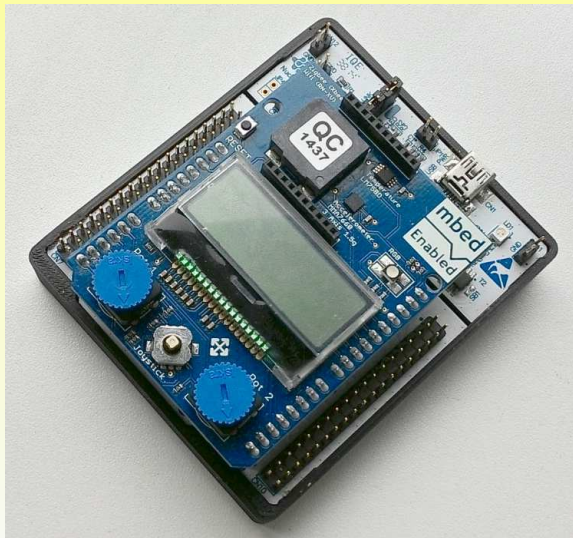


KAE/MINA - cvičení v 2018/19

Ing. Petr Weissar, Ph.D.

Ing. Petr Krist, Ph.D.



Plán cvičení – aktivity v laboratoři

1. Úvod, rozdělení kitů, opakování samostatně
 - GPIO - blikání LED, tlačítko,
 - SysTick,
 - USART
2. Pokročilé periférie ARM Cortex M
3. FPU - nastavení kompilátoru, porovnání rychlosti
4. Problém atomických operací, řešení pomocí bit-banding
5. Přerušení advanced - priority, blokování a řešení problémů
6. DMA - přenos bloku paměti, USART
7. DMA 2 - využití A/D převodníku a bufferu dat
8. Problematika Low-Power režimů
9. Privilegovaný a neprivilegovaný režim
10. SVC
11. Pokročilé techniky programování – zásobníky, semafore, instrukce LDREX/STREX, paměťové bariéry ...
12. RTOS - praktické řešení
13. Rezerva

Plán domácí přípravy

- I. Navázání znalosti z KAE/MPP
 - Spuštění známého prostředí Atollic TrueStudio (IDE+kompilátor+debugger) - Windows (příp. Linux)
 - Zopakování základní bloků práce s mikrokontrolérem
 - Jednoduchá práce s GPIO - blikání LED, čtení tlačítek.
 - Časování pomocí SysTick na 1ms
 - Sériová komunikace s PC pomocí USART a (USB virtuálního) COM portu
- II. Procvičení známých částí z KAE/MPP
 - Timer, PWM, řízení jasu LED, generování tónu
 - Použití LCD – připojen na SPI, má vlastní řadič
- III. Vyzkoušení efektu použití FPU pro float výpočty – výpočet Mandelbrotovy množiny
- IV. Ověření funkce bit-bandingu
- V. Ověření funkce priorit přerušení
- VI. Ověření činnosti DMA
- VII. Ověření spolupráce DMA a A/D převodníku
- VIII. Rezerva - doplnění chybějících věcí ze cvičení/příprav
- IX. Ověření funkce SVC a dalších pokročilých technik
- X. Prostor pro samostatnou práci - TODO
- XI. Prostor pro samostatnou práci - TODO
- XII. Prostor pro samostatnou práci - TODO

Organizace výuky, práce v laboratoři

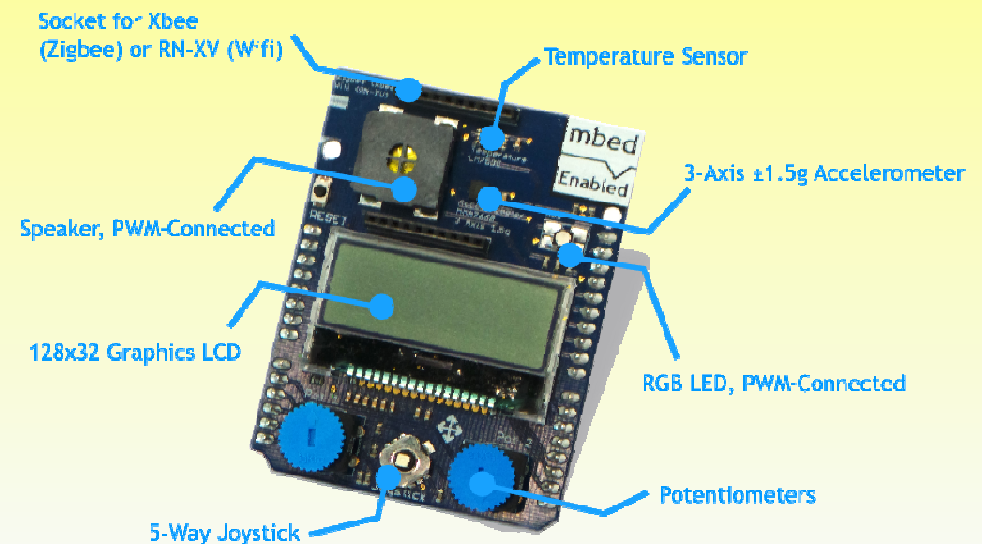
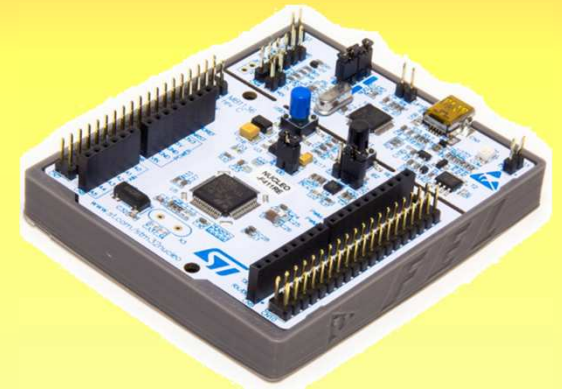
- V laboratoři pracujeme samostatně/ve dvojicích
 - Společný sdílený síťový adresář – disk T:
 - Domácí adresář studentů H:
 - Přilogování pomocí Orion-přístupu
 - Další síťové disky (R/O) - Z: (SW, podklady), X: (bat)
- HW společný – zapůjčen studentům
 - Předpokládá se část práce (příprava) doma
 - Uzavřena „smlouva o výpůjčce“
- Možno vlastní projekt založený na ARM Cortex-Mx procesoru
 - Nutno na počátku semestru schválit vyučujícím
 - Rozsahem musí odpovídat minimálně úkolům/látce ze cvičení
- Bezpečnost práce
 - Studenti musí mít 50-tku v laboratořích
- Bezpečnost elektroniky a zařízení
 - Kromě dodaných kitů veškeré další připojování HW až po **souhlasu cvičícího**

Podmínky zápočtu

- Splnění úkolů na cvičení
 - Možno (někdy nutno) dokončit v rámci domácí přípravy
- Znalost probírané problematiky na cvičeních i z "domácí přípravy"
 - Splněné kroky domácí přípravy
 - Na konci slajdů „domácí příprava“ seznam vyzkoušeného
 - Na počátku slajdů „cvičení“ seznam, co se má umět
 - Nesplnění jednotlivých etap přípravy ovlivní známku ze cvičení, se kterou se pak „jde“ ke zkoušce
- Vracení HW v pořádku
- U vlastního HW vhodné konzultace o samostatné práci
 - Doporučeno po 3-4 týdnech

Administrativa zapůjčení kitů

- Připraven formulář o "Zápůjčce"
 - Jedna kopie zůstává, druhou má student
- Kit obsahuje v krabičce
 - Nucleo STM32F411RE
 - MBED Application shield
 - Mini USB kabel
- Vracení HW na konci semestru
 - Ideálně ve 13. týdnu po předvedení funkčních úkolů/cvičení



Zdroje informací

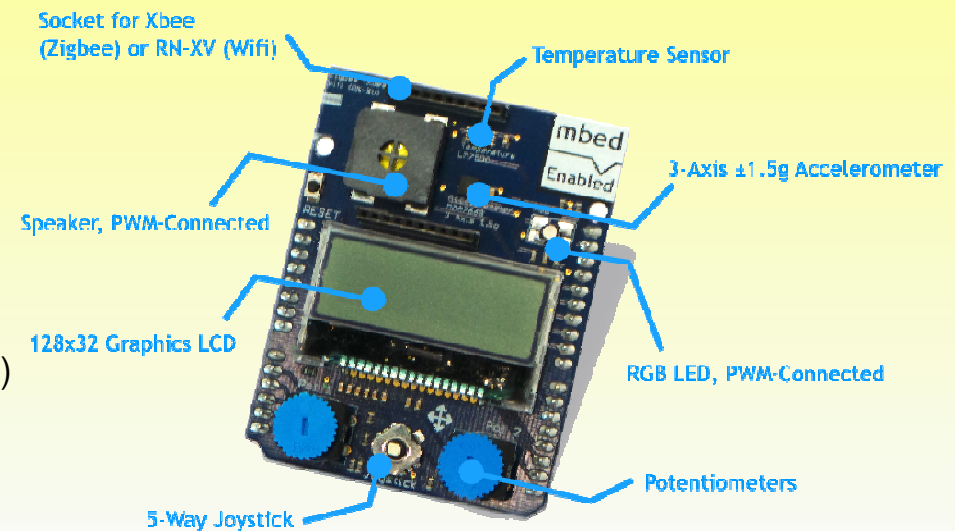
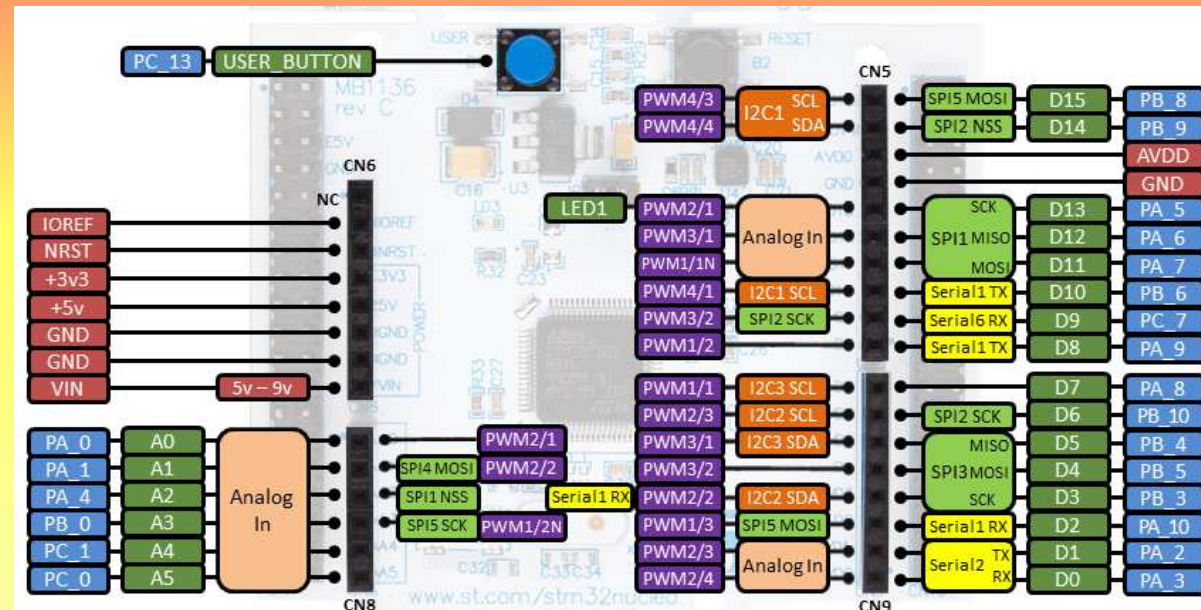
- Dostupné v elektronické podobě – typicky PDF:
 - Internet – www.st.com, mbed.org, ...
 - Vybrané na CourseWare nebo zde na Z:\podklady\MPP\...
- Podklady procesor:
 - Reference manual – RM0383 - STM32F411xC_E advanced ARM - V1_0 - en.DM00119316.pdf
 - Datasheet - DS10314 - STM32F411xC STM32F411xE - v6_0 en.DM00115249.pdf
 - Společný pro STM32F411xC STM32F411xE
 - ARM® Cortex®-M4 32b MCU+FPU, 125 DMIPS, 512KB Flash, 128KB RAM, USB OTG FS, 11 TIMs, 1 ADC, 13 comm. interfaces
 - Programming manual - PM0214 - F3 F4 L4 Programming manual - V5_0 en.DM00046982.pdf
 - STM32F3, STM32F4 and STM32L4 Series Cortex®-M4 programming manual
- Podklady Nucleo modul (<https://developer.mbed.org/platforms/ST-Nucleo-F411RE/>)
 - UM1724 - STM32 Nucleo-64 board - V11_0 en.DM00105823.pdf
 - Schematic - Nucleo schematic - rev C - MB1136.pdf
 - Pozor, Nucleo desky jsou rev. C-2 nebo C-3
- Podklady pro MBED Shield:
 - Schematic - NSHLD_SCH_1600xx_A.0_Basic.PDF
- Komponenty
 - Akcelerometr - MMA7660FC.pdf
 - <http://www.nxp.com/products/sensors/accelerometers/3-axis-accelerometers/1.5g-low-g-digital-accelerometer:MMA7660FC>

Opakování z KAE/MPP

- Pro psaní aplikací využíváme Atollic TrueStudio for STM32
 - V laboratoři v. 9.0.1, příklady funkční od 7.1.x
 - Stáhnout buď <https://atollic.com/truestudio/> nebo DropBox
 - https://www.dropbox.com/s/y764ln2incmhp8g/Atollic_TrueSTUDIO_for_STM32_windows_x86_v9.0.1_20180420-1214.exe?dl=0
- Debug pomocí vestavěného ST-Link modulu na Nucleo desce
- Využití UART procesoru
 - Rx a Tx signály připojeny na ST-Link, který komunikuje s ovladačem virtuálního COM portu v PC
 - Na straně PC použít terminál Putty nebo Hercules
- Blikání on-board LED
 - Připojena na PA5
 - Časování pomocí for cyklu
 - Časování pomocí SysTick – nastavit na přerušení 1ms a inkrementovat "ticks"

MBED Shield

- RGB LED
 - RED - PB4 - TIM3CH1
 - BLUE - PA9 - TIM1CH2
 - GREEN - PC7 - TIM3CH2
- 5-směrný joystick
 - PA4 = UP
 - PB0 = DOWN
 - PC1 = LEFT
 - PC0 = RIGHT
 - PB5 = CENTER
- Speaker – připojen na PB10 (= TIM2CH3)
- 2x potenciometr – piny PA0 a PA1
- LCD s rozlišení 128x32 (řadič ST7565R)
 - Připojen přes SPI, pouze signály SCK a MOSI (= nelze číst)
- Akcelerometr - MMA7660 – připojen přes I²C
- Teploměr – LM75B – připojen přes I²C



Úkoly na MBED shield

- Blikat složkami RGB LED
- Číst stav "tlačítek" joysticku
 - Vypisovat na UART – terminál v PC
 - Měnit "barvu blikání" LED
- Připravit si makra pro snazší využití
 - Např. soubor mbed_shield.h
 - Inspirace v nucleo_core.h
 - #define BOARD_LED GPIOA,5
 - Použito např. ve funkcích GPIOWrite a Nucleo_SetPinGPIO

Plán cvičení – aktivity v laboratoři

1. Úvod, rozdělení kitů, opakování samostatně
2. Pokročilé periférie ARM Cortex M
 - Timery, PWM
 - Využití LCD (SPI komunikace)
3. FPU - nastavení kompilátoru, porovnání rychlosti
4. Problém atomických operací, řešení pomocí bit-banding
5. Přerušení advanced - priority, blokování a řešení problémů
6. DMA - přenos bloku paměti, USART
7. DMA 2 - využití A/D převodníku a bufferu dat
8. Problematika Low-Power režimů
9. Privilegovaný a neprivilegovaný režim
10. SVC
11. Pokročilé techniky programování – zásobníky, semafore, instrukce LDREX/STREX, paměťové bariéry ...
12. RTOS - praktické řešení
13. Rezerva

Shrnutí domácí přípravy

- Máme připravené konstanty pro využití MBED Shield
- Umíme ovládat RGB LED "digitálně"
- Umíme číst stav joysticku
- Umíme komunikovat přes UART

Řízení LCD na MBED Shield

- Připojen přes SPI
 - Signály PA5 a PA7 odpovídají v SPI1 signálům SCK a MOSI (není MISO = nelze číst z displeje)
 - Komunikace 8-bitová, max. takt SCK = 10MHz (100ns perioda)
 - Signály PA6 = /RST, PB6 = /CS, PA8 = A0 (0 = zápis příkazů, 1 = zápis dat)
- Řadič ST7565R
 - Max. rozlišení 132x65, fyzické rozměry LCD 128x32 pixelů
 - Data organizována po řádcích po 8 pixelech
 - Sloupec odpovídá 1 bajtu při přenosu
 - MSB nahoře

Knihovna pro LCD

- K dispozici na Z: nebo <https://github.com/weissar/lcd-mbed-shield-stm32f>
 - Včetně jednoduchého příkladu použití
- Obsahuje 4 soubory
 - mbed_shield_lcd.c – výkonná těla funkcí
 - mbed_shield_lcd.h – hlavičky veřejného rozhraní
 - mbed_shield_lcd_defines.h – interní "include" definující HW rozhraní a "systémové" funkce
 - Umožňuje definovat, které funkce nastaví GPIO (včetně konstant) a Alternate Function
 - font_8x8.h – interní konstantní data s definicí 128 znaků fontu 8x8
- Obsahuje interní buffer 128x4 bajty (= video RAM)
 - Řeší problém nemožnosti číst z LCD při pixelových (= bitovém maskování) operacích
 - Nutno "ručně" zavolat funkci MBED_LCD_VideoRam2LCD
 - Výhledově připraveno pro automatický refresh pomocí časovače a DMA/SPI přenosů

Ověření funkce LCD knihovny

- Vypisujte měnící se text na displeji
 - Např. počítadlo nebo znaky přicházející po UARTu
- Vykreslete obdélník a kruh
 - Příp. další grafická primitiva (čára, pixel)

Ovládání složek RGB LED pomocí PWM

- Využijeme kód z MPP příkladů
 - Zjistit AF hodnoty v DS
- Časovače využity následovně
 - RED - PB4 – TIM3CH1
 - BLUE - PA9 – TIM1CH2
 - GREEN – PC7 - TIM3CH2
- **POZOR** – TIM1 má navíc povolovací bit MOE v registru BDTR
- Doporučená rychlost PWM v řádu kHz
- Ideální rozlišení 256 hodnot PWM
 - Snadno se generují 24-bitové barvy (3x složky po 8b)

Generování frekvence pomocí PWM

- Pro speaker uvažujeme digitální signál
 - Pin PB10 odpovídá TIM2CH3 (naštěstí jiný než RGB LED)
- Nastavit střihu 50% pro "rozumnou" frekvenci
- "Pípnout" např. při stisknutí spínače joysticku

Dokončit doma

- Pomocí terminálu umět nastavit libovolnou RGB barvu
- Na displeji vypisovat hodnotu RGB složek
- Pípnout při určité hodnotě nebo "na požádání"

Plán cvičení – aktivity v laboratoři

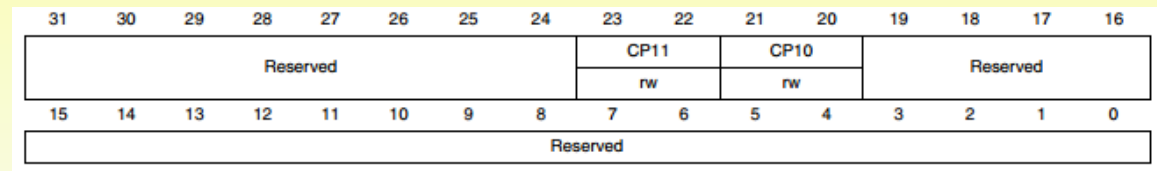
1. Úvod, rozdělení kitů, opakování samostatně
2. Pokročilé periférie ARM Cortex M
3. FPU - nastavení kompilátoru, porovnání rychlosti
 - Volby při vytváření projektu, úprava existujícího projektu
 - Příklad algoritmu – výpočet Mandelbrotovy množiny (s vizualizací na LCD), využití příkladu na PC (C#)
 - Porovnání rychlosti výpočtu (příp. pro 10x opakování)
4. Problém atomických operací, řešení pomocí bit-banding
5. Přerušení advanced - priority, blokování a řešení problémů
6. DMA - přenos bloku paměti, USART
7. DMA 2 - využití A/D převodníku a bufferu dat
8. Problematika Low-Power režimů
9. Privilegovaný a neprivilegovaný režim
10. SVC
11. Pokročilé techniky programování – zásobníky, semafore, instrukce LDREX/STREX, paměťové bariéry ...
12. RTOS - praktické řešení
13. Rezerva

Kontrola DCV

- Nastavení libovolné RGB barvy na LED
- Zobrazení hodnot RGB složek na LCD

FPU – Floating Point Unit

- FPU jednotka je společná všem Cortex-M4F (tj. série STM32F3, F4 a L4)
 - Dokumentace uvedena v Programming Manual (PM) – kap. 4.6
 - Přímo podporuje operace v "single-precision" – sčítání, odčítání, násobení, dělení, druhou odmocninu a "násobení a součet" (např. pro výpočet filtrů)
 - Reprezentace 32b podle standardu IEEE754
- Z programu je FPU přístupná přes 32 datových registrů a řídicích registrů v "System Control Block" (SCB)
 - CPACR - Coprocessor access control register = řízení přístupu v bitech CPn
 - Teoreticky pro n koprocetorů, ale přítomen je pouze 1
 - 00 – access denied = přístup k FPU generuje vyjímku UsageFault
 - 01 – jen privilegovaný přístup = v neprivilegovaném režimu generuje vyjímku
 - 10 – vyhrazeno = nedokumentované chování
 - 11 – plný přístup



- Pozor, bity CPn nejsou pojmenovány v .H souborech, možný test např.:

```
if (SCB->CPACR & ((1 << 20) | (1 << 21))) // CPACR registr, CP10 bits, see. PM 4.6.1 / pg. 252
    isFPUHW = true;
```

Použití FPU v aplikaci a detekce FPU režimu

- Za běžných podmínek je použití FPU volbou pro kompilace a linkování s příslušnými knihovnami (podporujícími HW nebo SW řešení float operací)
- Přítomnost podpory FPU lze v GCC překladači otestovat existencí makra/symbolu `__FPU_USED`
- Nastavení linkeru a kompilátoru pro "náš" procesor STM32F4xx přidává do příkazové řádky pro HW FPU přepínače `-mfloat-abi=hard -mfpv4-sp-d16`
 - Možno příkazovou řádku zkontrolovat v Settings/C Compiler – All options (resp. C Linker)
- Reálně nebudeme FPU za běhu programu zapínat/vypínat, protože bychom museli mít vlastní knihovny, které by pro každou "floatovou" operaci musely kontrolovat, zda je FPU právě aktivní

Implementace v kompilátoru

- Standardní chování v GCC

- Volby pro HW FPU se testují v kódu funkce `void SystemInit(void)` v souboru `system_stm32f4xx.c`

```
/* FPU settings -----*/  
#if (__FPU_PRESENT == 1) && (__FPU_USED == 1)  
    SCB->CPACR |= ((3UL << 10*2)|(3UL << 11*2)); /* set CP10 and CP11 Full Access */  
#endif
```

- Při volbě HW implementace se používají přímo FPU instrukce

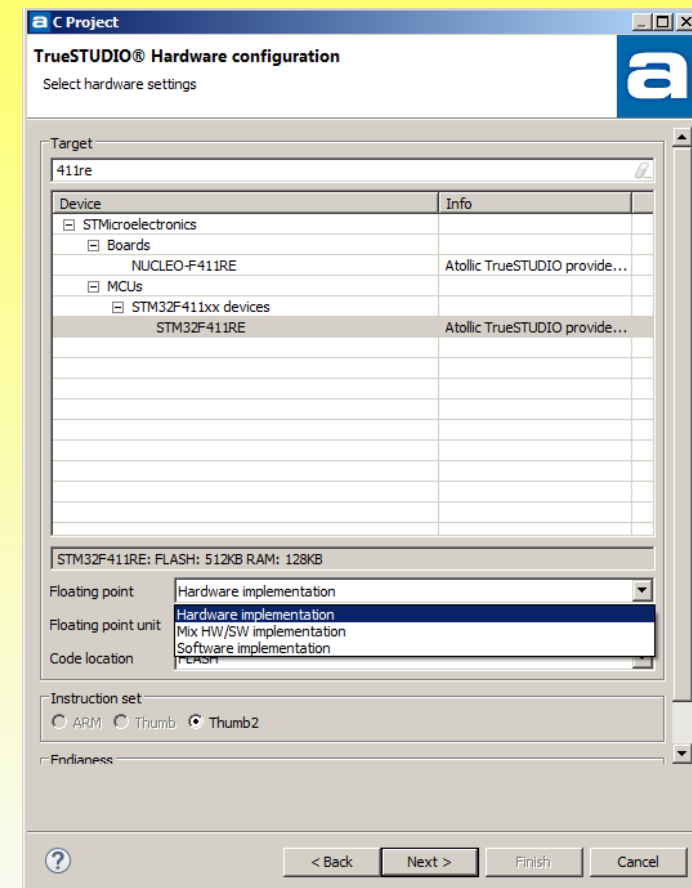
```
32          float x0 = xx;  
080007d0:   ldr     r3, [r7, #40]    ; 0x28  
080007d2:   vmov    s15, r3  
080007d6:   vcvtf32.s32    s15, s15  
080007da:   vstr     s15, [r7, #20]
```

- Při volbě SW implementace se volají knihovní funkce s SW implementací float výpočtů = pomalé

```
32          float x0 = xx;  
08000d78:   ldr     r0, [r7, #40]    ; 0x28  
08000d7a:   bl      0x8000874 <__floatsisf>  
08000d7e:   mov     r3, r0  
08000d80:   str     r3, [r7, #20]
```

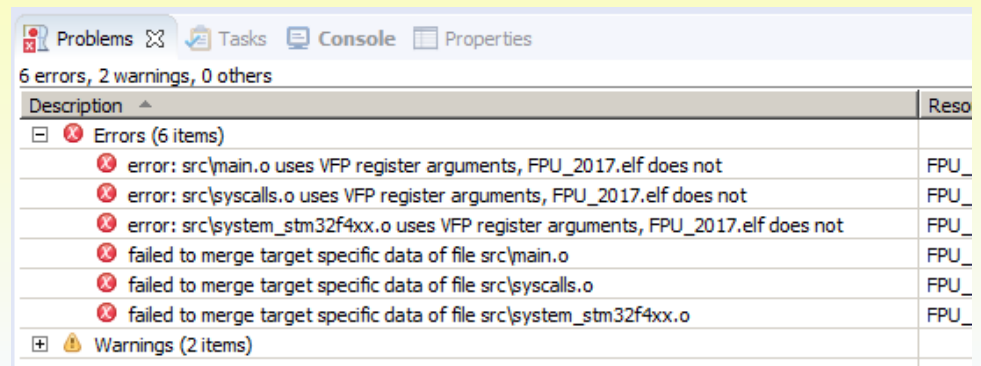
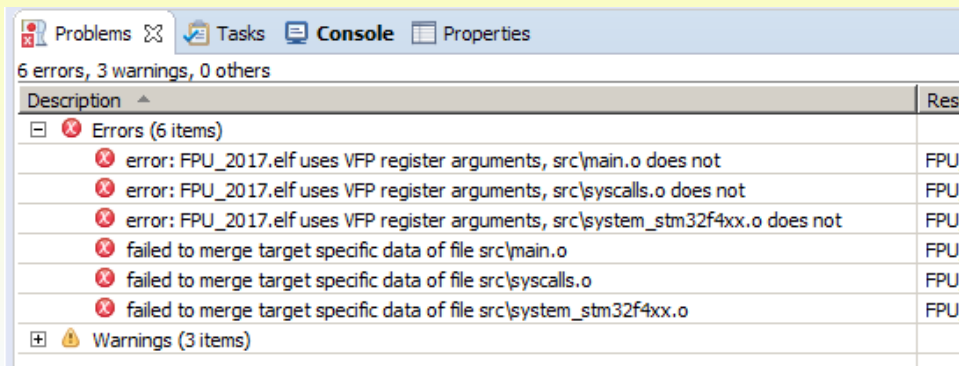
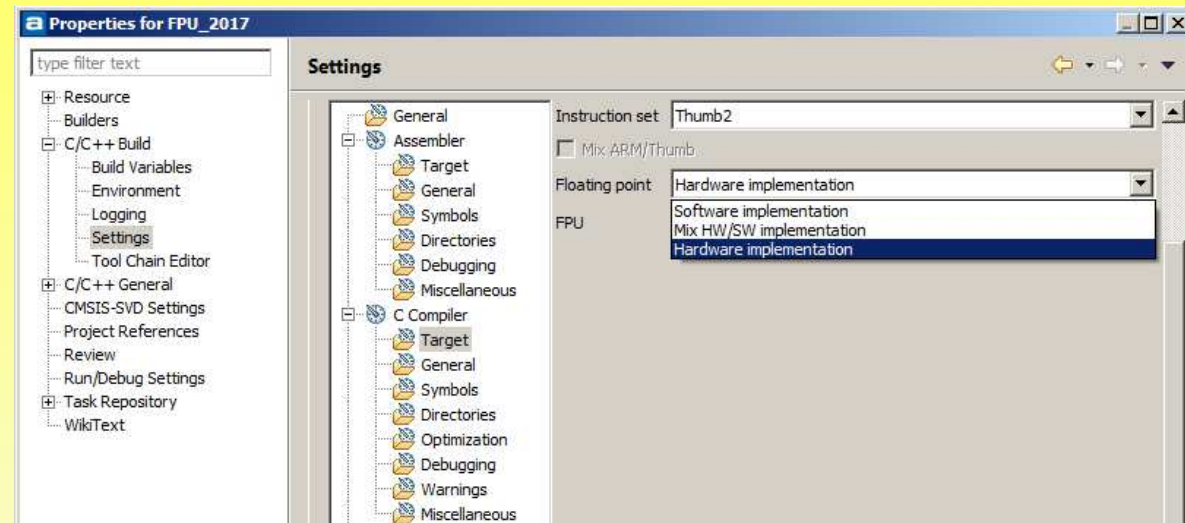
Nastavení při vytváření projektu

- Stačí zvolit v rámci výběru procesoru
- Defaultně je nastaven použití HW FPU

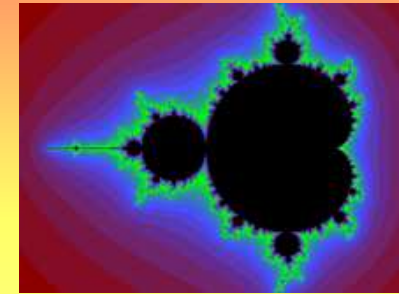


Změna ve stávajícím projektu

- Nutno nastavit jak C Compiler, tak C Linker
- Při "neshodě" generuje build řadu chyb
 1. Compiler SW, Linker HW
 2. Compiler HW, Linker SW



Výpočet a zobrazení Mandelbrotovy množiny



- Jedna z mnoha fraktálních matematických "hříček" vyžadující hodně výpočtů
 - Pro body komplexní roviny (2D prostor) se přiřadí rekurzivní posloupnost hodnot a body množiny mají tuto posloupnost omezenou
 - Fraktálem je hranice takové množiny
 - https://cs.wikipedia.org/wiki/Mandelbrotova_mno%C5%BEina
- C# demo
 - Přizpůsobené displeji 128x32 pixelů
 - Umožňuje posun X a Y a Zoom, vypisuje aktuální "střed" a zoom, dále dobu výpočtu [ms]
 - Funkci výpočtu pochází z C/C++ implementace a lze ji přímo zkopírovat do projektu pro ARM
 - Zdrojový projekt na Z:\podklady\MINA a na <https://github.com/weissar/Mandelbrot-demo-128x32-LCD>
- Úprava zobrazovacího kódu – zjednodušení:

```
if (z < 4)
    MBED_LCD_PutPixel(xx, yy, true);
else
    MBED_LCD_PutPixel(xx, yy, false);
```

Přenos z C# (PC) do ARMu

- Měření času – přesnost na ms
 - Před výpočtem zjistit stav počet ms v _ticks
 - Po výpočtu odečíst uložený stav od aktuálních _ticks a výsledkem je doba běhu algoritmu
 - Vypsát do vykresleného grafu dobu výpočtu
- Po dobu výpočtu rozsvítit složku RGB LED
- Zobrazení na LCD po změně joy-tlačítek
 - Implementovat nahoru, dolů, vlevo a vpravo, střed bude Zoom+, neumíme Zoom-
 - Alternativně možno využít potenciometry ?
- Dokončení jako DCV a změřit dobu výpočtu
 - Pro HW a SW typ výpočtu
 - Pro defaultní hodiny 16MHz HSI a pro max. 100MHz hodiny (PLL z HSI 16MHz nebo HSE 8MHz)

Plán cvičení – aktivity v laboratoři

1. Úvod, rozdělení kitů, opakování samostatně
2. Pokročilé periférie ARM Cortex M
3. FPU - nastavení kompilátoru, porovnání rychlosti
4. Problém atomických operací, řešení pomocí bit-banding
 - Přístup k registrům pomocí maskování = více instrukcí = zdroj problémů
 - Atomické operace podpořené GPIO registrem
 - Bit-banding pro registry i RAM
5. Přerušení advanced - priority, blokování a řešení problémů
6. DMA - přenos bloku paměti, USART
7. DMA 2 - využití A/D převodníku a bufferu dat
8. Problematika Low-Power režimů
9. Privilegovaný a neprivilegovaný režim
10. SVC
11. Pokročilé techniky programování – zásobníky, semafore, instrukce LDREX/STREX, paměťové bariéry ...
12. RTOS - praktické řešení
13. Rezerva

Kontrola DCV

- Možnost posuvu grafického zobrazení Mandelbrotovy množiny kurzorem
- Výpis doby výpočtu na displeji (nebo terminálu)
- Změřené hodnoty

	16MHz (HSI)	100MHz
FPU float		
SW float		

Problém atomických operací

- Při požadavku změny hodnoty proměnné v RAM nebo registru (tj. adresa) generuje překladač sekvenci instrukcí
- Pokud během této sekvence přijde přerušení, kde se pracuje se stejnou adresou, může dojít k problému:
 - Negovat port PC3 v IRQ od TIM3
 - Negovat port PC2 v hlavní smyčce programu



```
void TIM3_IRQHandler(void)
{
    TIM3->SR &= ~TIM_SR_UIF;

    GPIOC->ODR ^= 1 << 3;
}
```

```
Nucleo_SetPinGPIO(GPIOC, 2, ioPortOutputPP);
Nucleo_SetPinGPIO(GPIOC, 3, ioPortOutputPP);

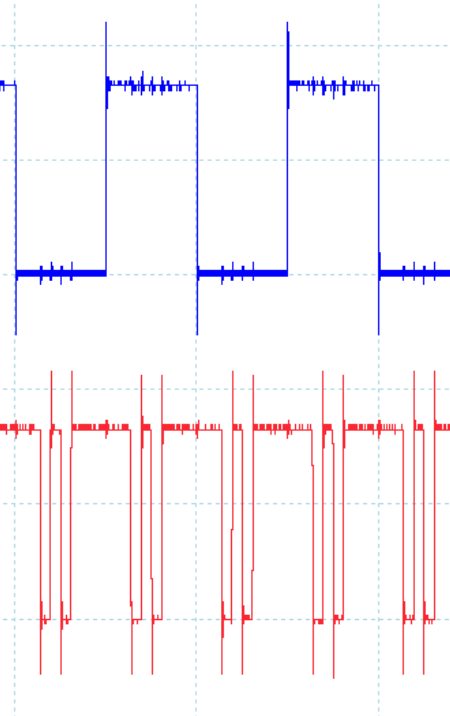
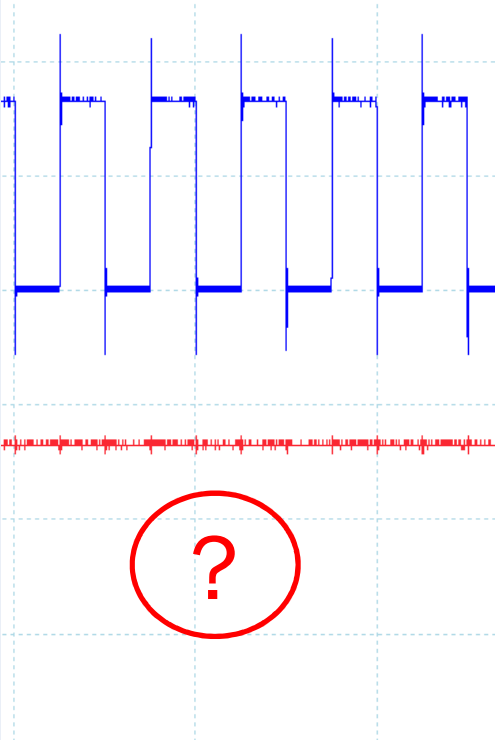
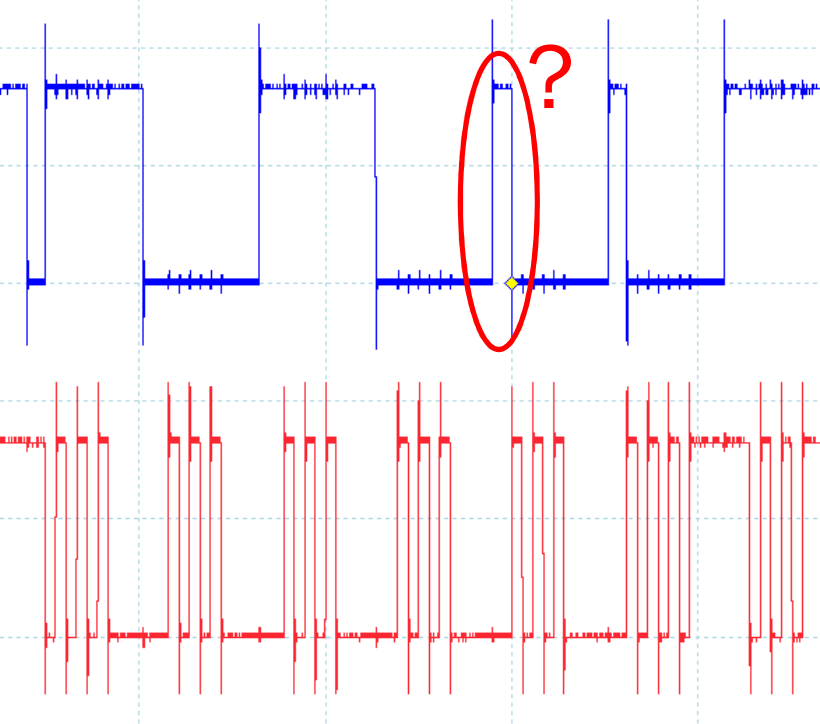
RCC->APB1ENR |= RCC_APB1ENR_TIM3EN;

TIM3->CR1 = TIM_CR1_DIR;    // DIR = 1 = cnt-down
TIM3->PSC = 1;              // : 2 = 8MHz
TIM3->ARR = 40 - 1;         // : 40 = 200kHz
TIM3->CR1 |= TIM_CR1_CEN;   // CEN = 1 = enable

TIM3->DIER = TIM_DIER_UIE;
NVIC_EnableIRQ(TIM3_IRQn);

while (1)
{
    GPIOC->ODR ^= 1 << 2;
}
```

Vliv četnosti přerušení na výsledek

PSC = 1, ARR = 40 - 1	PSC = 1, ARR = 20 - 1	PSC = 1, ARR = 50 - 1
TIM3 IRQ = 200kHz, 5us	TIM3 IRQ = 400kHz, 2,5us	TIM3 IRQ = 160kHz, 6,25us
		

Analýza časování

- Přibližně vyčteno z osciloskopu pro $ARR = 40 - 1$
 - While smyčka trvá 0,6us (čas mezi hranami na PC2, stopa 2)
 - Doba bez hrany při vykonání IRQ cca 3,3us
 - Tedy doba samotného IRQ = $3,3 - 0,6 = 2,7us$
 - Problém při periodě TIM3 IRQ = 2,5us !!!
- Chyby v průběhu pro $ARR = 50 - 1$
 - Záložka Disassembly v Debug pohledu
 - V instrukcích je viditelná sekvence Read-Modify-Write
 - Stejný kód v main i v IRQ
 - "Modify" operace je zde **eor** (=XOR)
 - Přerušení se může "trefit" mezi **ldr** (Load/Read) a **str** (Store/Write)
 - Do ODR registru se uloží hodnota načtená před přerušením
 - Ten v IRQ změněný stav se "ztratí"

```

66          GPIOC->ODR ^= 1 << 2;
08000380:  ldr    r2, [pc, #20] ; ...
08000382:  ldr    r3, [pc, #20] ; ...
08000384:  ldr    r3, [r3, #20]
08000386:  eor.w  r3, r3, #4
0800038a:  str    r3, [r2, #20]
  
```

Main/while	R3 (nebo jiný)	IRQ	R3	ODR
Load ODR	xx 0000			xx 0 0 0 0
XOR 0010	xx 0010			xx 0 0 0 0
		Ulož registry	xx	xx 0 0 0 0
		Load ODR	xx 0000	xx 0 0 0 0
		XOR 0100	xx 0100	xx 0 0 0 0
		Store ODR	xx 0100	xx 0 1 0 0
	xx 0010	Vrať registry		xx 0 1 0 0
Store ODR	xx 0010			xx 0 0 1 0
...				xx 0 0 1 0
while(1) návrat				

Řešení chyb při read-modify-write přístupu

- Pro GPIO použít registr BSRR

- Zápisem log. 1 na určený bit vynuluje nebo nastaví výstup bez ovlivnění ostatních
- + vhodné řešení
- - existuje jen jako náhrada ODR u GPIO
- - toggle nutno realizovat if/else-em

```
if (GPIOC->ODR & (1 << 2))  
    GPIOC->BSRR = 1 << (2 + 16); // BRy part  
else  
    GPIOC->BSRR = 1 << (2 + 16); // BSx part
```

- Během akce v main zakázat přerušení

- - reálně nepoužitelné

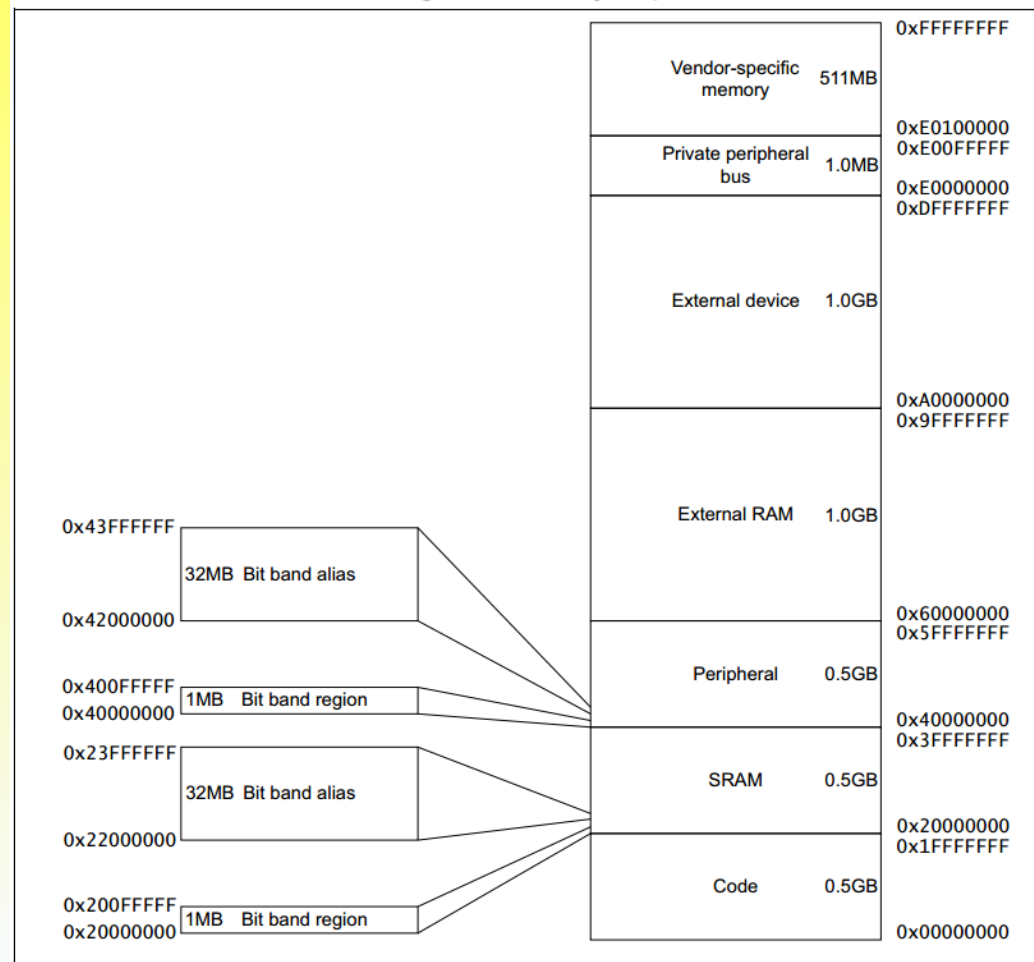
- Využít bit-banding

- + určené pro takové použití
- + relativně jednoduché použití v kódu
- + lze použít pro jakýkoliv registr i pro adresy paměti RAM
- - relativně složité makro při prvním přiblížení
- - podle možností překladače méně optimální kód

Bit-banding adresy

- PM kap. 2.2 a 2.2.5
 - Bit-banding = každému bitu odpovídá jedna 32-bitová adresa
- 1MB adresy registrů
 - Rozsah 0x4000 0000 – 0x400F FFFF
- 32MB – Bit-band alias
 - Rozsah 0x4200 0000 – 0x43FF FFFF
- 1MB RAM (SRAM na čipu)
 - Rozsah 0x2000 0000 – 0x200F FFFF
- 32MB – Bit-band alias
 - Rozsah 0x2200 0000 – 0x23FF FFFF

Figure 8. Memory map



Výpočet bit-banding aliasu

- Dle PM 2.2.5

```
bit_word_offset = (byte_offset x 32) + (bit_number x 4)
bit_word_addr = bit_band_base + bit_word_offset
```

- Příklad mapování RAM do BB:

- Bit 7 adresy 0x2000 0000
- Odpovídá 32b slovu na adrese $0x2200\ 0000 + 0 * 32 + 4 * 7 (=28=0x1c) = 0x2200\ 001c$

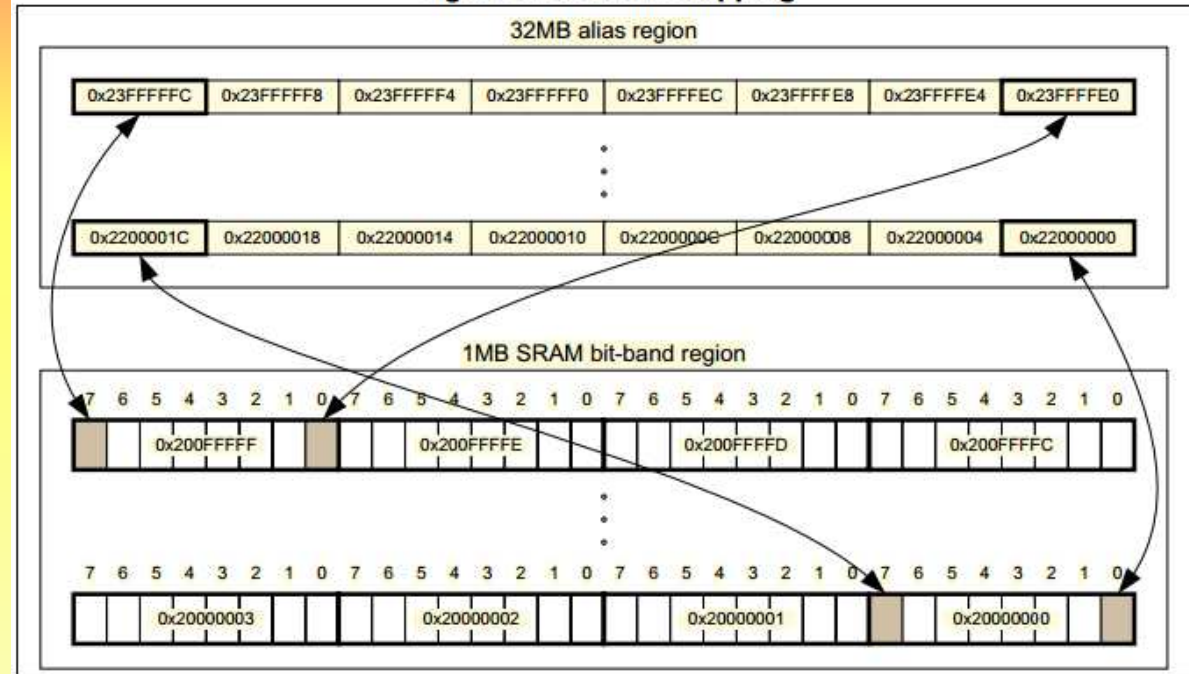
- V souboru **stm32f411xe.h** (inkludován z **stm32f4xx.h**) jsou konstanty

```
#define FLASH_BASE      ((uint32_t)0x08000000) /*!< FLASH(up to 1 MB) base address in the alias region */
#define FLASH_END      ((uint32_t)0x0807FFFF) /*!< FLASH end address */
#define SRAM1_BASE      ((uint32_t)0x20000000) /*!< SRAM1(112 KB) base address in the alias region */
#define SRAM1_BB_BASE   ((uint32_t)0x22000000) /*!< SRAM1(112 KB) base address in the bit-band region */

#define PERIPH_BASE      ((uint32_t)0x40000000) /*!< Peripheral base address in the alias region */
#define PERIPH_BB_BASE  ((uint32_t)0x42000000) /*!< Peripheral base address in the bit-band region */

#define SRAM_BASE        SRAM1_BASE
#define SRAM_BB_BASE     SRAM1_BB_BASE
```

Figure 9. Bit-band mapping



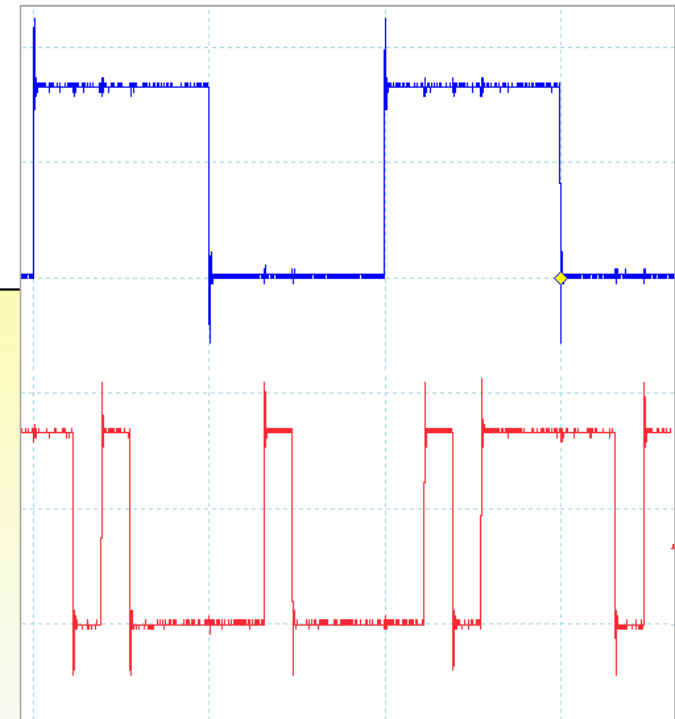
Využití bit-bandingu v úvodním kódu

```
#define GPIOC_ODR_B2 (*(uint32_t *) (0x42000000 + ((uint32_t) (&(GPIOC->ODR)) - 0x40000000) * 32 + 4 * 2))
#define GPIOC_ODR_B3 (*(uint32_t *) (0x42000000 + (GPIOC_BASE + 0x14 - 0x40000000) * 32 + 4 * 3))

...
void TIM3_IRQHandler(void)
{
    ...
    // GPIOC->ODR ^= 1 << 3;
    GPIOC_ODR_B3 = !GPIOC_ODR_B3;
    ...
    while (1)
    {
        // GPIOC->ODR ^= 1 << 2;
        GPIOC_ODR_B2 = !GPIOC_ODR_B2;
        ...
    }
}
```

- Problém vyřešen 😊
- Pro B2 je využito získání adresy registru ODR
 - Operátor reference &
- Pro B3 je využita bazová adresa registrů periférie GPIOC a pozice registru ODR
 - Viz. RM 8.4.6

8.4.6 GPIO port output data register (GPIOx_ODR) (x = A..E and H)
Address offset: 0x14
Reset value: 0x0000 0000



Makra pro bit-banding

```
#define BB_REG(reg, bit) (*(uint32_t *) (PERIPH_BB_BASE + ((uint32_t)(&(reg)) - PERIPH_BASE) * 32 + 4 * (bit)))  
#define BB_RAM(ad, bit) (*(uint32_t *) (SRAM_BB_BASE + ((uint32_t)(ad) - SRAM_BASE) * 32 + 4 * (bit)))
```

- V makrech je potřeba přetypovávat a závorkovat !!!
- Možné použití

```
BB_REG(GPIOC->ODR, 3) = ~BB_REG(GPIOC->ODR, 3);  
.. Nebo ..  
BB_REG(GPIOC->ODR, 3) = !BB_REG(GPIOC->ODR, 3);  
.. Nebo ..  
BB_REG(GPIOC->ODR, 3) ^= 1;
```

- Dle PM je rozhodující nejnižší bit, ostatní jej při zápisu "kopírují", proto se dá použít i XOR 1
- Výsledný kód v assembleru se ale liší použitou instrukcí – viz záložka Disassembly
- MVNS = "Move NOT"
- CMP, ITE eq = Compare, IfThenElse equal
- UXTB extracts bits[7:0] and zero extends to 32 bits (??)
- EOR = Exclusive OR (= XOR)

```
40      BB_REG(GPIOC->ODR, 3) = ~BB_REG(GPIOC->ODR, 3);  
080002cc: ldr     r2, [pc, #48] ;  
080002ce: ldr     r3, [pc, #48] ;  
080002d0: ldr     r3, [r3, #0]  
080002d2: mvns    r3, r3  
080002d4: str     r3, [r2, #0]  
41      BB_REG(GPIOC->ODR, 3) = !BB_REG(GPIOC->ODR, 3);  
080002d6: ldr     r2, [pc, #40] ;  
080002d8: ldr     r3, [pc, #36] ;  
080002da: ldr     r3, [r3, #0]  
080002dc: cmp     r3, #0  
080002de: ite     eq  
080002e0: moveq   r3, #1  
080002e2: movne   r3, #0  
080002e4: uxtb    r3, r3  
080002e6: str     r3, [r2, #0]  
42      BB_REG(GPIOC->ODR, 3) ^= 1;  
080002e8: ldr     r2, [pc, #20] ;  
080002ea: ldr     r3, [pc, #20] ;  
080002ec: ldr     r3, [r3, #0]  
080002ee: eor.w   r3, r3, #1  
080002f2: str     r3, [r2, #0]
```

Využití bit-banding pro přístup k RAM

- Každá proměnná (nejen) v RAM má svůj bitband alias
 - Je nutné jen zjistit adresu proměnné v paměti – operátor reference
 - "Naše" makro vyžaduje v parametru právě tuto adresu
- Příklad opakovaně vypisuje na terminál písmena A-Z
- Pomocí bit-band přístupu negujeme hodnotu bitu 5 (rozdíl mezi malým a velkým písmenem v ASCII)
 - Je možné použít opět celkovou logickou i bitovou negaci nebo XOR
 - Funguje i pro 8-bitové proměnné, ale předpokládáme alignment na 32 bitů
- Zamyšlení – proč je podmínka if právě taková ?

```
...
int main(void)
{
    uint32_t tm = 0;
    uint8_t b = 'A';

    SystemCoreClockUpdate();
    SysTick_Config(SystemCoreClock / 1000);
    Usart2InitStd(38400);

    while (1)
    {
        if (_ticks >= tm)
        {
            tm = _ticks + 100;

            putchar(b);
            b++;
            if (b > 'z')
                b = 'a';

            BB_RAM(&b, 5) = ~BB_RAM(&b, 5);
            // BB_RAM(&b, 5) ^= 1;
        }
    }
}
```

Bit-banding pro paměť – optimalizace kódu

- Použitý překladač GCC výpočet v makru pro paměť provádí při každém průchodu a to dokonce 2x – pro load i store ☹
 - Operace negace ani XOR to nezlepší
- Řešení samostatně
 - Použít přímo ukazatel na bit-band adresu odpovídající 5. bitu proměnné **b** (resp. její adrese v RAM)
 - Porovnat přeložený kód v Disassembleru

```
105          BB_RAM(&b, 5) = ~BB_RAM(&b, 5);
080003a6:  adds    r3, r7, #7
080003a8:  add.w   r3, r3, #17825792
080003ac:  lsls    r3, r3, #5
080003ae:  adds    r3, #20
080003b0:  mov     r2, r3
080003b2:  adds    r3, r7, #7
080003b4:  add.w   r3, r3, #17825792
080003b8:  lsls    r3, r3, #5
080003ba:  adds    r3, #20
080003bc:  ldr     r3, [r3, #0]
080003be:  mvns    r3, r3
080003c0:  str     r3, [r2, #0]
106          BB_RAM(&b, 5) ^= 1;
080003c2:  adds    r3, r7, #7
080003c4:  add.w   r3, r3, #17825792
080003c8:  lsls    r3, r3, #5
080003ca:  adds    r3, #20
080003cc:  mov     r2, r3
080003ce:  adds    r3, r7, #7
080003d0:  add.w   r3, r3, #17825792
080003d4:  lsls    r3, r3, #5
080003d6:  adds    r3, #20
080003d8:  ldr     r3, [r3, #0]
080003da:  eor.w   r3, r3, #1
080003de:  str     r3, [r2, #0]
```

DCV

- Dokončit příklady včetně dřívějších úkolů

Plán cvičení – aktivity v laboratoři

1. Úvod, rozdělení kitů, opakování samostatně
2. Pokročilé periférie ARM Cortex M
3. FPU - nastavení kompilátoru, porovnání rychlosti
4. Problém atomických operací, řešení pomocí bit-banding
5. Přerušování advanced - priority, blokování a řešení problémů
6. DMA - přenos bloku paměti, USART
7. DMA 2 - využití A/D převodníku a bufferu dat
8. Problematika Low-Power režimů
9. Privilegovaný a neprivilegovaný režim
10. SVC
11. Pokročilé techniky programování – zásobníky, semafore, instrukce LDREX/STREX, paměťové bariéry ...
12. RTOS - praktické řešení
13. Rezerva

Kontrola DCV

- Jak spočítat adresu Bit-band registru pro bit v paměti
- Jak ji použít jako proměnnou typu ukazatel pro přístup typu negace (XOR)
- Jak se liší výsledný kód (počet instrukcí) ve srovnání s použitím makra BB_RAM ?
 - Předpokládejme vypnuté optimalizace

Více aktivních přerušení, priority

- Testovací aplikace obsahuje minimálně 2 přerušení
 - Časovač (např. TIM3) v taktu 100ms
 - V přerušení negovat výstup LED – např. LED_R = PB4
 - Frekvence blikání = ?? ☺
 - Externí přerušení od tlačítka – střed joysticku = PB5
 - Přerušení bude blokující, dokud není uvolněno tlačítko, kód neopustí přerušovací funkci
 - Před blokováním rozsvítit LED_G, po uvolnění zhasnout = kontrola činnosti = PC7
 - **POZOR, LED svítí při log. 0, stisknuté tlačítko je log. 1**
 - Alternativně přerušení od SysTick = počítá stále tiky ?
 - Lze ověřit např. posíláním hodnoty na terminál v hlavní smyčce
- Očekávaný výsledek
 - Pokud mají obě přerušení stejnou prioritu, externí přerušení zablokuje blikání
 - Pokud má externí přerušení vyšší prioritu, dtto
 - Pokud má TIMx přerušení větší prioritu, dokáže přerušit obsluhu externího = bliká stále
- Vyzkoušet si měnit priority a sledovat chování (možno i v Debug-u)

Stručně blikání LED v přerušení

- Vyzkoušet jako první krok
- Využít Bit-Banding !!
 - Není třeba koordinovat přístupy k ODR registrům

```
...
void TIM3_IRQHandler(void)
{
    TIM3->SR &= ~TIM_SR_UIF;

    BB_REG(GPIOB->ODR, 4) ^= 1;      // MBED_LED_R
}
...
Nucleo_SetPinGPIO(MBED_LED_R, ioPortOutputPP);
BB_REG(GPIOB->ODR, 4) = 1;
...
TIM3->PSC = GetBusClock(timersClockAPB1) / 1000;      // 1ms
TIM3->ARR = 100 - 1;  // 0.1 sec
TIM3->CR1 |= TIM_CR1_CEN;  // CEN = 1 = enable

TIM3->DIER = TIM_DIER_UIE;
NVIC_EnableIRQ(TIM3_IRQn);
...
```

Externí přerušení

- Každý GPIO může generovat přerušení
 - Ze všech GPIO je přerušení pro bity 0, bit 1, ... 4
 - Společné přerušení pro bity 5-9 a bity 10-15 (nutno v IRQ detekovat "který to byl")
 - Pro daný bit se vybírá gpioX v registrech EXTICRx v bloku SYSCFG
 - Registry jsou 4, v RM 7.2.3+ číslované EXTICR1-EXTICR4, v C definováno jako:

```
__IO uint32_t EXTICR[4];    /*!< SYSCFG external interrupt configuration registers, Address offset: 0x08-0x14 */
```

- Tj. zápis SYSCFG->EXTICR[0] odpovídá registru EXTICR1 !!!
 - Vždy 4 bity odpovídají jednomu Ext IRQ, celkem 4 na registr (tj. je efektivně 16b)
- Blok SYSCFG potřebuje povolit hodiny z APB2
 - "Klasická" periférie
- Blok EXTI má hodiny trvale aktivní
 - Umí např. vzbudit procesor ze stavu spánku

7.2.4 SYSCFG external interrupt configuration register 2 (SYSCFG_EXTICR2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EXTI7[3:0]				EXTI6[3:0]				EXTI5[3:0]				EXTI4[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:16 Reserved, must be kept at reset value.

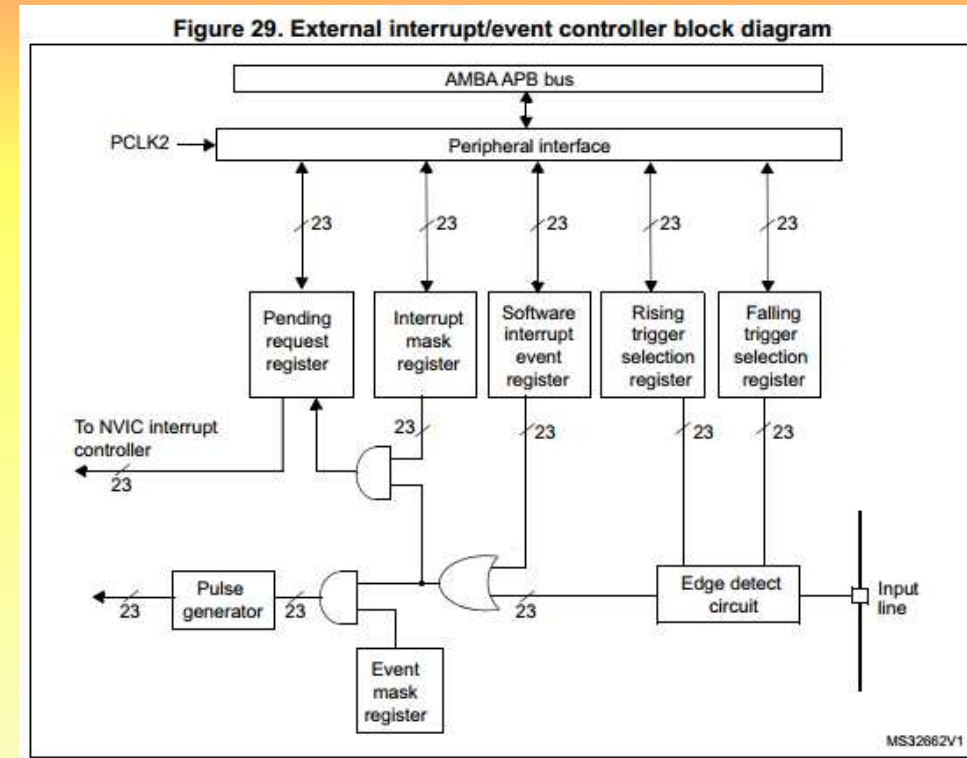
Bits 15:0 **EXTIx[3:0]**: EXTI x configuration (x = 4 to 7)

These bits are written by software to select the source input for the EXTIx external interrupt.

0000: PA[x] pin
0001: PB[x] pin
0010: PC[x] pin
0011: PD[x] pin
0100: PE[x] pin
0101: Reserved
0110: Reserved
0111: PH[x] pin

Struktura EXTI

- Signál postupuje od pinu zleva
- Detekce hrany
 - Volitelně spouští H-L nebo L-H nebo obě
- Maskovací bity pro IRQ
- Maskovací bity pro Event
 - Event probudí procesor z čekání v instrukci WFE
 - Nemá obslužnou rutinu/funkci/vektor
- Pending registr (= indikace požadavků)
 - Použít při testování "který bit"
 - Nuluje se zápisem 1 na pozici "pending" bitu



Registry EXTI

- IMR = Int. Mask Register – RM 10.3.1
 - Log. 1 povolí daný zdroj (bit) pro generování IRQ – dle RM "odmaskuje"
- EMR = Event Mask Register – RM 10.3.2
 - Stejně pro generování Eventu
- RSTR = Rising trigger selection register
 - Povolí spouštění L-H (vzestupnou) hranou
- FSTR = Falling trigger selection register
 - Stejně pro H-L (sestupnou) hranu
- SWIER = Software interrupt event register – RM 10.3.5
 - Pokud je povoleno v IMR, zápisem log. 1 do příslušného bitu se vygeneruje příslušné IRQ programově
- PR = Pending register
 - Nastaven do log. 1 při detekci hrany na vstupu (dle R/FSTR) nebo zápisem do SWIER
 - Možno použít pro zjištění zdroje přerušení
 - Bit se smaže zápisem log. 1 do příslušného bitu
- Všechny registry mají navíc bity 16, 17, 18, 21 a 22
 - EXTI line 16 is connected to the PVD output
 - EXTI line 17 is connected to the RTC Alarm event
 - EXTI line 18 is connected to the USB OTG FS Wakeup event
 - EXTI line 21 is connected to the RTC Tamper and TimeStamp events
 - EXTI line 22 is connected to the RTC Wakeup event

Fragment kódu pro externí přerušení

- Zápis do ODR přes Bit-banding
- Čtení z IDR nemusí BB používat
- Pozor na indexování EXTICR
- Detekována L-H hrana = stisk tlačítka
- While cyklus čeká dokud je na vstupu log. 1
 - Po dobu cyklu svítí LED_G = PC7

```
...
void EXTI9_5_IRQHandler(void)
{
    if (EXTI->PR & EXTI_PR_PR5)
    {
        EXTI->PR = EXTI_PR_PR5;    // RM10.3.6 - This bit is cleared by programming it to '1'.

        BB_REG(GPIOC->ODR, 7) = 0;    // ON
        while(GPIOB->IDR & (1 << 5))
        ;
        BB_REG(GPIOC->ODR, 7) = 1;    // OFF
    }
}
...
if (!(RCC->APB2ENR & RCC_APB2ENR_SYSCFGEN))
{
    RCC->APB2ENR |= RCC_APB2ENR_SYSCFGEN;
    RCC->APB2RSTR |= RCC_APB2RSTR_SYSCFGRST;
    RCC->APB2RSTR &= ~RCC_APB2RSTR_SYSCFGRST;
}

SYSCFG->EXTICR[1] |= SYSCFG_EXTICR2_EXTI5_PB;    // !!! registry 1..4, index 0..3 !!!!

EXTI->RTSR |= EXTI_RTSR_TR5;
EXTI->IMR |= EXTI_IMR_MR5;
NVIC_EnableIRQ(EXTI9_5_IRQn);
...
```


Nastavení priorit

- Priority přerušení řeší NVIC – rozsáhle viz. PM 4.3
- Součástí CMSIS knihovny jsou kromě povolení IRQ i funkce pro prioritu
 - Prioritu lze i číst
 - V podstatě je to jen zápis do příslušného registru v NVIC

Table 47. CMSIS functions for NVIC control

CMSIS interrupt control function	Description
void NVIC_SetPriorityGrouping(uint32_t priority_grouping)	Set the priority grouping
void NVIC_EnableIRQ(IRQn_t IRQn)	Enable IRQn
void NVIC_DisableIRQ(IRQn_t IRQn)	Disable IRQn
uint32_t NVIC_GetPendingIRQ (IRQn_t IRQn)	Return true (IRQ-Number) if IRQn is pending
void NVIC_SetPendingIRQ (IRQn_t IRQn)	Set IRQn pending
void NVIC_ClearPendingIRQ (IRQn_t IRQn)	Clear IRQn pending status
uint32_t NVIC_GetActive (IRQn_t IRQn)	Return the IRQ number of the active interrupt
void NVIC_SetPriority (IRQn_t IRQn, uint32_t priority)	Set priority for IRQn
uint32_t NVIC_GetPriority (IRQn_t IRQn)	Read priority of IRQn
void NVIC_SystemReset (void)	Reset the system

Detailní popis NVIC registrů

- ...

DCV – přerušení

- Dokončit uvedené příklady a vyzkoušet vliv priorit
- Přidat přerušení SysTick a počítání ms
 - Pravidelně vypisovat na terminál hodnotu `_ticks`
 - Prozkoumat, zda se zpozdí během stisku tlačítka
 - Jak se projeví případné změny priority ?
 - Výpis v hlavní `while(1)` smyčce je samozřejmě během obsluhy IRQ zastaven ...

Plán cvičení – aktivity v laboratoři

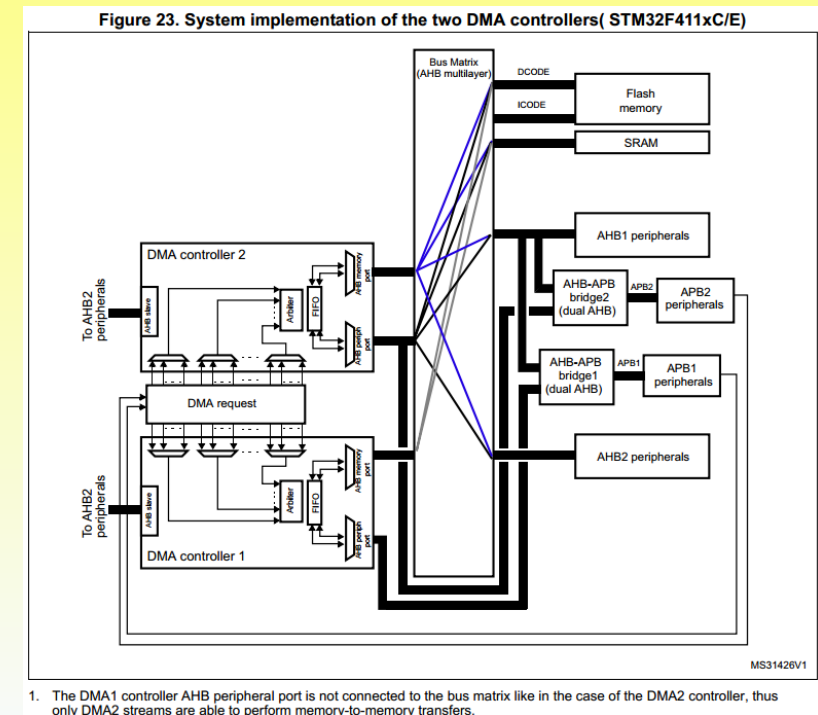
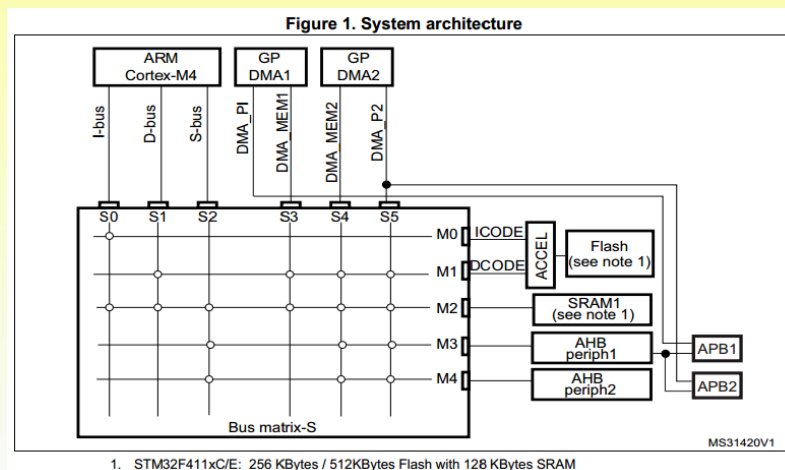
1. Úvod, rozdělení kitů, opakování samostatně
2. Pokročilé periférie ARM Cortex M
3. FPU - nastavení kompilátoru, porovnání rychlosti
4. Problém atomických operací, řešení pomocí bit-banding
5. Přerušování advanced - priority, blokování a řešení problémů
6. DMA - přenos bloku paměti, USART
7. DMA 2 - využití A/D převodníku a bufferu dat
8. Problematika Low-Power režimů
9. Privilegovaný a neprivilegovaný režim
10. SVC
11. Pokročilé techniky programování – zásobníky, semafore, instrukce LDREX/STREX, paměťové bariéry ...
12. RTOS - praktické řešení
13. Rezerva

Ověření DCV – priority přerušení

- Jak se projeví nastavení priorit nejen pro Timer3/EXTI ale i pro SysTick ?
 - Zastaví či nezastaví se `_tick++` ?
 - Zastaví se vysílání na UARTu realizované v hlavní smyčce ?

DMA = Direct Memory Access

- Viz. RM – kap.9
- V procesorech F4xx máme k dispozici 2 řadiče DMA
- Přenosy mohou být:
 - Register-to-Memory
 - Memory-to-Register
 - Memory-to-Memory
 - Pouze DMA2 umí Mem-to-Mem přenos
 - Viz. bus-matrix v RM kap.2.1.6



DMA – možnosti nastavení

- Při přenosech lze nastavit
 - Inkrementaci adresy paměti
 - Vhodné pro vysílání nebo příjem bloku dat
 - Naopak při "neinkrementu" lze posílat "konstantu", příp. vyplnit blok paměti konstantou
 - Inkrementaci adresy registru
 - Možná pro vyčtení A/D v Junction režimu (až 4 registry DR)
 - Při přenosu mem2mem je "registrová adresa" použita jako adresa zdrojových dat
 - Velikost datové jednotky – 8, 16 nebo 32 bitů
 - Velikost bloku dat 1 – 65535 "přenosů" (16b registr)
 - Příp. cyklický režim, který přenos stále opakuje
 - Další funkcionality – FIFO blok, burst transfer (blokový)
 - Spouštění jednotlivých přenosů od události – např. časovač nebo SW (pro mem2mem)
- Lze využít příznaky
 - Half-transfer, Complete, Error, FIFO Error, Direct mode error
 - A propojit je také s interrupty

Volba DMA pro periférii

- Oba DMA řadiče jsou na AHB1 – povolení hodin v RCC
- Oba DMA řadiče obsahují 8 nezávislých kanálů
- Každý kanál má 8 proudů (streamů)
 - V rámci daného kanálu může být využit najednou pouze 1 stream
 - Nutno vybrat podle periférie – někdy více možností
- Pro přenos mem2mem lze použít libovolný "volný" kanál/stream v DMA2

Table 27. DMA1 request mapping (STM32F411xC/E)

Peripheral requests	Stream 0	Stream 1	Stream 2	Stream 3	Stream 4	Stream 5	Stream 6	Stream 7
Channel 0	SPI3_RX	I2C1_TX	SPI3_RX	SPI2_RX	SPI2_TX	SPI3_TX		SPI3_TX
Channel 1	I2C1_RX	I2C3_RX				I2C1_RX	I2C1_TX	I2C1_TX
Channel 2	TIM4_CH1		I2S3_EXT_RX	TIM4_CH2	I2S2_EXT_TX	I2S3_EXT_TX	TIM4_UP	TIM4_CH3
Channel 3	I2S3_EXT_RX	TIM2_UP TIM2_CH3	I2C3_RX	I2S2_EXT_RX	I2C3_TX	TIM2_CH1	TIM2_CH2 TIM2_CH4	TIM2_UP TIM2_CH4
Channel 4						USART2_RX	USART2_TX	
Channel 5			TIM3_CH4 TIM3_UP		TIM3_CH1 TIM3_TRIG	TIM3_CH2		TIM3_CH3
Channel 6	TIM5_CH3 TIM5_UP	TIM5_CH4 TIM5_TRIG	TIM5_CH1	TIM5_CH4 TIM5_TRIG	TIM5_CH2	I2C3_TX	TIM5_UP	USART2_RX
Channel 7			I2C2_RX	I2C2_RX				I2C2_TX

Table 28. DMA2 request mapping (STM32F411xC/E)

Peripheral requests	Stream 0	Stream 1	Stream 2	Stream 3	Stream 4	Stream 5	Stream 6	Stream 7
Channel 0	ADC1				ADC1		TIM1_CH1 TIM1_CH2 TIM1_CH3	
Channel 1								
Channel 2			SPI1_TX	SPI5_RX	SPI5_TX			
Channel 3	SPI1_RX		SPI1_RX	SPI1_TX		SPI1_TX		
Channel 4	SPI4_RX	SPI4_TX	USART1_RX	SDIO	SPI4_RX	USART1_RX	SDIO	USART1_TX
Channel 5		USART6_RX	USART6_RX	SPI4_RX	SPI4_TX	SPI5_TX	USART6_TX	USART6_TX
Channel 6	TIM1_TRIG	TIM1_CH1	TIM1_CH2	TIM1_CH1	TIM1_CH4 TIM1_TRIG TIM1_COM	TIM1_UP	TIM1_CH3	
Channel 7						SPI5_RX	SPI5_TX	

Příznakové registry DMA

- Společné pro všechny kanály na řadiči
- Interrupt status register – **LISR** a **HISR**
 - Stavové bity pro kanály 3-0 (LISR) a 7-4 (HISR)
 - **TCIFx**: Stream x transfer complete interrupt flag
 - **HTIFx**: Stream x half transfer interrupt flag
 - **TEIFx**: Stream x transfer error interrupt flag
 - **DMEIFx**: Stream x direct mode error interrupt flag
 - **FEIFx**: Stream x FIFO error interrupt flag
- Interrupt flag clear register – **LIFCR** a **HIFCR**
 - Zápisem log.1 na příslušnou pozici nuluje odpovídající příznak v xISR
 - **CTCIFx**: Stream x clear transfer complete interrupt flag
 - ...
 - Před přenosem vhodné vše vynulovat = zapsat log. 1 do xIFCR bitů kanálu

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				TCIF3	HTIF3	TEIF3	DMEIF3	Reserved	FEIF3	TCIF2	HTIF2	TEIF2	DMEIF2	Reserved	FEIF2
r	r	r	r	r	r	r	r		r	r	r	r	r		r

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved				TCIF1	HTIF1	TEIF1	DMEIF1	Reserved	FEIF1	TCIF0	HTIF0	TEIF0	DMEIF0	Reserved	FEIF0
r	r	r	r	r	r	r	r		r	r	r	r	r		r

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				TCIF7	HTIF7	TEIF7	DMEIF7	Reserved	FEIF7	TCIF6	HTIF6	TEIF6	DMEIF6	Reserved	FEIF6
				r	r	r	r		r	r	r	r	r		r

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved				TCIF5	HTIF5	TEIF5	DMEIF5	Reserved	FEIF5	TCIF4	HTIF4	TEIF4	DMEIF4	Reserved	FEIF4
				r	r	r	r		r	r	r	r	r		r

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				CTCIF3	CHTIF3	CTEIF3	CDMEIF3	Reserved	CFEIF3	CTCIF2	CHTIF2	CTEIF2	CDMEIF2	Reserved	CFEIF2
				w	w	w	w		w	w	w	w	w		w

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved				CTCIF1	CHTIF1	CTEIF1	CDMEIF1	Reserved	CFEIF1	CTCIF0	CHTIF0	CTEIF0	CDMEIF0	Reserved	CFEIF0
				w	w	w	w		w	w	w	w	w		w

```
DMA1->HIFCR = DMA_HIFCR_CTCIF6 | DMA_HIFCR_CHTIF6 | DMA_HIFCR_CTEIF6 | DMA_HIFCR_CDMEIF6 | DMA_HIFCR_CFEIF6;
```

Řídící registry DMA kanálů

- [illegible]

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				CHSEL[3:0]			MBURST[1:0]		PBURST[1:0]		Reserved	CT	DBM or reserved	PL[1:0]	
				rw	rw	rw	rw	rw	rw	rw		rw	rw or r	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PINCOS	MSIZE[1:0]		PSIZE[1:0]		MINC	PINC	CIRC	DIR[1:0]		PFCTRL	TCIE	HTIE	TEIE	DMEIE	EN
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Pracovní registry DMA kanálů

- **DMA_SxNDTR** – number of data register
 - POZOR, pouze 16-bitový, tj. max 65535 přenosů
 - Během přenosu lze vyčíst "ještě k přenesení"
 - Po skončení přenosu je = 0 (kromě circular modu)
- **DMA_SxPAR** - DMA stream x peripheral address register
 - Použit jako zdrojová adresa při mem2mem (na DMA2)
- **DMA_SxM0AR** - DMA stream x memory 0 address register
 - **DMA_SxM1AR** – používá se pouze v Double buffer režimu
- **DMA_SxFCR** - DMA stream x FIFO control register
 - FEIE = FIFO error interrupt enable
 - FS = FIFO status = úroveň zaplnění
 - DMDIS = Direct mode disable
 - FTH = FIFO threshold = velikost využití FIFO

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								FEIE	Reserved	FS[2:0]			DMDIS	FTH[1:0]	
								rw		r	r	r	rw	rw	rw

Příklad DMA – přenos po USART2 do PC

- Inicializace USART2 pro přenosy
- Povolení hodin DMA1
- Nastavení kanálu 4, streamu 6
- Režim mem2periph (DR registr USART2)
- **POZOR**, na USART2 povolit DMAT v CR3
- Přenášet řetězec vygenerovaný např. Lorem Ipsum
 - Délku vložit do NDTR (pro strlen nutný `#include <string.h>` !!)
- Přenos se spustí nastavením EN v `DMA1_Stream6->CR`
- **!!!** Při krokování přenos proběhne i když program stojí v Debug !!!

```

Usart2InitStd(38400);
puts("Start APP - MINA DMA test " __DATE__ " " __TIME__ "\r");

char *lorem = "Lorem ipsum ... "; // https://loremipsumgenerator.com/generator/?n=10&t=s

if (!(RCC->AHB1ENR & RCC_AHB1ENR_DMA1EN))
{
    RCC->AHB1ENR |= RCC_AHB1ENR_DMA1EN;
    RCC->AHB1RSTR |= RCC_AHB1RSTR_DMA1RST;
    RCC->AHB1RSTR &= ~RCC_AHB1RSTR_DMA1RST;
}

DMA1_Stream6->CR &= DMA_SxCR_EN;           // EN = 0 !!, jinak nepujdou zmeny
DMA1_Stream6->CR = 0
    | DMA_SxCR_CHSEL_2           // CHSEL = 100
    | 0 // MSIZE = 00 = 8bit
    | 0 // PSIZE = 00 = 8bit
    | DMA_SxCR_MINC           // MINC = 1 = memory increment
    | 0 // PINC = 0
    | DMA_SxCR_DIR_0         // DIR = 01 = memory 2 peripheral
    ;

DMA1_Stream6->NDTR = strlen(lorem);
DMA1_Stream6->PAR = (uint32_t)&(USART2->DR);
DMA1_Stream6->M0AR = (uint32_t)lorem;

USART2->CR3 |= USART_CR3_DMAT;

DMA1->HIFCR = DMA_HIFCR_CTCIF6 | DMA_HIFCR_CHTIF6 | DMA_HIFCR_CTEIF6 | DMA_HIFCR_CDMEIF6 | DMA_HIFCR_CFEIF6;

    // place for breakpoint
DMA1_Stream6->CR |= DMA_SxCR_EN;
while(!(DMA1->HISR & DMA_HISR_TCIF6))
    ;

DMA1_Stream6->CR &= ~DMA_SxCR_EN;

DMA1->HIFCR = DMA_HIFCR_CTCIF6 | DMA_HIFCR_CHTIF6 | DMA_HIFCR_CTEIF6 | DMA_HIFCR_CDMEIF6 | DMA_HIFCR_CFEIF6;

USART2->CR3 &= ~USART_CR3_DMAT;

puts("\nFinished");

```

Kopírování bloku paměti

- Budeme kopírovat blok paměti o velikosti 8kB do jiného 8kB bloku
 - Nelze mít lokální proměnnou na zásobníku, ten je defaultně velký 2kB (0x800)
 - Globální proměnné jsou v BSS segmentu = velikost omezena jen pamětí RAM procesoru
- Primitivní kopírování možné pomocí for cyklu
- Měření doby kopírování
 - Využít "milisekundovou" proměnnou `_ticks`
 - Kopírování příliš rychlé, proto opakovat např. 100x
 - Změřit dobu všech opakování v ms a vypsát
- Optimalizace 1
 - Použít inkrementující ukazatele, nepočítat offsety v []
- Optimalizace 2
 - Použít ukazatele `uint32_t *` a přenášet 4 bajty najednou
 - Počet přenosů je `BLOCK_BYTES_LEN / 4`

```
#define BLOCK_BYTES_LEN 8192
#define BLOCK_COPY_COUNT 100

uint8_t blok_src[BLOCK_BYTES_LEN];
uint8_t blok_dest[BLOCK_BYTES_LEN];

int main(void)
{
    ...
    for(int i = 0; i < BLOCK_BYTES_LEN; i++)
    {
        blok_src[i] = i;    // naplnení zdrojového pole
        blok_dest[i] = 0;  // vynulování cílového pole
    }
    ...
    for (int i = 0; i < BLOCK_BYTES_LEN; i++)
        blok_dest[i] = blok_src[i];
    ...
}
```

```
...
uint8_t *pd = blok_dest, *ps = blok_src;

for (int i = 0; i < BLOCK_BYTES_LEN; i++)
    *pd++ = *ps++;
...
```

```
...
uint32_t *pd = (uint32_t *)blok_dest, *ps = (uint32_t *)blok_src,
          *pf = ps + BLOCK_BYTES_LEN / 4;
// for (int i = 0; i < BLOCK_BYTES_LEN / 4; i++)
while(ps < pf)
    *pd++ = *ps++;
...
```

Kopírování bloku paměti pomocí DMA

- Libovolný kanál/stream na DMA2
- Nezapomenout před přenosem nastavit NDTR
- Změřit dobu pro stejný počet opakování

```
if (!(RCC->AHB1ENR & RCC_AHB1ENR_DMA2EN))
...

DMA2_Stream6->CR &= DMA_SxCR_EN; // EN = 0 !!, jinak nepujdou zmeny
DMA2_Stream6->CR = 0
| DMA_SxCR_CHSEL_2          // CHSEL = 100
| DMA_SxCR_MSIZE_1         // MSIZE = 10 = 32bit
| DMA_SxCR_PSIZE_1         // PSIZE = 10
| DMA_SxCR_MINC             // MINC = 1 = memory increment
| DMA_SxCR_PINC             // PINC = 0
| DMA_SxCR_DIR_1           // DIR = 10 = memory 2 mem = PAR 2 M0AR
;

DMA2_Stream6->PAR = (uint32_t)blok_src;
DMA2_Stream6->M0AR = (uint32_t)blok_dest;

... Opakovani ...
{
    DMA2_Stream6->NDTR = BLOCK_BYTES_LEN / 4; // prenosy 32b !!
    DMA2->HIFCR = DMA_HIFCR CTCIF6 | ...;

    DMA2_Stream6->CR |= DMA_SxCR_EN;
    while(!(DMA2->HISR & DMA_HISR_TCIF6))
    ;

    DMA2_Stream6->CR &= ~DMA_SxCR_EN;
    DMA2->HIFCR = DMA_HIFCR CTCIF6 | ...;
}
```

DCV s DMA a kopírováním paměti

- Zvolte počet opakování přenosů (např. 100x)
- Změřte dobu trvání (stačí v ms)
 - For cyklem s []
 - Cyklem s ukazateli na byte
 - Cyklem s ukazateli na uint32
 - DMA 8 bitově
 - DMA 32 bitově
- Všechny varianty pro hodiny 16MHz (default HSI) a 100MHz (např. PLL z HSE)
- Doplnující úloha
 - Změřit dobu výpočtů Mandelbrotovy množiny během probíhajícího při DMA přenosu a bez něj
 - Opět pro hodiny 16MHz a 100MHz

Plán cvičení – aktivity v laboratoři

1. Úvod, rozdělení kitů, opakování samostatně
2. Pokročilé periférie ARM Cortex M
3. FPU - nastavení kompilátoru, porovnání rychlosti
4. Problém atomických operací, řešení pomocí bit-banding
5. Přerušení advanced - priority, blokování a řešení problémů
6. DMA - přenos bloku paměti, USART
7. DMA 2 - využití A/D převodníku a bufferu dat
8. Problematika Low-Power režimů
9. Privilegovaný a neprivilegovaný režim
10. SVC
11. Pokročilé techniky programování – zásobníky, semafore, instrukce LDREX/STREX, paměťové bariéry ...
12. RTOS - praktické řešení
13. Rezerva

DCV – změřené časy DMA kopírování

- Jak to vyšlo ? Měření pro 16MHz a 100MHz takt
 - For cyklem s []
 - Cyklem s ukazateli na byte
 - Cyklem s ukazateli na uint32
 - DMA 8 bitově
 - DMA 32 bitově

A/D převodník – opakování

- Je možné převádět z 16 externích vstupů a 3 interních
- Reference pevně spojena na VDD (3v3)
- Pro více kanálů je možné:
 - Režim *Regular* = provádí měření v sekvenci, zpracování více hodnot z DR ideálně pomocí DMA
 - Režim *Injection* = změří až 4 kanály těsně po sobě a výsledek poskytne najednou v JDR1 – JDR4
- Každý kanál může mít jinou rychlost převodu
- Blok **ADC1** připojen na APB2 sběrnici
- Na MBED shield připojeny 2 potenciometry mezi 3v3 a GND
 - Odpovídající GPIO porty **PA0** a **PA1** nutno nastavit na analog režim

A/D převodník - registry

- Registr **SR** – Status
 - JEOC a EOC – příznak dokončení převodu pro Injection, resp. Regular režim
 - OVR – došlo ke ztrátě dat = nebyla odebrána data z DR před dokončením dalšího převodu
- Registr **CR1** – Control Register 1 – viz. RM 11.12.2
 - RES – budeme používat 12b rozlišení, pro výpočty a pole hodnoty pak uint16_t
 - SCAN – Scan mód musí být povolen
 - JEOCIE a EOICIE – povolení přerušení od Injected resp. Regular převodu
- Registr **CR2** – Control Register 2 – viz. RM 11.12.3
 - SWSTART a JSWSTART – nastavením log. 1 spouští převod "ručně"
 - EXTEN a JEXTEN – typ události pro externí spouštění
 - EXTSEL a JEXTSEL – zdroj události ext. spouštění (typicky výstup časovače nebo pin)
 - ALIGN – budeme používat zarovnání vpravo
 - EOCS – příznak dokončení převodu celé sekvence v Regular režimu
 - DDS – povolení opakovaných přenosů při DMA
 - DMA – povolení aktivace převodu hodnoty DMA kanálem po převodu
 - ADON – zapnutí celého ADC

A/D převodník – registry II

- **SMPR1 a SMPR2** – doba převodu pro každý kanál – 3 bity (8 možností)
 - Určuje počet cyklů "hodin" ADC
 - Společné jak pro Regular tak pro Injection režim
 - Do celkového času převodu ještě nutno započítat 12 cyklů (pro 12 bitů dat) – viz. RM 11.5 a 11.7
- **SQR1 – SQR3** – čísla kanálů pro Regular sekvenci
 - Počet převodů v L bitech v **SQR1**, 0000 = jeden převod
 - Číslo kanálu je 5-bitové, umístěno "od konce", tj. "nultý" převáděný kanál je v **SQR3**
- **DR** – data registr pro Regular převod
- **JSQR** – čísla kanálů pro Injection režim
 - Počet měřených kanálů v **JL** (00 = jeden)
 - POZOR – čísla kanály se uvádí od **JSQ4** – viz. poznámka v 11.12.12
- **CCR** – Common Control
 - **ADCPRE** – předdělička z **APB2**, 00 = /2 – tj. nejmenší dělení

A/D převodník – Regular mod, 1 kanál

- Pro časování všech příkladů uvažujte 100MHz hodiny a funkční "stdio"
- Nastavte A/D převodník pro Regular převod 1 kanálu
 - Číslo kanálu 0
 - Doba převodu není kritická, např. 15 cyklů v **SMPR**
 - Nezapomenout **SCAN** bit
 - Max. rychlost hodin ADC je 36MHz – viz. DS 6.3.20
 - Při taktu 100MHz je APB2 také 100M, minimální dělení **ADPPRE** je /2 = příliš rychlé
 - Nastavit **ADCPRE** na 01, takt ADC potom bude 25MHz
- Pravidelně spouštějte v hlavním while cyklu měření a vypisujte na terminál
 - Pravidelně každých např. 20 ms
 - **SWSTART** spouští převod
 - Čekám na **EOC**
 - V registru **DR** je hodnota
 - Výpis např. pomocí printf

A/D převodník – Injection mód, 2 kanály

- Stejné základní nastavení, změníme na měření v Injection režimu
 - Spouštění **JSWSTART**, konec **JEOC**
 - Nastavit kanál 0 a kanál 1 do **JSQ4** a **JSQ3** v registru **JSQR**
 - Data po převodu jsou v **JDR1** a **JDR2**
 - Vypisovat obě hodnoty (např. printf)

A/D převodník – Injection mód, přerušení, takt časovačem

- Nastavení stejné jako pro Injection "manuálně"
- Přerušení od AD
 - Povolení přerušení od dokončení převodu – **JEOPIE** v **CR1**
 - Povolení v NVIC pro zdroj **ADC_IRQn**
 - Obslužná funkce **void ADC_IRQHandler(void)**
 - Shodí **JEOP** příznak v **SR**
 - Vyčte hodnoty **JDR1** a **JDR2** a zapíše do globálních **uint16_t** proměnných
- Časovač – např. **TIM2** (je na **APB1**)
 - Nastavit na přetékání 1ms (vhodně zvolit **PSC** a **ARR**, pro jistotu vynulovat **CNT**)
 - Povolit přerušení při přetečení – bit **UIE** v registru **DIER**
 - Povolit v NVICu zdroj **TIM2_IRQn**
- Přerušení od časovače – funkce **void TIM2_IRQHandler(void)**
 - Vynulovat **UIF** v registru **SR**
 - Spustit **AD** převod **JSWSTART**
- V hlavní while smyčce v intervalu např. 20ms vypisujeme globální proměnné = "poslední měřená hodnota"
- Takže každých 1ms se spustí převod a po dokončení se data vloží do globální proměnné, odkud je "občas vypíše hlavní smyčka"

A/D – průměrování měřených hodnot

- Doplnit počítání průměru měřených hodnot
 - Pole pro AD0 a AD1, např. velikost 16 hodnot
 - V ADC přerušení ukládat do pole přes počítadlo, které přičítá "modulo" velikost pole
 - "Hloupý" výpočet průměru v každém kroku přes všechny prvky pole
 - Úmyslně nějakou dobu trvá
- Měření na osciloskopu
 - Pomocné GPIO výstupy **PC2** a **PC3** = jsou v levém dolním rohu velkého Nucleo konektoru
 - V hlavní smyčce generovat pulsy na **PC2** – změna každým průchodem = velmi rychlé
 - Vykonávání jakéhokoliv přerušení je vidět jako "výpadek" pulsů
 - Raději dočasně zrušit printf výpisy ve smyčce
 - V interruptu od ADC nastavit **PC3** do log. 1 před začátkem výpočtu průměru a shodit na konci do log. 0
 - Šířka pulsu odpovídá době výpočtu
 - Pro delší pole se příslušně prodlouží ☹
 - Vzdálenost pulsů by měla být 1ms = takt spouštění převodu
- Reálné rychlé vzorkování – jak se na osciloskopu projeví např. spouštění po 20us (tj. 50kHz = "skoro" audio vzorkování 48kHz) ?

A/D s DMA

- Výpočetně náročnější operace (např. FFT) je nutné dělat nad celým blokem dat a ne při každém měření
 - Pomocí DMA plnit pole a po dokončení (jedna událost) provést výpočet
 - DMA umí hlásit i naplnění $\frac{1}{2}$ bufferu
 - V tomto okamžiku počítat s první půlkou bufferu
 - Mezitím se plní "druhá" polovinu bufferu
 - Při události "Complete" počítat ve druhé polovině bufferu
 - Ideálně využít Cirkular módu DMA (bit **CIRC**)
- POZOR, pro jednoduchost budeme měřit jen na jednom kanále !!
- Spouštění převodu opět pomocí časovače (TIM2)
- Pro ADC nutno použít DMA2, Stream 0 nebo 4, pro oba Channel 0

Nastavení periférií pro DMA + A/D

- DMA2 - Stream0 – Channel0

- Velikost dat 16b
- Přenos peripheral 2 memory
- Memory inkrement
- **MOAR** je adresu bufferu
- **PAR** je adresa registru **DR** nebo **JDR1** (podle Regular nebo Injection režimu)
- **NDTR** velikost bufferu
- Cirkulační mód – bit **CIRC** v registru **CR**
- Povolení přerušení **TCIE** a **HTIE**
- Povolení v NVIC pro **DMA2_Stream0_IRQn**
- "Shodit" příznak v **LIFSR** zápisem do **LIFCR**
- "Zapnout" **EN** v registru **CR**
 - Zatím se nic nepřenáší, dokud nebudou data

- Přerušení od 1/2 a plného bufferu

- Funkce **void DMA2_Stream0_IRQHandler(void)**
- Shodí příznaky přerušení
- Spočte průměr první nebo druhé 1/2 bufferu a uloží do globální proměnné
 - "Okolo" výpočtu aktivní **PC3** pro osciloskop

Nastavení periférií pro DMA + A/D – část 2

- Převod AD musí být spouštěn událostí mimo ADC blok
 - EXTEN nebo JEXTEN nastavit na "ne 00", doporučeno 01 = trigger rising edge
 - EXTSEL nebo JEXTSEL nastavit Timer2 TRGO event. (0110 resp. 0011 pro JE...)
- Bit DMA v registru CR2 – použití DMA
- Bit DDS v registru CR2 – aby stále převáděl i v Cirkulačním režimu DMA
- Povolit ADC
 - Stále se ještě nepřevádí
- TIM2 musí generovat TRGO výstup – viz. fig. 87 v RM kap. 13.2
 - Nastavit MMS v CR2 na 010 = generuje TRGO při update (přetečení/podtečení podle směru)
- Po zapnutí TIM2 by se to celé mělo rozběhnout 😊

DCV pro ADC+DMA

- Dokončit všechny varianty
- Nepovinné "na hraní" – osciloskop na LCD
 - Data z bufferu vykreslovat
 - Zkopírovat DMA buffer pro vykreslení
 - Vztít pouze horních 6 bitů z 12b hodnoty (= 32 hodnot odpovídá 32 pixelů vertikálně)
 - Pomocí PutPixel nebo DrawLine vykreslit "stopu"
 - Vhodně řídit rychlost vzorkování a překreslování

Plán cvičení – aktivity v laboratoři

1. Úvod, rozdělení kitů, opakování samostatně
2. Pokročilé periférie ARM Cortex M
3. FPU - nastavení kompilátoru, porovnání rychlosti
4. Problém atomických operací, řešení pomocí bit-banding
5. Přerušování advanced - priority, blokování a řešení problémů
6. DMA - přenos bloku paměti, USART
7. DMA 2 - využití A/D převodníku a bufferu dat
8. Problematika Low-Power režimů
9. Privilegovaný a neprivilegovaný režim
10. SVC
11. Pokročilé techniky programování – zásobníky, semafore, instrukce LDREX/STREX, paměťové bariéry ...
12. RTOS - praktické řešení
13. Rezerva

DCV – DMA a ADC

- Jaké problémy se vyskytly ?
- Injection převod negeneruje DMA ?

Low-power – snížení spotřeby

- V procesoru
 - Snížení rychlosti jádra
 - Vypnutí nepoužívaných periférií
 - Nastavení GPIO pinů do "analog" režimu
 - Nižší napájecí napětí
 - Aktivace úsporných režimů
- Mimo procesor
 - LED s menším proudem
 - Lepší napájecí obvody s malou vlastní spotřebou
 - Nižší napájecí napětí
 - "Vypínání" aktuálně nepoužívaných obvodů (převodníky, budiče, ...)

STM32F4xx – spotřeba dle DS

- DS – 6.3.6 – Supply Current – Typical and maximum current consumption
 - Table 20 (pg. 68) pro 1.7V a Table 21 (pg. 69) pro 3.6V pro kód v RAM, příp. Table 22 a 23 pro Flash
 - Table 33 (pg. 79) - On-chip peripheral current consumption
 - Uvádí se v $\mu\text{A}/\text{MHz}$
- DS – 6.3.7 - Wakeup time from low-power modes
 - Jednotlivé časy pro přechod z úsporných režimů

STM32F4xx – úsporné režimy

- Dle PM (kap. 2.5) existují 2 základní režimy společné pro Cortex-M
 - Sleep mode = zastaví hodiny procesoru, ostatní periférie zůstávají funkční
 - Deep sleep mode = zastaví také hodiny pro periférie
 - POZOR – ukončení možné pouze vybranými akcemi
 - Externí interrupt – nutno v SYSCFG správně povolit
 - RESET – externí nebo překročení/podtečení napájecího napětí
 - WKUP pin - PA0 – nutno povolit v PWR_CSR
 - Watchdog
- STM32F4 má další speciální "režimy" – viz. RM kap. 5.3 – Low-power modes
 - Stop mode – odpovídá Deep-sleep s možností upřesnit co a jak se vypne
 - Viz. Table 17, kap. 5.3.4 v RM
 - Standby mode – vypíná i 1.2V doménu (napájení RAM, RTC, HSI, HSE, ...)
 - Zůstává pouze obsah RAM v backup-domain (včetně RTC)

Přechod do úsporných režimů

- Napřed nakonfigurovat registry v RCC a dalších perifériích procesoru a registry SCB jádra Cortex-M
- Vykonat instrukci, která zastaví vykonávání kódu
 - **WFE** – Wait for event. – PM kap. 3.11.11
 - Dokud nenastane event
 - Nastavit vznik přerušení v periférii, ale nepovolovat **NVIC**
 - Pro externí vstup/pin nastavit příslušný bit v **EXTI_EMR**
 - **Pozor** - Nutno 2x – testuje interní Event Register (1-bitový) a při 0 jde do sleep a při 1 jen shodí do 0
 - Určeno pro multiprocesorové systémy apod. – viz. popis v PM kap. 2.5.1 Entering sleep mode
 - **WFI** – Wait for interrupt – PM kap. 3.11.12
 - "Čeká" na přerušení příp. exception
- Vložení instrukce assembleru do kódu v C/C++
 - ARM-GCC (Atollic TrueStudio) – pomocí `asm("xxx");` - zde tedy `asm("WFE");` a `asm("WFI");`
 - Keil-MDK – v `intrinsic.h` existují "pseudofunkce" pro instrukce – zde `__WFE();` a `__WFI();`

SCR - System control register

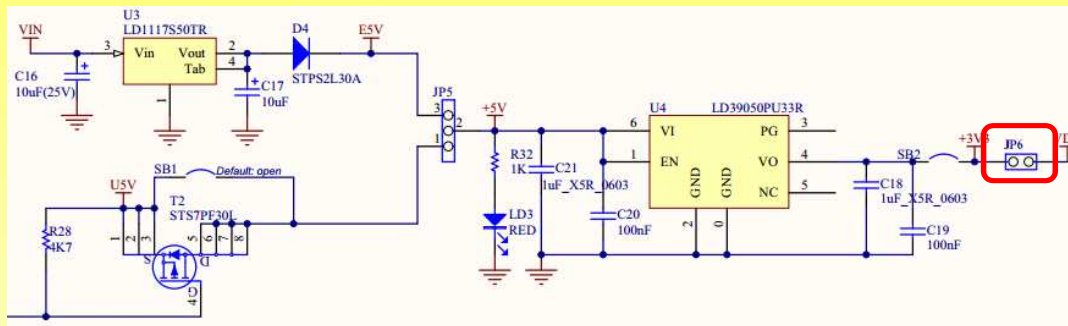
- Součást SCB (Systém Control Block) v jádře Cortex-M
 - Viz. PM kap. 4.4.6
 - SEVEONPEND (= Send Event on Pending bit)
 - 0 = probudit procesor mohou pouze povolené přerušení a "eventy"
 - 1 = probudit procesor může povolený event nebo kterýkoliv IRQ
 - SLEEPDEEP – typ "spánku"
 - 0 = sleep (zastaveno pouze jádro)
 - 1 = deep sleep – zastaveny všechny hodiny
 - SLEEPONEXIT – kdy přejde do některého sleep-režimu
 - 0 = ihned po dokončení instrukce WFI nebo WFE
 - 1 = po dokončení přerušení s nejmenší prioritou (pokud je nějaké vykonáváno)

Probuzení

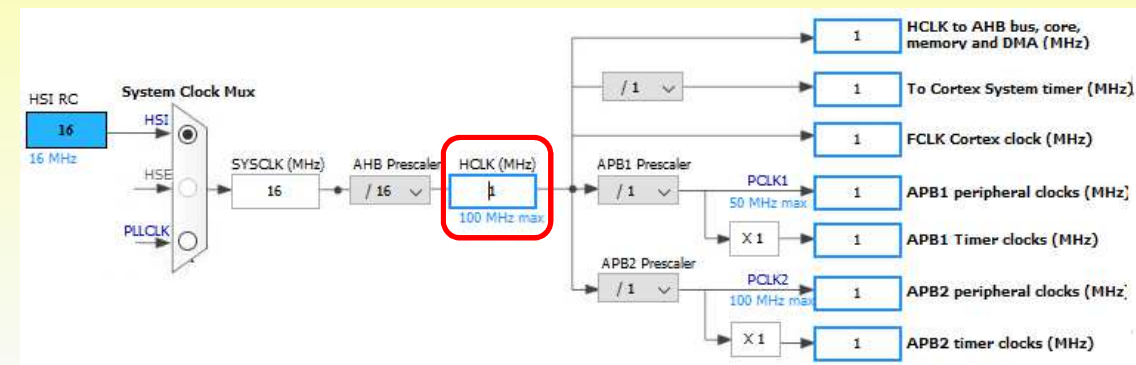
- Po probuzení ze sleep zůstává "minulá" konfigurace registrů platná
- Po probuzení z deep-sleep registry také zůstávají
 - ??? Jede se z HSI ???
- Po standby je nutno vše nakonfigurovat (jako po RESETu)
 - Stejně se většinou vzbouzí RESETem
 - Je nutné počítat s wakeup časy – viz. DS

Měření spotřeby procesoru

- Na Nucleo desce je dvojice pinů VDD s "jumperem" = připojit A-metr



- Změřte spotřebu pro aplikaci "blikání LED" na PA5 – odvozené od SysTick
- Zvolte takty hodin:
 - 16MHz (default z HSI)
 - 100MHz (z HSI 16MHz nebo z HSE 8MHz)
 - 1MHz – dělením z HSI – viz. HPRE v RCC_CFGR
- Co když nebude aktivní LED ?



Sleep mode – příklad 1

- Při činnosti bliká LED v hlavní smyčce
- Časování přibližné pomocí for cyklu
 - Nemůžeme použít přerušení SysTick-u, protože by probouzel procesor
- Čekáme na tlačítko (vzestupná hrana)
 - Nastaví **SCB_SCR** a vykoná **WFE** (2x)
- Po dalším stisku tlačítka se probudí a pokračuje v činnosti

- Nastavení
 - OnBoard LED jako výstup – **PA5**
 - Tlačítko jako vstup – **PC13**
 - Externí vstup tlačítka generuje Event

```
...
Nucleo_SetPinGPIO(BOARD_LED, ioPortOutputPP);
Nucleo_SetPinGPIO(BTN_BLUE, ioPortInputFloat);

if (!(RCC->APB2ENR & RCC_APB2ENR_SYSCFGEN))
{
    RCC->APB2ENR |= RCC_APB2ENR_SYSCFGEN;
    RCC->APB2RSTR |= RCC_APB2RSTR_SYSCFGRST;
    RCC->APB2RSTR &= ~RCC_APB2RSTR_SYSCFGRST;
}

SYSCFG->EXTICR[3] &= SYSCFG_EXTICR4_EXTI13;
SYSCFG->EXTICR[3] |= SYSCFG_EXTICR4_EXTI13_PC;

EXTI->EMR |= EXTI_EMR_MR13;    // PC13
EXTI->RTSR |= EXTI_RTSTR_TR13; // pust = hrana L-H
...
```

```
...
while (1)
{
    for (int i = 0; i < 2000; i++) // pri 16MHz
    ;
    _ticks++;

    if (_ticks >= tm)
    {
        tm = _ticks + 50;
        BB_REG(GPIOA->ODR, 5) ^= 1; // OnBoard LED
    }

    if (_ticks >= tmBut)
    {
        tmBut = _ticks + 5;

        bool bb = GPIORead(BTN_BLUE);
        if (lastBut != bb)
        {
            lastBut = bb;
            if (bb) // L-H hrana ?
            {
                SCB->SCR &= ~SCB_SCR_SLEEPONEXIT_Msk;
                SCB->SCR &= ~SCB_SCR_SLEEPDEEP_Msk;

                BB_REG(GPIOA->ODR, 5) = 0; // zhasni LED

                asm("WFE");
                asm("WFE");
            }
        }
    }
}
```

Sleep mode – příklad 1 - poznámky

- Změřte spotřebu také při neaktivní LED
- Zkuste nastavit bit SLEEPDEEP na log. 1 – jaká je spotřeba
- Instrukce WFE se musí vykonat 2x – viz. DS

Sleep mode – příklad 2

- V aktivním stavu bliká LED, opět bez SysTick
- Tlačítkem uvedeme do stavu spánku
 - Napřed spustíme čítač **TIM3** na nastavenou dobu (5 sec)
 - Instrukce **WFI**
- V přerušení od časovače zastavíme TIM3
 - Nezapomenout shodit příznak **UIF** v registru **SR**
 - Zastavení časovače nastavením **EN = 0** v registru **CR1**
 - Výsledným efektem přerušení je konec sleep mode – opět bliká LED
- Měříme spotřebu
- Nastavení
 - Tlačítko jen jako vstup, nic dalšího
 - Časovač nastavenou celkovou dobu periody na 5sec
 - Pozor na **PSC**, je jen 16-bitový
 - Povoleno přerušení od přetečení – **UIE** v **DIER** a **NVIC_EnableIRQ(TIM3_IRQn)**;
 - Časovač zatím neběží (**EN = 0**)
- Pokus – co se stane v případě Deep-sleep ?

```
... Kod detekce pustení tlačítka
BB_REG(GPIOA->ODR, 5) = 0;          // zhasni LED

TIM3->CNT = 1;                      // fresh start
TIM3->CR1 |= TIM_CR1_CEN;

SCB->SCR &= ~SCB_SCR_SLEEPONEXIT_Msk;
SCB->SCR &= ~SCB_SCR_SLEEPDEEP_Msk;
// SCB->SCR |= SCB_SCR_SLEEPDEEP_Msk;

asm("WFI");
...
```

```
void TIM3_IRQHandler(void)
{
    if(TIM3->SR & TIM_SR_UIF)
    {
        TIM3->SR &= ~TIM_SR_UIF;

        BB_REG(TIM3->CR1, 0) = 0;    // TIM_CR1_CEN
    }
}
```

Plán cvičení – aktivity v laboratoři

1. Úvod, rozdělení kitů, opakování samostatně
2. Pokročilé periférie ARM Cortex M
3. FPU - nastavení kompilátoru, porovnání rychlosti
4. Problém atomických operací, řešení pomocí bit-banding
5. Přerušování advanced - priority, blokování a řešení problémů
6. DMA - přenos bloku paměti, USART
7. DMA 2 - využití A/D převodníku a bufferu dat
8. Problematika Low-Power režimů
9. Privilegovaný a neprivilegovaný režim
10. SVC
11. Pokročilé techniky programování – zásobníky, semaforey, instrukce LDREX/STREX, paměťové bariéry ...
12. RTOS - praktické řešení
13. Rezerva