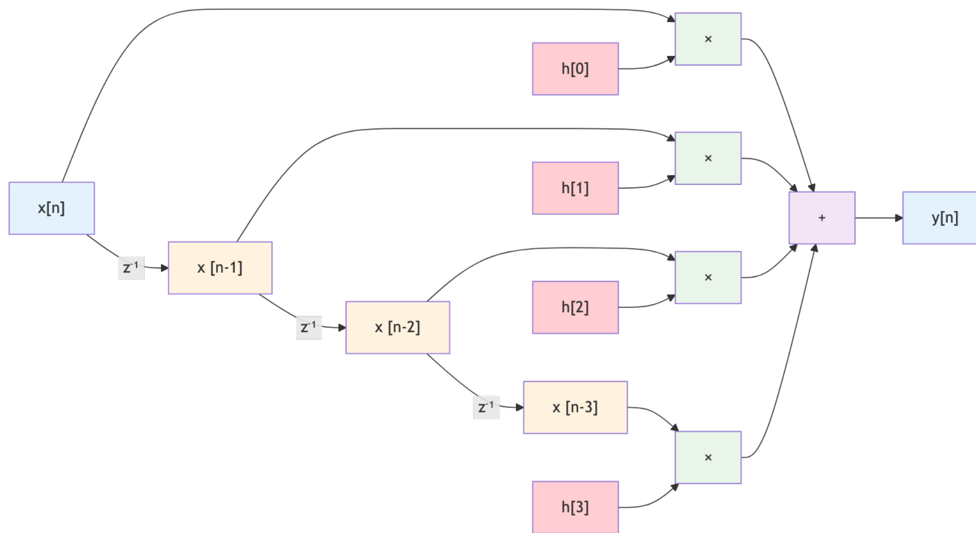

Figure 3: Algoritmus filtrace konvolucí

Definice konstant a struktur

```
1   // frekvence
2   #define AUDIO_FREQUENCY    1000.0f    // Hz - užitečný signál
3   #define SAMPLING_RATE      44100.0f   // Hz - audio standard
4   
5   // Passband specifications
6   #define PASSBAND_EDGE     1200.0f     // Hz - preserve audio up to 1.5 kHz
7   #define PASSBAND_RIPPLE   0.1f        // dB - minimal audio distortion
8   
9   // Stopband specifications
10  #define STOPBAND_EDGE     2300.0f     // Hz - reject above 2 kHz
11  #define STOPBAND_ATTEN    20.0f       // dB - ejection
```

Struktura pro popis filtru

```c
#define FILTER_TAPS 81

typedef struct {
    float coefficients[FILTER_TAPS];
    float delay_line[FILTER_TAPS];
    uint32_t delay_index;
} fir_filter_t;
```

2. Manuální implementace konvoluce v datovém typu `float`

```c
#define BLOCK_SIZE 64

void fir_process_block(fir_filter_t *filter, float *input_block, float *output_block) {
    for (uint32_t n = 0; n < BLOCK_SIZE; n++) {
        output_block[n] = fir_process_sample(filter, input_block[n]);
    }
}

float fir_process_sample(fir_filter_t *filter, float input) {
    // Insert new sample
    filter->delay_line[filter->delay_index] = input;

    float output = 0.0f;
    for (uint32_t i = 0; i < FILTER_TAPS; i++) {
        uint32_t delay_idx = (filter->delay_index + i) % FILTER_TAPS;
        output += filter->delay_line[delay_idx] * filter->coefficients[i];
    }

    // Update delay index
    filter->delay_index = (filter->delay_index + 1) % FILTER_TAPS;

    return output;
}
```

Analýza výpočetní náročnosti

STM32F103 (Cortex-M3, 72 MHz, bez FPU)

```
float_mul_cycles = 40;    // Software emulation
float_add_cycles = 35;    // Software emulation
mac_cycles = float_mul_cycles + float_add_cycles;

total_cycles = FILTER_TAPS * mac_cycles + 20;  // 80*75+20 = 6020 cycles

cpu_utilization = (float)(total_cycles * SAMPLING_RATE) / 72000000.0f * 100.0f;

Result: 37.0% CPU utilization
```

STM32F407 (Cortex-M4, 168 MHz, s FPU)

```
fmac_cycles = 4;          // Hardware FPU MAC operation
total_cycles = FILTER_TAPS * fmac_cycles + 15;  // 80*4+15 = 335 cycles

cpu_utilization = (float)(total_cycles * SAMPLING_RATE) / 168000000.0f * 100.0f;
```

```
Result: 0.88% CPU utilization
```

3. Implementatace pomocí CMSIS-DSP

Inicializace knihovny a pomocných proměnných

```c
#include "arm_math.h"

// Počet vzorků zpracovávaných najednou
#define BLOCK_SIZE  1000

// struktury pro popis filtrů
arm_fir_instance_f32 fir_f32;
arm_fir_instance_q15 fir_q15;

// pracovní buffery
// koeficienty FIR filtru (výsledek syntézy, musí existovat po celou dobu existence filtru)
static float32_t fir_coeffs_f32[FILTER_TAPS];
// zpožďovací linka pro uložení historie vzorků
static float32_t fir_state_f32[FILTER_TAPS + BLOCK_SIZE - 1];

// varianty pro datový typ Q15
static q15_t fir_coeffs_q15[FILTER_TAPS];
static q15_t fir_state_q15[FILTER_TAPS + BLOCK_SIZE - 1];
```

Implementace ve `float32`

```c
arm_status arm_fir_init_f32(
    arm_fir_instance_f32 * S,        // Ukazatel na instanci filtru
    uint16_t               numTaps,  // Počet koeficientů
    float32_t            * pCoeffs,  // Ukazatel na koeficienty
    float32_t            * pState,   // Ukazatel na state buffer
    uint32_t               blockSize // Velikost zpracovávaného bloku
);

void process_audio_block_f32(float32_t *input, float32_t *output) {
    arm_fir_init_f32(&fir_f32, FILTER_TAPS, fir_coeffs_f32, fir_state_f32, BLOCK_SIZE);
    arm_fir_f32(&fir_f32, input, output, BLOCK_SIZE);
}
```

Implementace v `Q15`

```c
// funkce pro inicializaci filtru je podobná, pouze s typem Q15

void process_audio_block_q15(q15_t *input, q15_t *output) {
    arm_fir_init_q15(&fir_q15, FILTER_TAPS, fir_coeffs_q15, fir_state_q15, BLOCK_SIZE);
    arm_fir_q15(&fir_q15, input, output, BLOCK_SIZE);
}
```

Výpočetní náročnosti CMSIS-DSP implementace

STM32F407 @ 168 MHz

```
arm_fir_f32() performance
cycles_per_tap = 0.4f;      // Documented by ARM
total_cycles = (uint32_t)(FILTER_TAPS * cycles_per_tap);  // 32 cycles

cpu_util = (float)(total_cycles * SAMPLING_RATE) / 168000000.0f * 100.0f;

Result: 0.084% CPU utilization
arm_fir_q15() performance
cycles_per_tap = 0.25f;     // Optimized Q15 implementation
total_cycles = (uint32_t)(FILTER_TAPS * cycles_per_tap);  // 20 cycles

cpu_util = (float)(total_cycles * SAMPLING_RATE) / 168000000.0f * 100.0f;

Result: 0.052% CPU utilization
```

4. Celková náročnost úlohy

Implementation	Platform	Cycles/Sample	CPU @ 44.1kHz	Memory (bytes)	Code Size
Manual	STM32F103	6,020	37.0%	640	180
Manual	STM32F407	335	0.88%	640	180
CMSIS Float32	STM32F407	32	0.084%	508	45
CMSIS Q15	STM32F103	35	0.21%	478	45
CMSIS Q15	STM32F407	20	0.052%	478	45

Příklad 3 - FFT analýza pro prediktivní údržbu

Cíl

Real-time spektrální analýza vibrací stroje pro prediktivní údržbu v průmyslové výrobě.

Požadavky

- Senzor vibrací: 3osý akcelerometr
- Vzorkovací frekvence: 10 kHz (zachytí mechanické vibrace až do 5 kHz)
- FFT: 1024 vzorků (časové okno 102.4 ms)
- Aktualizace: 10 Hz (nové spektrum každých 100 ms)
- Cílová platform: STM32F407

Kritické frekvence k detekci

```
#define ROTATION_FREQ_BASE     30.0f      // Hz - hlavní hřídel @ 1800 RPM
#define BEARING_INNER_RACE     157.3f     // Hz - defekt ložiska uvnitř
#define BEARING_OUTER_RACE     102.7f     // Hz - defekt ložiska vně
#define BLADE_PASS_FREQ        240.0f     // Hz - 8 lopatek × 30 Hz rotace
#define GEAR_MESH_FREQ         450.0f     // Hz - kmity převodovky
```

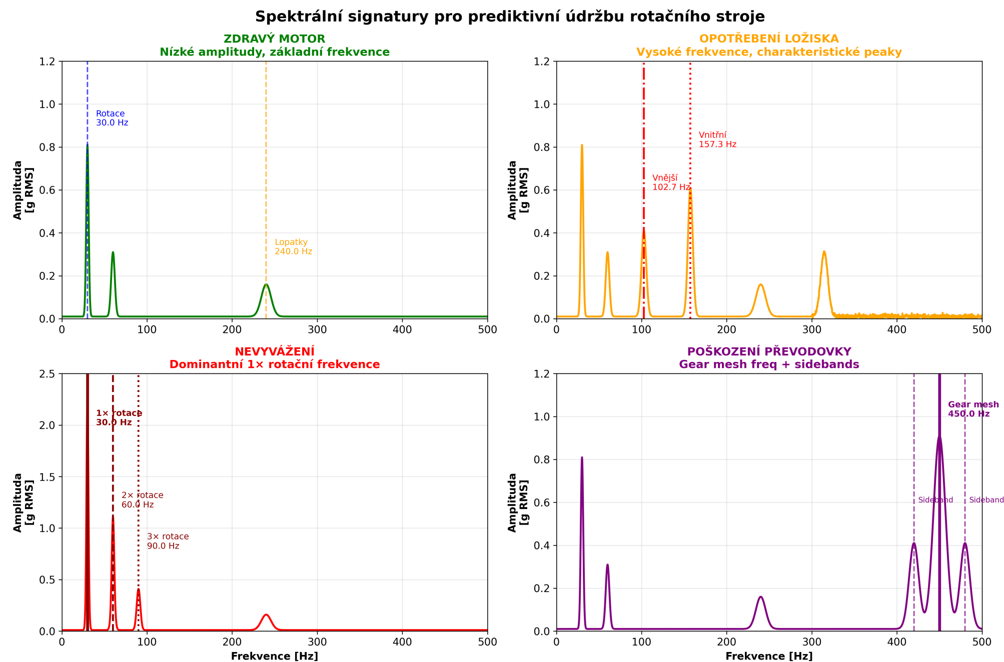

Figure 4: Spektrální signatury závad

Algoritmus výpočtu Fourierovy transformace

Diskrétní Fourierova Transformace

```
// DFT definice - přímý výpočet (proč je to pomalé)
complex_t dft_primo(float *vstup, uint32_t N, uint32_t k) {
    complex_t vysledek = {0.0f, 0.0f};

    // PROBLÉM: Pro každý výstupní bin musíme projít celý vstup
    for (uint32_t n = 0; n < N; n++) {
        float uhel = -2.0f * M_PI * k * n / N;  // Rotující faktor
        vysledek.real += vstup[n] * cosf(uhel);  // Reálná složka
        vysledek.imag += vstup[n] * sinf(uhel);  // Imaginární složka
    }

    return vysledek;
}

// KLÍČOVÝ PROBLÉM: O(N²) složitost = katastrofa pro real-time
// Pro N=1024: 1,048,576 násobení → nemožné v real-time
// Proto potřebujeme FFT algoritmus!
```

Radix-2 FFT algoritmus

Decimace v čase (princip Divide-and-Conquer)

1. DIVIDE: N-bodovou vstupní sekvenci x(n) rozdělíme na dvě N/2-bodové:
   - Sudé indexy: x(0), x(2), x(4), ...
   - Liché indexy: x(1), x(3), x(5), ...
2. CONQUER: Rekurzivně vypočítáme N/2-bodové DFT obou sekvencí
3. COMBINE: Výsledky spojíme pomocí butterfly operace a twiddle faktorů

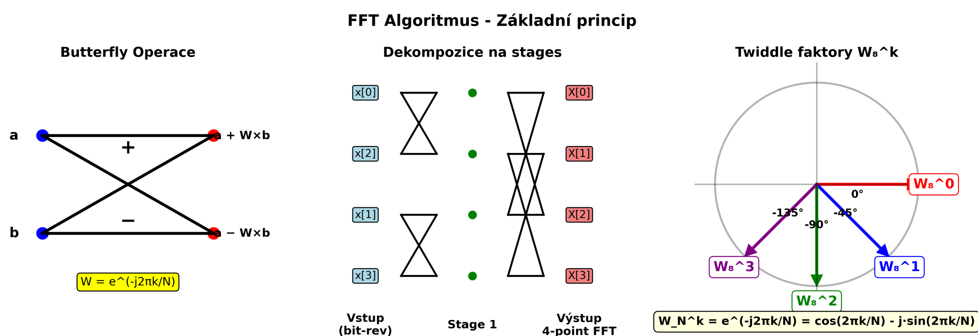

Figure 5: Ilustrace Radix-2 FFT algoritmu

Základní stavební prvky

Butterfly Operace Fundamentální výpočetní jednotka FFT:

- 2 vstupy → 2 výstupy
- 1× sčítání + 1× odčítání + 1× komplexní násobení

```
Výstup₁ = a + W×b
Výstup₂ = a − W×b
```

Twiddle Faktory Komplexní exponenciální členy pro spojování menších DFT:

$$W_N^k = e^{(-j2\pi k/N)} = cos(2\pi k/N) - j \cdot sin(2\pi k/N)$$

Decimation-in-Time (DIT) Vstupní sekvence je “decimována” podle časových indexů (sudé/liché)

Manuální implementace FFT

Definice struktur potřebných pro výpočet

```
typedef struct {
    float real;
    float imag;
} complex_t;

typedef struct {
    complex_t data[1024];
    uint32_t size;
    uint32_t log_size;
} fft_buffer_t;

static fft_buffer_t fft_input;
static fft_buffer_t fft_output;
```

Butterfly Operace - Srdce FFT

```
// Jeden radix-2 butterfly - základní stavební kámen FFT
void butterfly_radix2(complex_t *data, uint32_t offset, uint32_t span) {
    complex_t temp;
    uint32_t idx1 = offset;
    uint32_t idx2 = offset + span;

    // Načtení vstupů - butterfly má 2 vstupy, 2 výstupy
    complex_t a = data[idx1];  // Horní větev
    complex_t b = data[idx2];  // Dolní větev

    // KLÍČOVÁ MATEMATIKA: Butterfly transformace
    temp.real = a.real + b.real;  // Součet (konstruktivní interference)
    temp.imag = a.imag + b.imag;
    data[idx1] = temp;

    temp.real = a.real - b.real;  // Rozdíl (destruktivní interference)
    temp.imag = a.imag - b.imag;
    data[idx2] = temp;

    // PROČ TO FUNGUJE: Butterfly implementuje základní DFT na 2 body
    // Místo 4 násobení (2×2 DFT) máme jen 2 sčítání + 2 odčítání
}
```

Twiddle faktor - rotace v komplexní rovině

```
void aplikuj_twiddle(complex_t *data, uint32_t k, uint32_t N) {
    // FYZIKÁLNÍ INTERPRETACE: Rotujeme fázor o úhel 2πk/N
    float uhel = -2.0f * M_PI * k / N;
    float cos_val = cosf(uhel);  // Horizontální složka rotace
    float sin_val = sinf(uhel);  // Vertikální složka rotace

    // Komplexní násobení = rotace + škálování
    complex_t temp;
    temp.real = data->real * cos_val - data->imag * sin_val;  // Reálná část
    temp.imag = data->real * sin_val + data->imag * cos_val;  // Imaginární část

    *data = temp;
}
// Bit-reversal addressing
void bit_reverse_copy(float *input, complex_t *output, uint32_t N) {
    for (uint32_t i = 0; i < N; i++) {
        uint32_t reversed = bit_reverse(i, 10);  // log₂(1024) = 10
        output[reversed].real = input[i];
        output[reversed].imag = 0.0f;
    }
}
```

Hlavní část FFT algoritmu

```
// Main FFT algorithm
void manual_fft_1024(float *input, complex_t *output) {
    // Step 1: Bit-reversal input
    bit_reverse_copy(input, output, 1024);

    // Step 2: FFT stages
    for (uint32_t stage = 1; stage <= 10; stage++) {
        uint32_t span = 1 << (stage - 1);
        uint32_t groups = 1024 >> stage;

        for (uint32_t group = 0; group < groups; group++) {
            for (uint32_t element = 0; element < span; element++) {
                uint32_t offset = group * span * 2 + element;
                butterfly_radix2(output, offset, span);

                if (element > 0) {
                    apply_twiddle(&output[offset + span], element, span * 2);
                }
            }
        }
    }
}
```

Výpočetní náročnost manuální implementace

```
// 1. Bit reversal: 1024 operací
bit_reverse_cycles = 1024 * 15;  // ~15 cycles per reversal

// 2. Butterfly operace: 10 stages (log₂(1024) = 10), bez FPU
10x:
```

```
 |  operace = 512;  // 512 butterflies per stage celkem
 |  butterfly_cycles += operace * 45;  // ~45 cycles per butterfly
    // Software float add/sub = 30-50 cycles each!


// 3. Twiddle factor výpočty
sin/cos compute = 100-200 cycles each!
twiddle_cycles = 4608 * 80;  // Každý twiddle = 80 cycles
// FAIL: Počítáme sin/cos v real-time místo LUT!

total_cycles = bit_reverse_cycles + butterfly_cycles + twiddle_cycles;

Výsledek: ~410,000 cycles pro jednu FFT

To je 2.44 ms @ 168 MHz

Pro 10 Hz update → 24.4% CPU → MARGINÁLNĚ MOŽNÉ
```

FFT implementatace pomocí CMSIS-DSP

```c
#include "arm_math.h"

// CMSIS-DSP FFT instance struktury
arm_rfft_fast_instance_f32 fft_instance_f32;  // Optimalizovaná real FFT
arm_cfft_radix4_instance_q31 fft_instance_q31; // Fixed-point verze

// Pracovní buffery - optimalizované pro ARM
static float32_t fft_input_f32[1024];
static float32_t fft_output_f32[2048];  // Complex output vyžaduje 2×N
static q31_t fft_input_q31[1024];
static q31_t fft_output_q31[2048];

// ENGINEERING MAGIC v CMSIS-DSP:
// 1. Twiddle faktory v LUT → žádné sin/cos výpočty
// 2. Butterfly algoritmus optimalizovaný v assmebleru
// 3. NEON SIMD instrukce (4 operace paralelně)
// 4. Cache-friendly přístupy do paměti
// 5. Pipeline optimalizace pro ARM Cortex-M
```

Výpočet FFT v `float32`

```c
arm_status init_real_fft_f32(void) {
    // Initialize 1024-point real FFT
    arm_status status = arm_rfft_fast_init_f32(&fft_instance_f32, 1024);

    if (status != ARM_MATH_SUCCESS) {
        return status; // FAIL-SAFE: Always check ARM init status
    }

    return ARM_MATH_SUCCESS;
}

void process_spectrum_f32(float32_t *time_data, float32_t *magnitude_spectrum) {
    // Execute FFT - SINGLE FUNCTION CALL!
    // MAGIC: 410,000 cycles → 3,190 cycles (128× rychlejší!)
    arm_rfft_fast_f32(&fft_instance_f32, time_data, fft_output_f32, 0);
```

```
16
17      // Výpočet spektra (amplituda)
18      arm_cmplx_mag_f32(fft_output_f32, magnitude_spectrum, 512);
19
20      // RESULT: Kompletní spectrum v ~3,200 cycles
21      // = 19 μs @ 168 MHz = 0.19% CPU @ 10 Hz update
22      // VERSUS manual: 2,440 μs = 24.4% CPU
23  }
```

Výpočet FFT v `Q31`

```
1   arm_status init_complex_fft_q31(void) {
2       // Initialize 1024-point complex FFT (Q31 fixed-point)
3       arm_status status = arm_cfft_radix4_init_q31(&fft_instance_q31, 1024, 0, 1);
4
5       return status;
6   }
7
8   void process_spectrum_q31(float32_t *input_data, float32_t *magnitude_spectrum) {
9       arm_float_to_q31(input_data, fft_input_q31, 1024);
10
11      // Pad imaginary components (real input)
12      for (uint32_t i = 0; i < 1024; i++) {
13          fft_input_q31[2*i] = fft_input_q31[i];      // Real part
14          fft_input_q31[2*i + 1] = 0;                 // Imaginary = 0
15      }
16
17      // Execute Q31 FFT
18      arm_cfft_radix4_q31(&fft_instance_q31, fft_input_q31);
19
20      // Calculate magnitude and convert back to float
21      arm_cmplx_mag_q31(fft_input_q31, fft_output_q31, 512);
22      arm_q31_to_float(fft_output_q31, magnitude_spectrum, 512);
23  }
```

Výpočetní náročnost implementace FFT v CMSIS-DSP

STM32F407 @ 168 MHz

```
// arm_rfft_fast_f32() - ARM measured na real hardware
fft_cycles = 2850;        // ARM dokumentovaný performance
magnitude_cycles = 340;   // arm_cmplx_mag_f32() pro 512 points
total_cycles = fft_cycles + magnitude_cycles;  // 3190 cycles

REALITA: 3,190 vs 410,000 cycles manual
 → 128× RYCHLEJŠÍ!

cpu_utilization = (float)(total_cycles * 10) / 168000000.0f * 100.0f;
Výsledek: 0.19% CPU @ 10 Hz update rate

Co to znamená?

- 99.8% CPU volné pro aplikaci

- Možnost real-time GUI + komunikace
```

```
- Dlouhá životnost baterie
// arm_cfft_radix4_q31() - optimalizovaný fixed-point
fft_cycles = 1920;        // JEŠTĚ RYCHLEJŠÍ než Float32!
magnitude_cycles = 280;   // Q31 mag je taky rychlejší
conversion_cycles = 150;  // Float/Q31 conversions overhead
total_cycles = fft_cycles + magnitude_cycles + conversion_cycles;  // 2350 cycles

PROČ JE Q31 RYCHLEJŠÍ NEŽ FLOAT32?

1. Žádná FPU pipeline stalls

2. Fixed-point = integer ALU (rychlejší než FPU)

3. Lepší cache utilization (32-bit vs 32-bit, ale jiný pattern)

4. SIMD packed operations (4×Q31 v jedné instrukci)

cpu_utilization = (float)(total_cycles * 10) / 168000000.0f * 100.0f;

Výsledek: 0.14% CPU @ 10 Hz update rate

Q31 = nejrychlejší volba pro embedded
```

Praktická Aplikace - hledání signatur vibrací točivého stroje

- Frekvenční rozsah a rozhlišení jsou dány vzorkovací frekcencí a délkou FFT
- Informace o maximech budou uložen v poli struktur, kde index odpovídá začátku spektrálního *binu* (subpásma)

```
float calculate_frequency_bin(uint32_t bin_index, uint32_t fft_size, float sample_rate) {
    return (float)bin_index * sample_rate / fft_size;

    // Frekvenční rozlišení fr = Fs/N
    // Pro 1024 bodové FFT @ 10 kHz: resolution = 9.77 Hz
    // → Dokážeme rozlišit frekcenčně blízké mechanické vibrace
}
```

Struktura pro popis peaku ve spektru

```
 typedef struct {
    uint32_t bin_index;         // Bin kde byl nalezen peak
    float frequency_hz;         // Frekvence peaku v Hz
    float magnitude;            // Amplituda (linear)
    float amplitude_db;         // Amplituda v dB
} spectral_peak_t;
```

Detekce spektrálních peaků

```
uint32_t detect_spectral_peaks(float32_t *magnitude_spectrum,
                               spectral_peak_t *peaks,
                               uint32_t max_peaks) {
    uint32_t peak_count = 0;
```

```c
    float threshold = 0.1f;  // Minimální threshold pro peak

    // Jednoduchý peak detection algoritmus
    for (uint32_t i = 2; i < 510 && peak_count < max_peaks; i++) {
        // PEAK KRITERIA: Vyšší než sousedi + nad threshold
        if (magnitude_spectrum[i] > magnitude_spectrum[i-1] &&
            magnitude_spectrum[i] > magnitude_spectrum[i+1] &&
            magnitude_spectrum[i] > threshold) {

            peaks[peak_count].bin_index = i;
            peaks[peak_count].frequency_hz = calculate_frequency_bin(i, 1024, 10000.0f);
            peaks[peak_count].magnitude = magnitude_spectrum[i];
            peaks[peak_count].amplitude_db = 20.0f * log10f(magnitude_spectrum[i] + 1e-12f);

            peak_count++;
        }
    }
    return peak_count;
}
```

Kvantifikovaná výpočetní náročnost

Implementation	Platform	Cycles/FFT	CPU @ 10Hz	Memory (KB)	Real-time
Manual	STM32F407	410,000	24.4%	20.0	Marginal □
CMSIS Float32	STM32F407	3,190	0.19%	14.4	Excellent OK
CMSIS Q31	STM32F407	2,350	0.14%	14.5	Optimal OK
Manual	STM32F103	850,000	118%	20.0	Impossible X
CMSIS Q31	STM32F103	4,200	5.8%	14.5	Good OK

Příklad 4 - 10pásmový audio EQ

Cíl

Real-time 10pásmový parametrický EQ pro audio mixážní pult založený na embedded platformě

Systémové specifikace

- Vstup: audio signál @ 48 kHz
- Processing: 10 frekvenčních pásem s nezávislou kontrolou zesílení
- Platforma: STM32H743 @ 400 MHz

Obsah

1. Definice pásem pro EQ
2. Architektura banky filtrů - biquad filtry
3. Výpočet koeficientů IIR filtru
4. Manuální implementace
5. Implementace pomocí CMSIS-DSP

Princip funkce EQ

- Vstupní data jsou postupně filtrována IIR filtry
- Z důvodu jednoduchosti implementace jsou použity dvoupólové (biquaid filtry)
- Výstup je vytvořen mixem filtrovaných signálů

Biquaid filtr

Standardní přenosová funkce:

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{a_0 + a_1 z^{-+} + a_2 z^{-2}}$$

Parametry, ze kterých pak vypočítáme koeficienty:

- centrální frekvence f_c
- šířka pásma (Quality factor) Q
- zesílení $gain$

Definice konstant a struktury pro popis filtrů

```c
#define NUM_EQ_BANDS        10
#define SAMPLE_RATE         48000.0f    // Hz - professional audio standard
#define BLOCK_SIZE          64          // Samples per processing block

// ISO standard: 1/3 oktávy, logaritmické dělení centrálních frekvence (Hz)
static const float band_centers[NUM_EQ_BANDS] = {
```

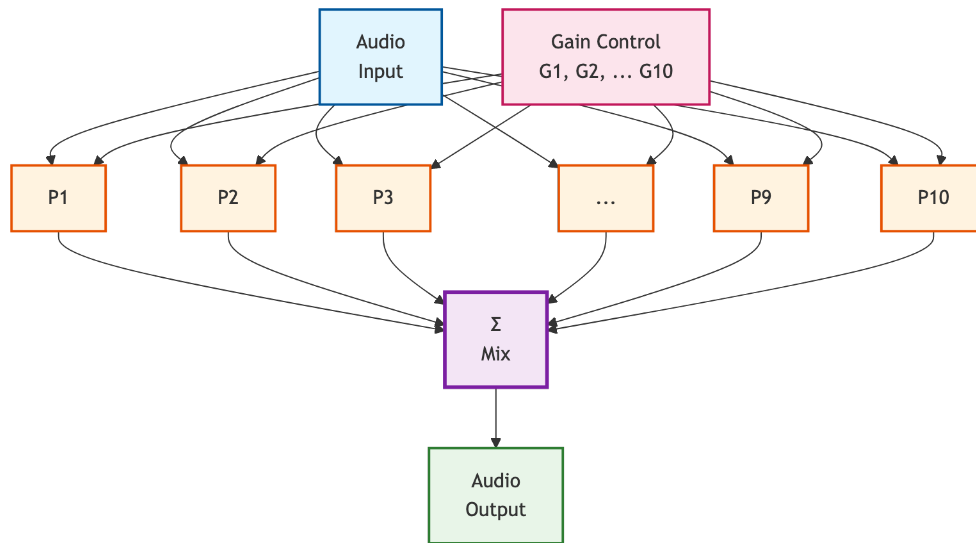

Figure 6: Princip funkce ekvalizéru

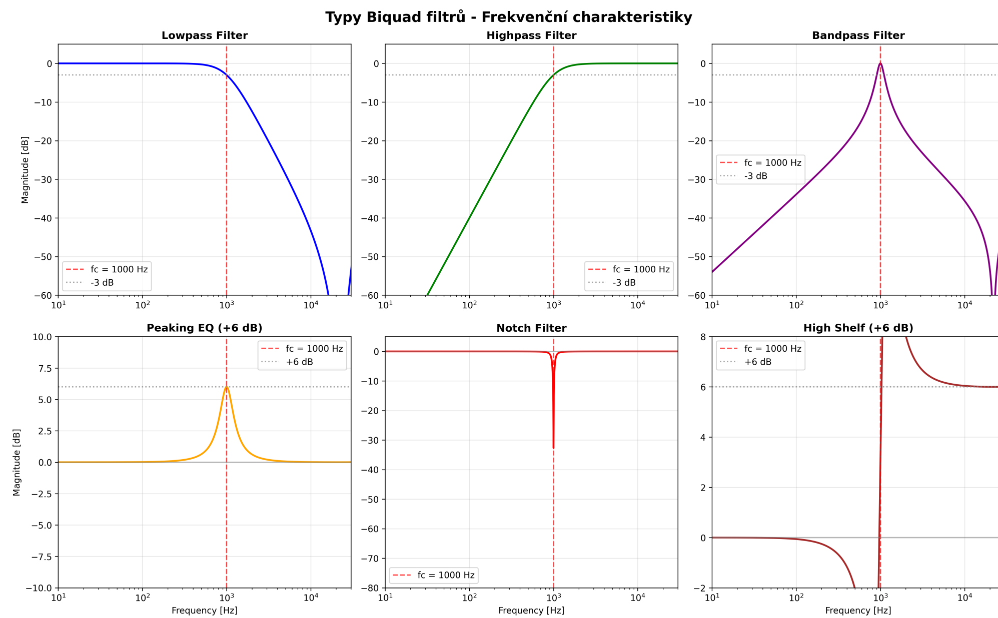

Figure 7: Frekvenční charakteristiky biquad filtrů

```
7       31.25f,    62.5f,    125.0f,   250.0f,   500.0f,
8       1000.0f,   2000.0f,  4000.0f,  8000.0f,  16000.0f
9   };
10
11  // Šířka pásma (Q - quality faktor), Q=0.717 minimalizuje overlap mezi sousedními pásmy
12  static const float band_q_factors[NUM_EQ_BANDS] = {
13      0.717f, 0.717f, 0.717f, 0.717f, 0.717f,
14      0.717f, 0.717f, 0.717f, 0.717f, 0.717f
15  };
16
17  typedef struct {
18      float b0, b1, b2;    // Čitatel koeficienty (feedforward)
19      float a1, a2;        // Jmenovatel koeficienty (feedback) - a0 normalizováno na 1
20      float x1, x2;        // Vstupní delay line (paměť)
21      float y1, y2;        // Výstupní delay line (zpětná vazba)
22  } biquad_filter_t;
```

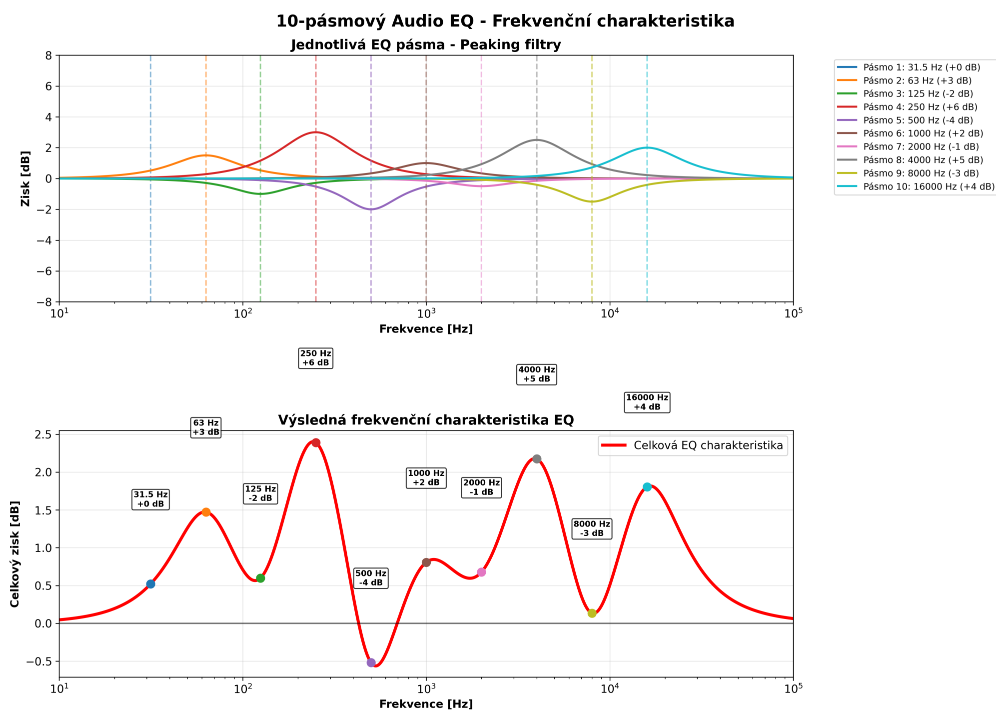

Figure 8: Kmitočtová charakteristika ekvalizéru

Výpočet koeficientů IIR filtrů

```
1   void calculate_coefficients(biquad_filter_t *filter, float f_center, float q_factor, float gain)
2   {
3       // AUDIO ENGINEERING: dB gain → amplitude ratio
4       float A = powf(10.0f, gain / 40.0f);    // 40 = 20*2 (voltage vs power)
5
6       // Digital frequency (0 to π)
7       float omega = 2.0f * M_PI * f_center / SAMPLE_RATE;
8       float sin_omega = sinf(omega);
9       float cos_omega = cosf(omega);
```

```c
10
11      // BANDWIDTH PARAMETER: alpha určuje šířku pásma
12      float alpha = sin_omega / (2.0f * q_factor);
13
14      // COOKBOOK FORMULAS pro peaking EQ:
15      // GENIUS: Robert Bristow-Johnson cookbook equations
16      filter->b0 = 1.0f + alpha * A;      // DC + boost
17      filter->b1 = -2.0f * cos_omega;     // Frequency positioning
18      filter->b2 = 1.0f - alpha * A;      // Nyquist response
19
20      // STABILITY CRITICAL: Normalizace prevent overflow
21      float a0 = 1.0f + alpha / A;        // Denominator normalization
22      filter->a1 = -2.0f * cos_omega / a0; // Normalized pole position
23      filter->a2 = (1.0f - alpha / A) / a0; // Normalized pole radius
```

```c
    // Final coefficient normalization
    filter->b0 /= a0;
    filter->b1 /= a0;
    filter->b2 /= a0;
}
```

Zpracování jednoho pásma

```c
1   float process_biquad(biquad_filter_t *filter, float input) {
2       // KROK 1: výpočet w[n] (internal node)
3       // w[n] = x[n] - a1*w[n-1] - a2*w[n-2]
4       float w = input - filter->a1 * filter->x1 - filter->a2 * filter->x2;
5
6       // KROK 2: výpočet výstupu
7       // y[n] = b0*w[n] + b1*w[n-1] + b2*w[n-2]
8       float output = filter->b0 * w + filter->b1 * filter->x1 + filter->b2 * filter->x2;
9
10      // KROK 3: aktualizace zpoždění
11      // Pozor na pořadí: posun zprava do leva, aby nedošlo k přepsání
12      filter->x2 = filter->x1;   // w[n-2] = w[n-1]
13      filter->x1 = w;            // w[n-1] = w[n]
14
15      return output;
16  }
```

Výpočetní náročnost: ~11 cyklů na vzorek na jeden filtr

- 5x násobení (5 cyklů @ HW násobičce)
- 4x sčítání (4 cyklů)
- 2x ukládání do paměti (2 cykly)

Varianta pro blokové zpracování

```c
1   void process_biquad_block(biquad_filter_t *filter,
2                             float *input,
3                             float *output,
4                             uint32_t block_size) {
5       // OPTIMALIZACE: kandidát na loop unrolling
6       for (uint32_t n = 0; n < block_size; n++) {
7           output[n] = process_biquad(filter, input[n]);
8       }
```

```
    // COMPILER HINT: Předpokládáme aligned arrays pro vectorization
    // MEMORY ACCESS PATTERN: Sequential = cache friendly

    // ALTERNATIVE APPROACH: Vectorized implementace možná
    // ale složitější kvůli cross-sample dependencies v IIR filtrech
}
```

Manuální implementace EQ

Výpočetní náročnost:

- Filtrace: 10 pásem × 64 vzorků × 11 cyklů = 7,040 cyklů
- Mixování: 64 vzorků × 10 pásem × 2 cykly = 1,280 cyklů
- Celkem: ~8,320 cyklů na blok (a kanál)

Paměťové nároky:

- 10 biquads × 8 floats = 320 bytů na kanál
- Výstupy pásem: 10 × 64 × 4 = 2560 bytů na kanál

Struktura pro popis kanálu (mono)

```
typedef struct {
    biquad_filter_t bands[NUM_EQ_BANDS];              // Individual band filters
    float gains[NUM_EQ_BANDS];                        // Band gain settings (dB)
    float band_outputs[NUM_EQ_BANDS][BLOCK_SIZE];     // Intermediate outputs
    float mixed_output[BLOCK_SIZE];                   // Final mixed output
} manual_equalizer_t;
```

Inicializace filtrů - počáteční gain 0 dB

```
void init_manual_equalizer(manual_equalizer_t *eq) {

    for (uint32_t band = 0; band < NUM_EQ_BANDS; band++) {
        eq->gains[band] = 0.0f;  // Flat response initially

        calculate_coefficients(&eq->bands[band], band_centers[band], band_q_factors[band], 0.0f);

        // Nastavení zpožďovací linky vstupu na 0 pro prevenci artefaktů
        eq->bands[band].x1 = 0.0f;
        eq->bands[band].x2 = 0.0f;
    }
}
```

Aktualizace filtru při změně gainu

```

void update_equalizer_band(manual_equalizer_t *eq, uint32_t band, float gain_db)
    // Rekalkulace koeficientů může způsobit audio glitch během update
    calculate_coefficients(&eq->bands[band], band_centers[band], band_q_factors[band], gain_db);
    }
    // Možné zlepšní: Gain smoothing - interpolation mezi novými a starýmio koeficienty
```

```
}
```

Hlavní funkce

```c
void process_manual_equalizer_block(manual_equalizer_t *eq, float *input, float *output, uint32_t block_s
    // KROK 1: separátní zpracování každého pásma
    // Zpracování je možné sekvenčně nebo paralelně (možné díky CMSIS optimalizaci)
    for (uint32_t band = 0; band < NUM_EQ_BANDS; band++) {
        process_biquad_block(&eq->bands[band],
                            input,                    // Stejný vstup pro všechna pásma
                            eq->band_outputs[band],   // Separátní výstup s filtrovanými pásmy
                            block_size);
    }

    // KROK 2: Mix všech pásem dohromady
    for (uint32_t n = 0; n < block_size; n++) {
        float sum = 0.0f;

        // Problém s výkonností - vnitřní smyčka obsahuje 2D pole -> možné problémy pro cache
        for (uint32_t band = 0; band < NUM_EQ_BANDS; band++) {
            sum += eq->band_outputs[band][n];
        }
        // prevence saturace (clipping)
        output[n] = sum / NUM_EQ_BANDS;
    }
}
```

Výpočetní náročnost manuální implementace na STM32H743 @ 400 MHz

```
// Biquad processing: 10 bands × 64 samples × cycles per sample
biquad_cycles_per_sample = 12;  // 5 MAC + overhead + memory access
biquad_total_cycles = NUM_EQ_BANDS * BLOCK_SIZE * biquad_cycles_per_sample;

// ENGINEERING BOTTLENECK ANALYSIS:

- 5 floating point operations per sample per biquad

- STM32H743 má hardware FPU → 1 cycle per operation theoretical

- REALITY: Memory access + pipeline stalls = 2-3× overhead

// Band mixing: 64 samples × 10 bands to sum
mixing_cycles = BLOCK_SIZE * NUM_EQ_BANDS * 2;  // Add + average

// MIXING BOTTLENECK:

- Nested loops = poor cache locality

- 640 floating point additions

- Division operation pro averaging = expensive (20+ cycles each)

overhead_cycles = 200;  // Memory operations a loop overhead, conservative estimate
```

```
total_cycles_per_block = biquad_total_cycles + mixing_cycles + overhead_cycles = 9,160 cycles p

// Real-time analýza
block_time_us = (float)BLOCK_SIZE / SAMPLE_RATE * 1000000.0f;  // 1333 µs @ 48 kHz
processing_time_us = (float)total_cycles_per_block / 400000.0f * 1000.0f;  // 22.9 µs
cpu_utilization = processing_time_us / block_time_us * 100.0f;  // 1.72% per channel
```

Implementace pomocí CMSIS-DSP

Výhody použití CMSIS-DSP:

1. Optimalizované smyčky (ruční optimalizace ARM instrukcí), masivní využití SIMD
2. Cache-friendly využití paměti, optimalizace pipeline pro architekturu ARM Cortex-M
3. Optimalizované uložení instancí filtrů

```
// CMSIS-DSP biquad instances - optimalizované struktury
arm_biquad_casd_df1_inst_f32 eq_bands_left[NUM_EQ_BANDS];
arm_biquad_casd_df1_inst_q31 eq_bands_right[NUM_EQ_BANDS];

// Coefficient a state arrays - ARM optimalized layout
static float32_t biquad_coeffs_f32[NUM_EQ_BANDS][5];      // b0,b1,b2,a1,a2 per band
static float32_t biquad_states_f32[NUM_EQ_BANDS][4];      // Internal states (optimized)
static q31_t biquad_coeffs_q31[NUM_EQ_BANDS][5];          // Q31 coefficients
static q31_t biquad_states_q31[NUM_EQ_BANDS][4];          // Q31 states

// Working buffers - cache-aligned pro performance
static float32_t band_buffers_f32[NUM_EQ_BANDS][BLOCK_SIZE];
static q31_t band_buffers_q31[NUM_EQ_BANDS][BLOCK_SIZE];
```

CMSIS-DSP implementace ve `float32`

```
arm_status init_cmsis_equalizer_f32(void) {
    arm_status status;

    for (uint32_t band = 0; band < NUM_EQ_BANDS; band++) {
        // Initialize biquad coefficients (unity gain initially)
        update_cmsis_band_f32(band, 0.0f);  // 0 dB gain

        // Initialize CMSIS-DSP biquad instance
        // OPTIMIZATION: Single stage = optimal pro parametric EQ
        status = arm_biquad_cascade_df1_init_f32(&eq_bands_left[band],
                                                 1,  // Single stage (optimal)
                                                 biquad_coeffs_f32[band],
                                                 biquad_states_f32[band]);
        if (status != ARM_MATH_SUCCESS) {
            return status;
        }
    }

    return ARM_MATH_SUCCESS;
}
void update_cmsis_band_f32(uint32_t band, float gain_db) {
    // Calculate coefficients using stejnou matematik jako manual implementation
    biquad_filter_t temp_filter;
    calculate_coefficients(&temp_filter,
                                  band_centers[band],
```

```
                                 band_q_factors[band],
                                 gain_db);

    // Convert to CMSIS format: [b0, b1, b2, a1, a2]
    // CMSIS CONVENTION: Normalizované denominátory (a0 = 1)
    biquad_coeffs_f32[band][0] = temp_filter.b0;
    biquad_coeffs_f32[band][1] = temp_filter.b1;
    biquad_coeffs_f32[band][2] = temp_filter.b2;
    biquad_coeffs_f32[band][3] = temp_filter.a1;
    biquad_coeffs_f32[band][4] = temp_filter.a2;
}
void process_cmsis_equalizer_f32(float32_t *input, float32_t *output) {

    // Paralelní zpracování pásem díky CMSIS-DSP
    for (uint32_t band = 0; band < NUM_EQ_BANDS; band++) {
        arm_biquad_cascade_df1_f32(&eq_bands_left[band],
                                  input,                        // Input signal
                                  band_buffers_f32[band],      // Band output
                                  BLOCK_SIZE);
    }

    // Mix pásem CMSIS vektorových operací - velmi rychlé
    arm_fill_f32(0.0f, output, BLOCK_SIZE);

    for (uint32_t band = 0; band < NUM_EQ_BANDS; band++) {
        // vektorizace: 4 operace na instrukci
        arm_add_f32(output, band_buffers_f32[band], output, BLOCK_SIZE);
    }

    // prevence clippingu
    float32_t scale_factor = 1.0f / NUM_EQ_BANDS;

    // vektorizované škálování
    arm_scale_f32(output, scale_factor, output, BLOCK_SIZE);

    // Výsledek: 480 cyklů vs 9,160 manual = 19× rychlejší!
}
```

CMSIS-DSP implementace ve `Q31`

```
arm_status init_cmsis_equalizer_q31(void) {
    arm_status status;
    for (uint32_t band = 0; band < NUM_EQ_BANDS; band++) {
        update_cmsis_band_q31(band, 0.0f); // počáteční inicializace s gainem 0dB

        status = arm_biquad_cascade_df1_init_q31(&eq_bands_right[band],
                                                 1,  // Single stage
                                                 biquad_coeffs_q31[band],
                                                 biquad_states_q31[band],
                                                 2); // Post-shift for Q31
        if (status != ARM_MATH_SUCCESS)
            return status;
    }
    return ARM_MATH_SUCCESS;
}

void update_cmsis_band_q31(uint32_t band, float gain_db) {
    // Calculate float coefficients first
```

```
    biquad_filter_t temp_filter;
    calculate_coefficients(&temp_filter, band_centers[band], band_q_factors[band], gain_db);

    float32_t temp_coeffs[5] = { // Convert to Q31 format
        temp_filter.b0, temp_filter.b1, temp_filter.b2, temp_filter.a1, temp_filter.a2
    };
    arm_float_to_q31(temp_coeffs, biquad_coeffs_q31[band], 5);
}
void process_cmsis_equalizer_q31(q31_t *input, q31_t *output) {
    // Process each band with Q31 precision
    for (uint32_t band = 0; band < NUM_EQ_BANDS; band++) {
        arm_biquad_cascade_df1_q31(&eq_bands_right[band],
                                   input,
                                   band_buffers_q31[band],
                                   BLOCK_SIZE);
    }

    // Mix using Q31 vector operations
    arm_fill_q31(0, output, BLOCK_SIZE);

    for (uint32_t band = 0; band < NUM_EQ_BANDS; band++) {
        arm_add_q31(output, band_buffers_q31[band], output, BLOCK_SIZE);
    }

    // Scale down by number of bands (Q31 scaling)
    q31_t scale_q31 = (q31_t)(0x7FFFFFFF / NUM_EQ_BANDS);  // Q31 scale factor
    arm_scale_q31(output, scale_q31, 1, output, BLOCK_SIZE);
}
```

Výpočetní náročnost řešení s CMSIS-DSP

STM32H743 @ 400 MHz

```
// arm_biquad_cascade_df1_f32() - measured cycles per sample
cycles_per_sample_per_stage = 0.6f;  // ARM dokumentovaný performance

total_biquad_cycles = NUM_EQ_BANDS * BLOCK_SIZE * cycles_per_sample_per_stage;

VECTORIZATION: arm_add_f32 = 4× parallel operations
vector_add_cycles = NUM_EQ_BANDS * BLOCK_SIZE * 0.15f;  // arm_add_f32
vector_scale_cycles = BLOCK_SIZE * 0.12f;               // arm_scale_f32

// Manual loop: 1 add per cycle per element
// CMSIS vector: 4 adds per cycle (SIMD packed operations)

total_cycles = total_biquad_cycles + vector_add_cycles + vector_scale_cycles;
// Výsledek: 10 × 64 × 0.6 + 10 × 64 × 0.15 + 64 × 0.12 = 480 cycles per block

processing_time_us = (float)total_cycles / 400.0f;  // 1.2 µs
block_time_us = (float)BLOCK_SIZE / SAMPLE_RATE * 1000000.0f;  // 1333 µs
cpu_utilization = processing_time_us / block_time_us * 100.0f;  // 0.09% per channel

Výsledek: 22.9 µs → CMSIS: 1.2 µs = 19× FASTER
// arm_biquad_cascade_df1_q31() - optimalizovaný fixed-point
cycles_per_sample_per_stage = 0.4f;  // Lepší performance než Float32
```

```
total_biquad_cycles = NUM_EQ_BANDS * BLOCK_SIZE * cycles_per_sample_per_stage;

// Q31 vector operations (slightly faster than Float32)
vector_add_cycles = NUM_EQ_BANDS * BLOCK_SIZE * 0.12f;
vector_scale_cycles = BLOCK_SIZE * 0.10f;

total_cycles = total_biquad_cycles + vector_add_cycles + vector_scale_cycles;
// Výsledek: 10 × 64 × 0.4 + 10 × 64 × 0.12 + 64 × 0.10 = 333 cycles per block

cpu_utilization = total_cycles / 400.0f / 1333.0f * 100.0f;  // 0.06% per channel
```

Real-time Audio Processing Application

```
#define AUDIO_CHANNELS      2           // Stereo
#define DMA_BUFFER_SIZE     (BLOCK_SIZE * 2)  // Double buffer

// Audio I/O buffers
static float32_t audio_input_buffer[DMA_BUFFER_SIZE * AUDIO_CHANNELS];
static float32_t audio_output_buffer[DMA_BUFFER_SIZE * AUDIO_CHANNELS];
static volatile uint32_t audio_buffer_ready = 0;

// User interface control
typedef struct {
    float band_gains[NUM_EQ_BANDS];     // Current band gain settings (dB)
    uint32_t bypass_enabled;            // EQ bypass flag
    float master_gain;                  // Master output gain (dB)
} eq_control_interface_t;

static eq_control_interface_t eq_controls = {
    .band_gains = {0.0f, 0.0f, 0.0f, 0.0f, 0.0f,   // Initialize to 0 dB
                   0.0f, 0.0f, 0.0f, 0.0f, 0.0f},
    .bypass_enabled = 0,
    .master_gain = 0.0f
};
```

ISR

```
// Audio DMA interrupt - triggered every 1.33 ms (64 samples @ 48 kHz)
void AUDIO_DMA_HalfComplete_IRQ(void) {
    audio_buffer_ready = 1;  // First half ready for processing
}

void AUDIO_DMA_Complete_IRQ(void) {
    audio_buffer_ready = 2;  // Second half ready for processing
}

// Main processing function called from DMA ISR
void process_audio_block_isr(void) {
    uint32_t buffer_offset = (audio_buffer_ready == 1) ? 0 : BLOCK_SIZE;

    // Extract left and right channels
    float32_t left_input[BLOCK_SIZE], right_input[BLOCK_SIZE];
    float32_t left_output[BLOCK_SIZE], right_output[BLOCK_SIZE];

    // Deinterleave stereo input
    for (uint32_t n = 0; n < BLOCK_SIZE; n++) {
        left_input[n] = audio_input_buffer[buffer_offset + n * 2];
```

```
        right_input[n] = audio_input_buffer[buffer_offset + n * 2 + 1];
    }
    // Apply equalizer processing
    if (!eq_controls.bypass_enabled) {
        process_cmsis_equalizer_f32(left_input, left_output);
        process_cmsis_equalizer_f32(right_input, right_output);
    } else {
        // Bypass mode - copy input to output
        arm_copy_f32(left_input, left_output, BLOCK_SIZE);
        arm_copy_f32(right_input, right_output, BLOCK_SIZE);
    }

    // Apply master gain
    if (eq_controls.master_gain != 0.0f) {
        float32_t gain_linear = powf(10.0f, eq_controls.master_gain / 20.0f);
        arm_scale_f32(left_output, gain_linear, left_output, BLOCK_SIZE);
        arm_scale_f32(right_output, gain_linear, right_output, BLOCK_SIZE);
    }

    // Interleave stereo output
    for (uint32_t n = 0; n < BLOCK_SIZE; n++) {
        audio_output_buffer[buffer_offset + n * 2] = left_output[n];
        audio_output_buffer[buffer_offset + n * 2 + 1] = right_output[n];
    }
}
```

Uživatelské presety

```
typedef enum {
    EQ_PRESET_FLAT,
    EQ_PRESET_ROCK,
    EQ_PRESET_JAZZ,
    EQ_PRESET_CLASSICAL,
    EQ_PRESET_VOCAL
} eq_preset_t;

void apply_equalizer_preset(eq_preset_t preset) {
    float preset_gains[5][NUM_EQ_BANDS] = {
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},           // Flat
        {3, 2, -1, -2, 1, 2, 3, 4, 3, 2},        // Rock
        {1, 1, 0, 1, 2, 1, 0, -1, 1, 2},         // Jazz
        {2, 1, -1, 0, 1, 0, -1, 2, 3, 2},        // Classical
        {-2, -1, 1, 3, 4, 3, 1, -1, -2, -1}      // Vocal
    };

    if (preset < 5) {
        for (uint32_t band = 0; band < NUM_EQ_BANDS; band++) {
            set_equalizer_band_gain(band, preset_gains[preset][band]);
        }
    }
}
```

Kvantifikované výsledky implementací

Implementation	Platform	Cycles/Block	CPU per Channel	Stereo CPU	Memory (KB)
Manual	STM32H743	9,160	1.72%	3.44%	5.2
CMSIS Float32	STM32H743	480	0.09%	0.18%	3.3
CMSIS Q31	STM32H743	333	0.06%	**0.12%**	3.1
Manual	STM32F407	15,200	7.2%	14.4%	5.2
CMSIS Q31	STM32F407	520	0.25%	0.50%	3.1