KAE/MPP – domácí příprava

Mikroprocesory a mikropočítače

Verze 2016/17

Ing. Petr Weissar, Ph.D.
Ing. Petr Krist, Ph.D.
Ing. Kamil Kosturik, Ph.D.

Plán domácí přípravy

I. **Instalace prostředí, ... První program – blikání LED na GPIO**
II. Zobecnění GPIO operací
III. Systick
IV. Timer
V. UART
VI. Rezerva
VII. SPI
VIII. PWM
IX. A/D
X. I2C (využití "knihovny")
XI. Časování – HSI/HSE/PLL
XII. Rezerva

Plán cvičení – aktivity v laboratoři

1. Úvod
2. Procvičení práce s GPIO
3. Procvičení pokročilé práce s GPIO
4. Procvičení časování pomocí SysTick a přerušení
5. Procvičení – práce časovačem (TIM3) - generování tónu i časování
6. Procvičení - sériová komunikace (UART) s PC s spojení s GPIO
7. Procvičení probraných částí, vyrovnání neprobraného
8. Procvičení - komunikace přes SPI - spojení s GPIO - řízení LED
9. Procvičení - pomocí PWM generování barev na RGB LED
10. Procvičení - použití A/D převodníku, spojení s PWM a GPIO
11. Procvičení - komunikace s akcelerometrem přes I2C - aplikace "vodováha"
12. Ověření výběru zdroje hodin a nastavení taktu procesoru (např. pomocí osciloskopu)
13. Rezerva, zápočty

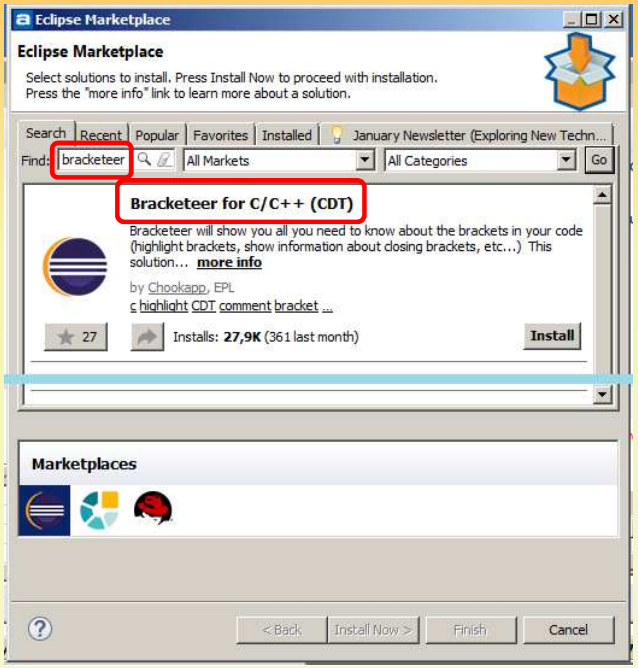
Cíle přípravy

- Nainstalovat Atollic TrueStudio
- Vytvořit Workspace a nastavit globální vlasnosti
- Vytvořit první projekt pro "náš" HW
- Napsat kód pro základní blikání LED na Nucleo desce
- Prozkoumat řídící registry související s GPIO
  - Naučit se číst v dokumentaci
- Zopakovat si konstrukce jazyka C použité v příkladech

- Předvedení kroků na YT

Instalace Atollic TrueStudio

- Windows
  - Video na YT - https://youtu.be/_pdHqeFJvJA
  - Instalační soubor buď
    - Přímo https://atollic.com/resources/download/windows/
    - Můj DropBox: https://dl.dropboxusercontent.com/u/22184251/Atollic_TrueSTUDIO_for_ARM_windows_x86_v7.1.0_20170201-1344.exe
  - Doporučuji doplnit plugin Bracketeer - viz. Eclipse Marketplace
    - Doplňuje k ukončovacím závorkám bloku informaci o začátku
- Linux
  - Instalace obdobná v grafickém prostředí jako Windows
  - Podle výrobce (Atollic) nyní podporován 64b Ubuntu a Fedora
    - Ověřeno na XUbuntu
    - https://atollic.com/resources/download/linux/
  - Použití shodné jako ve Windows, odlišnosti:
    - Ve verzi 7.0.1 není podporován ST-Link
    - Naštěstí je podporován J-Link (Segger)
    - Do Nucleo je možné nahrát speciální verzi J-Link
      - a zase vrátit původní ST-Link ...
    - https://www.segger.com/jlink-st-link.html
  - V předvedených postupech se jen použije jiná "Debug Probe"



Version	Status	Resources	Download link
7.0.1	Alpha1	Revision history Known problems Installation guide	Join Beta Program

Hardware requirements

- 2 GB RAM minimum and 4 GB RAM recommended
- 4 GB of free hard disk space
- For the Atollic TrueSTUDIO® Pro upgrade:
  - A valid License Number. Delivered upon purchase.
  - Internet access. For license activation and license updates.
  - An email account. Required to get product information.

System requirements

- Ubuntu 16.04 LTS 64-bit
- Fedora 24 64-bit
- Any Linux version (no technical support provided by support-team) fulfilling these requirements:
  - 64-bit distribution with 32-bit C runtime libraries installed
  - SysV compatible Linux system

Note: TrueSTUDIO on Linux is only available as a 64-bit application.

Workspace

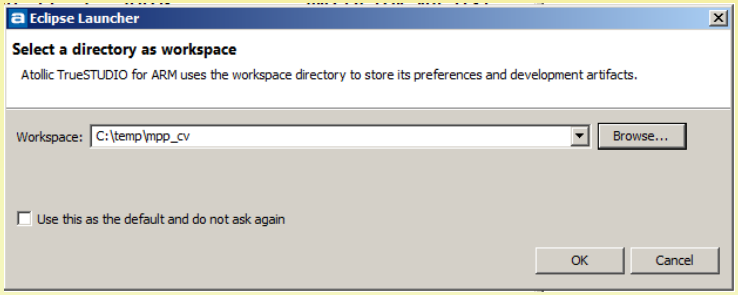
- IDE (= Integrated Devel. Environment) je založené na "Eclipse"
  - Adresářově/souborově orientované
    - Zpracovává (= překládá) vše co je v daném adresáři
    - Lze "ručně" vyloučit soubory nebo adresáře
  - Konfigurace je typicky v adresáři .settings
    - Interní formát
    - Zvlášť pro Projekty i pro Workspace

- Hlavní "jednotkou" je pracovní prostor = **Workspace**
  - Obsahuje společná nastavení (barvy, pravidla zobrazení, ...)
    - Možno exportovat/importovat
    - Při přechodu do jiného Workspace možno zkopírovat aktuální
  - Obsahuje jeden či více projektů
  - Eclipse se vždy otevírá v nějakém Workspace (adresáři)
  - Změna – menu File/Select Workspace
- **Projekt** je typicky jeden program = celek, který se pak spustí v HW
  - Může to být i knihovna, která se po zkompilování použije v dalších projektech
  - Vždy jeden projekt je aktivní = bude spuštěn nebo laděn
    - Vybírá se podle aktuálně otevřeného souboru
  - Projekty mohou být skryty (uzavřeny), aby se omylem nespustily
    - Lze snadno opět otevřít

Omezení Free verze Atollic

- Není aktivní kompletní podpora pro debugging
  - Hlavně SWV
- Nelze využívat některé nástroje pro týmovou práci
- Nejsou aktivní pokročilé diagnostické nástroje
- Na začátku je 5sec nepřeskočitelná reklama

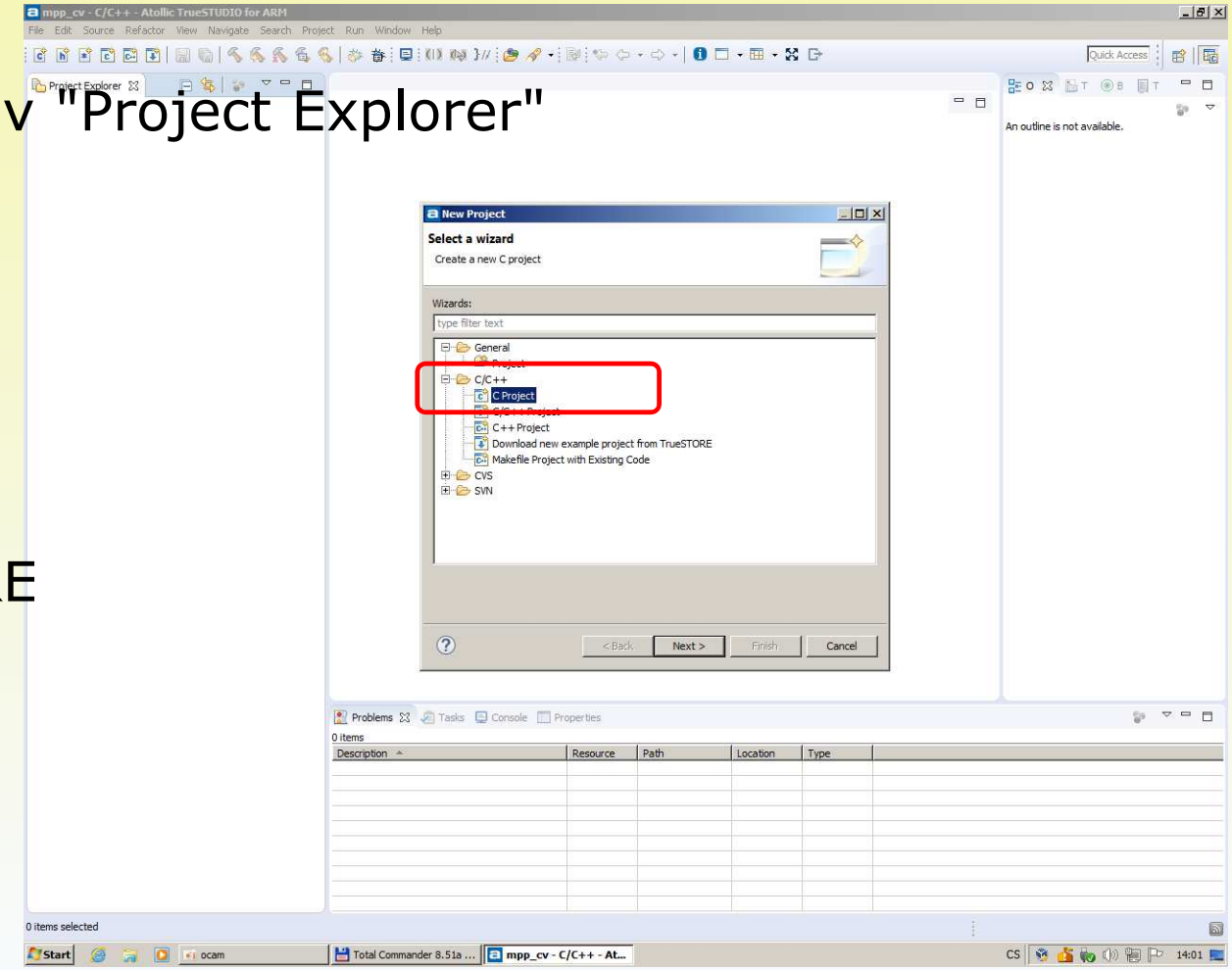
Vytvoření nového Workspace + prvotní nastavení



- Video https://youtu.be/pSTtUWc3f9k
- Doporučené prvotní nastavení:
  - Zavřít okno "C/C++ perspective - Build Analyzer" – vpravo dole
  - V nabídce Window/Preferences lze hledat napsáním prvních pár znaků hesla (viz. video)
    - Local History - 1 - 500 – 1 = omezí počet dočasných souborů
    - Text Editors - Tab width = 2, Insert spaces for tab
    - Typing – přejít na link "Code style preference page" (přejde na "Formatter")
      - Zvolit "BSD/Allmann" a přejít na "Edit"
        - Nelze měnit std. Profily, ale stačí změnit název a vytvoří se nový "Profile name" a pak půjdou tlačítka "Apply" a "Export"
      - Nastavit - Spaces Only, Indentation size = 2, Tab size = 2
      - V "Indentation" – checkbox "public, ..." – zaškrtnout
      - V "Control statement" - odškrtnout "keep else if on one line"
    - V "Colors and Fonts" pro C/C++ by mělo být něco lepšího než "Courier"
      - Doporučuji "Consolas" nebo "Lucida Console" apod.
- Tato nastavení jsou pro všechny projekty daného Workspace

Vytvoření nového projektu

- Menu File/New Project
  - Nebo pravým tlačítkem v "Project Explorer"
- Vybrat "C Project"
  - Adresář
    - Vytvoří se "sám"
  - Embedded C Project
    - Atollic ARM
  - Procesor - STM32F411RE
    - Máme na Nucleo
  - Debug probe - ST-Link
    - Příp. pro Linux – J-Link



C Project

C Project

Create C project of selected type

1

Project name: cv1

Use default location

Location: C:\temp\mpp_cv\cv1

Project type:

- Executable
  - Empty Project
  - CMSIS C/C++ Project
  - PC C Project
  - Embedded C Project
- Shared Library
- Static Library
- Makefile project

Toolchains:

Atollic ARM To

Show project types and toolchains only if they are supported on

< Back | Next >

C Project

TrueSTUDIO® Hardware configuration

Select hardware settings

2

Target: f411

Device	Info
STMicroelectronics	
Boards	
NUCLEO-F411RE	Atollic TrueSTUDIO provide...
MCUs	
STM32F411xx devices	
STM32F411CC	Atollic TrueSTUDIO provide...
STM32F411CE	Atollic TrueSTUDIO provide...
STM32F411RC	Atollic TrueSTUDIO provide...
STM32F411RE	Atollic TrueSTUDIO provide...
STM32F411VC	Atollic TrueSTUDIO provide...
STM32F411VE	Atollic TrueSTUDIO provide...

Floating point: Hardware implementation

Floating point unit: FPv4-SP-D16

Code location: FLASH

Instruction set: ARM, Thumb, Thumb2

Endianess: Big endian, Little endian

Memory information: STM32F411RE: FLASH: 512KB, RAM: 128KB

< Back | Next > | Finish | Cancel

C Project

TrueSTUDIO® Software configuration

Select software settings

3

Runtime library

Library: Newlib-nano (Reduced functionality)

Use tiny printf/sprintf/fprintf (small code size)

Generate system calls file (enable I/O redirection and OS integration)

Dynamic Heap size (Default)

Optimization

Remove unused code (dead code removal)

Remove unused data (dead data removal)

C Project

TrueSTUDIO® Debugger configuration

Select hardware debug settings

4

Debug probe selection

Debug Probe: ST-LINK

ST-LINK

This supports debugging with ST-Link on STM32 boards.

C Project

Select Configurations

Select platforms and configurations you wish to deploy on

5

Project type: Executable

Toolchains: Atollic ARM Tools

Configurations:

- Debug
- Release

Select all

Deselect all

Kostra programu

- Funkce Main
- Inkrementující cyklus nekonečný

mpp_cv - C/C++ - cv1/src/main.c - Atollic TrueSTUDIO for ARM

File Edit Source Refactor View Navigate Search Project Run Window Help

Project Explorer

- cv1
  - Binaries
  - Includes
  - Drivers
  - src
    - main.c
    - startup_stm32f411xe.s
    - system_stm32f4xx.c
    - tiny_printf.c
  - Debug
  - stm32_flash.ld

main.c

```c
/*
******************************************************************************
File:     main.c
Info:     Generated by Atollic TrueSTUDIO(R) 7.0.1   2017-01-26

The MIT License (MIT)
Copyright (c) 2009-2016 Atollic AB

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
******************************************************************************
*/

/* Includes */
#include "stm32f4xx.h"

/* Private macro */
/* Private variables */
/* Private function prototypes */
/* Private functions */

/**
**===========================================================================
**
**  Abstract: main program
**
**===========================================================================
*/
int main(void)
{
  int i = 0;
```

stm32f4xx.h

main(void) : int

Problems | Tasks | Console | Properties

0 items

Description	Resource	Path	Location	Type

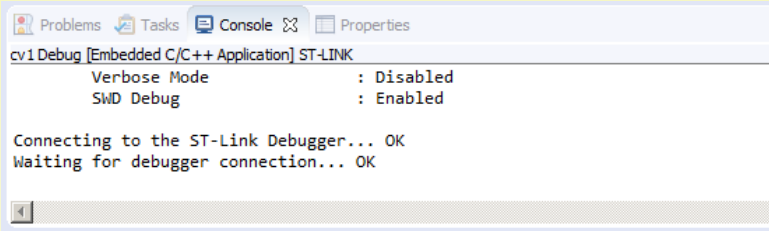
/cv1/src/main.c

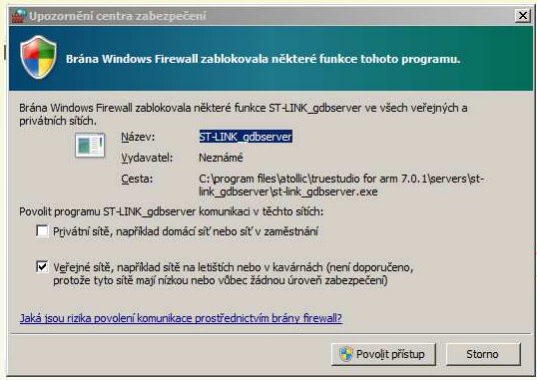
C/C++ Indexer: (0%)

Start | ocam | Total Commander 8.51a ... | mpp_cv - C/C++ - cv... | CS | 14:02

Spuštění aplikace

- Po provedení změn v programu je potřeba provést několik kroků k jeho spuštění
  - Build
    - Překlad (kompilace) jednotlivých C zdrojových souborů
    - Linkování se standardními i vlastními knihovnami
  - Start Debug
    - Při prvním spuštění se musí nastavit "Debug Probe"
    - Spuštění "Debug perspective" = jiný layout oken
    - Spuštění GDB = speciální program pro debugování
      - Při prvním spuštění chce povolit přístup přes TCP/IP (vnitřně)
    - Nahrání přeloženého programu do paměti mikroprocesoru na desce
    - Aktivace voleb (menu a ikony) v prostředí
  - Nyní je možno spustit vykonávání
  - Debug proces lze přerušit (pauza) a potom:
    - Vidět, kde se právě vykonává program
    - Prohlédnout si hodnoty proměnných
    - Prohlédnout si stavy řídících registrů mikrokontroléru





- Případná změna zdrojového kódu programu znamená opakování všech kroků

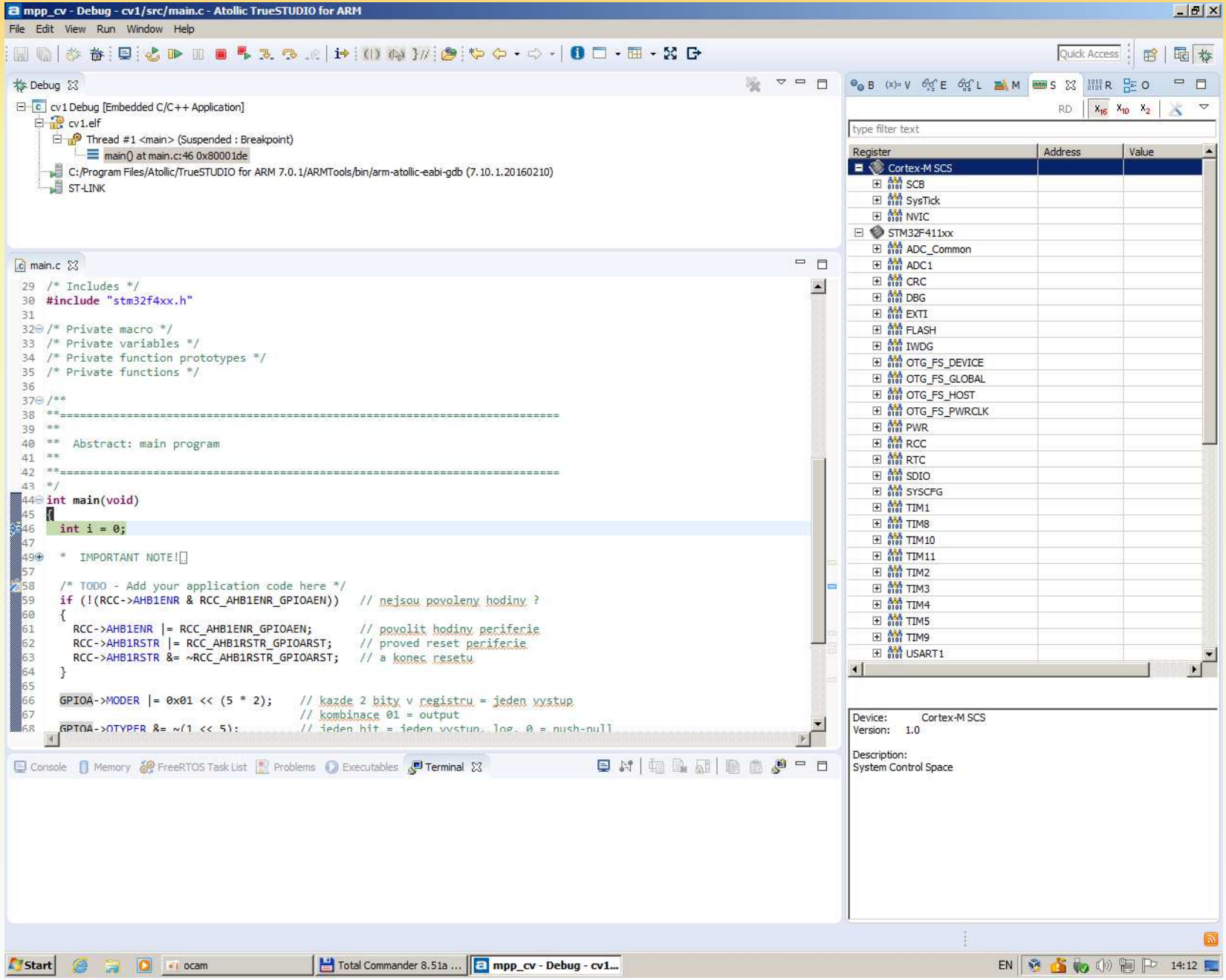
Ikony v C/C++ i Debug perspektivě

- Přepínání perspektiv (= layout oken a ikony) je buď automatické nebo lze ručně – vpravo nahoře

	Vytvoří v projektu nový C soubor		Znovu spuštění debug procesu
	Vytvoří v projektu nový H soubor		Restart běžícího programu v mikrokontroléru
	Ulož aktuální soubor (pokud je změněn)		Spuštění nahraného programu
	Ulož všechny změněné soubory		Zastavení běhu programu (pauza)
	Build – překlad změněných a linkování		Zastavení programu a ukončení debug
	Build All – znovu provést komplet Build		Zastavení, nová kompilace a nové spuštění
	Clean – smaže přeložené a dočasné soubory		Krok – v případě funkce dovnitř
	Spuštění Debug		Krok – v případě funkce ji vykonat a "jít dál"
	Nastavení Debug = výběr "Debug Probe", ...		Dokončení funkce a krok "ven"

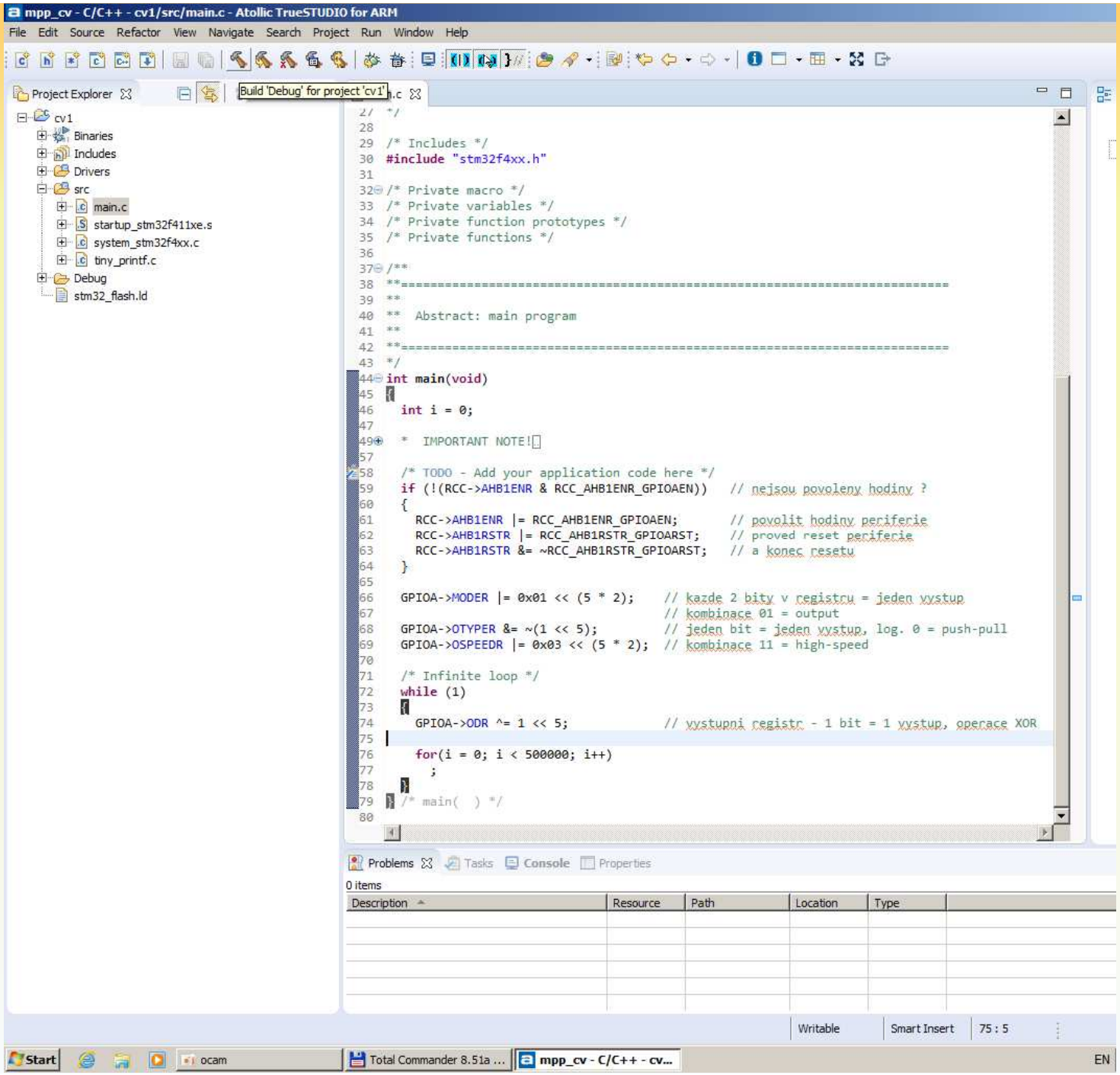
Debug perspektiva

- Při prvním spuštění je vhodné pozavírat nepoužívaná okna a doladit layout
  - Viz. YT video
- Program zatím nic nevykonává, ale měl by se bez chyb nahrát do desky a měl by jít spustit



První program = blikání LED

- Také na YT video
- Při psaní kódu prostředí "napovídá"
  - Nabídne po -> prvky struktury
  - U delších identifikátorů nabízí možnosti doplnění
  - Pokud nenapoví automaticky, možno nabídku vyvolat Ctrl+Space
- Automaticky se doplňují párové závorky
  - Občas ne úplně správně
  - Chování nastavitelné v Preferences
- Poloautomaticky se vytváří odsazování
  - Opět dle pravidel v Preferences
  - Příp. ručně v menu Source
    - Correct Indentation
    - Format



Dokumentace a zdroje informací

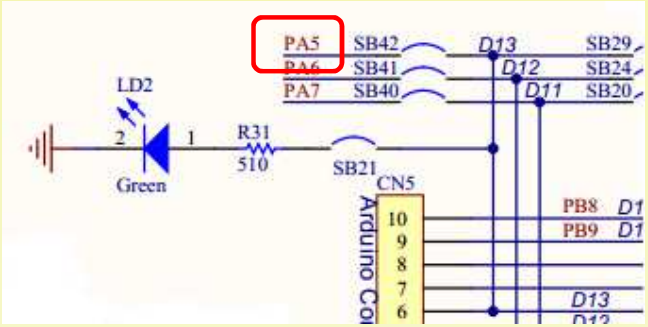
- V následujících slajdech jsou uvedeny fragmenty z PDF dokumentace
  - Dokument, kapitola a příp. strana nebo číslo obrázku
  - Dokumenty dostupné z Courseware nebo webu (na http://st.com pozor na novější/jiné verze)

- **RM** = Reference manual = společný pro více typů/skupinu
  - RM0383 - STM32F411xE - rev 1 - DM00119316.pdf
  - Reference manual STM32F411xC/E advanced ARM®-based 32-bit MCUs
- **DS** = Datasheet = specifické vlastnosti konkrétního typu
  - DS10314 - STM32F411xE - rev 6 - DM00115249.pdf
  - STM32F411xC STM32F411xE
  - ARM® Cortex®-M4 32b MCU+FPU, 125 DMIPS, 512KB Flash, 128KB RAM, USB OTG FS, 11 TIMs, 1 ADC, 13 comm. interfaces
- **PM** = Programming manual = popis funkcionalit jádra společný pro celou rodinu
  - PM0214 - STM32F4 F3 L4 - rev 5 - DM00046982.pdf
  - STM32F3, STM32F4 and STM32L4 Series Cortex®-M4 programming manual
- Nucleo **UM** = User manual a Schematic = dokumentace kitu
  - Nucleo UserManual rev10 - DM00105823.pdf
  - Schematic - Nucleo schematic - rev C - MB1136.pdf
- Podklady pro **KAE/MPP Shield** = dokumentace rozšiřujícího modulu
  - Shield Schematic - NSHLD_SCH_1600xx_A.0_Basic.PDF
  - DS od akcelerometru - MMA7660FC.pdf
  - DS od budiče pro zelené LED - STP08CP05 - CD00160230.pdf
  - Darlingtony pro červené LED a RGB - uln2803a.pdf,

Analýza prvního programu 1 – kód

- Úvodní #include připojuje soubor s definicí názvů registrů pro náš procesor
  - Bývá společný pro celou skupinu (řadu), zde všechny STM32F4
    - Uvnitř jsou některé části neaktivní pomocí #ifdef dle konkrétního typu mikrokontroléru
  - Názvy typicky velkými písmeny (GPIOA)
  - Bloky registrů jsou ve skupinách – realizováno pomocí struct
  - Stejných bloků může být více, proto odkaz z blok je realizován jako ukazatel
    - Přístup k registru v bloku pomocí -> (prvek struktury dané adresou)
  - Registr je typově celočíselná proměnná (16b nebo 32b) s pevně danou adresou v paměti (ve formě offsetu od počátku bloku)
  - Podle řídících bitů v registru jsou pojmenovány konstanty – možno použít dle pravidla REGISTR_BIT
- If blok testuje, zda má periférie GPIOA aktivní (povolené) hodiny
  - Pokud ne, povolí vstup hodin do periférie v registru AHB1ENR bloku RCC
  - Dále provede RESET periférie GPIOA bitem v registru AHB1RSTR
- Nastaví se vlastnosti GPIOA pomocí registrů
  - GPIOx obsahuje 16 I/O signálů
  - Některé vlastnosti jsou 1-bitové (typ výstupu OTYPER) nebo 2-bitové (MODER, OSPEEDR)
  - Nastavení bitů pomocí bitového maskování
- Nekonečná smyčka
  - Typická konstrukce v mikrokontrolérech = program nikdy nekončí (až do vypnutí nebo RESETu)
- Negace vybraného bitu pomocí bitové operace XOR
- Časování FOR cyklem = hodně opakování "naprázdno"
  - Mez cyklu určena zkušeností, výsledný čas záleží na rychlosti (hodinách) procesoru

Analýza prvního programu 2 – HW



- Na Nucleo desce je LED připojena na signál PA5
  - bit 5 I/O brány GPIOA (General-purpose I/O)
  - Viz. Nucleo Schematic, str. 4

- Nutno uvést do provozu GPIOA
  - Jednotlivé části (bloky nebo také periférie) mikrokontroléru mají své "hodiny" = nutný signál pro činnost
    - V bloku RCC se dají v registrech xxxENR příslušnými bity
    - Je nutno vědět, na který zdroj hodin je periférie interně připojena
    - Defaultně jsou prakticky všechny hodiny periférie vypnuty
  - V RCC bloku jsou obdobně registry pro RESET periférií - xxxRSTR
    - Stejné pořadí bitů jako u ENR registrů
    - Nastaví vnitřní stav (a řídící registry) periferie do výchozího stavu

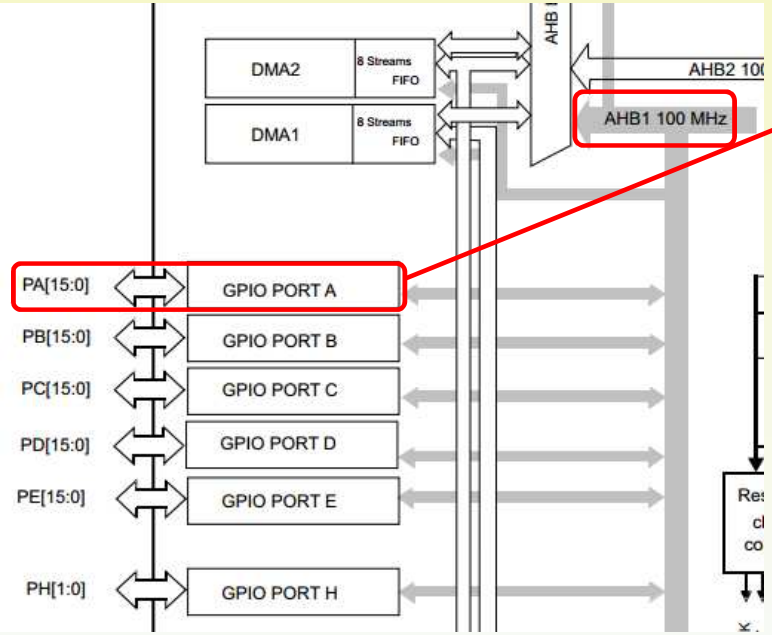
- Nakonfigurovat signál č. 5 jako výstupní
  - Mód – registr MODER – vybírá vstup, výstup, speciální režim pro jinou periférii (= alternate function) nebo analogový vstup
  - Typ výstupu – OTYPER – push-pull nebo otevřený kolektor
  - Rychlost výstupu (= strmost hrany) – OSPEEDR – Low/Medium/Fast/High speed

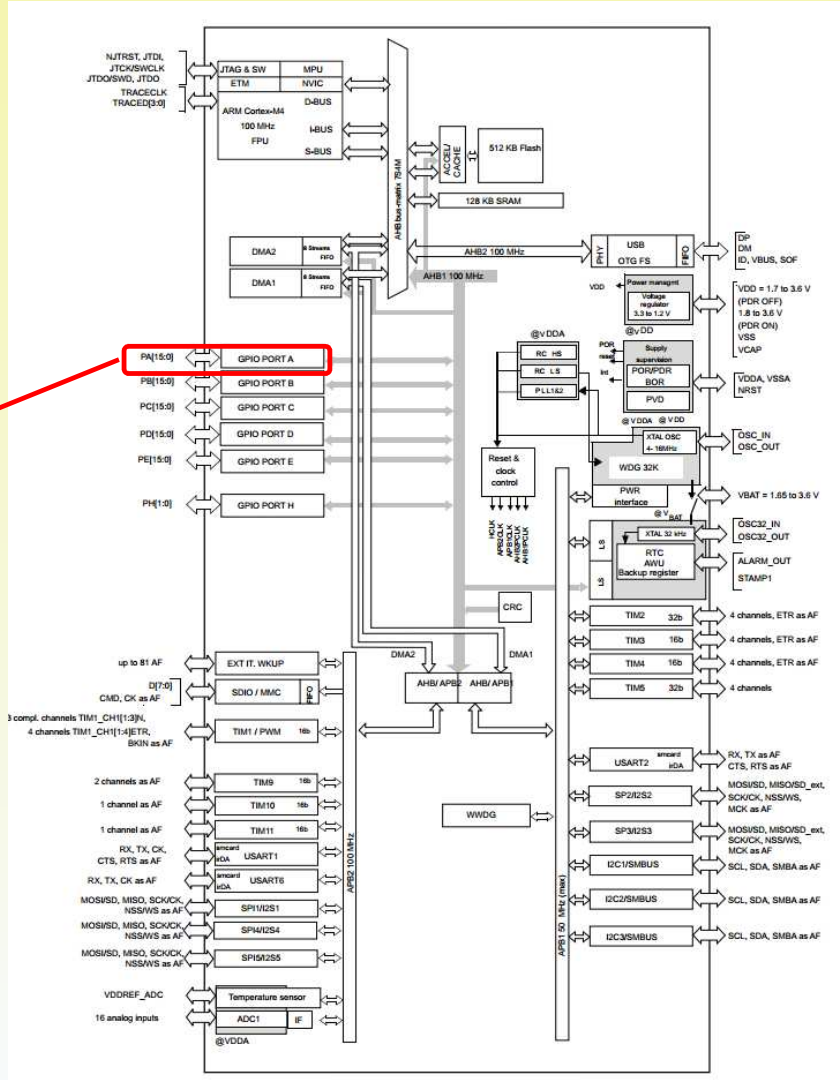
Distribuce hodin v mikrokontroléru (STM32F4xx)

- Základní hodinový takt je společný celému mikrokontroléru a vychází z High-Speed zdrojů
  - Vnitřní = HSI (16HMz)
    - Nastaven po RESETu, tj. procesor se vždy "rozběhne"
  - Vnější (např. krystal) – HSE (4-26MHz)
    - Pokud se program pokusí přepnout na HSE a není k dispozici, zůstává HSI a nastaví se příznak chyby
- Z HSx se pak přímo nebo přes PLL blok získává SystemCoreClock = základní takt
  - PLL slouží k dělení a násobení HSx téměř libovolným číslem = možno jemně ladit výsledný takt
- Z CoreClock se získává základní takt pro periférie přes "sběrnice"
  - AHBx – ARM HighSpeed Bus – AHB1 a AHB2
  - APBx – ARM Peripheral Bus – APB1 a APB2
  - Každá sběrnice má svoji předděličku
    - Nutno respektovat při práci s časováním periférií – typicky komunikační rychlosti
    - Bude se řešit až v době nastavování hodin procesoru
  - Pozor, pro jiné typy procesorů některé AxBx nemusí být k dispozici a také podobné periférie mohou být na jiných sběrnicích

Distribuce hodin v mikrokontroléru (STM32F4xx)

- Určení sběrnice, ke které je připojena periférie
  - Např. DS, str. 15/149 - Figure 3. STM32F411xC/xE block diagram





Použité registry RCC – viz. RM, kap. 6.3

- **AHB1ENR**
  - 1 = enable
  - 0 = disable

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved									DMA2EN	DMA1EN	Reserved				
									rw	rw					

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved			CRCEN	Reserved				GPIOH EN	Reserved		GPIOEEN	GPIOD EN	GPIOC EN	GPIOB EN	GPIOA EN
			rw					rw			rw	rw	rw	rw	rw

- **AHB1RSTR**
  - 1 = reset
  - 0 = pracovní režim

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved									DMA2 RST	DMA1 RST	Reserved				
									rw	rw					

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved			CRCRST	Reserved				GPIOH RST	Reserved		GPIOE RST	GPIOD RST	GPIOC RST	GPIOB RST	GPIOA RST
			rw					rw			rw	rw	rw	rw	rw

- Šikovně jsou bity na stejných pozicích
- Při resetu je potřeba nastavit log 1 a následně vrátit log 0, jinak nebude periférie fungovat
  - Výhodně možno nastavit maskováním bitem pomocí OR
  - Obdobně nulování pomocí AND a bitově negovaná konstanta čísla bitu

Použité registry GPIO(A) – viz. RM, kap. 8.4

- **MODER** - režim
  - Každé 2 bity = typ GPIO
  - 00: Input (reset state)
  - 01: General purpose output mode
  - 10: Alternate function mode
  - 11: Analog mode

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

- **OTYPER** – typ výstupu
  - Každý bit řídí 1 výstup
  - 0: Output push-pull (reset state)
  - 1: Output open-drain

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
OT15	OT14	OT13	OT12	OT11	OT10	OT9	OT8	OT7	OT6	OT5	OT4	OT3	OT2	OT1	OT0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

- **OSPEEDR** – rychlost hrany
  - 00: Low speed
  - 01: Medium speed
  - 10: Fast speed
  - 11: High speed

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
OSPEEDR15 [1:0]		OSPEEDR14 [1:0]		OSPEEDR13 [1:0]		OSPEEDR12 [1:0]		OSPEEDR11 [1:0]		OSPEEDR10 [1:0]		OSPEEDR9 [1:0]		OSPEEDR8 [1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OSPEEDR7[1:0]		OSPEEDR6[1:0]		OSPEEDR5[1:0]		OSPEEDR4[1:0]		OSPEEDR3[1:0]		OSPEEDR2[1:0]		OSPEEDR1 [1:0]		OSPEEDR0 1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

- **ODR**
  - Každý bit řídí hodnotu výstupu

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
ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Jak číst dokumentaci ?

- Především RM je rozdělen podle periférií
  - Jména pak odpovídají názvům struktur popisujících registry
  - Může být více periférií stejného typu (stejná funkčnost a stejné registry)
    - V PDF je např. GPIOx
    - Reálně se používá GPIOA až GPIOF
    - Příp. se čísluje – SPIx může být SPI1, SPI2, ...
- Až na vyjímky budou uváděny hlavně popisy registrů a jejich bitů, ale funkčnost a význam je v dokumentaci popsán "před nimi"
- Některé pojmy
  - "reset state" = stav po RESETu (celého obvodu nebo jen periférie)
  - "r", "w", "rw" = bit jen pro čtení, jen pro zápis, čtení i zápis
    - V jednom registru mohou všech typů najednou
  - "reset value" = hodnota registru po RESETu
  - "address offset" = relativní adresa registru vzhledem k počáteční adrese periférie
    - Díky definicím v .H souboru stačí používat jen symbolická jména

Jazyk C – struktury a ukazatele

- Symbol **GPIOA** reprezentuje skupinu registrů brány PA
  - V kódu lze snadno přejít na definici symbolu
    - Stiskem F3 je-li kurzor na "slově"
    - Pravý klik a volba "Open Declaration"
    - Funguje i mezi dalšími soubory v rámci projektu
      - Lze použít i na #include = otevře příslušný soubor
  - V **stm32f411xe.h** je uvedeno na řádku **808**

```
#define GPIOA    ((GPIO_TypeDef *) GPIOA_BASE)
```

    - Kde adresa je odvozena od bloku adres periférií

```
#define GPIOA_BASE (AHB1PERIPH_BASE + 0x0000)
```

    - A typ je reprezentován strukturou na řádku 286

- Abychom mohli přistoupit k registru, je třeba vzít ukazatel na blok registrů (= strukturu) a na prvek struktury přistoupit pomocí operátoru ->

```
GPIOA->MODER |= ...
```

```
typedef struct
{
  __IO uint32_t MODER;    /*!< GPIO port mode
register, Address offset: 0x00      */
  __IO uint32_t OTYPER;   /*!< GPIO port output
type register, Address offset: 0x04      */
  __IO uint32_t OSPEEDR;  /*!< GPIO port output
speed register, Address offset: 0x08      */
  __IO uint32_t PUPDR;    /*!< GPIO port pull-
up/pull-down register, Address offset: 0x0C */
  __IO uint32_t IDR;      /*!< GPIO port input
data register, Address offset: 0x10      */
  __IO uint32_t ODR;      /*!< GPIO port output
data register, Address offset: 0x14      */
  __IO uint32_t BSRR;     /*!< GPIO port bit
set/reset register, Address offset: 0x18      */
  __IO uint32_t LCKR;     /*!< GPIO port
configuration lock register, Address offset: 0x1C
*/
  __IO uint32_t AFR[2];   /*!< GPIO alternate
function registers, Address offset: 0x20-0x24 */
} GPIO_TypeDef;
```

Jazyk C - bitové operace

- Většina operací s registry periférií má bitový význam
  - Musíme umět nastavit nebo vynulovat konkrétní bit (nebo bity)
  - Bitové operace pracují s jednotlivými bity hodnoty (konstanty nebo výrazu) operandů
    - **|** = bitový OR = pokud je alespoň jedna 1, výsledek je 1 ("vyhrává 1")
    - **&** = bitový AND = musí být obě 1, aby byl výsledek 1, jinak 0 ("vyhrává 0")
    - **^** = bitový XOR = pokud jsou různé, výsledek = 1, při shodě výsledek = 0
      - Negace bitu v prvním operandu, kde je 1 v druhém operandu
    - **~** = bitová negace = zneguje každý bit hodnoty, jedná se o unární operátor
  - Pokud mají operandy různou velikost typu (8b, 16b, 32b), "roztáhne" se automaticky na větší z nich
  - Pozor na znaménko (může se uplatnit znaménková korekce)
    - Zásadně se bitové operace provádějí s **unsigned** typy (v např. **uint32_t**)
- Hodnoty se výhodně zapisují hexadecimálně
  - Pak jedna číslice (0-9, A-F nebo a-f) odpovídá 4 bitům

A	0xAA	1	0	1	0	1	0	1	0
B	0x0F	0	0	0	0	1	1	1	1
A&B	**0x0A**	**0**	**0**	**0**	**0**	**1**	**0**	**1**	**0**
A|B	**0xAF**	**1**	**0**	**1**	**0**	**1**	**1**	**1**	**1**
A^B	**0xA5**	**1**	**0**	**1**	**0**	**0**	**1**	**0**	**1**
~A	**0x55**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**

Jazyk C – zkrácené výrazy, posuvy

- Jazyk C umožňuje zkrácený výraz pro matematické operace
  - Operace **a = a + b;** lze zkráceně zapsat jako **a += b;**
  - Není třeba opakovat operand (proměnnou), pokud je výsledek vložen "zpět"
  - Zjednodušený zápis u delších názvů proměnných
  - Logičtěji ukazuje požadovanou akci

- Operace posuvu vlevo **<<** a vpravo **>>** pracuje s bity
  - Na uprázdněné místo se doplní 0 (na kraji bitového slova)
  - Je velmi doporučeno používat **unsigned** datové typy
  - Lze použít také na konstanty – např. výraz **1 << 5**
    - Odpovídá hodnotě **0x0020** (bitově 0000 0000 0010 0000)
    - Je ale čitelnější = "log. 1 na 5. bitu hodnoty"
  - Možno spojit pomocí bitového OR - např. **((1 << 5) | (1 << 7))**
    - Odpovídá **0x00A0** (bitově 0000 0000 1010 0000)
    - Je vidět, že nastavuji 5. a 7. bit do log. 1

  - Pokud potřebuji log. 0 na konkrétní pozici, je výhodné vzít log. 1 na stejné pozici a provést bitovou negaci – např. **~(1 << 9)**
    - Odpovídá **0xFDFF** (bitově 1111 1101 1111 1111 jako negace 0000 0010 0000 0000)
    - Zde je čitelnost mnohem vyšší, pokud jsme na takový zápis zvyklí

Detekce a odstranění chyb v kódu

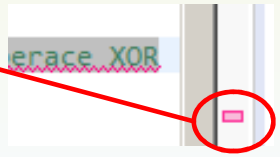
- Pokud překladač při překladu narazí na chybu, vypíše ji a pokusí se pokračovat
  - Chyby proto nutno odstraňovat od začátku
  - Někdy je faktická chyba je detekována jako více chyb - **Error**
  - Všechny chyby musí být odstraněny, jinak není úspěšný překlad
- Překladač může narazit i na konstrukci, kterou lze přeložit, ale je v něčem podezřelá nebo nejasná
  - Překladač hlásí varování (**Warning**), přeložený kód je vytvořen, ale nemusí pracovat správně
  - **!! Zásadně se snažíme odstraňovat i varování !!**
- Informace o chybách je v okně "Problems"
  - Ukazuje se typ a popis chyby, řádek a soubor, kde se na ni narazilo – pozor, může to být až ZA skutečnou příčinou chyby

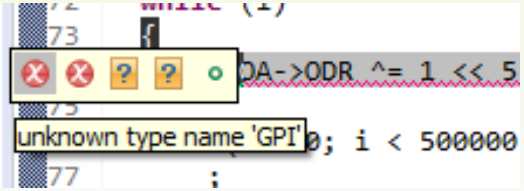
Problems | Tasks | Console | Properties

2 errors, 0 warnings, 0 others

Description	Resource	Path	Location	Type
Errors (2 items)				
expected '=', ',', ';', 'asm' or '__attribute__' before '->' token	main.c	/cv1/src	line 74	C/C++ Pro...
unknown type name 'GPI'	main.c	/cv1/src	line 74	C/C++ Pro...

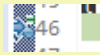
- Zároveň je chyba znázorněna v kódu a po najetí myši se ukáže popis
  - Je značka i na scrollbaru



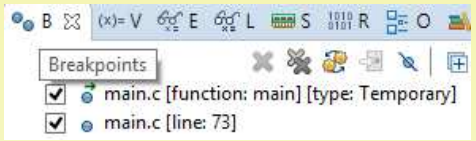


Pokročilejší práce s debuggerem

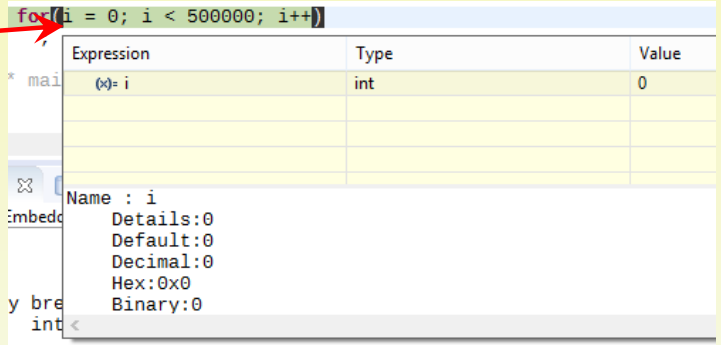
- Pokud je program zastaven (vykonávání dospělo k Breakpointu nebo byla stisknuta "pauza") je možné zkoumat stav programu i procesoru
  - Řádek, který bude vykonán je označen šipkou
  - Řádek s Breakpointem má modré kolečko
    - V okně/záložce Breakpoints jsou všechny uvedeny, je možno je zapínat/vypínat
    - Architektura debuggeru a procesoru umožňuje max. 4 (?) breakpointy

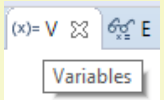



```
46  int i = 0;
```

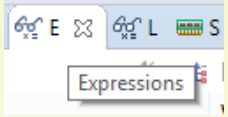


- Najetím myší nad proměnnou se ukáže její hodnota

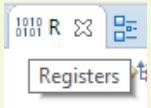




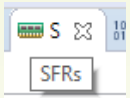
- Stejné informace obsahuje záložka "**Variables**" – lokální proměnné



- Je možné počítat i výrazy – viz. záložka "**Expressions**"



- V záložce "**Registers**" je obsah registrů jádra procesoru



- V záložce "**SFRs**" jsou jednotlivé periférie a jejich registry
  - Registr je možné "rozbalit" na bitové hodnoty
  - Ukazují se změny po posledním kroku
  - Možno změnit obsah

Shrnutí domácí přípravy - I

- Naučili jsme se
  - Založit Workspace a nový projekt pro STM32F411
  - Nastavit činnost GPIO
    - Povolit hodiny v RCC periférii
    - Nastavit režim příslušného výstupu
  - V nekonečném cyklu "blikat" LED-kou
    - Blikání pomocí bitového XOR v ODR registru
    - Čekání realizováno cyklem for
- Zopakovali jsme konstrukce jazyka C
  - Bitové operace – AND, OR
  - Maskování pomocí bitových operací
  - Posuvy a využití pro nastavení bitů v registrech pomocí konstant
  - Přístup k prvkům struktury přes ukazatel (její adresu)

Plán domácí přípravy

I. Instalace prostředí, ... První program – blikání LED na GPIO

II. Zobecnění GPIO operací

- **Kopírování projektů**
- **Vlastní funkce – nastavení GPIO, čtení (z Input), zápis (do Output), negování (Output)**
- **Rozdělení do souborů, linkování, #include**

III. Systick

IV. Timer

V. UART

VI. Rezerva

VII. SPI

VIII. PWM

IX. A/D

X. I2C (využití "knihovny")

XI. Časování – HSI/HSE/PLL

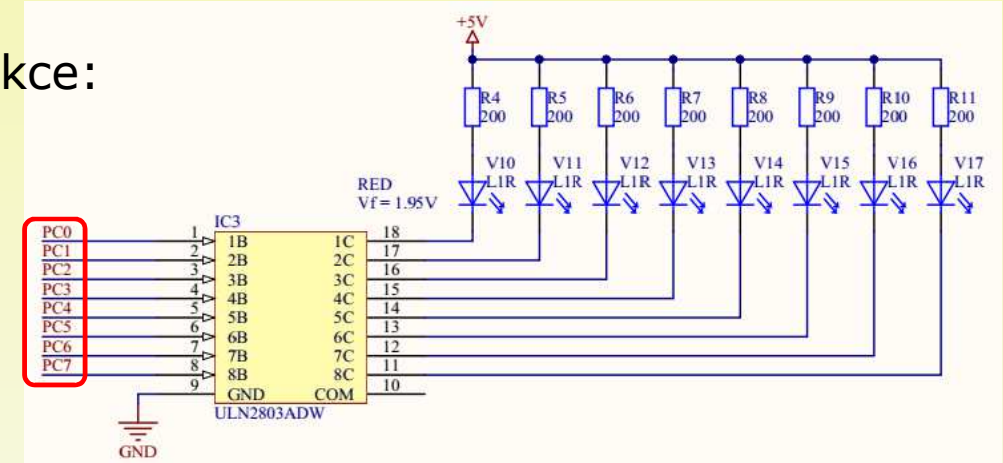
XII. Rezerva

Cíle přípravy

- Dále prozkoumat řídící registry GPIO
  - Včetně vnitřní struktury v HW
- Prohlédnout schéma zapojení a podle toho připravit kód
  - Pracovat s 8x červené LED "najednou"
- Rozšířit kód do funkcí pro snadné další využití
  - Nastavení a výstup hodnoty pro 8xLED
  - Univerzální nastavení jednotlivých GPIO
  - Funkce pro vstup, výstup a změnu (toggle) I/O bitu
- Vytvořit samostatné soubory s kódem funkcí a s deklaracemi (hlavičkami) funkcí
  - Bude pak stačit kopírovat soubory do dalších projektů

Funkce pro využití 8x LED přímo na GPIO

- Osmice červených LED je přes budič připojena na 8 bitů (vstupů/výstupů) brány PC (GPIOC)
- Pro práci s celou "osmicí" si připravíme funkce:
  - Nastavení PC0 – PC8 jako výstup
    - Registry MODER, OTYPER a SPEEDR
  - Zápis 8-bitové hodnoty do výstupu
    - Zápis do dolní poloviny registru ODR



- Začneme novým projektem
  - Viz. YT ??? a dcv 1

- POZOR - Překladač C je v principu jednoprůchodový
  - Funkce musí být známa před použitím
  - Nejjednodušší je umístit je před main
  - V kostře kódu je komentář:
    /* Private functions */
  - A za ním umístíme funkce

Využití 8x LED – nastavení osmice I/O

```
void Init8LED(void)
{
  if (!(RCC->AHB1ENR & RCC_AHB1ENR_GPIOCEN))    // neni aktivni GPIOC ?
  {
    RCC->AHB1ENR |= RCC_AHB1ENR_GPIOCEN;        // povol hodiny
    RCC->AHB1RSTR |= RCC_AHB1RSTR_GPIOCRST;     // udelej RESET puls periferii
    RCC->AHB1RSTR &= ~RCC_AHB1RSTR_GPIOCRST;
  }

  GPIOC->MODER &= 0xffff0000;    // nizsich 8 I/O nastaveno na 00 kazdy = defaultni stav
  GPIOC->MODER |= 0x00005555;    // a pak jim nastav kazdemu 01 = output, bitove zapis ... 0101 0101 0101

  GPIOC->OTYPER &= 0xff00;       // kazdy vystup reprezentuje 1 bit, 0 = push-pull

  GPIOC->OSPEEDR |= 0x0000ffff;  // 2 bity na vystup, kombinace 11 = high speed
  return;
}
```

- Vždy je nutné správně maskovat bity tak, aby ostatní nebyly "zasaženy", protože mohou být připojeny na jiný HW
  - MODER a OSPEEDR mají 2 bity na signál (výstup)
  - OTYPER má 1 bit na signál
- Popis registrů – RM kap. 8.4 nebo Dcv 1
- Nezapomenout na hodiny periférie

Využití 8x LED – výstup celé osmice I/O

```
void Write8LED(uint8_t val)
{
  GPIOC->ODR = (GPIOC->ODR & 0xff00) | ((uint16_t)val);  // zachovat hornich 8b a prepsat spodnich 8b hodnotou
  return;
}
```

- Zápis do ODR by šlo provést i ve více krocích
  - Zde uvedený postup je poměrně efektivní
- Tabulka naznačuje operace
  - Bity zapisujeme nejvýznamnějším (MSB) vlevo
  - Podle RM má v ODR význam jen "spodních" 16b
  - Přetypování uint8 na uint16 "roztáhne" 8-bitovou hodnotu na 16-bitovou a protože je všechno unsigned, doplní zleva nulovými bity

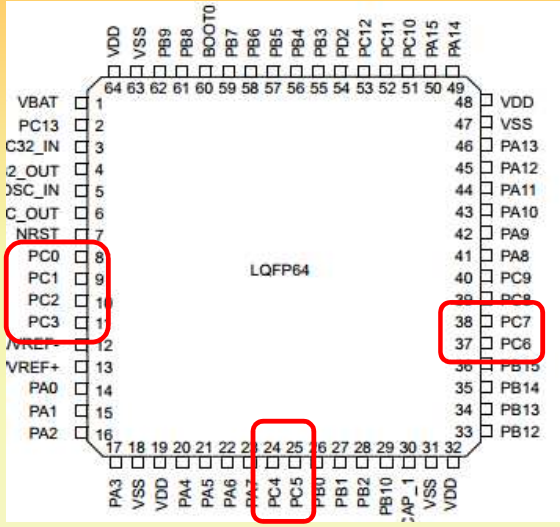
Bit	**15**	**14**		**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
ODR	C	C	...	C	C	C	C	C	C	C	C	C
0xff00	1	1	...	1	0	0	0	0	0	0	0	0
ODR & 0xff00	C	C	...	C	0	0	0	0	0	0	0	0
val	0	0	...	0	V	V	V	V	V	V	V	V
(...) | val	C	C	...	C	V	V	V	V	V	V	V	V

Generování vzoru "počítadlo" na 8xLED

```
... Main ...
{
  uint8_t b = 0;
...
  while(1)
  {
    b++;          // obycejne pocitadlo v 8b promenne
    Write8LED(b);

    ... "brzdici" FOR cyklus
  }
}
```

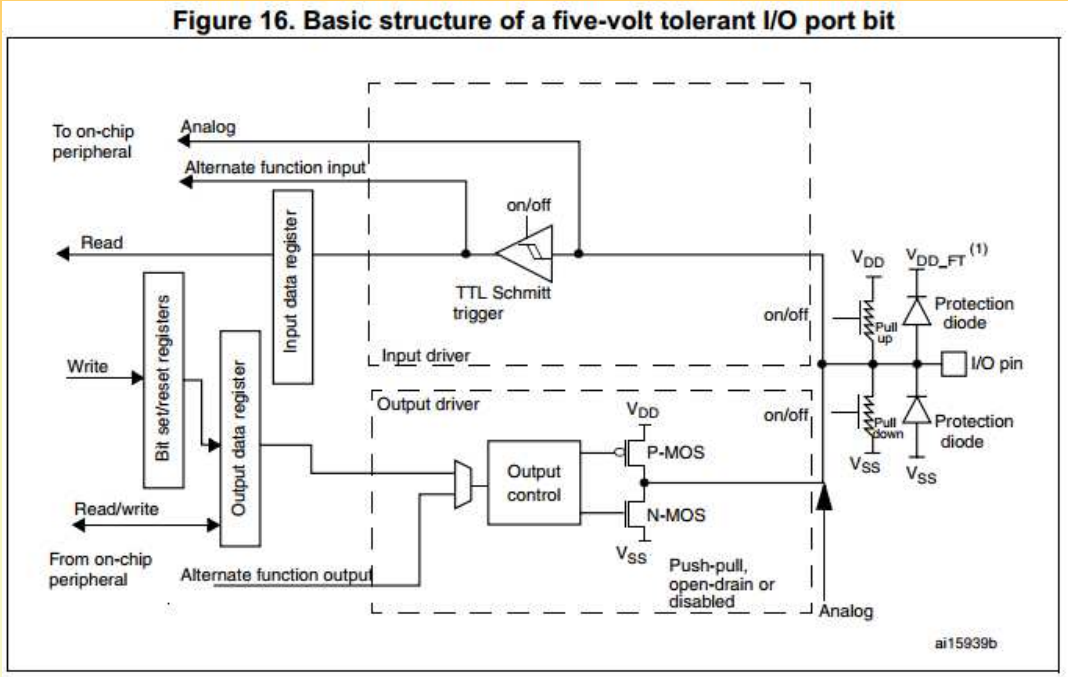
- Hodnota pro 8xLED je uložena v proměnné b
  - 8-bitová bez znaménka
- Není potřeba testovat max. hodnotu
  - V celočíselných proměnných je počet hodnot omezen počtem kombinací
  - V 8b proměnné mohou být čísla pouze v rozsahu 0000 0000 až 1111 1111 (binárně)
    - 0 – 255 dekadicky nebo 0x00 – 0xff hexadecimálně
  - Při přičtení 1 k nejvyšší hodnotě (0xff) následuje nejnižší hodnota
    - Počítá se "modulo" k rozsahu typu
  - Pro 16- a 32-bitové typy to funguje podobně
- Nezapomenout "zbrždění" vykonávání while-cyklu vložením prázdného for cyklu
  - Blikání by jinak bylo velmi rychlé
  - Zkuste různé hodnoty hranice FOR cyklu

Univerzální funkce pro 1x I/O



- Typické připojení externího HW využívá buď
  - Skupiny I/O – viz. 8xLED
    - Ideálně logicky "vedle" sebe – nemusí být stejné na pouzdře
      - Viz. DS Fig. 11, str. 35/149
    - Pracujeme s celou skupinou (pokud to jde)
  - Jednotlivé I/O –např. LED na Nucleo – připojená na PA5
    - Viz. dcv 1
    - Potřebujeme funkce pro individuální nastavení směru/vlastností a funkce pro čtení nebo zápis hodnoty (podle směru)
- Pro výstup použijeme registry ODR, OSPEEDR a OTYPER
  - Ještě lze použít BSRR (RM - kap. 8.4.7)
- Chování vstupů řídí registry IDR a PUPDR

GPIO – struktura HW



- Viz. RM Fig. 16 – str. 143/836
- Většina výstupů 5V tolerant
  - I když napájení je 3,3V
  - Snadné připojení běžných obvodů
- Při sledování zprava
  - Výstup je chráněn diodami
  - Programem lze "připnout" Pull-Up a/nebo Pull-Down odpory
- Základní režimy
  - Analogový vstup – žádné další ovlivnění signálu
  - Digitální vstup – možno zapnout Schmittův obvod
    - Čtení v příslušném bitu registru IDR (Input Data Register), pouze ke čtení
  - Digitální výstup – hodnota buď do ODR registru (lze vyčíst obsah)
    - Možno jen zapsat do BSRR (viz. dále)
    - Výstupní blok používá oba tranzistory = PushPull nebo jen "spodní" = OpenDrain (použito ke spojení více výstupů technikou "open collector", vyžaduje vnější Pull-Up
  - Alternativní režim (AF = Alternate Function)
    - GPIO sdílí vývody s dalšími periferiemi (časovač, komunikační sběrnice, ...)
    - Která periférie využívá daný vývod určují registry AFRL (RM 8.4.9) a AFRH (RM 8.4.10)
    - Některá nastavení jsou společná s GPIO – např. PP/OD, Pull-Up/Down – viz. Fig. 16

Funkce pro práci s I/O

```
void GPIOToggle(GPIO_TypeDef *gpio, uint32_t bitnum)
{
  gpio->ODR ^= (1 << bitnum);
}

bool GPIORead(GPIO_TypeDef *gpio, uint32_t bitnum)
{
  return ((gpio->IDR & (1 << bitnum)) != 0);    // porovnani = vynucena logicka hodnota
}
```

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
IDR15	IDR14	IDR13	IDR12	IDR11	IDR10	IDR9	IDR8	IDR7	IDR6	IDR5	IDR4	IDR3	IDR2	IDR1	IDR0
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

- Předpokládáme správně nakonfigurovaný vývod (pin)
  - Nekontroluje se, programátor "ví co dělá" ☺
- První parametr – "který" GPIO port
  - Všechny jsou definovány pomocí struktury GPIO_TypeDef
  - Liší se pouze adresy umístění bloku registrů – viz. stm32f4xx.h a dcv. 1
- Druhý parametr - "který" bit
  - Protože je 32b procesor, základním typem je uint32
  - Fungovalo by stejně i s uint16
  - Udává se pořadové číslo bitu (0 až 15, všechny porty jsou max. 16-bitové na STM32)
  - Pokud potřebujeme masku do registru, použijeme posun log. 1 na příslušnou pozici
- Toggle – XOR bitu v ODR viz. dcv. 1
- Čtení stavu vstupu
  - Přečtu hodnotu v IDR, logický AND s bitem, který mne zajímá (maska posunem)
  - Návratová hodnota má být logická, proto vynutím test nenulovosti
    - Možno využít C-čkovských pravidel (nula = nesplněno, ne-nula = splněno), ale toto je "čistší"

Zápis bitu výstupu pomocí BSRR

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BR15	BR14	BR13	BR12	BR11	BR10	BR9	BR8	BR7	BR6	BR5	BR4	BR3	BR2	BR1	BR0
w	w	w	w	w	w	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
BS15	BS14	BS13	BS12	BS11	BS10	BS9	BS8	BS7	BS6	BS5	BS4	BS3	BS2	BS1	BS0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

Bits 31:16 **BRy:** Port x reset bit y (y = 0..15)
These bits are write-only and can be accessed in word, half-word or byte mode. A read to these bits returns the value 0x0000.
0: No action on the corresponding ODRx bit
1: Resets the corresponding ODRx bit
Note: If both BSx and BRx are set, BSx has priority.

Bits 15:0 **BSy:** Port x set bit y (y= 0..15)
These bits are write-only and can be accessed in word, half-word or byte mode. A read to these bits returns the value 0x0000.
0: No action on the corresponding ODRx bit
1: Sets the corresponding ODRx bit

```
void GPIOWrite(GPIO_TypeDef *gpio, uint32_t bitnum, bool state)
{
  gpio->BSRR = (state) ? (1 << (bitnum)) : ((1 << (bitnum)) << 16);
        // BSRR register - lower 16 bits = Set to 1
        // higher 16 bits = Reset to 0
}
```

- Klasický zápis hodnoty jednoho bitu do ODR registru vyžaduje interně 3 akce
  - Přečíst stav, provést masku (OR pro nastavení, AND pro nulování) a zapsat zpět
  - V dcv týkajících se přerušení si ukážeme, že to není ideální postup
- Pomocí BSRR registru lze totéž provést jednou akcí
  - Horních 16 bitů (BRy) nastavuje log. 0 na daný výstup (Bit Clear)
  - Dolních 16 bitů (BSy) nastavuje log. 1 (Bit Set)
  - Požadovaná akce se vyvolá zápisem log. 1 do příslušného bitu v registru BSRR
    - Může jich být více najednou
  - Příklad – zápis 0x80800420
    - Bitově 1000 0000 1000 0000 0000 0100 0010 0000
    - Log. 1 jsou BR15, BR7, BS10 a BS5
      - Vynuluje se výstup Px15 a Px7
      - Nastaví se do log. 1 výstup Px10 a Px5
  - V případě shody má BSx vyšší prioritu
- V kódu použit ternární operátor – tj. Podmínka ? Výraz_1 : Výraz_2
  - Pokud je podmínka splněna, použije se hodnota výrazu 1, jinak výraz 2
  - Zapíše se log. 1 správně do BSRR
- Třetí parametr říká, zda se má zapsat log. 1 nebo log. 0 na výstup

Další registry pro GPIO Funkce Nastavení

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PUPDR15[1:0]		PUPDR14[1:0]		PUPDR13[1:0]		PUPDR12[1:0]		PUPDR11[1:0]		PUPDR10[1:0]		PUPDR9[1:0]		PUPDR8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PUPDR7[1:0]		PUPDR6[1:0]		PUPDR5[1:0]		PUPDR4[1:0]		PUPDR3[1:0]		PUPDR2[1:0]		PUPDR1[1:0]		PUPDR0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 2y:2y+1 **PUPDRy[1:0]:** Port x configuration bits (y = 0..15)
These bits are written by software to configure the I/O pull-up or pull-down
00: No pull-up, pull-down
01: Pull-up
10: Pull-down
11: Reserved

- **PUPDR**
  - Pull-Up, Pull-Down
  - 2 bity na I/O

- Funkce pro nastavení bude mít parametry port, číslo pinu a typ I/O

```
bool Nucleo_SetPinGPIO(GPIO_TypeDef *gpio, uint32_t bitnum, eIoPortModes mode)
```

- Typ si připravíme jako typedef enum
  - Typedef vytváří nový typ – zde **eIoPortModes**
  - Enum (výčet hodnot) reprezentuje pojmenované konstanty
    - Interně je to celočíselný typ
    - Čísluje se od 0 (není-li dáno jinak) a další je +1

```
typedef enum {
  ioPortOutputPP,        // vystup typu Push-Pull
  ioPortOutputOC,        // vystup typu Open Collector
  ioPortAnalog,          // analogovy vstup - pro prevodnik
  ioPortInputFloat,      // vstup bez pull-up/down
  ioPortInputPU,         // vstup s pull-up
  ioPortInputPD,         // vstup s pull-down
  ioPortAlternatePP,     // alternativni vystup - push/pull
  ioPortAlternateOC      // alternativni vystup - open drain
} eIoPortModes;
```

Funkce nastavení

- Podle portu povolí hodiny do GPIOx pokud nebyly a udělá RESET
  - Výběr pomocí switch-case konstrukce
  - Naštěstí jsou všechny GPIO na sběrnici AHB1
    - Stejný ENR a RSTR registr
- Uvedený kód hlásí chybu
  - GPIOx je typu ukazatel a nemůže být v switch-case
  - Naštěstí ukazatel=adresa a to je v principu celočíselná hodnota
    - Adresový prostor je 32-bitový
- Nutno přetypovat hodnoty v switch-case na uint32_t

```
bool Nucleo_SetPinGPIO(GPIO_TypeDef *gpio, uint32_t bitnum, eIoPortModes mode)
{
  uint32_t enr_mask = 0;      // hodnota do xxENR registru
  uint32_t rstr_mask = 0;     // hodnota do xxRSTR registru

  switch(gpio)                // detekce, ktery GPIO
  {
    case GPIOA:
      enr_mask = RCC_AHB1ENR_GPIOAEN;
      rstr_mask = RCC_AHB1RSTR_GPIOARST;
      break;
    case GPIOB:
      enr_mask = RCC_AHB1ENR_GPIOBEN;
      rstr_mask = RCC_AHB1RSTR_GPIOBRST;
      break;
    ...
    ... az do GPIOH
  }

  if ((enr_mask == 0) || (rstr_mask == 0))    // nevybran GPIO
    return false;                             // vrat priznak chyby

  if (!(RCC->AHB1ENR & enr_mask))   // inicializace vybraneho
  {
    RCC->AHB1ENR |= enr_mask;       // povolit hodiny periferie
    RCC->AHB1RSTR |= rstr_mask;     // proved reset periferie
    RCC->AHB1RSTR &= ~rstr_mask;    // a konec resetu
  }
...
```

```
66 bool Nucleo_SetPinGPIO(GPIO_TypeDef *gpio, uint32_t bitnum, eIoPortModes mode)
67 {
68   // detekce, ktery GPIO
69   switch(gpio)
70   {
71     case GPIOA:
72       break;
73   }
```

Problems | Tasks | Console | Properties

4 errors, 1 warning, 1 other

Description	Resource	Path	Location	Type
Errors (4 items)				
'true' undeclared (first use in this function)	main.c	/dom2/src	line 78	C/C++ Pro...
pointers are not permitted as case values	main.c	/dom2/src	line 71	C/C++ Pro...
switch quantity not an integer	main.c	/dom2/src	line 69	C/C++ Pro...
unknown type name 'bool'	main.c	/dom2/src	line 66	C/C++ Pro...

Funkce nastavení

- Přetypování vyřeší potřebu celočíselné hodnoty ve switch-case
  - Ideálně unsigned 32-bitová hodnota

```
  switch((uint32_t)gpio) // detekce, ktery GPIO
  {
    case (uint32_t)GPIOA:
      enr_mask = RCC_AHB1ENR_GPIOAEN;
      rstr_mask = RCC_AHB1RSTR_GPIOARST;
      break;
    case (uint32_t)GPIOB:
...
```

- Další chyba se projeví u GPIOF a GPIOG
  - Řada STM32F4xx je může mít všechny, ale "náš" F411RE je nemá
  - Aby byl kód univerzální, detekujeme, jaká je cílová platforma
  - Symbol STM32F411xE je definován v projektu při vytváření
    - Pro jiné procesory by bylo možno přidat další podmínky pro jiné procesory F4xx

```
...
#if defined(STM32F411xE) // 411RE nema GPIOF a G
#else
    case (uint32_t)GPIOF:
      enr_mask = RCC_AHB1ENR_GPIOFEN;
      rstr_mask = RCC_AHB1RSTR_GPIOFRST;
      break;
    case (uint32_t)GPIOG:
      enr_mask = RCC_AHB1ENR_GPIOGEN;
      rstr_mask = RCC_AHB1RSTR_GPIOGRST;
      break;
#endif
...
```

Funkce nastavení

```
// nastaveni konfiguracnich bitu do defaultniho stavu (nemususelo byt od drive)
gpio->MODER &= ~(0x03 << (2 * bitnum));   // vynuluj prislusne 2 bity v registru
gpio->PUPDR &= ~(0x03 << (2 * bitnum));   // vynuluj prislusne 2 bity v registru
gpio->OSPEEDR &= ~(0x03 << (2 * bitnum));   // vynuluj prislusne 2 bity v registru
```

- V další části je potřeba nastavit bity v řídích registrech do výchozí hodnoty
  - I/O mohl být nastaven z dřívějšího programu
- Podle požadovaného režimu se nastaví bity v registrech
  - Různě mají 1- a 2-bitový význam
  - Zohlednit v posuvech
- Nutno respektovat zachování ostatních bitů
  - Bity posuvem
  - Maskování AND a bitovou negací pro získání log. 0 na pozici bitu(ů)
  - Maskování OR a log. 1

```
  // nastaveni registru podle typu vystupu/vstupu
  switch(mode)
  {
    case ioPortOutputOC:
    case ioPortOutputPP:
      gpio->MODER |= 0x01 << (2 * bitnum);    // 01 = output
      gpio->OSPEEDR |= 0x03 << (2 * bitnum);  // 11 = high speed
      gpio->PUPDR &= ~(0x03 << (2 * bitnum)); // 00 = no pu/pd

      if (mode == ioPortOutputOC)         // open collector (drain) ?
        gpio->OTYPER |= 0x01 << bitnum;   // 1 = OC/Open drain
      else
        gpio->OTYPER &= ~(0x01 << bitnum);    // 0 = push-pull
      break;
    case ioPortInputPU:                           // moder bits 00 = input
      gpio->PUPDR &= ~(0x03 << (2 * bitnum));     // clear bits
      gpio->PUPDR |= 0x01 << (2 * bitnum);        // 01 = pull-up
      break;
    case ioPortInputPD:
      gpio->PUPDR &= ~(0x03 << (2 * bitnum));     // clear bits
      gpio->PUPDR |= 0x02 << (2 * bitnum);        // 10 = pull-down
      break;
    case ioPortInputFloat:     // 00 = input mode, nothing to do
      gpio->PUPDR &= ~(0x03 << (2 * bitnum)); // 00 = no pull-up/dn
      break;
    case ioPortAnalog:    // 11 - analog mode
      gpio->MODER |= 0x03 << (2 * bitnum);    // set bits
      break;
    default:          // neznamy rezim ?
      return false;   // priznak chyby
  }

  return true;      // priznak OK
}
```

Funkce nastavení - finále

```
case ioPortAlternatePP:
case ioPortAlternateOC:
  gpio->MODER |= 0x02 << (2 * bitnum);    // set bits

  if (mode == ioPortAlternateOC)
    gpio->OTYPER |= 0x01 << bitnum;      // 1 = open-drain
  else
    gpio->OTYPER &= ~(0x01 << bitnum);   // 0 = push-pull

  gpio->OSPEEDR |= 0x03 << (2 * bitnum);  // high-speed = 11
  gpio->PUPDR &= ~(0x03 << (2 * bitnum)); // 00 = no pull-up/pull-down
                                  // don't forget set AFR registers !!!
  break;
```

- Ve switch-case chybí volba pro alternativní funkce (**AF**)
  - Doplníme již nyní i když je budeme potřebovat až později
  - Nastavení typu AF funkce zatím neřešíme

- Sekvence v nastavovací funkci je následující
  1. Vyber bit pro ENR a RSTR registr podle GPIOx
  2. Nevybráno = konec s chybou
  3. Test povolených hodin (z AHB1)
     - Pokud nejsou povol a udělej reset GPIOx
  4. Uveď bity řídích registrů do výchozího stavu
  5. Podle typu I/O nastav
     - MODER podle In/Out/Analog/AF
     - OSPEEDR a OTYPER pro výstupy
     - PUPDR – pull-up a pull-down odpory
  6. Pokud je vše OK, vrať true jako příznak "v pořádku"

Ověření funkcí práce s GPIO

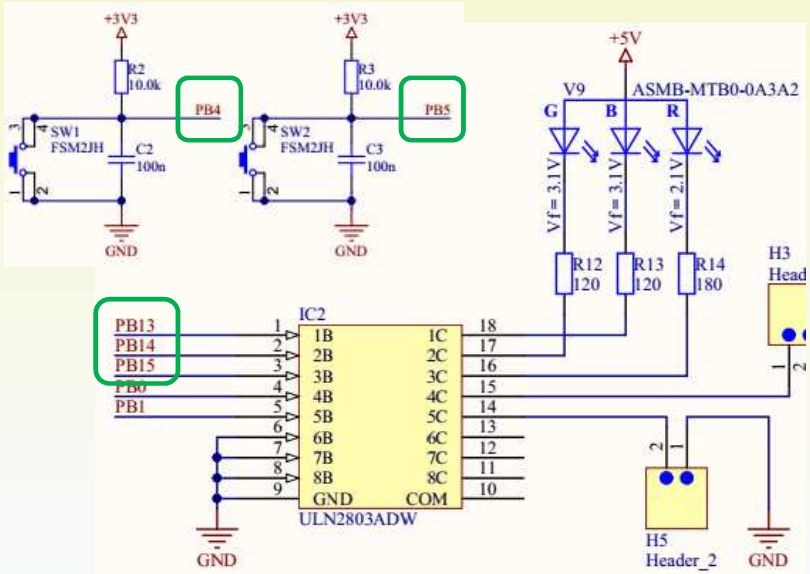
- Všechny funkce jsou před main()
- Program počítadlo 8xLED doplníme o blikání RGB LED
  - Nastavit na výstup
  - Toggle

```
int main(void)
{
  int i = 0;
  uint8_t b = 0;

  /* TODO - Add your application code here */
  Init8LED();

//  Nucleo_SetPinGPIO(GPIOB, 13, ioPortOutputPP);     // RGB-Blue
  Nucleo_SetPinGPIO(GPIOB, 14, ioPortOutputPP);     // RGB-Green

  /* Infinite loop */
  while (1)
  {
    b++;    // obycejne pocitadlo
    Write8LED(b);

//    GPIOToggle(GPIOB, 13);
    GPIOToggle(GPIOB, 14);

    for(i = 0; i < 200000; i++)
      ;
  }
}
```

- Pro jednoduchost můžeme doplnit definici "**port,pin**" pro připojené komponenty
  - Použití je pak mnohem intuitivnější

```
#define RGB_BLUE    GPIOB,13
#define RGB_GREEN   GPIOB,14
#define RGB_RED     GPIOB,15

#define BTN_LEFT    GPIOB, 4
#define BTN_RIGHT   GPIOB, 5
```

```
//  Nucleo_SetPinGPIO(RGB_BLUE, ioPortOutputPP);
Nucleo_SetPinGPIO(RGB_GREEN, ioPortOutputPP);
...
GPIOToggle(RGB_GREEN);
...
```

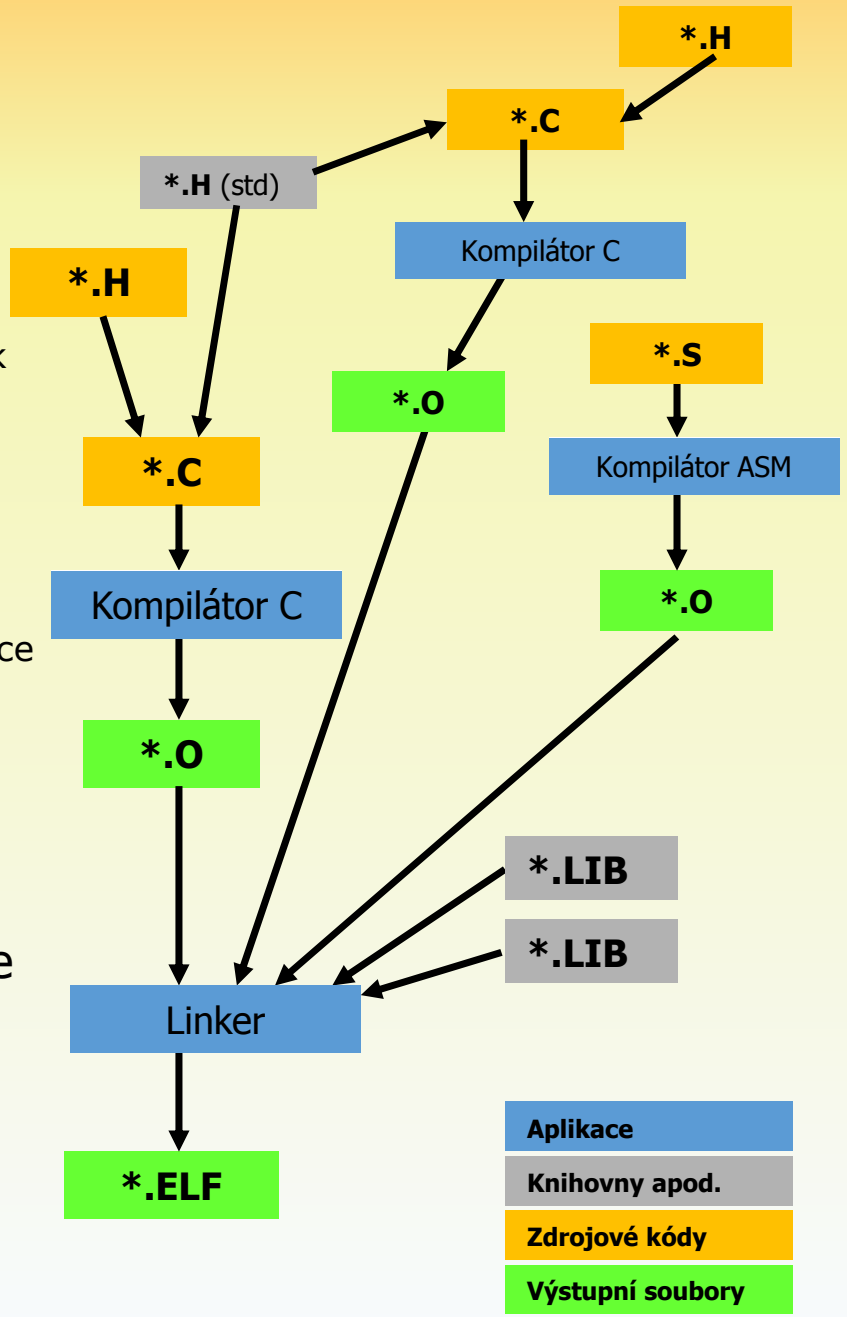


Rozdělení funkcí do modulů

- Projekty v C-čku jsou typicky rozděleny do více souborů
  - Zdrojové soubory
    - Přípona .C (nebo .CPP pro C++)
    - Obsahují výkonný kód
  - Hlavičkové soubory
    - Přípona .H
    - Obsahují deklarace (hlavičky) funkcí, definice symbolů (#define)
    - Typicky jsou připojeny (#include) do *.C souborů (nebo do více)
  - Soubory asembleru
    - Přípona .S (nebo .ASM)
    - Obsahují typicky speciální části kódu, které:
      - Se mají vykonat před C-čkovým kódem (tzv. startup)
      - Využívají speciální instrukce procesoru, které překladač C neumí díky své univerzálnosti použít
- Jednotlivé .C (.CPP) se kompilují samostatně pomocí překladače C
- Jednotlivé .S (.ASM) se kompilují samostatně pomocí "asembleru"
- Výsledné přeložené bloky se "slinkují" do výsledného souboru ještě typicky spolu s knihovnami, které dodává buď výrobce překladače nebo jiný dodavatel
- Uvedený systém je použit jak pro PC aplikace, tak pro mikrokontroléry
  - Liší se samozřejmě formát programu a překladač pro danou platformu

Proces zpracování modulů

- Rozdělení do modulů má výhody
  - Každý modul se překládá zvlášť
    - V případě změny se přeloží jen dotčený modul a výsledek se linkuje s nezměněnými
    - Je možné používat modul ve více projekt jednoduše zkopírováním souboru nebo odkazem na sdílený adresář
  - Je možné dodávat zkompilované knihovny bez zdrojových kódů
    - Stačí poskytnout *.H soubor s popisem funkcí
    - Minimálně se s tím setkáme u standardní knihovny (funkce z stdio, stdlib atd.)
- Pokud dojde k chybě při "akci", je vygenerována chyba
  - U kompilace i s číslem řádku v souboru
  - U linkování se řeší jen symboly
- Výsledek linkování je pro ARM mikrokontroléry ve formátu ELF
- Navíc se vytváří pomocné informační soubory
  - Mapa využití datové i kódové paměti .MAP
  - Seznam symbolů .SU a .LIST
  - Případně další



Rozdělení "našeho" projektu

- V main zůstává jen funkce main a #include hlaviček s popisem funkcí v ostatních modulech
- Využíváme soubory přidané při vytvoření projektu
  - **startup_stm32f411xe.s** – kód v asembleru
    - Vykoná se před spuštěním main
    - Nastaví procesor do potřebného výchozího stavu
    - Využijeme ještě při práce s přerušením
  - **system_stm32f4xx.c** – speciální funkce pro práci s jádrem procesoru – nastavení/detekce hodin
- Naše výkonné funkce rozdělíme do 2 modulů
  - **stm_core.c** – funkce týkající se bloků přímo v procesoru
  - **mpp_shield.c** – funkce speciálně jen pro "MPP Shield"
    - Logicky využívají funkcionalitu v stm_core
- Každý .C kód má "svůj" .H soubor

Přidání souboru

- Vývojové prostředí založené na Eclipse je souborově a adresářově orientované, takže stačí do adresáře src v projektu nakopírovat .C nebo .H soubor(y) a při příštím "build" se zpracují
- Přidat soubor přímo v prostředí
  - Menu File/New ... – volba "Source File" a "Header File"
  - Pravý klik na adresáři src v projektu, volba "New ..." a "Source File", "Header File"
  - Základní kostra .H souboru obsahuje ochranu před vícenásobným vložením
    - Pokud není definován jednoznačný symbol
    - Symbol je nadefinován
    - Před #endif je kód, který se má překládat právě jednou
    - Pokud se někde použije tento .H vícenásobně, bude symbol podruhé a dále již definován a obsah se nevykoná

```
/* ... */

#ifndef MPP_SHIELD_H_
#define MPP_SHIELD_H_

#endif /* MPP_SHIELD_H_ */
```

```c
#ifndef MPP_SHIELD_H_
#define MPP_SHIELD_H_

#include "stm_core.h"

void Init8LED(void);
void Write8LED(uint8_t val);

#define RGB_BLUE    GPIOB,13
#define RGB_GREEN   GPIOB,14
#define RGB_RED     GPIOB,15

#define BTN_LEFT    GPIOB, 5
#define BTN_RIGHT   GPIOB, 4

#endif /* MPP_SHIELD_H_ */
```

```c
#ifndef STM_CORE_H_
#define STM_CORE_H_

#include "stm32f4xx.h"
#include <stdbool.h>

typedef enum {
  ioPortOutputPP,       // vystup typu Push-Pull
  ...
} eIoPortModes;

bool Nucleo_SetPinGPIO(GPIO_TypeDef *gpio, uint32_t bitnum, eIoPortModes
mode);
void GPIOToggle(GPIO_TypeDef *gpio, uint32_t bitnum);
bool GPIORead(GPIO_TypeDef *gpio, uint32_t bitnum);
void GPIOWrite(GPIO_TypeDef *gpio, uint32_t bitnum, bool state);

#define BOARD_BTN_BLUE  GPIOC,13
#define BOARD_LED       GPIOA,5

#endif /* STM_CORE_H_ */
```

```c
#include "mpp_shield.h"

void Init8LED(void)
{
  ...
}

void Write8LED(uint8_t val)
{
  ...
}
```

```c
#include "stm_core.h"

bool Nucleo_SetPinGPIO(GPIO_TypeDef *gpio, uint32_t bitnum, eIoPortModes mode)
{
  ...
}

void GPIOToggle(GPIO_TypeDef *gpio, uint32_t bitnum)
{  ...  }

bool GPIORead(GPIO_TypeDef *gpio, uint32_t bitnum)
{  ...  }

void GPIOWrite(GPIO_TypeDef *gpio, uint32_t bitnum, bool state)
{  ...  }
```

Poznámky k rozdělení na moduly

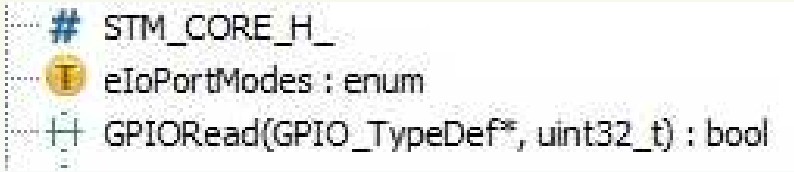
- Kopírovat soubory mezi projekty lze snadno pomocí Drag-Drop myší (se stisknutým CTRL se objeví "+" = kopírování)

- Použití #include se zjednoduší
  - V souboru main.c stačí připojit #include "mpp_shield.h"
  - V souboru mpp_shield.c připojit #include "mpp_shield.h"
  - V souboru mpp_shield.h je třeba připojit #include "stm_core.h"
  - V souboru stm_core.c připojit #include "stm_core.h"
  - Až v stm_core.h jsou klíčové připojení
    - #include "stm32f4xx.h" – definice HW symbolů apod.
    - #include <stdbool.h> - budeme používat bool typů, zde uvedeny

Zobrazení v Project Explorer

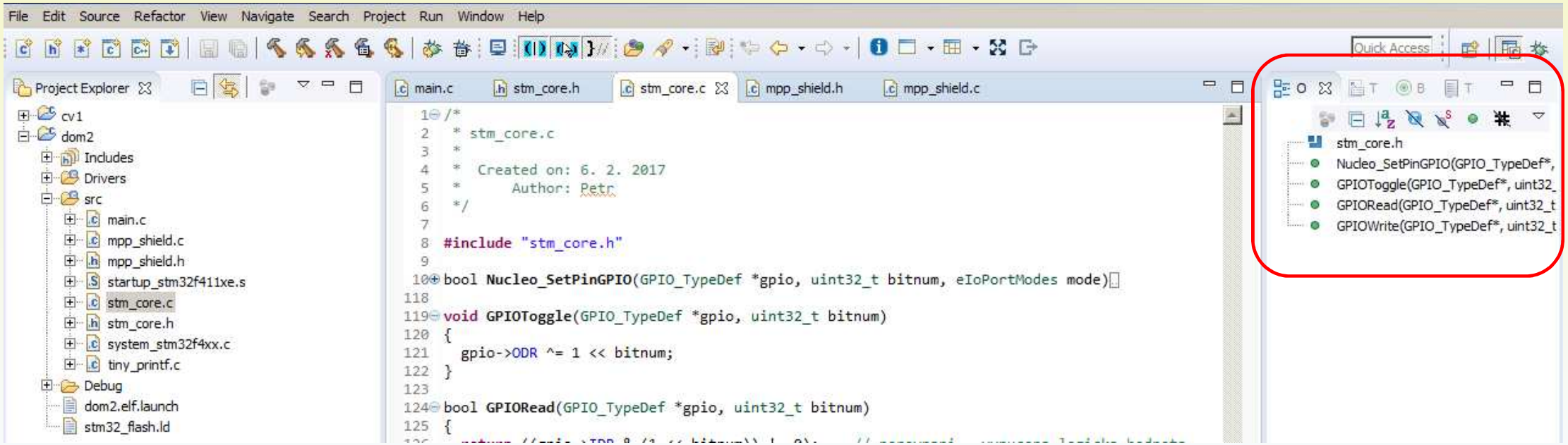
- Zdrojové soubory *.c lze "rozkliknout" a ukáže
  - "includované" (nejen) .H soubory
  - Veřejné symboly = funkce

- "Rozkliknutí" *.h souboru ukáže
  - Deklarované funkce
  - Definované symboly
  - Výčty (enum)



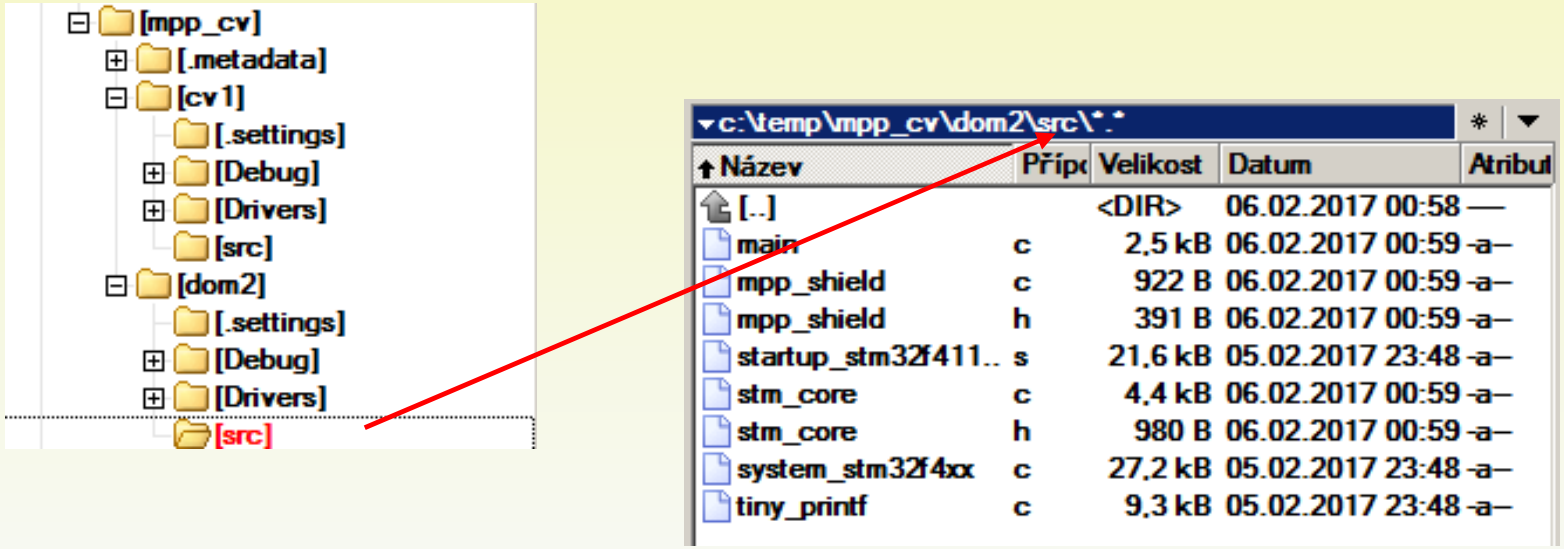
Okno "Outline"

- Jedno z "pravých" oken
- Ukazuje strukturu aktuálního dokumentu
  - Lze upravit třídění
  - Kliknutím na symbol se vybere ve "zdrojáku"
- Pokud chybí – zapnout v menu Window/Show View/Outline



Finální stav výsledného projektu

- Viz. YT???
- Všechno funguje podle předvedení
- Soubory v adresáři src



Shrnutí domácí přípravy - II

- Umíme nastavit všechny základní režimy GPIO
  - Máme připravenou univerzální funkci pro korektní nastavení jednoho IO pinu
- Máme připravenou funkci pro nastavení výstupu a čtení vstupu
- Máme připravenou funkci pro nastavení GPIO pro 8xLED
  - Připojené na PC0-7
  - Nastavují se najednou
- Máme připravené makro pro nastavení hodnoty PC0-7

- Funkce a makra máme v samostatných souborech (názvy podle ukázky)
  - Pro procesor obecně v **stm_core.c** a **stm_core.h**
  - Pro "náš" KAE/MPP shield v **mpp_shield.c** a **mpp_shield.h**
- Umíme připojit do svého main.c pomocí #include
- Máme nadefinovaná makra pro RGB LED a tlačítka

Plán domácí přípravy

I. Instalace prostředí, ... První program – blikání LED na GPIO

II. Zobecnění GPIO operací

III. Systick

- **SysTick jako periférie jádra ARM**
- **Zjištění rychlosti hodin jádra - SystemCoreClock**
- **Spuštění SysTick, nastavení registrů**
- **Testování příznaku přetečení**
- **Použití přerušení pro generování příznaku**
- **Použití přerušení pro počítání "tiků" a různé možnosti práce s touto hodnotou**

IV. Timer

V. UART

VI. Rezerva

VII. SPI

VIII. PWM

IX. A/D

X. I2C (využití "knihovny")

XI. Časování – HSI/HSE/PLL

XII. Rezerva

Cíle přípravy

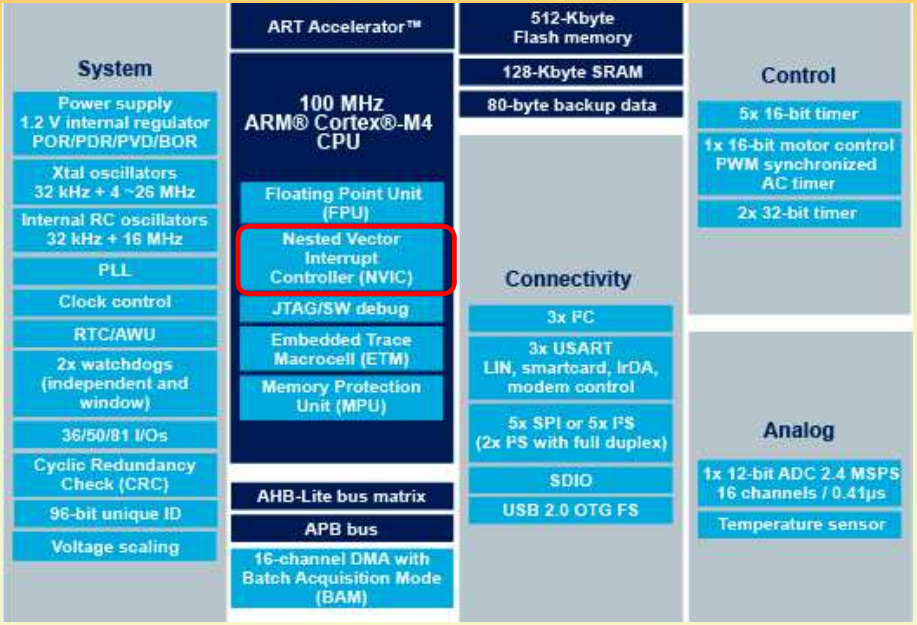
- Prozkoumat SysTick – časovač jádra (core)
  - Činnost
  - Řídící registry a bity
  - Výpočet hodnoty pro reload čítače
- Testování "dopočítání" vlastním kódem (polling)
- Použití přerušení
  - Pro nastavení příznaku
  - Pro počítání taktu v msec přes globální proměnnou
- Využití systémových funkcí (z CMSIS knihovny)
- Zobrazení aktuálního nastavení systémových hodin

Kopírování projektů a souborů mezi projekty

- Projekt lze klasicky kopírovat
  - Pravým klikem na projektu – Copy, pravým ve volném místě "Project Explorer" – Paste
  - Automaticky se pokusí navrhnout nové jméno projektu
  - POZOR, nejde kopírovat ze "zavřeného projektu"
- Pokud máme otevřeno více projektů (ve Workspace), je možné kopírovat soubory (typicky z SRC adresáře)
  - Přetáhnutím myší se stisknutým CTRL (u šipky je "plus")
  - Klasicky CTRL+C, CTRL+V
  - Pravým klikem na souboru – Copy, následně na cílovém adresáři – Paste

- !!! obrázek

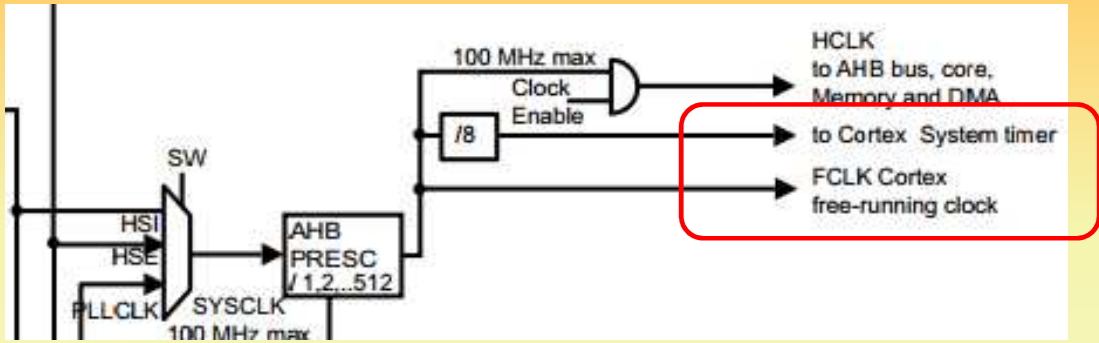
Využití HW bloků – např. pro časování



- V procesoru "na křemíku" je:
  - Samotné výpočetní jádro
    - Společné pro všechny Cortex-M4 procesory
    - Nejen v STM, ale i NXP, Nordic, ...
  - Velké množství bloků pro speciální účely
    - **Už jsme použili GPIO**

- Pro obecné časování je možné použít Systick
  - Tento blok mají všechny Cortex-M procesory (nejen "naše" M4 !!)
  - Má jednoduché použití
- Pro realizaci obsluhy "uplynutí času" se nejlépe využije přerušení
  - = nezávislá (asynchronní) událost, která nastává při splnění HW podmínek bez ohledu na právě vykonávaný kód
  - Obsluha je typicky realizována jako speciální funkce
- Nastavení a řízení přerušení realizuje blok NVIC
  - = Nested vectored interrupt controller
- POZOR – jak SysTick, tak NVIC jsou součástí jádra a proto jej jejich popis v příslušném PM (= Program Manual)
  - Dokument je společný pro řadu STM32F4, F3 i F0

Hodiny pro SysTick



- SysTick jako blok v jádru má přímé připojení k systémovému zdroji hodin
  - Viz. RM Fig. 12, kap. 6.2 – výřez zde
  - SysTick se často skrývá pod jinými názvy ("System Timer", "free-running clock" apod.)
  - Hodiny samotné se nemusí "povolovat", povoluje se jen činnost SysTicku v jeho registru CTRL

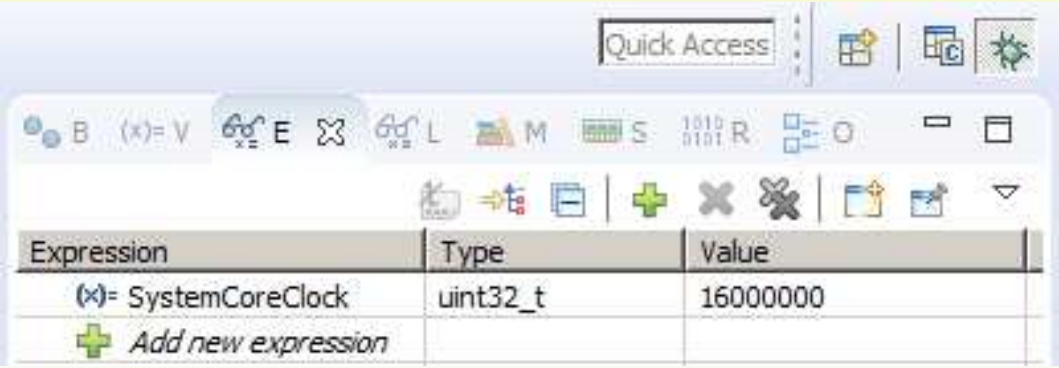
- Pro určení hodnot dělících poměrů je nutno zjistit CoreClock
  - "Náš" F411 má základní SYSCLK max. 100MHz
  - Je vydělen děličkou AHB sběrnice, většinou se používá 1:1
    - Blok AHB Presc (Prescaler = předdělička)

SystemCoreClock

- V rámci knihoven pro ARM Cortex-M je k dispozici řada informací usnadňujících práci s Core (jádrem)
  - Součást **CMSIS** knihoven (= Cortex Microcontroller Software Interface Standard)
  - Některé k nalezení v souboru **system_stm32f4xx.c**
- Pro určení hodin jádra existuje globální proměnná **SystemCoreClock**
  - Je nastavena na počátku programu před main()
    - POZOR, jedná se o defaultní hodnotu, nemusí odpovídat reálnému stavu
  - Po změně nastavení hodin je nutno aktualizovat
  - Doporučeno aktualizovat na počátku v kódu main()
- Funkce **SystemCoreClockUpdate()** přepočítá hodnotu proměnné **SystemCoreClock** podle aktuálního nastavení děliček hodinového signálu

Zjištění SystemCoreClock při Debug

- V "čistém" projektu spustit Debug
- Nastavit breakpoint na řádek while-cyklu
  - Menu Run/Toggle BreakPoint
  - Nebo 2x levý klik vlevo od čísla řádku
  - Nebo pravý klik a Toggle BreaPoint
- Po zastavení programu obsahuje okno Variables proměnné
  - Kromě externích proměnných
- Okno Expressions umožňuje zobrazení jakéhokoliv výrazu, tj. i obsah globální (externí) proměnné
  - Zvolit Add new expression
  - Napsat **SystemCoreClock**
  - Ukáže aktuální hodnotu proměnné



- Mělo by být vidět 16000000 = 16MHz
  - Dole pod seznamem ukazuje i jiném zobrazení
  - Jedná se o defaultní hodnotu

SysTick = "počítadlo"

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

- Interně je SysTick realizován jako 24-bitový registr čítající dolů (count-down)
  - Při přechodu z 1 do 0 se nastavuje příznak dopočítání
  - Při přechodu z 0 se načte obsah RELOAD registru do čítače a následuje opět počítání dolů
  - Největší hodnota je 2^{24}-1, tj. 16 777 215
- Viz. PM 4.5.2
  - Počítá se včetně hodnoty 0, proto se musí naplnit hodnotou N-1, tj. pro 100 taktů RELOAD = 99

The RELOAD value can be any value in the range 0x00000001-0x00FFFFFF. A start value of 0 is possible, but has no effect because the SysTick exception request and COUNTFLAG are activated when counting from 1 to 0.

The RELOAD value is calculated according to its use:

- To generate a multi-shot timer with a period of N processor clock cycles, use a RELOAD value of N-1. For example, if the SysTick interrupt is required every 100 clock pulses, set RELOAD to 99.
- To deliver a single SysTick interrupt after a delay of N processor clock cycles, use a RELOAD of value N. For example, if a SysTick interrupt is required after 100 clock pulses, set RELOAD to 99.

SysTick - řízení

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															COUNT FLAG
															rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved													CLKSO URCE	TICK INT	EN ABLE
													rw	rw	rw

- Řídící registr popsán v PM 4.5.1 STK_CTRL
- COUNTFLAG – příznak dopočítání
  - Nastaven na 1 pokud od posledního čtení dosáhl čítač 0
- CLKSOURCE – zdroj hodin
  - Může být buď přímo AHB (log. 1) nebo AHB/8 (log. 0)
- TICKINT – povolení generování přerušení
- ENABLE – povolení čítání SysTick čítače
  - Při nastavení log. 1 se přesune RELOAD do čítače a spustí se

- Není potřeba žádné další nastavování = jednoduché využití

SysTick testovaný programově

- Využijeme minulé příklady – kód pro nastavení GPIO (LED)
- Požadavek přetékání každou 1 sec
  - Při (známém) taktu 16M je nutno nastavit **RELOAD** = 16000000 – 1;
- Registry přístupné přes strukturu
  - **SysTick** je (SysTick_Type *)
- Řídící bity jsou definovány v *.h
  - Zde v core_cm4.h
  - Symbolicky s příponou ...Msk
  - Možno rovnou OR-ovat
- Nastavení Enable a Source AHB
- Prázdný while testuje dopočítání
  - Test bitu Countflag
  - "Čekej dokud NENÍ nastaven"

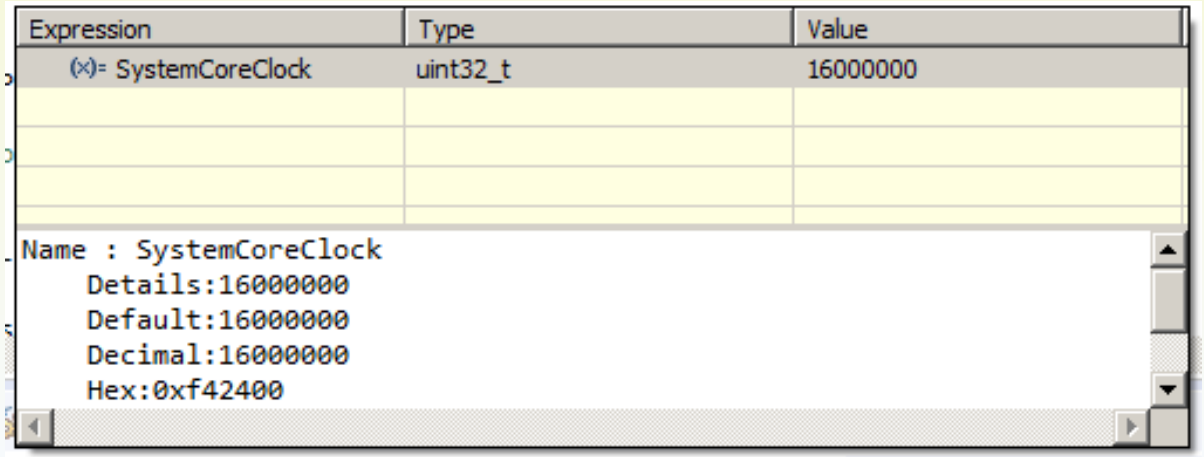
```
#include <mpp_shield.h>
...
int main(void)
{
  ...
  SysTick->LOAD = 1600000 - 1;
  SysTick->CTRL |=
    SysTick_CTRL_ENABLE_Msk | SysTick_CTRL_CLKSOURCE_Msk;

  Nucleo_SetPinGPIO(RGB_BLUE, ioPortOutputPP);

  while (1)
  {
    GPIOToggle(RGB_BLUE);

    while(!(SysTick->CTRL & SysTick_CTRL_COUNTFLAG_Msk))
      ;
  }
}
```

SysTick testovaný programově – II

- Lze využít zjištěného SystemCoreClock
  - Na začátku main doporučeno aktualizovat podle nastavení hodin, příp. po programové změně hodin
  - Změníme na 1/10 sec
    - Zde tedy 1 600 000 – 1

```
...
int main(void)
{
  SystemCoreClockUpdate();

  SysTick->LOAD = SystemCoreClock / 10 - 1;
...
```

- Je možné se při debug nebo krokování podívat na obsah proměnné
  - Když je vykonávání zastaveno, najet myší nad proměnnou = ukáže typ a obsah v různých formátech



Přerušení – zjednodušený úvod + NVIC

- Zjednodušeně - přerušení jsou akce/události, které vznikají mimo běh hlavního programu
  - Typicky splněním HW podmínek – např. dopočítání SysTick – viz. další popis
- HW blok procesoru nastaví požadavek přerušení
  - Na úrovni periférie lze vygenerování požadavku povolit (TICKINT) nebo testovat "nějaký" příznak programově (COUNTFLAG)

- Součástí jádra (core) procesoru je řadič přerušení NVIC
  - = Nested vectored interrupt controller
  - Pokud je povoleno zdroj přerušení v NVIC:
    1. Dokončí se vykonávaná instrukce programu
    2. Načte se adresa obslužného kódu podle typu události
    3. Vykoná se kód počínaje touto adresou
    4. Po skončení (= instrukce "návratu") se vykonávání vrátí do původně vykonávaného místa
  - NVIC řeší i priority = co dělat pokud přijde přerušení během obsluhy jiného přerušení
- Tj. je nutno nastavit přerušení v periférii i NVICu a adresu "obslužné" funkce
  - Nastavení adresy zajistí typicky překladač

Nastavení funkce obsluhy přerušení

- V C-čkových překladačích pro ARM je to typicky funkce bez parametrů a bez návratové hodnoty
  - Formálně void jmeno(void)
- Pokud má funkce "správné" jméno, použije překladač její adresu jako obsluhu přerušení
  - Adresy jsou umístěny v tabulce vektorů přerušení
  - Pořadí v tabulce určuje, kterému přerušení adresa odpovídá
- V použitém překladači (založený na GCC) je tabulka vektorů přerušení naplněna adresami defaultních funkcí s příznakem .weak
  - Najdeme v kódu asembleru zavaděče startup_stm32f411xe.s
  - Tabulka začíná symbolem g_pfnVectors na řádku 142
  - Linker vyřeší použití "naší" funkce stejného jména, jako je u symbolu s příznakem .weak, jinak vezme defaultní

Přerušení od SysTick

```
#include <mpp_shield.h>

void SysTick_Handler(void)
{
  GPIOToggle(RGB_BLUE);
}

int main(void)
{
  SystemCoreClockUpdate();

  SysTick_Config(SystemCoreClock / 10);

  Nucleo_SetPinGPIO(RGB_BLUE, ioPortOutputPP);

  while (1)
  {
  }
}
```

- Funkce se MUSÍ jmenovat **SysTick_Handler**
  - Není potřeba další "servisní" akce
    - SysTick příznak se nuluje "automaticky"

- V knihovně CMSIS existuje šikovná funkce **SysTick_Config**, která
  - Očekává počet tiků, které má SysTick napočítat (dá do RELOAD číslo -1)
  - Nastaví řídící bity v CTRL registru
  - Nastaví prioritu přerušení v NVIC
  - Protože je vše součástí jádra, postačuje to

- **POZOR – nikde neexistuje kontrola, zda je správně vytvořena obsluha přerušení (= správný název funkce) při povolení zpracování přerušení v NVIC**

SysTick jako časová základna

- SysTick je výhodné použít spíše jako zdroj časové informace nebo měření intervalu
  - Výkonná akce (zde blikání) je pak jinde v programu
  - V obsluze SysTick se jen nastaví příznak "uplynutí" času
  - Pozor, MUSÍ být **VOLATILE**
    - Proměnná je vždy v paměti
    - Není překladačem optimalizován přístup
    - Nutné pro data sdílená programem a přerušením
- V main čekám až čas uplyne
  - Pak "shodím" příznak
  - Jedná se o stejný blokující přístup jako první příklad

```
#include <mpp_shield.h>

volatile bool tickElapsed = false;

void SysTick_Handler(void)
{
  tickElapsed = true;
}

int main(void)
{
  SystemCoreClockUpdate();

  SysTick_Config(SystemCoreClock / 10);

  Nucleo_SetPinGPIO(RGB_BLUE, ioPortOutputPP);

  while (1)
  {
    GPIOToggle(RGB_BLUE);

    while(!tickElapsed)
      ;
    tickElapsed = false;
  }
}
```

- Lépe je jen testovat příznak a pokud je nastaven, "shodit" a vykonat intervalovou akci
  - Jinak lze v main dělat něco jiného
  - Takových příznaků může být více

```
...
  while (1)
  {
    if (tickElapsed)
    {
      tickElapsed = false;

      GPIOToggle(RGB_BLUE);
    }
  }
}
```

SysTick jako časová základna – II

- Optimální využití SysTicku je nastavit jej na např. 1ms a inkrementovat globální proměnnou
  - Obsahuje počet ms od spuštění (SysTicku)
  - Při použití uint32_t přetéká po $(2^{32} - 1)/1000$ sec = 4Msec = cca 49 dní
  - Opět MUSÍ být **volatile**
- Systick nastaven na 1 / 1000 sec
- V programu je potřeba mít vždy jednu proměnnou pro uschování časového okamžiku
  - Testujeme, zda již uplynulo o 500ms víc od posledního "časového vzorku" (timestamp)
- Alternativně je možno uložit čas "příští" události
- Takových intervalů může být několik – viz. YT ukázka

```
...
  while (1)
  {
    if (_ticks >= tm)
    {
      tm = _ticks + 500;
...
```

```
#include <mpp_shield.h>

volatile uint32_t _ticks = 0;

void SysTick_Handler(void)
{
  _ticks++;
}

int main(void)
{
  uint32_t tm = 0;

  SystemCoreClockUpdate();

  SysTick_Config(SystemCoreClock / 1000);

  Nucleo_SetPinGPIO(RGB_BLUE, ioPortOutputPP);

  while (1)
  {
    if ((_ticks - 500) >= tm)
    {
      tm = _ticks;

      GPIOToggle(RGB_BLUE);
    }
  }
}
```

Shrnutí domácí přípravy - III

- Umíme používat **SysTick**
  - Nejčastěji v režimu s přerušením
  - Každým "tikem" se inkrementuje počítadlo "tiků"
  - V aplikaci je možné ho použít jako časovou základnu
- Umíme se podívat v debuggeru na hodnotu **SystemCoreClock**
- Umíme hodnotu **SystemCoreClock** použít pro nastavení **SysTick**
- Umíme jednoho časového základu využít pro měření různých intervalů

Plán domácí přípravy

I. Instalace prostředí, ... První program – blikání LED na GPIO

II. Zobecnění GPIO operací

III. Systick

IV. Timer

- **Timer3 – jeden z univerzálních časovačů**
- **Timer – vybrané registry a význam bitů**
- **Nastavení časovače pro běh, testování příznaku přetečení/podtečení**
- **Využití přerušení – identifikace názvu obslužné funkce, povolení v NVICu**
- **Generování tónu řízením I/O v přerušení**

V. UART

VI. Rezerva

VII. SPI

VIII. PWM

IX. A/D

X. I2C (využití "knihovny")

XI. Časování – HSI/HSE/PLL

XII. Rezerva

Cíle přípravy

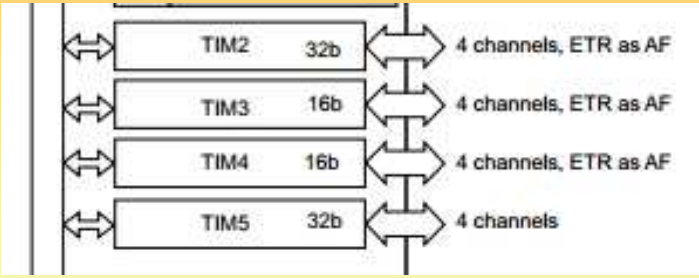
- Naučit se používat základní funkce bloku čítače/časovače
  - Nastavení registrů
  - Testování příznakových bitů
- Vytvořit obsluhu přerušení
  - Využít k časování blikání LED
  - Využít ke generování obdélníkového průběhu na piezo-"pípáku"
- Program bude využívat i funkcí z minulých cvičení
  - Např. SysTick a nastavení GPIO

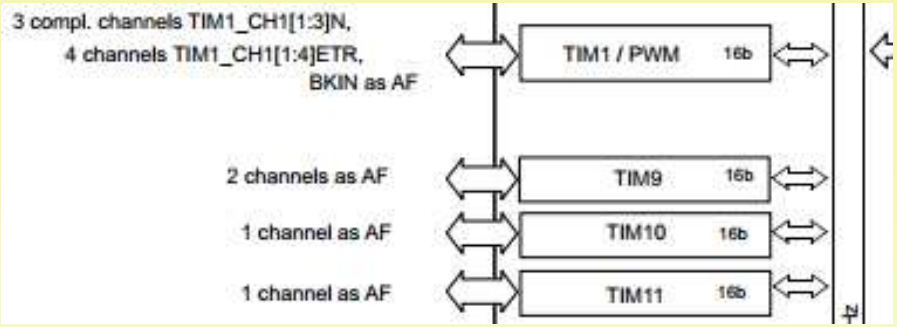
- Video – viz. https://youtu.be/xokHtahdLlY

Čítače/časovače – důležité periférie

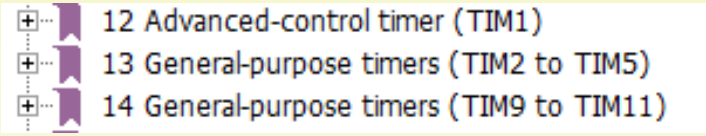
- Pojem čítač (=counter) a časovač (=timer) se často zaměňují
  - Základem je registr (hodnota), který umí přičítat (nebo odečítat) pulsy
  - Při přetečení (podtečení) typicky generuje událost
    - Možno testovat programově nebo využít přerušení
  - Časovač = pulsy jsou odvozeny od vnitřního zdroje hodin
  - Čítač = pulsy jsou nejčastěji z vnějšího vstupu
    - Např. možno měřit frekvenci jako počet pulsů za čas
- Čítačů je v mikrokontroléru typicky několik nezávislých a liší se:
  - Zdrojem hodin (= na kterou sběrnici je připojen)
  - "Délkou" čítacího registru – běžně 16 nebo 32 bitů
  - Možnostmi nastavení (nahoru/dolů, události, taktování DMA, ...)
- Na jeden čítač může být navázáno několik kanálů
  - Realizují funkčnost CAPture-COMpare (CAPCOM - viz. přednášky a cvičení PWM)
- Existují speciální čítače např. pro řízení DC motorů (bezpečné spínání H-můstků) apod.
- Některé čítače se mohou i řetězit = výstup jednoho je vstupem druhého

Čítače na STM32F411RE





- DS str. 15/149
  - Na APB1 – TIM2, 3, 4, 5
  - Na APB2 – TIM1, 9, 10, 11
- Dle RM dělení
  - Advanced – TIM1
  - "Běžné" – skupina 1 (TIM2-5) a skupina 2 (TIM9-11)



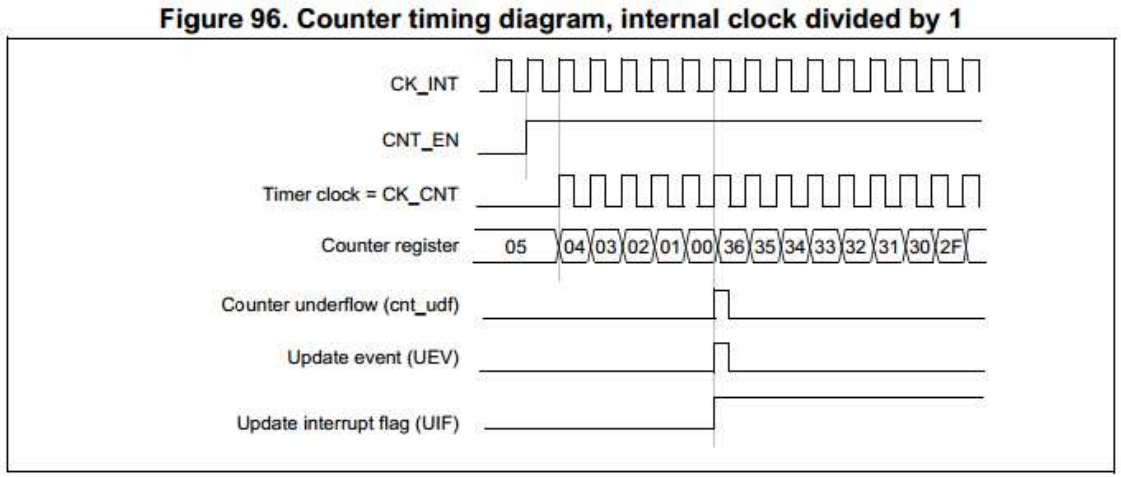
- Všechny čítače mají 3 základní registry:
  - PSC (=prescaler) - předdělič vstupních pulsů, pro TIM3 je 16b
    - RM 13.4.11
    - Počítá "včetně 0"

PSC[15:0]: Prescaler value

The counter clock frequency CK_CNT is equal to f_{CK_PSC} / (PSC[15:0] + 1).
PSC contains the value to be loaded in the active prescaler register at each update event.

  - ARR (= autoreload) – hodnota pro reload , TIM3 - 16b
    - Jinak by počítal plný rozsah (2^{16} nebo 2^{32})
  - CNT (= counter) – zde se přičítá/odečítá hodnota, TIM3 opět 16b

Průběh čítání čítače



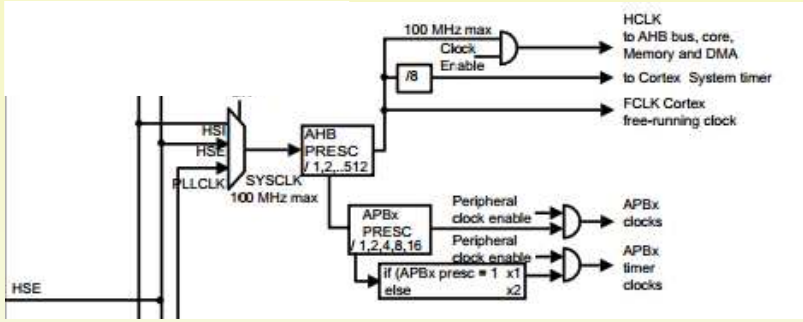
- Různé průběhy jsou ukázány v RM – kap. 13.1
  - Pro čítání nahoru i dolů jsou obdobné
- Čítač dolů čítá do 0 (včetně), pak je automaticky načten obsah ARR registru a počítá se opět dolů
  - Počet "čítnutí" je tedy ARR+1 (opět jakoby C-čkovsky včetně nuly)
  - Proto pro např. 100 tiků zadáme 99 (= (100-1))
  - Při překlopení 00->ARR se vygeneruje událost UEV a nastaví se příznak UIF
  - Viz. obr. 96, str. 313/837
- Komplexnější režimy zatím nevyužijeme

- Zdroj hodin určují bity SMS v registru SMCR
  - Defaultně 000 = internal clock

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ETP	ECE	ETPS[1:0]		ETF[3:0]				MSM	TS[2:0]			Res.	SMS[2:0]		
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw		rw	rw	rw

Určení taktu sběrnice APBx

- Základní takt SYSCLK je dělen AHB předděličem
  - Na něj navázáno přímo jádro (SysTick apod.)
  - Z AHB sběrnice předděličky pro obě APB sběrnice



- Vše má na starosti registr CFGR bloku RCC
  - RM 6.3.3 – str.103/837

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MCO2		MCO2 PRE[2:0]			MCO1 PRE[2:0]			I2SSCR	MCO1		RTCPRE[4:0]				
rw		rw	rw	rw	rw	rw	rw	rw	rw		rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PPRE2[2:0]			PPRE1[2:0]			Reserved		HPRE[3:0]				SWS1	SWS0	SW1	SW0
rw	rw	rw	rw	rw	rw			rw	rw	rw	rw	r	r	rw	rw

Bits 12:10 **PPRE1:** APB Low speed prescaler (APB1)

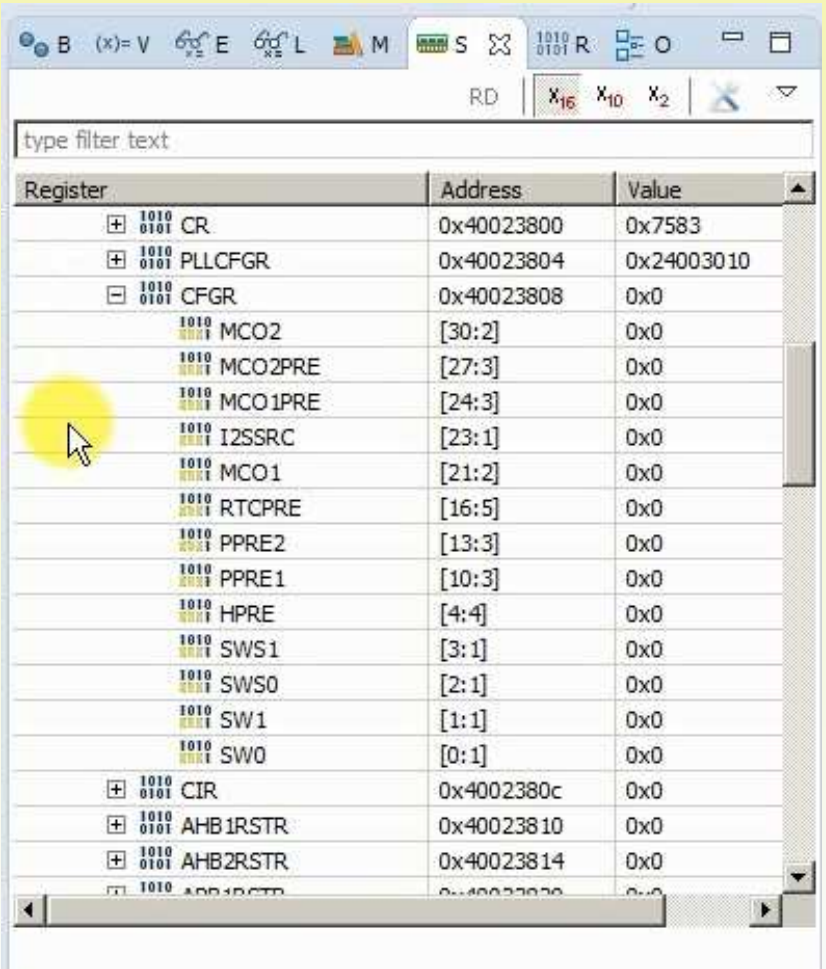
Set and cleared by software to control APB low-speed clock division factor.

Caution: The software has to set these bits correctly not to exceed 42 MHz on this domain. The clocks are divided with the new prescaler factor from 1 to 16 AHB cycles after PPRE1 write.

0xx: AHB clock not divided
100: AHB clock divided by 2
101: AHB clock divided by 4
110: AHB clock divided by 8
111: AHB clock divided by 16

Určení taktu sběrnice APBx – II

- Uvažujme defaultní nastavení procesoru
  - SystemCoreClock = 16M(Hz)
- V debug režimu zobrazíme okno (záložku) SFRs
  - = Special Funcion Registers
  - Pokud není, menu Window/Show View/SFRs
- Rozbalit blok RCC a registr CFGR
  - Je vidět hodnota PPRE1, PPRE2 apod.
  - Případně pohled na bity celého registru a z toho spočítat bitové složky (viz. RM)
- Hodnota 000 = not divided
  - Tedy takt APB1 = 16 000 000



Register	Address	Value
CR	0x40023800	0x7583
PLLCFGR	0x40023804	0x24003010
CFGR	0x40023808	0x0
MCO2	[30:2]	0x0
MCO2PRE	[27:3]	0x0
MCO1PRE	[24:3]	0x0
I2SSRC	[23:1]	0x0
MCO1	[21:2]	0x0
RTCPRE	[16:5]	0x0
PPRE2	[13:3]	0x0
PPRE1	[10:3]	0x0
HPRE	[4:4]	0x0
SWS1	[3:1]	0x0
SWS0	[2:1]	0x0
SW1	[1:1]	0x0
SW0	[0:1]	0x0
CIR	0x4002380c	0x0
AHB1RSTR	0x40023810	0x0
AHB2RSTR	0x40023814	0x0

Základní registry čítače TIM3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved						CKD[1:0]		ARPE	CMS		DIR	OPM	URS	UDIS	CEN
						rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

- **CR1** = Control 1
  - **DIR** – Direction – 0 = čítá vzhůru, 1 = čítá dolů
  - **OPM** – One Pulse Mode – 1 = po přetečení zastaví čítač
  - **URS** – Update Req. Src – 1 = pouze pře/podtečení, 0 = také "manuální" události (viz. registr EGR)
  - **UDIS** – Update Disable – 1 = blokuje generování událostí
  - **CEN** – Counter Enable – povoluje běh čítače
    - Některá nastavení lze provádět pouze při CEN = 0 (zastaveno)
    - Proto typicky nastaven jako poslední (OR maskou)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								TI1S	MMS[2:0]			CCDS	Reserved		
								rw	rw	rw	rw	rw			

- **CR2** = Control 2
  - Běžně stačí jen vynulovat

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	TDE	Res	CC4DE	CC3DE	CC2DE	CC1DE	UDE	Res.	TIE	Res	CC4IE	CC3IE	CC2IE	CC1IE	UIE
	rw		rw	rw	rw	rw	rw		rw		rw	rw	rw	rw	rw

- **DIER** = DMA a IRQ Control
  - **UIE** – Update Int. Enable – 1 = povoleno generování přerušení při události

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved			CC4OF	CC3OF	CC2OF	CC1OF	Reserved		TIF	Res	CC4IF	CC3IF	CC2IF	CC1IF	UIF
			rc_w0	rc_w0	rc_w0	rc_w0			rc_w0		rc_w0	rc_w0	rc_w0	rc_w0	rc_w0

- **SR** = Status
  - **UIF** – Update interrupt flag – příznak události (pře/podtečení apod.), nulování kódem

Testování příznaku přetečení TIM3

- Předpokládejme funkční kód z minula
  - Nastavení GPIO pro RGB-LED
  - Funkční SysTick generující 1ms "inkrementy"
  - Blikání zelené RGB po 200ms od SysTicku
- Nastavit TIM3
  - Povolit hodiny pro TIM3 v ENR registru ("klasicky") – sběrnice APB1
  - Nastavit směr čítání čítače (dolů)
  - Nastavit předdělič na 1000ms (=1kHz)
  - Nastavit ARR na 100 opakování
    - Hodnota 100-1 !!
  - Spustit čítač bitem CEN
- V hlavní smyčce testovat podtečení
  - Bit UIF v registru SR – test maskou
  - Shodit příznak (nutno ručně)
  - Provést akci (bude 1x za 100ms)

```
if (!(RCC->APB1ENR & RCC_APB1ENR_TIM3EN))
{
  RCC->APB1ENR |= RCC_APB1ENR_TIM3EN;
  RCC->APB1RSTR |= RCC_APB1RSTR_TIM3RST;
  RCC->APB1RSTR &= ~RCC_APB1RSTR_TIM3RST;
}

TIM3->CR1 = TIM_CR1_DIR; // DIR = 1 = cnt-down, ostatni v 0
TIM3->CR2 = 0;
TIM3->PSC = SystemCoreClock / 1000 - 1;  // 1ms = 1kHz
TIM3->ARR = 99;  // (100 - 1) - tj. 10Hz, 100ms

TIM3->CR1 |= TIM_CR1_CEN;   // CEN = 1 = enable
```

```
while(1)
...
  if (TIM3->SR & TIM_SR_UIF)  // nenula = nastaven bit
  {
    TIM3->SR &= ~TIM_SR_UIF;  // shodit priznak !!!
    GPIOToggle(RGB_BLUE);
  }
...
```

Využití přerušení od TIM3

- Využití přerušení od některého periferního bloku (zde TIMx) vyžaduje 3 kroky
  - Kód funkce obsluhující přerušení
    - Typicky začíná shozením příznaku z periférie – zde UIF v registru SR
    - Obsluha může být pro více událostí, nutno pak detekovat, která to byla
  - Povolení přerušení "z periférie"
    - Zde bit UIE v registru DIER
  - Povolení zpracování přerušení v jádře v bloku NVIC

- Základní princip přerušení
  - Pokud je dané přerušení povoleno
    - Pozastaví vykonávání kódu hlavního programu (dokončí instrukci)
    - Uloží registry
    - Podle adresy (vektoru) daného přerušení "skočí" na adresu obsluhy přerušení
    - Po skončení vrátí načte registry a pokračuje vykonávání hlavního programu
  - Pokud ji probíhá obsluha přerušení a přišlo přerušení
    - S vyšší prioritou -> zastaví kód s nižší prioritou a vykoná obsluhu tohoto
    - S nižší prioritou -> počká na dokončení toho vyššího (pending) a pak teprve vykoná nižší

NVIC – zjednodušený pohled

- NVIC = Nested Vector Interrupt Controller
  - Zpracovává požadavky na přerušení od periférií nebo jádra (až 81 zdrojů, ne všechny využity, záleží na obvodu)
  - Umožňuje nastavit priority periferních přerušení (až 16 úrovní)
  - Přerušení z jádra mají nejvyšší priority
  - Řízení pomocí skupin registrů (3 od každého)
    - ISERx – Interrupt set-enable
    - ICERx – Interrupt clear-enable
    - ISPRx – Interrupt set-pending
    - ...
- Součástí základní knihoven (CMSIS)
  - Jednoduché funkce nastavení registrů NVIC
  - PM 4.3.1 – str. 208/260
  - Nejčastěji NVIC_EnableIRQ a NVIC_DisableIRQ
    - S parametrem typu IRQn_Type (= typedef enum)

Table 45. CMSIS access NVIC functions

CMSIS function[1]	Description
void NVIC_EnableIRQ(IRQn_Type IRQn)	Enables an interrupt or exception.
void NVIC_DisableIRQ(IRQn_Type IRQn)	Disables an interrupt or exception.
void NVIC_SetPendingIRQ(IRQn_Type IRQn)	Sets the pending status of interrupt or exception to 1.
void NVIC_ClearPendingIRQ(IRQn_Type IRQn)	Clears the pending status of interrupt or exception to 0.
uint32_t NVIC_GetPendingIRQ(IRQn_Type IRQn)	Reads the pending status of interrupt or exception. This function returns non-zero value if the pending status is set to 1.
void NVIC_SetPriority(IRQn_Type IRQn, uint32_t priority)	Sets the priority of an interrupt or exception with configurable priority level to 1.
uint32_t NVIC_GetPriority(IRQn_Type IRQn)	Reads the priority of an interrupt or exception with configurable priority level. This function return the current priority level.

1. The input parameter IRQn is the IRQ number,

NVIC na STM32F411RE

- Tabulka vektorů přerušení
  - Viz. RM – kap. 10
  - Str. 198/836, tabulka 37
- Větší číslo = menší priorita
- Každý zdroj přerušení má svůj vektor
  - = adresa v paměti, kde je uložena adresa obslužné funkce
  - V případě přerušení je z tabulky vektorů načtena adresa a na ní pokračuje vykonávání programu

Table 37. Vector table for STM32F411xC/E

Position	Priority	Type of priority	Acronym	Description	Address
	-	-	-	Reserved	0x0000 0000
	-3	fixed	Reset	Reset	0x0000 0004
	-2	fixed	NMI	Non maskable interrupt, Clock Security System	0x0000 0008
	-1	fixed	HardFault	All class of fault	0x0000 000C
	0	settable	MemManage	Memory management	0x0000 0010
	1	settable	BusFault	Pre-fetch fault, memory access fault	0x0000 0014
	2	settable	UsageFault	Undefined instruction or illegal state	0x0000 0018
	5	settable	PendSV	Pendable request for system service	0x0000 0038
	6	settable	Systick	System tick timer	0x0000 003C
0	7	settable	WWDG	Window Watchdog interrupt	0x0000 0040
1	8	settable	EXTI16 / PVD	EXTI Line 16 interrupt / PVD through EXTI line detection interrupt	0x0000 0044
2	9	settable	EXTI21 / TAMP_STAMP	EXTI Line 21 interrupt / Tamper and TimeStamp interrupts through the EXTI line	0x0000 0048
3	10	settable	EXTI22 / RTC_WKUP	EXTI Line 22 interrupt / RTC Wakeup interrupt through the EXTI line	0x0000 004C
4	11	settable	FLASH	Flash global interrupt	0x0000 0050
5	12	settable	RCC	RCC global interrupt	0x0000 0054
6	13	settable	EXTI0	EXTI Line0 interrupt	0x0000 0058
27	34	settable	TIM1_CC	TIM1 Capture Compare interrupt	0x0000 00AC
28	35	settable	TIM2	TIM2 global interrupt	0x0000 00B0
29	36	settable	TIM3	TIM3 global interrupt	0x0000 00B4
30	37	settable	TIM4	TIM4 global interrupt	0x0000 00B8

Kód pro využití přerušení od TIM3

- Mechanismus využití přerušení je stejný jako jsme používali minule u SysTick
  - V interruptu se nastaví příznak
  - V hlavní smyčce se testuje
  - Pokud je nastaven, vynuluje se a provede se akce
- Vznik interruptu – bit UIE v registru DIER
- Povolení v NVICu – funkce z CMSIS
  - Symbol (konstanta) s číslem přerušení najdeme v enum v souboru
  - Na funkci NVIC_EnableIRQ dám F3 a vyberu core_cm4.h
  - Na typu IRQn_Type dám F3 a vyberu stm32f411xe.h
  - Hledám nahoru TIM3...něco a vidím, že TIM3_IRQn = 29
- Jméno IRQ funkce zjistíme v souboru zavaděče
  - startup_stm32f411xe.s
  - Hledáme symbol TIM3...něco
  - Opět je .weak, tj.
    - Pokud máme takovou funkci, linker ji použije
    - Jinak defaultní (prázdná) obsluha
- V přerušení nezapomenout "shodit" příznak události
- Nastavit uživatelský příznak
  - Je testován a shozen v hlavní smyčce

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

```
volatile bool tim3irq = false;

void TIM3_IRQHandler(void)    // obsluha preruseni od TIM3
{
  TIM3->SR &= ~TIM_SR_UIF;    // shodit priznak !!
  tim3irq = true;
}
```

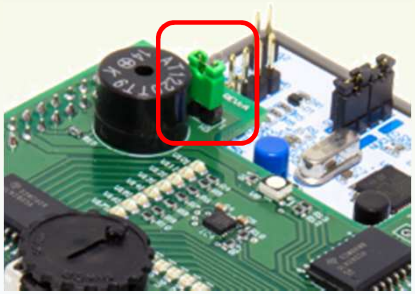
```
  if (tim3irq)              // priznak ze doslo k preteceni
  {
    tim3irq = false;        // shodit
    GPIOToggle(RGB_BLUE);
  }
```

Generování tónu v přerušení TIM3

- Nastavíme PSC tak, aby vstup čítače byl 1MHz
  - Zápis 1E6 si převede překladač na celočíselnou hodnotu 1000000
- Nastavím ARR na 1000 opakování
  - Čítač přetéká po 1ms = 1kHz
- V přerušení měníme stav výstupu připojeného na piezo-"pípák"
  - Nezapomeneme nastavit GPIO – dle schématu je to PB0
  - Pípák připojen přes zelený "jumper" = možno "ztichnout"

```
...
  Nucleo_SetPinGPIO(GPIOB, 0, ioPortOutputPP);  // pipak
...
  TIM3->PSC = SystemCoreClock / 1E6 - 1;
  TIM3->ARR = 1000 - 1;            // 1000x
...
```

```
void TIM3_IRQHandler(void)    // obsluha preruseni od TIM3
{
  TIM3->SR &= ~TIM_SR_UIF;    // shodit priznak !!

  GPIOToggle(GPIOB, 0);       // pipak
  tim3irq = true;
}
```



Vylepšení "rychlého" přerušení

- Vzhledem k tomu, že přerušení nyní nastávají po 1ms je blikání LED příliš rychlé (jakoby svítí stále)
- Můžeme napočítat 100x přerušení a až poté nastavit příznak **tim3irq**, od kterého se odvozuje blikání v hlavní smyčce
- Ideální je použít proměnnou typu **static**
  - Je lokální ve funkci (zde interruptu)
  - Zachovává hodnotu i po skončení funkce
  - Zde slouží jako počítadlo vyvolání funkce
  - Po 100 inkrementech vynuluje počítadlo a nastaví "blikací" příznak

```
volatile bool tim3irq = false;
void TIM3_IRQHandler(void)    // obsluha preruseni od TIM3
{
  static int cnt = 0;         // lokalni ale staticka !!!

  TIM3->SR &= ~TIM_SR_UIF;    // shodit priznak !!

  GPIOToggle(GPIOB, 0);       // pipak

  cnt++;
  if (cnt >= 100)             // pocitej kazde 100x volani
  {
    cnt = 0;
    tim3irq = true;
  }
}
```

Shrnutí domácí přípravy – IV

- Umíme nastavit čítač/časovač v základním režimu
  - Vyzkoušeno na TIM3
- Umíme otestovat příznak přetečení/podtečení
- Umíme použít přerušení od přetečení/podtečení
  - Nastaven základní režim NVICu
  - Nezapomenout "shodit" příznak události
- Umíme generovat tón na připojeném "pípáku" využitím přerušení
- Připomněli jsme si význam proměnné typu static
  - Použito pro počítání počtu vyvolání přerušení

Plán domácí přípravy

I. Instalace prostředí, ... První program – blikání LED na GPIO

II. Zobecnění GPIO operací

III. Systick

IV. Timer

V. UART

- **Sériová komunikace – princip, COM port na straně PC**
- **USART blok v STM32 – registry, časování, režimy**
- **HW propojení mezi procesorem – ST-link – virtuální USB COM port**
- **Funkce pro vysílání a příjem bajtů (dat)**
- **Implementace do standardní in-out knihovny (syscalls.c), použití printf a spol.**

VI. Rezerva

VII. SPI

VIII. PWM

IX. A/D

X. I2C (využití "knihovny")

XI. Časování – HSI/HSE/PLL

XII. Rezerva

Cíle přípravy

- Vyzkoušet sériovou komunikaci mezi PC a kontrolérem
  - Data přenášena přes virtuální COM port (fyzicky přes USB)
  - Na straně PC se chová jako zařízení COMx
  - Na straně kontroléru dělá ST-Link konverzi USB <-> Rx/Tx
- Nastavit periférii U(S)ART a příslušné GPIO
  - Spočítat správně dělící poměry pro komunikační rychlost
- Vytvořit funkce pro odesílání a příjem bajtů (znaků)
- Upravit předpřipravené systémové funkce pro použití standardní knihovny STDIO

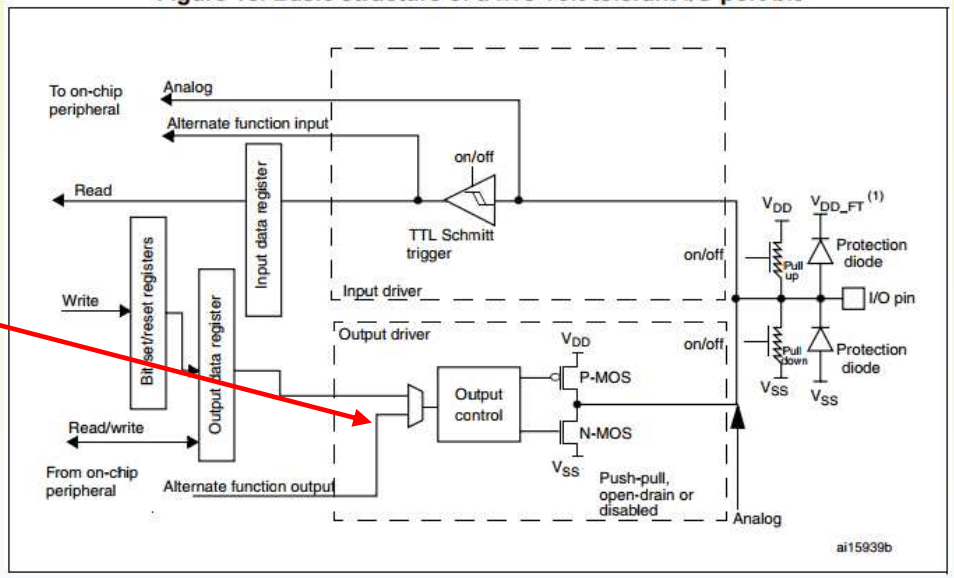
Nový projekt

- Nový projekt - F411RE a můžeme použít:
  - Newlib nano – základní stdio knihovna v minimálním provedení
  - "use tiny printf" – zjednodušený kód – úspornější a rychlejší
  - **!!! "generate syscalls"** – připojí kostru pro std. IO funkce

- Zkopírovat stm_core.h a .c a mpp_shield.h a .c
- Použít základní kód pro SysTick a blikání LED = ověření funkčnosti

- Blok (periférie) pro sériovou komunikaci sdílí vývod s GPIO blokem
  - Nutno nastavit alternativní funkci
    - Viz. Fig.16 v kap. 8.3 v RM

Figure 16. Basic structure of a five-volt tolerant I/O port bit



Alternativní funkce = AF

Table 9. Alternate function mapping

Port		AF00	AF01	AF02	AF03	AF04	AF05	AF06	AF07	AF08	AF09	AF10	AF11	AF12	AF13	AF14	AF15
		SYS_AF	TIM1/TIM2	TIM3/ TIM4/ TIM5	TIM9/ TIM10/ TIM11	I2C1/I2C2/ I2C3	SPI1/I2S1S PI2/ I2S2/SPI3/ I2S3	SPI2/I2S2/ SPI3/ I2S3/SPI4/ I2S4/SPI5/ I2S5	SPI3/I2S3/ USART1/ USART2	USART6	I2C2/ I2C3	OTG1_FS		SDIO			
Port A	PA0	-	TIM2_CH1/ TIM2_ETR	TIM5_CH1	-	-	-	-	USART2_ CTS	-	-	-	-	-	-	-	EVENT OUT
	PA1	-	TIM2_CH2	TIM5_CH2	-	-	SPI4_MOSI /I2S4_SD	-	USART2_ RTS	-	-	-	-	-	-	-	EVENT OUT
	PA2	-	TIM2_CH3	TIM5_CH3	TIM9_CH1	-	I2S2_CKIN	-	USART2_ TX	-	-	-	-	-	-	-	EVENT OUT
	PA3	-	TIM2_CH4	TIM5_CH4	TIM9_CH2	-	I2S2_MCK	-	USART2_ RX	-	-	-	-	-	-	-	EVENT OUT
	PA4	-	-	-	-	-	SPI1_NSS/I 2S1_WS	SPI3_NSS/I2 S3_WS	USART2_ CK	-	-	-	-	-	-	-	EVENT OUT
	PA5	-	TIM2_CH1/ TIM2_ETR	-	-	-	SPI1_SCK/I 2S1_CK	-	-	-	-	-	-	-	-	-	EVENT OUT
	PA6	-	TIM1_BKIN	TIM3_CH1	-	-	SPI1_MISO	I2S2_MCK	-	-	-	-	-	SDIO_ CMD	-	-	EVENT OUT
	PA7	-	TIM1_CH1N	TIM3_CH2	-	-	SPI1_MOSI /I2S1_SD	-	-	-	-	-	-	-	-	-	EVENT OUT
	PA8	MCO_1	TIM1_CH1	-	-	I2C3_ SCL	-	-	USART1_ CK	-	-	USB_FS_ SOF	-	SDIO_ D1	-	-	EVENT OUT

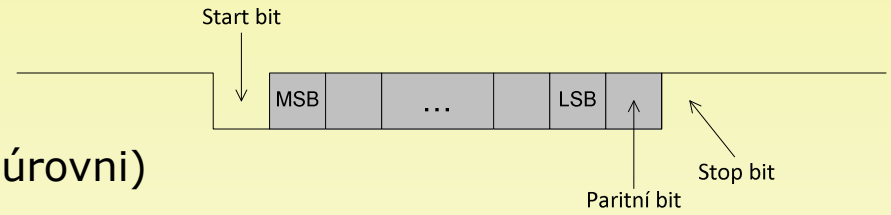
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
AFRL7[3:0]				AFRL6[3:0]				AFRL5[3:0]				AFRL4[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AFRL3[3:0]				AFRL2[3:0]				AFRL1[3:0]				AFRL0[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

- Přiřazení vývodů a AF
  - Tabulka 9 v kap. 4 v DS
- Režim vývodu AF v MODER registru
  - Kombinace 10: Alternate function mode
  - Volitelně OD/PP v OTYPER
  - Případně pull-up/down v PUPDR
- Číslo AF je nutno zapsat do AF registru
  - Rozdělen na AFRL (spodních 8 vývodů portu) nebo AFRH (vývody 8-15)
  - V C-struktuře je AFR polem o 2 prvcích, index 0 odpovídá AFRL, 1 pak AFRH
  - Samotná hodnota je 4 bitová, tedy teoreticky 16 variant AF
- Doplnění kódu do stm_core
  - Funkce očekává GPIO vývod a číslo AF
  - Výběr prvku pole AFR podle čísla vývodu
  - Na příslušnou pozici 4-bitová hodnota (posun omezen pouze na 0-7 maskováním bitově 00..0111)
- Pozor, předpokládá se nastavení AF v Nucleo_SetPinGPIO

```
bool Nucleo_SetAFGPIO(GPIO_TypeDef *gpio, uint32_t bitnum, uint32_t afValue)
{
  gpio->AFR[(bitnum < 8) ? 0 : 1] &= ~(0x0f << (4 * (bitnum & 0x07)));   // vynuluj AF bity
  gpio->AFR[(bitnum < 8) ? 0 : 1] |= ((afValue & 0x0f) << (4 * (bitnum & 0x07)));   // nastav AF bity
  return true;
}
```

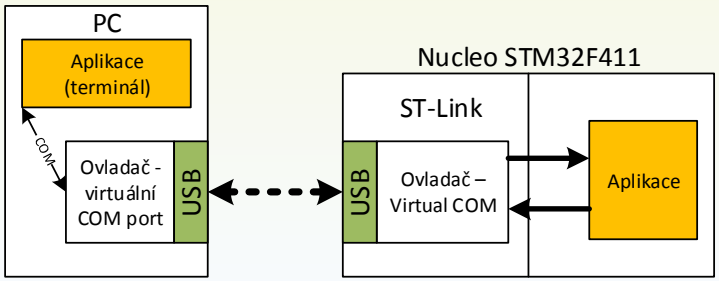
Sériová komunikace - princip

- Pro každý směr vlastní vodič, data odesílána sekvenčně bit po bitu
  - V klidu log. 1
  - Start-bit log. 0
  - 5-8 datových bitů, nejvýznamnější (MSB) první
  - Bit nepovinné parity (typicky generováno na HW úrovni)
  - Stop-bit o délce 1, 1.5 nebo 2 bity v log. 1
  - Doba trvání bitu vychází z přenosové rychlosti
  - Přenesení bajtu představuje typicky 10 bitů (1x start + 8x data + 1x stop)



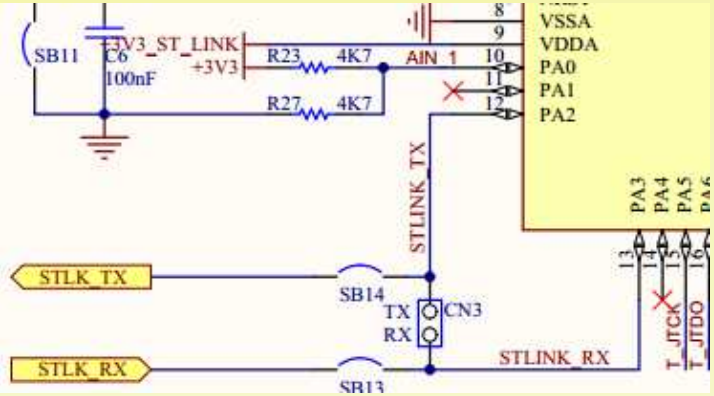
- Případně více detailnější slajdy s popisem sériové komunikace
  - https://www.slideshare.net/AnkitSingh13/uart-32550652

- "Naše" Nucleo není přímo připojeno s PC přes RS232
  - Na straně PC je ovladačem komunikace převedena z virtuálního COM portu na USB
  - Na straně HW převádí ST-Link USB data na sériovou komunikaci
  - V aplikacích tedy pracujeme se sériovým portem
  - Ve skutečnosti data "tečou" v rámci USB paketů

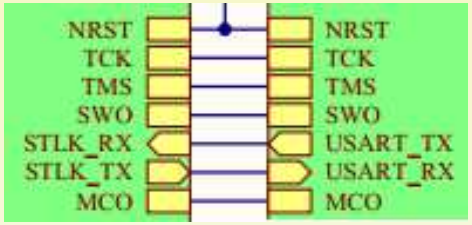


Použití sériové komunikace na Nucleo-F411

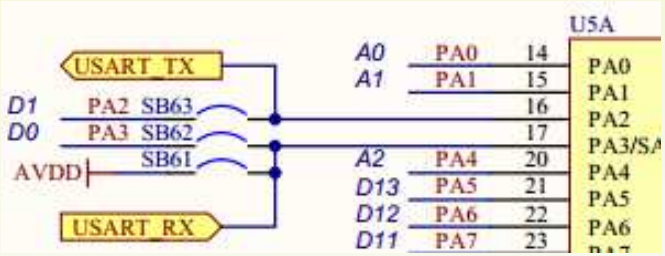
- Jak číst schéma Nucleo desky ?



- Str. 3 – zapojení ST-Link části
  - Převádí STLK_TX a _RX na VCP (Virtual COM Port)



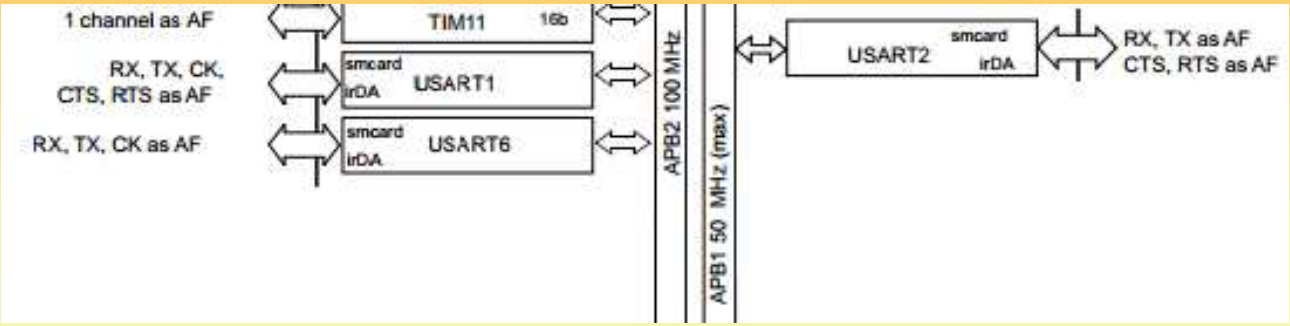
- Str. 1 – propojení mezi ST-Link a hlavním procesorem
  - STLK_RX na USART_TX a STLK_TX na USART_RX
  - Logické, výstup jednoho musí být vstupem druhého



- Str. 2 – využití vývodů F411
  - Připojeno k PA2 a PA3

- V tabulce 9 (DS kap. 4) najdeme možné AF pro tyto vývody
  - Jedině použitelný je USART2 a režim AF07 pro oba vývody

USART2

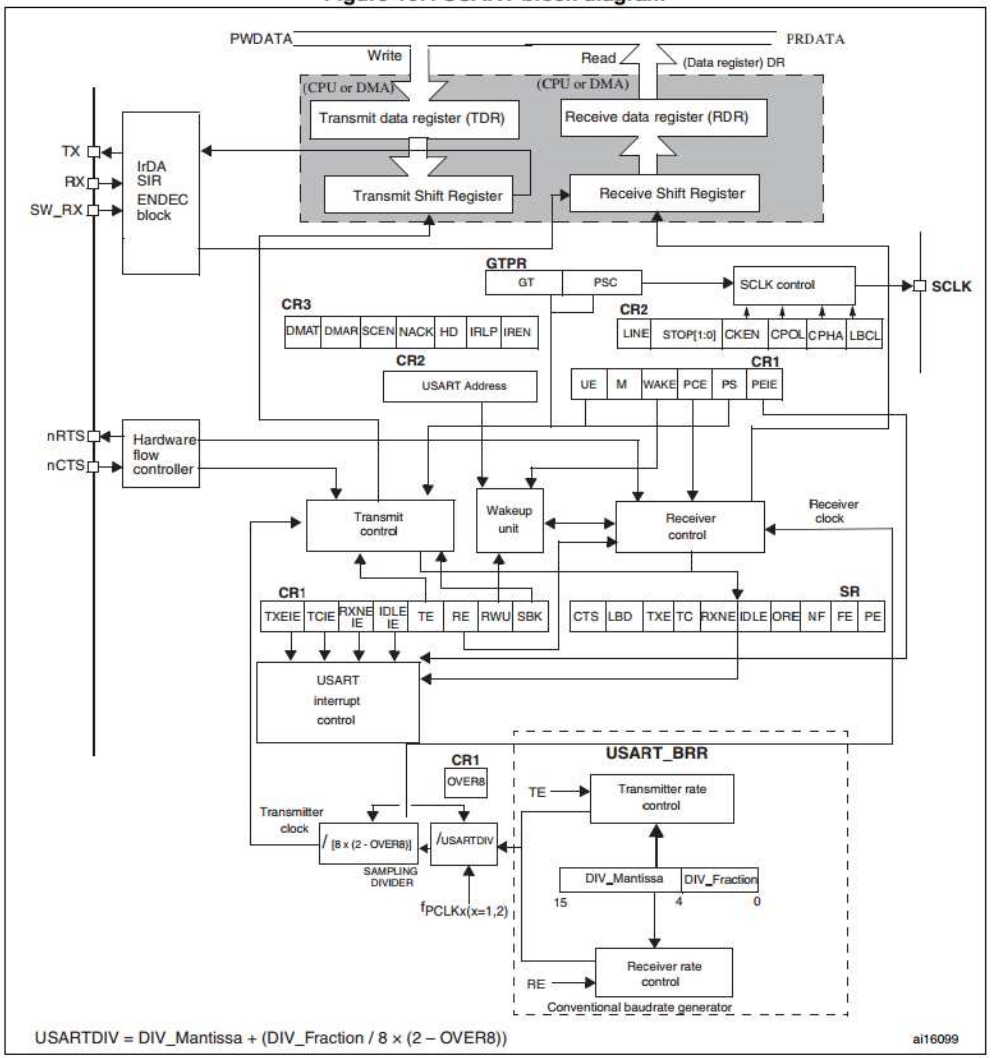


- Na mikrokontroléru F411RE máme k dispozici 3 U(S)ART
  - USART2 připojen k APB1 (na rozdíl od USART1 a 6)
- Základní funkcionalita velmi bohatá – viz. kap. 19.2 v RM
  - Nastavitelné slova 5-8 bitů
  - Použití parity (může se přidávat v 9. bitu)
  - Detekce několika typů chyb
  - Generování přerušení při přenosu i při chybách
  - Široce programovatelná přenosová rychlost
  - Synchronní a asynchronní režimy
  - Další speciální režimy, ...

USART blokově

- Viz. RM kap. 19.1
- Interně obsahuje
  - Pro vysílání shift-registr a datový registr Transmit Data Reg.
  - Pro příjem shift-registr a datový registr Receive DR
  - Řídící a stavové signály – některé přístupné přes registry
  - Generátor přenosové rychlosti BRR
- Při vysílání se po zápisu dat do TDR počká na doodvysílání dat ze shift registru a pak se začnou vysílat tato data a TDR je volný
  - Takže může být jeden znak vysílán a druhý čekat v TDR
- Při příjmu se sériová data "nasunují" do shift registru a při příjmu stop-bitu se zapíší do RDR k vyzvednutí
  - Potom může shift registr přijímat další bity, tj. program musí odebrat data před přijmutím dalšího znaku
- Při čtení začátečního start-bitu a hodnot bitů je použito 8- nebo 16-násobné převzorkování
  - Hodiny z BRR jsou násobkem přenosové rychlosti

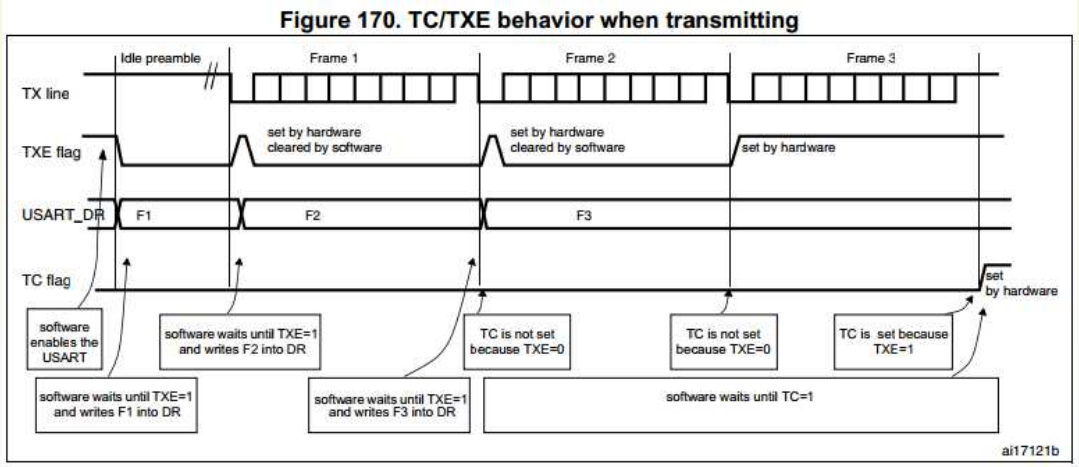
Figure 167. USART block diagram



Příznaky a akce při příjmu

- Pro vysílání se používají 2 příznaky nastavované HW a nulované programově
  - TXE = TDR empty = možno zapsat další data do TDR registru, protože ta původní začala být odesílána
    - Nastaven na počátku fyzického přenosu
  - TC = Trans. Complete = je odvysíláno a nic již nečeká v TDR
- Z toho plyne:
  - Než zapíšeme nová data do TDR, musíme vyčkat na nastavení TXE
  - Pokud stihneme dodávat data do TDR, nenastane TC
  - Pokud je program blokován po dobu přenosu bloku dat, je to minimálně
    - "počet znaků - 1" * "doba trvání znaku"
    - -1 může být proto, že první znak se hned začne posílat a může se do TDR zapsat další

- Viz. kap. 19.3.2 v RM



Výpočet přenosové rychlosti

- RM 19.3.4 str. 511/836
  - Oversampling - OVER8 = 0 pro 16x, = 1 pro 8x
  - f_{CK} hodiny periférie (pro USART2 je to APB1)

Equation 1: Baud rate for standard USART (SPI mode included)

$$\text{Tx/Rx baud} = \frac{f_{CK}}{8 \times (2 - \text{OVER8}) \times \text{USARTDIV}}$$

- Typicky potřebujeme obrácenou rovnici
  - USARTDIV = f_{CK} / (baud x 8 x (2 – OVER8))
- Máme-li 16MHz AHB a následně APB1 a chceme komunikovat rychlostí 38400 při 16x oversamplingu
  - USARTDIV = 16M / (38400 * 16) = 26,0416666
- BRR registr má 12-bitovou celou část a 4-bitový "zlomek"
  - Celá část USARTDIV je 26
  - Zlomková část je v šestnáctinách (4-bitově maximum)
  - Možno rychle odhadnout nejbližší hodı
    - 0/16 = 0
    - 1/16 = 0,0625
    - 2/16 = 0,125
    - ...
  - Pro 0,041666 je to 1 (nejblíže má k 0,0625)

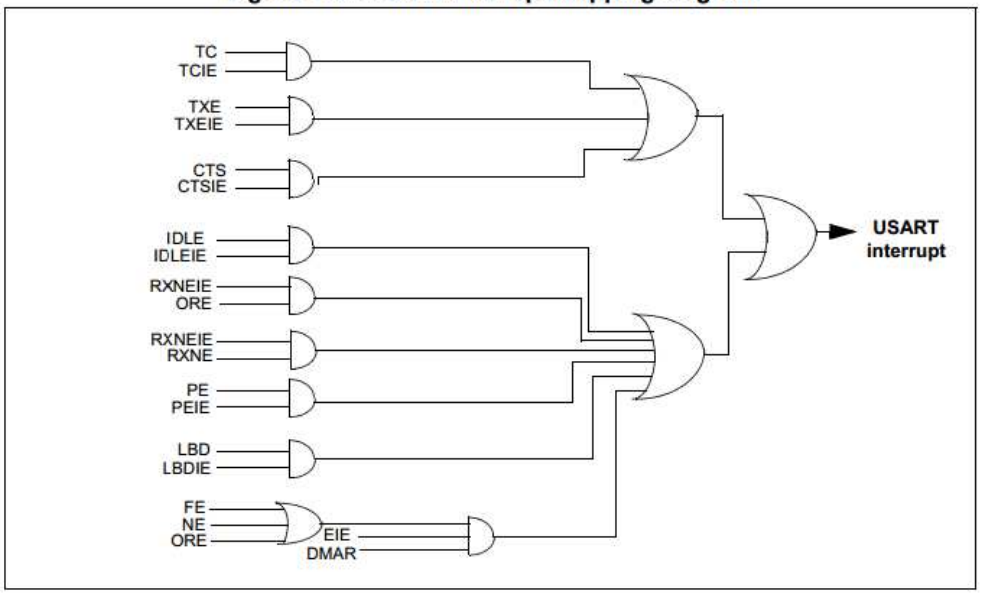
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DIV_Mantissa[11:0]												DIV_Fraction[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

- Obsah BRR vypočteme jako celou část posunutou vlevo o 4 (pozice mantisy) + 4-bitový zlomek na spodních 4 bitech
  - Elegantně zapíšeme hexadecimálně (kde 0x1A = 26): **0x01A1**,

Události

- Celý USART má společný IRQ výstup
  - V obslužné funkci nutno detekovat, která událost ji spustila
  - Nezapomenout "shodit" příslušný příznak
    - Některé se nulují automaticky (RXNE, TXE, ...)
  - Pokud je více požadavků, je možné je obsloužit "najednou" nebo se IRQ vyvolá vícekrát
- Každý zdroj má svůj xxIE (Interrupt Enable)
  - Povoluje vznik přerušení

Figure 191. USART interrupt mapping diagram



Status registr

Reset value: 0x00C0 0000

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						CTS	LBD	TXE	TC	RXNE	IDLE	ORE	NF	FE	PE
						rc_w0	rc_w0	r	rc_w0	rc_w0	r	r	r	r	r

- **SR** = status registr
  - **TXE** – Transmit Data Reg Empty = prázdný vysílací datový registr (TDR)
    - Přesunuta data z TDR do vysílacího shift-registru
    - Vynuluje se automaticky zápisem dalších dat
  - **TC** – Transmit Complete = vysílání (rámce) dokončeno
  - **RXNE** – Receive Data Reg. Not empty
    - Přijímací datový registr (RDR) obsahuje nově přijatá data
  - **ORE** – Overrun Error = chyba přetečení
    - Nebyl vyčten obsah datového registru před dokončením příjmu dalšího rámce
    - Nově přijatá data zahozena
  - **NF** – Noise Detect = příznak "šumu"
    - Blok přijímače nedetekoval konstantní log. úroveň v době detekce hodnoty bitu
  - **FE** – Framing Error = chyba rámce = typicky neodpovídají STOP-bity
  - **PE** – Parity Error = vypočtená parita neodpovídá nastavené
- Chybové příznaky se vynulují přečtením SR a následným čtení DR
  - Viz. jednotlivé Note v RM
- Defaultní hodnota 0xC0 (1100 0000), tj. nastavené TXE a TC (logicky)

CR1

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
OVER8	Reserved	UE	M	WAKE	PCE	PS	PEIE	TXEIE	TCIE	RXNEIE	IDLEIE	TE	RE	RWU	SBK
rw	Res.	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

- **OVER8** – vzorkování při příjmu
  - 0 = oversampling 16x, 1 = over 8x
  - Důležité také při výpočtu rychlosti (BRR)
- **UE** – USART enable = povolení periférie
- **PCE** – Parity Enable – povolení generování a detekce parity
- **PS** – Parity Select – 0 = Even (sudá), 1 = Odd (lichá)
- **PEIE**, **TCIE**, **RXENIE** – povolení přerušení od chyby parity, dokončení vysílání a přijatých dat
- **TE** – Transmitter Enable = povolení funkce vysílače
- **RE** – Receiver Enable = povolení funkce přijímače

- **Pozor – pro kompletní provoz musí být nastaven UE, RE i TE**

Další registry

- **DR** = Data register
  - Při čtení se vyčítá hodnota přijímacího datového registru RDR
  - Při zápisu se zapisuje do odesílacího datového registru TDR

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
DIV_Mantissa[11:0]												DIV_Fraction[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

- **BRR** = Baud Rate register
  - 16b hodnota složená z 12b mantisy (celočíselný základ) a 4b fraction (zlomkový zbytek)

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
Res.	LINEN	STOP[1:0]		CLKEN	CPOL	CPHA	LBCL	Res.	LBDIE	LBDL	Res.	ADD[3:0]			
	rw	rw	rw	rw	rw	rw	rw		rw	rw	rw	rw	rw	rw	rw

- **CR2** = Control Register 2
  - Použitý pro synchronní režimy a LIN

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				ONEBIT	CTSIE	CTSE	RTSE	DMAT	DMAR	SCEN	NACK	HDSEL	IRLP	IREN	EIE
				rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

- **CR3** = Control Register 3
  - Pro řízení DMA přenosů a povolení přerušení

USART2 - inicializace

- Povolit hodiny periférie z APB1
- Nastavit CR registry
- Nastavit BRR
  - Zatím vypočtená konstanta pro rychlost 38400 b/s
- Povolit komunikaci bitem UE v CR1

```
if (!(RCC->APB1ENR & RCC_APB1ENR_USART2EN))       // neni povolen USART2
  {
    RCC->APB1ENR |= RCC_APB1ENR_USART2EN;
    RCC->APB1RSTR |= RCC_APB1RSTR_USART2RST;
    RCC->APB1RSTR &= ~RCC_APB1RSTR_USART2RST;
  }

  USART2->CR1 = USART_CR1_RE | USART_CR1_TE;      // staci povoleni prijmu a vysilani
  USART2->CR2 = 0;      // nic specialniho
  USART2->CR3 = 0;      // nic specialniho

  //TODO doplnit vypocet BRR podle pozadovaneho a podle clocku
  USART2->BRR = 0x1A1;  // rychlost 38400 pri 16MHz - spocitano predem

  USART2->CR1 |= USART_CR1_UE;    // az na zaver povolen blok USARTu
```

Odesílání znaků přes USART2

- Funkce odesílání
  - Na počátku čekání na dokončení případného probíhajícího vysílání
    - Resp. TXE příznak = volný TDR
  - Zapsat do TDR nová data k odeslání

```
int Usart2Send(char c)
{
  while(!(USART2->SR & USART_SR_TXE))
    ;           // cekej dokud neni volny TDR

  USART2->DR = c;    // zapis do TDR k odeslani
  return c;
}
```

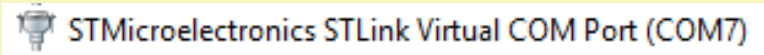
- Tedy pokud se vysílalo, interně se počká na doodvysílání
  - Jinak se ihned přesune obsah do vysílacího shift-registru

- Funkci volat opakovaně
  - Podle Systick časování
  - Posíláme znaky 'A'– 'Z'

```
if (_ticks >= tm2)    // casovani vysilani
{
  tm2 = _ticks + 100; // každých 100ms

  Usart2Send(c);  // funkce interne pocka na doodvysilani

  c++;
  if (c > 'Z')   // vysilame jen velka pismena
      c = 'A';
}
```

Vysílání a příjem dat na PC

- Zjištění COM portu – Správce zařízení
  - Pravý klik na Tento počítač a vybrat Spravovat
- Porty COM a LPT
  - Jeden bude od STLink



    - Realizováno ovladačemm který se nainstaloval při instalaci TrueStudia společně s ovladači pro debug přes STLink
    - Číslo COMx přiděleno poprvé podle zařízení a použitého USB portu, dále již zůstává při stejné konfiguraci
- Pro komunikaci používáme program typu "terminál"
  - Přijaté bajty (znaky) z COMx zobrazuje v hlavním okně
  - Při stisku kláves odesílá znaky po COMx z PC
  - Často jsou to univerzální programy také pro TCP/IP a UDP komunikaci
- **PUTTY** - http://www.putty.org/
- **Hercules** - http://www.hw-group.com/products/hercules/index_cz.html

- V připojeném terminálu se ukazují znaky A-Z v taktu 100ms

- **!! Občas vysílání přestane fungovat – problém ve firmware ST-Linku**
  - **Pomůže odpojit/připojit Nucleo desku na USB**

Vysílání dat z PC do mikrokontroléru

- Funkce na příjem
  - Přijatá data vrací jako hodnotu
  - Blokující čekáním na RXNE
- Vhodné si připravit test existence přijatých dat
  - Funkce "je přijato"
  - Vrací stav RXNE jako bool
    - Vynuceně podmínkou

```
int Usart2Recv(void)
{
  while(!(USART2->SR & USART_SR_RXNE)) // cekej dokud neprijde
    ;

  return USART2->DR;      // vycti a vrat jako hodnotu
}

bool IsUsart2Recv(void)   // priznak, ze je neco v bufferu
{
  return (USART2->SR & USART_SR_RXNE) != 0;
      // podminka vynuti true/false vysledek
}
```

- V hlavní smyčce programu
  - Testujeme přijetí dat
  - Vyčteme přijatá data (čekají v RDR)
  - Zobrazíme na 8xLED

```
  if (IsUsart2Recv())          // neco prislo ?
  {
    char x = Usart2Recv();     // blokujici funkce

    Write8LED(x);
  }
```

- Pro jednodušší výpis textů si připravíme funkci "pošli řetězec"
- Lze pak využívat např. sprintf, ale nezapomenout:
  - #include <stdio.h>
  - Buffer pro řetězec (o dostatečné velikosti)

```
void Usart2String(char *txt)
{
  while(*txt)
  {
    Usart2Send(*txt);
    txt++;
  }
}
```

Náhrada systémových volání v syscalls.c

- Pokud jsme nezvolili při vytváření projektu - Project-New...-Other...-System Calls-Minimum System Calls ...
  - Klidně nechat dynamic memory
  - Pokud se nevytvoří syscalls.c v /src tak tam přesunout do /src
- Najít funkce _read a _write
  - Pro přehlednost přesunout _read k _write
  - Obsahují pouze kód errno = ENOSYS; což je chyba "není implementováno"
- Podobné parametry
  - File – identifikace "souboru"
    - 0 = stdin, 1 = stdout, 2 = stderr
  - Ptr - adresa datového bufferu
  - Len - počet znaků ke přijetí/odeslání

```
int _write(int32_t file, uint8_t *ptr, int32_t len)
{
  errno = ENOSYS;
  return -1;
}

int _read(int32_t file, uint8_t *ptr, int32_t len)
{
  errno = ENOSYS;
  return -1;
}
```

Náhrada systémových volání v syscalls.c

- Přidat deklarace Usart2xxx funkcí pro snazší volání
  - Bez nutnosti includovat další soubory
- Nastavit "globální" errno na 0 = "bez chyb"
- Tělo **_write** je cyklus posílající po znacích
  - Délka je předaná jako len
- Tělo **_read** čte jen 1 znak na začátek bufferu
  - Vrací počet = 1
  - Bez ohledu na požadovaný len
- V main pak připojit stdio.h a lze používat puts, printf, ...
- Při příjmu i vysílání se defaultně projeví bufferování I/O operací v knihovně STDIO
  - Chová se "řádkově", tedy odesílá až po CR/LF
- Řešení - nastavit setvbuf na _IONBF = no buffering
  - Nejlépe jako součást inicializace
  - Viz. https://www.tutorialspoint.com/c_standard_library/c_function_setvbuf.htm

```
// pomocna deklarace, abych nemusel includovat
void Usart2Send(char c);
int Usart2Recv(void);

int _write(int32_t file, uint8_t *ptr, int32_t
len)
{
  int i;
  for(i = 0; i < len; i++)
    Usart2Send(ptr[i]);

  errno = 0;    // OK
  return len;   // kolik se odeslalo
}

int _read(int32_t file, uint8_t *ptr, int32_t len)
{
  *ptr = Usart2Recv();

  errno = 0;    // OK
  return 1;     // prijal se prave jeden
}
```

```
// zrusit bufferovani vystupu i vstupu
setvbuf(stdout, NULL, _IONBF, 0);
setvbuf(stdin, NULL, _IONBF, 0);
```

Oddělit funkce do samostatného .C a .H

- Obsah syscalls.c beze změn

- Vytvoříme nový nucleo_usart.c a nucleo_usart.h
- Do C přesuneme z main.c funkce Usart2Send, Usart2String, Usart2Recv, IsUsart2Recv
- Připravíme funkci Usart2Init(int baudSpeed) a přesuneme do ní komplet inicializaci USART2 včetně GPIO
- Připojíme nucleo_usart.h s deklaracemi
  - Ten pak je potřeba i v main.c

```
#ifndef NUCLEO_USART_H_
#define NUCLEO_USART_H_

#include "stm_core.h"
#include <stdio.h>

int Usart2Send(char c);
void Usart2String(char *txt);
int Usart2Recv(void);
bool IsUsart2Recv(void);
void Usart2Init(int baudSpeed);

#endif /* NUCLEO_USART_H_ */
```

Výpočet BRR algoritmicky

```
// bd = apb1 / (16 x DIV) .... DIV = APB1 / (16 x BD) = 16M / (16 x baud)
// pro baud = 38400 vyjde DIV = 26,041666, 1/16 = 0,625, tedy 26 = 0x1A a spodni 4b = 0x1
{  // blok pro lokalni promenne
  uint sampling = (USART2->CR1 & USART_CR1_OVER8) ? 8 : 16;
  uint32_t apb1, mant, tmp, frac;

  apb1 = SystemCoreClock;        //TODO zjistit presne z RCC->CFGR registru deleni

  mant = apb1 * 100 / (sampling * baudRate);  // v setinach
  tmp = mant / 100;

  frac = mant - (tmp * 100);                  // zbyvajici cast 0-99 nutno prevest na 0-15
  frac = ((((frac * sampling) + 50) / 100)); // +50 kvůli zaokrouhlovani ořezem intu

  USART2->BRR = (tmp << 4) | (frac & 0x0f);  // mantisa vyssich 12b, zlomek dolni 4b, celkem 16b
}
```

- Takt APB1 zatím odvozen přímo z CoreClock, protože jedeme pouze na 16MHz
  - Správně je nutno se podívat na dělící poměry v RCC->CFGR registru
- Floatové dělení nahrazeno celočíselným s tím, že počítáme v setinách
  - Nelze v tisícinách – int je 32-bitový, tj. max. rozsah +-2miliardy (+-2^{31}-1), tj. max. apb by byl 2MHz, nyní (v setinách) je to 20MHz
  - Pro vyšší takty lze počítat v "dvacetipětinách", pak max. APB může být 80MHz (4x20M)
  - Pozor na to, že celočíselné zaokrouhlování odřízne desetinnou část, proto +½ rozsahu u výpočtu zlomku

Shrnutí domácí přípravy – V

- Umíme nastavit USART do režimu asynchronního přenosu
  - Používáme USART2, protože "jeho" GPIO jsou připojeny na ST-Link, který realizuje virtuální COM port přes USB
  - Umíme vypočítat dělící poměry v BRR registru pro danou komunikační rychlost a takt hodin
- Umíme otestovat příznaky při příjmu a vysílání
- Máme připravené funkce zabalující základní operace
- Umíme v terminálu na PC přijmout i odeslat znaky
- Umíme využít funkce z knihovny STDIO
  - Máme vyzkoušené funkce _read a _write v stdcalls.c

Plán domácí přípravy

I. Instalace prostředí, ... První program – blikání LED na GPIO
II. Zobecnění GPIO operací
III. Systick
IV. Timer
V. UART
VI. Rezerva
VII. SPI
VIII. PWM
IX. A/D
X. I2C (využití "knihovny")
XI. Časování – HSI/HSE/PLL
XII. Rezerva

Cíle přípravy a shrnutí VI

- Vyzkoušet si zatím všechny probrané bloky (periférie)
- Doplnit si chybějící či nejasné části

- Zkusit vlastní nápad z PC "něco" řídit na modulu a naopak hlásit stav zpět do terminálu na PC

Plán domácí přípravy

I. Instalace prostředí, ... První program – blikání LED na GPIO
II. Zobecnění GPIO operací
III. Systick
IV. Timer
V. UART
VI. Rezerva
VII. SPI
VIII. PWM
IX. A/D
X. I2C (využití "knihovny")
XI. Časování – HSI/HSE/PLL
XII. Rezerva

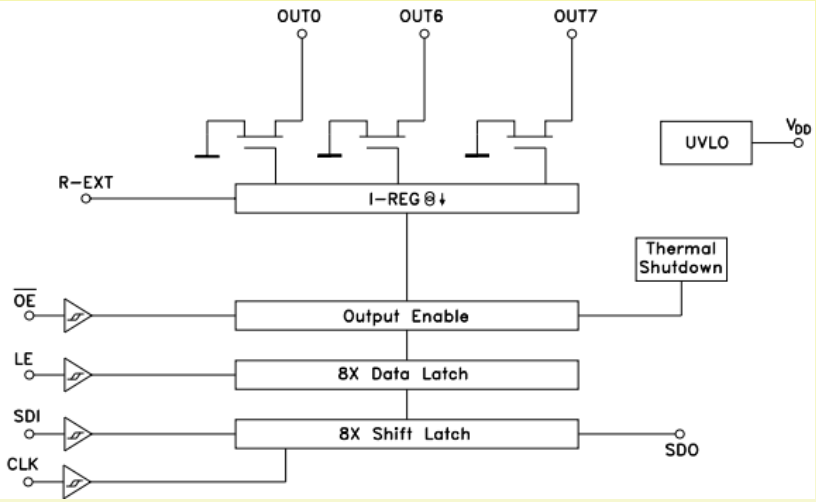
Cíle přípravy

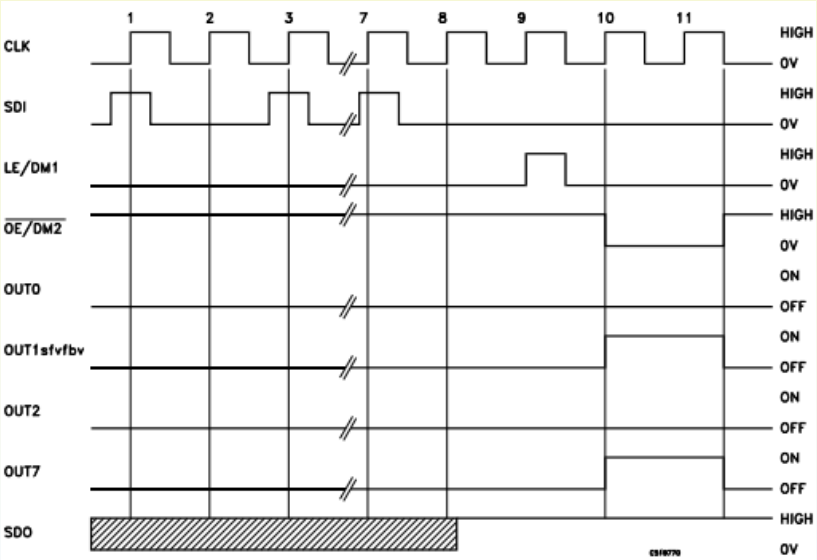
- Použít budič LED se sériovým vstupem dat
  - Generovat příslušné signály "ručně" pomocí GPIO operací
  - Generovat signály periférií SPI
- Seznámit se se základními funkcemi bloku SPI

- Připravit si funkce inicializace a zápisu dat pro další využití

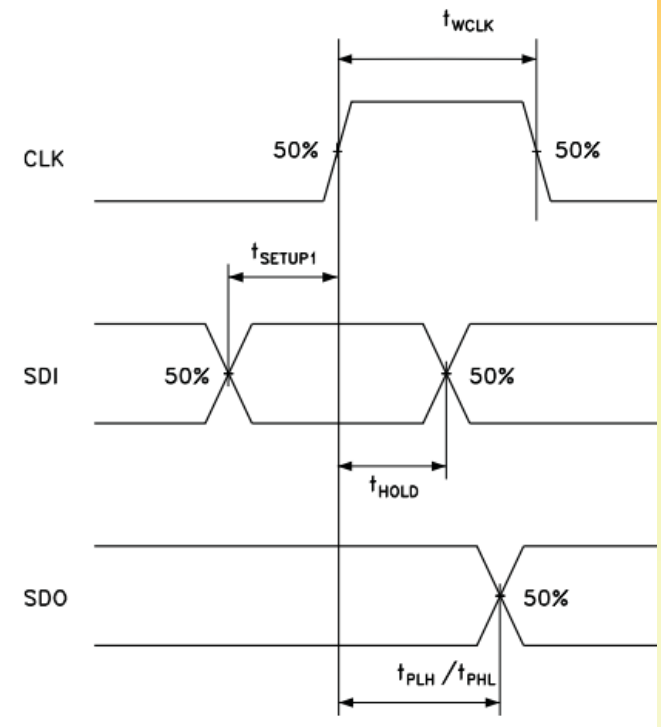
Budič LED STP08CP05

- Viz. DS - STP08CP05MTR - en.CD00160230.pdf
- Umí budit 8 LED konstantním proudem
  - Nastavitelný externím R – viz. fig. 11 na str. 14/31 v DS
  - Každý výstup má svůj proudový zdroj
- Vstup sériový se základními signály
  - **SDI** – Serial Data Input
  - **CLK** – hodiny – aktivní náběžná hrana = očekávají se platná data
  - **/OE** – Output Enable – řízení výstupních budičů LED, aktivní v log. 0
  - **LE** – Latch Input – přepisuje obsah vstupního posuvného registru do datového registru (paralelního)
  - **SDO** – Serial Data Output – možno řetězit více obvodů sériově
- Z časových průběhů (fig. 7 – timming diagram) je dobře vidět, jak mají signály vypadat
  - Aktivní hrana hodin, platná data (SDI), vliv LE a /OE signálů





Budič LED – časování



- Fig. 8 ukazuje časování v detailu
  - Data musí být platná před a po vzestupné hraně hodin
  - Na výstupy se nová hodnota dostane se zpožděním – to nás zatím nezajímá
- Tab. 6 obsahuje konkrétní hodnoty
  - Časové požadavky jsou minimální
    - Délka LE pulsu – 20ns
    - Délka CLK pulsu – určuje frekvenci hodin
    - Nutné přesahy dat a zpoždění
    - Frekvence vyplývá maximální
      - Uvedeno 30MHz
    - OE neřešíme, je trvale v log. 0

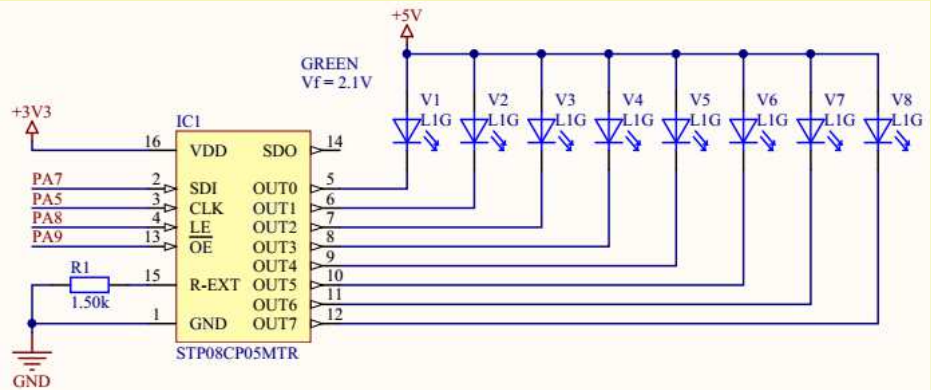
t_{wLAT}	LE pulse width	V_{DD} = 3.0 to 5.0 V	20			ns
t_{wCLK}	CLK pulse width		20			ns
t_{wEN}	$\overline{OE}$ pulse width		200			ns
$t_{SETUP(D)}$	Setup time for DATA		7			ns
$t_{HOLD(D)}$	Hold time for DATA		4			ns
$t_{SETUP(L)}$	Setup time for LATCH		15			ns
f_{CLK}	Clock frequency	Cascade operation (1)			30	MHz

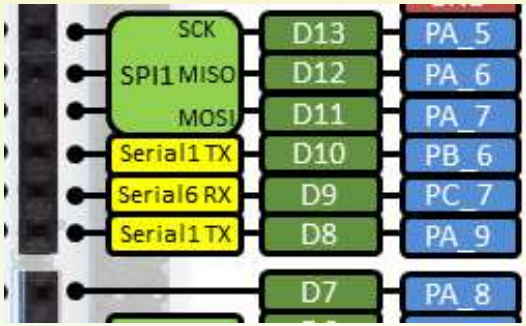
1. In order to achieve high cascade data transfer, please consider t_r/t_f timings carefully.

- Časové průběhy můžeme ověřit na osciloskopu, zda nepřekračujeme meze
  - Při "ručním" generování k tomu nedojde – GPIO operace vyžadují několik instrukcí
  - Problém může nastat při generování přes SPI z rychlejších hodin sběrnice než standardních 16MHz

Zapojení budiče na MPP-Shieldu

- Budič připojen tak, aby šlo jednoduše využít SPI
  - SDI na PA7
    - Odpovídá MOSI = výstup sériových dat
  - CLK na PA5
    - Odpovídá SCK = hodiny sériové sběrnice
  - LE na PA8 a OE na PA9
    - Volné GPIO poblíž, lze použít jakékoliv
- Při prvním využití si vystačíme s programovým ovládáním GPIO, až následně použijeme periférii SPI1





Ovládání přes GPIO – funkce inicializace

```
void InitSPILED(void)
{
  Nucleo_SetPinGPIO(GPIOA, 8, ioPortOutputPP);     // SHIFT - LE
  GPIOWrite(GPIOA, 8, 0);
  Nucleo_SetPinGPIO(GPIOA, 9, ioPortOutputPP);     // SHIFT - OE
  GPIOWrite(GPIOA, 9, 0);
  Nucleo_SetPinGPIO(GPIOA, 5, ioPortOutputPP);     // SHIFT -
CLK
  GPIOWrite(GPIOA, 5, 0);
  Nucleo_SetPinGPIO(GPIOA, 7, ioPortOutputPP);     // SHIFT -
SDI
}
```

- Všechny signály z procesoru jsou "output"
- Při inicializaci je potřeba nastavit výchozí hodnoty podle požadovaných signálů budiče
  - LE – klidový stav log. 0
  - CLK – klidový stav log. 0, první akce bude vzestupná hrana
  - SDI – není třeba předem nastavovat, musí se nastavit pro každý vysílaný bit před CLK hranou
  - OE – aktivní v log. 0, nastavíme takto, aby se jakákoliv změna datového registru ihned projevila na výstupech

Ovládání přes GPIO – funkce výstupu

- Nutno 8x zopakovat
  - Nastavit hodnotu bitu na SDI
  - Udělat L-H (vzestupnou) hranu
  - Logicky následováno sestupnou (H-L) hranou, aby příště mohla být vzestupná
- Bity se vysílají MSB (nejvýznamnější) napřed
  - Maskujeme bitově 1000 0000
  - Posouváme vlevo
- Po odeslání všech bitů nutno provést vzestupnou hranu LE signálu
  - Resp. puls, aby mohla být příště opět vzestupná hrana
- Signál OE negenerujeme, je trvale aktivní (log. 0 při inicializaci)

```
void WriteSPILED(uint8_t val)
{
  int i;

  for(i = 0; i < 8; i++)
  {
    GPIOWrite(GPIOA, 7, (val & 0x80) != 0); // maska 1000 0000

    GPIOWrite(GPIOA, 5, 1);                 // puls na CLK
    GPIOWrite(GPIOA, 5, 0);

    val <<= 1;
  }

  GPIOWrite(GPIOA, 8, 1);                   // puls na LE
  GPIOWrite(GPIOA, 8, 0);
}
```

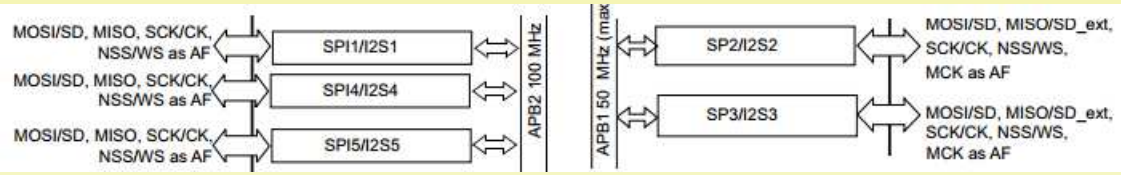
SPI LED budič – main

```
int main(void)
{
  uint32_t tm = 0, tm2 = 0; // timestamps
  uint8_t b = 0;            // pocitadlo pro 8xLED

... Init SysTick

  InitSPILED();

  while (1)
  {
    if (_ticks >= tm2)
    {
      tm2 = _ticks + 50;

      b++;
      WriteSPILED(b);
    }
...
```

- Kód vychází z červených 8xLED
- Pro časování využit tik 1ms
  - Generováno přes SysTick
- Na zelených LED vzor "počítadlo"
- Neřešíme přetečení proměnné b
  - Je 8-bitová, takže nabývá max. hodnoty 255

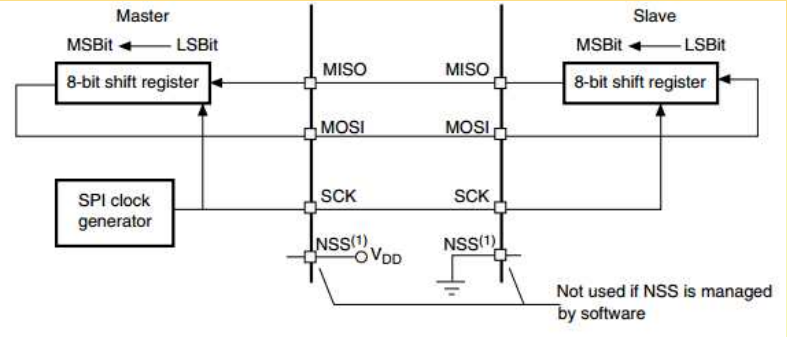
SPI – synchronní sériová sběrnice

- Na procesoru STM32F411 máme celkem 5 bloků SPI sběrnice
  - SPI1, 4 a 5 na APB2
  - SPI2 a 3 na APB1



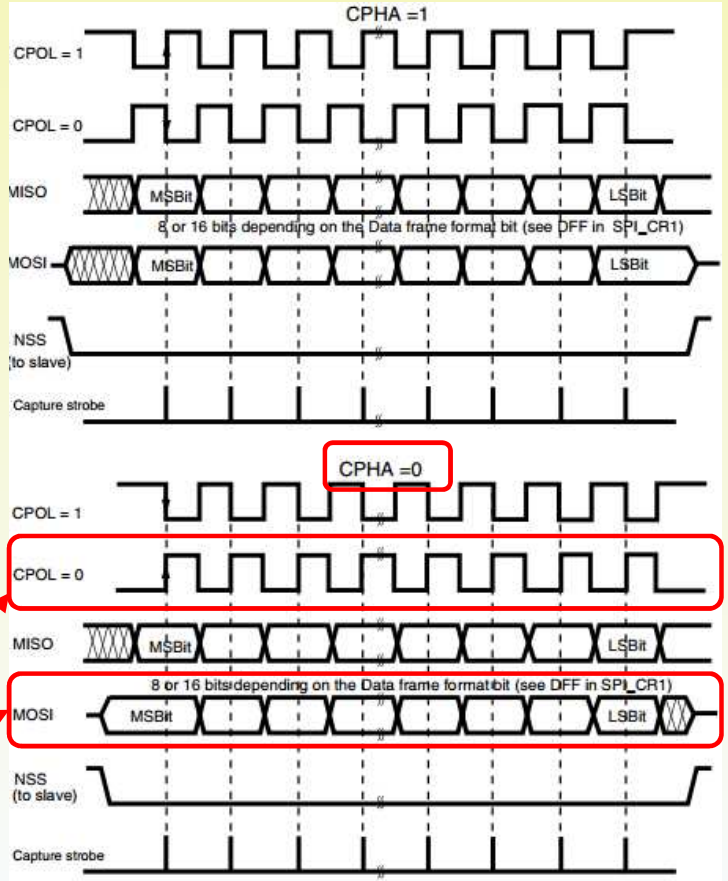
- Funkčnost mají stejnou, ne všechny jsou k dispozici na našem 64-pinovém pouzdře
  - SPI dokáží navíc pracovat v režimu I2S – speciální synchronní protokol typicky pro připojení obvodů audio kodeků
- V RM viz. kapitola 20, str. 549/837
- SPI je synchronní sběrnice
  - Jeden signál hodiny s nastavitelnou aktivní hranou
  - Jeden signál data "ven"
    - Pokud je procesor Master, má význam MOSI = Master Output, Slave Input
  - Jeden signál data "dovnitř" (pro LED nepoužíváme)
    - Pro procesor Master má význam MISO = Master Input, Slave Output
- Na SPI možno připojit více zařízení
  - Mají společné hodiny a datové signály
    - Každé zařízení může komunikovat různou rychlostí, před konkrétním přenosem nutno změnit předdělič
  - Výběr konkrétního obvodu se provádí typicky signálem "Chip Select" (CS)
    - Lze generovat "ručně" – pro více obvodů
    - Příp. to umí SPI HW automaticky (jen pro konkrétní GPIO)
- Procesor může pracovat jako Master (generuje hodiny) i Slave (přijímá hodiny od jiného Mastera)

SPI - komunikace



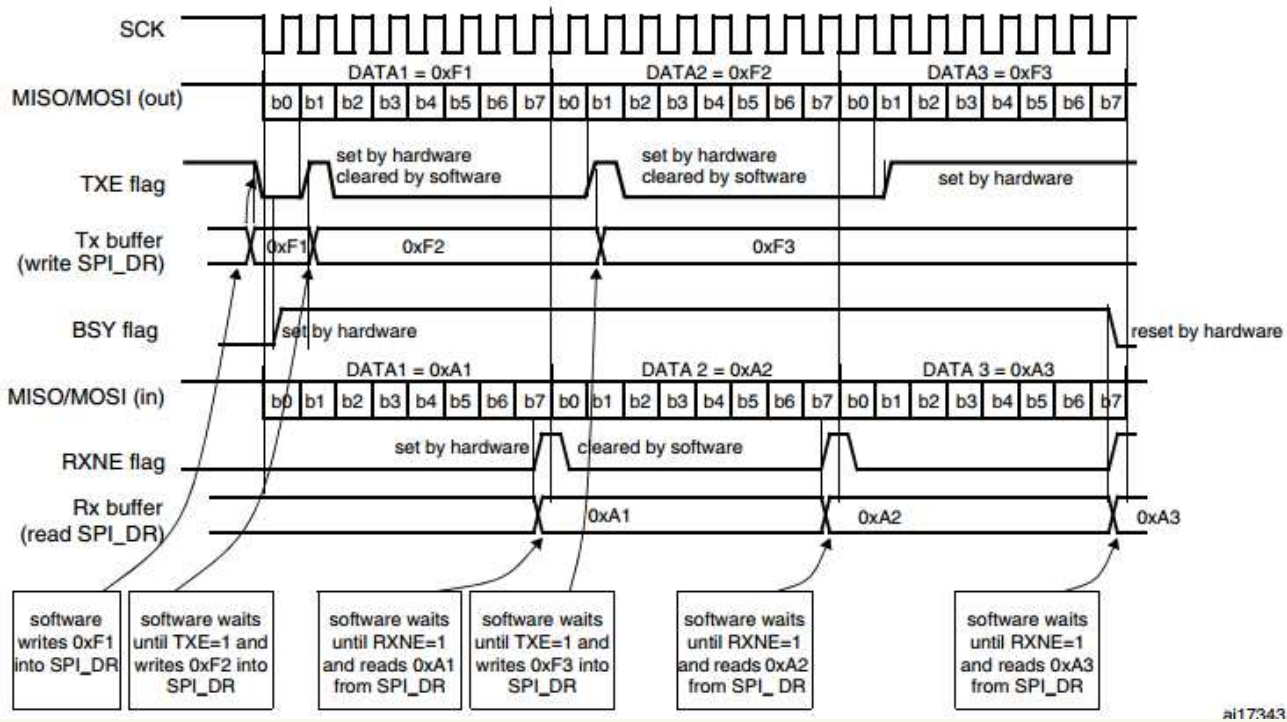
- Při vysílání se přepíše obsah datového vysílacího registru do shift registru, ze kterého jsou data bit po bitu vysílána synchronně s generovanými hodinami
- Zároveň se do přijímacího shift registru přijímají vstupní data a pro 8 (nebo 16) bitech se přepíší do přijímacího datového registru

- Data mohou být platná při vzestupné (L-H) nebo sestupné (H-L) hraně
- Výchozí stav hodin může být log. 0 nebo log. 1
- Je potřeba vybrat vhodnou kombinaci CPOL (polarita) a CPHA (fáze) podle obrázku 194 v RM na str. 555/837
  - Hledá se nejpodobnější průběh požadovaný periférií – u nás LED budičem
  - Odpovídá stav log. 0 v klidu a aktivní vzestupná hrana



SPI – časování a příznaky

- Při vysílání se uplatní příznaky nastavované HW:
  - TXE = Volný vysílací datový registr
    - Možno zapsat další data
    - Při zápisu nových dat se vynuluje než se začnou vysílat
  - BSY = Busy = probíhá vysílání
    - Nastaven během vysílání
    - POZOR, nastavuje se s se zpožděním po začátku vysílání prvního bitu



  - Příklad sice pro CPOL=1 a CPHA = 1, ale příznaky jsou stejné

SPI – řídící registr CR1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BIDI MODE	BIDI OE	CRC EN	CRC NEXT	DFF	RX ONLY	SSM	SSI	*LSB FIRST*	SPE	BR [2:0]			MSTR	CPOL	CPHA
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

- **BIDIMODE** a **BIDIOE** – řídí režimy obousměrného přenosu
  - Defaultní stav log. 0 = nejčastější obousměrný přenos
- **DFF** = Data Frame Format
  - Log. 0 = 8-bit, log. 1 = 16-bit
- **SSM** = SW Slave management a **SSI** = Internal Slave Select
  - Bity nastavují práci s řízením slave režimu, pro běžný provoz s manuálním řízením "chipselectu" nutno nastavit oba na log. 1
- **LSBFIRST** – defaultně 0 odpovídá MSB First
- **SPE** = SPI Enable
  - Doporučeno nastavit jako poslední
- **BR** = Baud Rate – dělič vstupních hodin
  - 000 odpovídá /2, 001 - /4, ... 111 - /256
- **MSTR** = Master Selection – log. 1 pro režim Master
- **CPOL** = Clock Polarity a **CPHASE** = Clock Phase
  - Nutno zvolit vhodnou kombinaci podle požadavku periférie a obrázku 194

SPI – registr CR2 a stavový SR

- Řídící registr CR2 slouží k povolení přerušení a nastavení DMA přenosů

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								TXEIE	RXNEIE	ERRIE	FRF	Res.	*SSOE*	TXDMAEN	RXDMAEN
								rw	rw	rw	rw		rw	rw	rw

- Stavový registr SR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved							FRE	BSY	OVR	MODF	CRC ERR	UDR	CHSIDE	TXE	RXNE
							r	r	r	r	rc_w0	r	r	r	r

  - BSY = Busy
    - Probíhá přenos – viz. obrázek časování
  - OVR = Overrun
    - Nebyla vyčtena přijatá dat před dokončením přijetí dalších dat
  - TXE = Transmit Buffer Empty
    - Možno zapsat nová data do vysílacího datového registru DR
  - RXNE = Receive Buffer Not Empty
    - Nová data v přijímacím datovém registru (opět DR)
    - V obousměrném přenosu se vždy během vysílání "něco" přijme

SPI – inicializace HW

- Funkci připravíme snadno přepínatelně mezi SPI a GPIO řízením
  - Testujeme existenci definovaného symbolu USE_SPI
  - PA8 (LE) a PA9 (OE) jsou stejné
  - PA5 (CLK) a PA7 (data) nastavíme Alternativní režim a AF mód 5
    - Viz. DS str. 47 – "Table 9. Alternate function mapping"
- SPI1 je na APB2
- CR1 nastavujeme režim Master a "povinné" SSM a SSI
  - BR = 0, tj. /2 – tedy SPI hodiny 8MHz (pro 16MHz takt sběrnice)
- V CR2 nic nepotřebujeme
- Na závěr povolíme bit SPE v CR1

```
#define USE_SPI

void InitSPILED(void)
{
  Nucleo_SetPinGPIO(GPIOA, 8, ioPortOutputPP);     // SHIFT - LE
  GPIOWrite(GPIOA, 8, 0);
  Nucleo_SetPinGPIO(GPIOA, 9, ioPortOutputPP);     // SHIFT - OE
  GPIOWrite(GPIOA, 9, 0);
#ifdef USE_SPI
  Nucleo_SetPinGPIO(GPIOA, 5, ioPortAlternatePP);  // SHIFT - CLK
  Nucleo_SetAFGPIO(GPIOA, 5, 5);            // AF05 je SPI1_SCK
  Nucleo_SetPinGPIO(GPIOA, 7, ioPortAlternatePP);  // SHIFT - CLK
  Nucleo_SetAFGPIO(GPIOA, 7, 5);            // AF05 je SPI1_MOSI

  if (!(RCC->APB2ENR & RCC_APB2ENR_SPI1EN))
  {
    RCC->APB2ENR |= RCC_APB2ENR_SPI1EN;
    RCC->APB2RSTR |= RCC_APB2RSTR_SPI1RST;
    RCC->APB2RSTR &= ~RCC_APB2RSTR_SPI1RST;
  }

  SPI1->CR1 = SPI_CR1_MSTR | SPI_CR1_SSI | SPI_CR1_SSM;
     // CPOL = 0, CPHA = 0, DFF = 0 = 8b data, BR = 0 = clk/2
  SPI1->CR2 = 0;              // zadne preruseni ani DMA
  SPI1->CR1 |= SPI_CR1_SPE;   // enable
#else
  Nucleo_SetPinGPIO(GPIOA, 5, ioPortOutputPP);     // SHIFT - CLK
  GPIOWrite(GPIOA, 5, 0);
  Nucleo_SetPinGPIO(GPIOA, 7, ioPortOutputPP);     // SHIFT - SDI
#endif
}
```

SPI – odeslání bajtu

```
void WriteSPILED(uint8_t val)
{
#ifdef USE_SPI
  //TODO otestuj, zda je odvysilano predchozi
  SPI1->DR = val;

  while(!(SPI1->SR & SPI_SR_TXE) || (SPI1->SR & SPI_SR_BSY))
    ;

  GPIOWrite(GPIOA, 8, 1);                    // puls na LE
  GPIOWrite(GPIOA, 8, 0);
#else
... Puvodni odesilani 8 bitu cyklem
#endif
}
```

- Opět přepínatelné SPI/GPIO
- Sekvence jednoduchá
  - Zapíšu nová data k odeslání
    - Pokud se nic nevysílá, začnou se ihned vysílat
  - Čekám na doodvysílání
  - Udělám LE puls
  - Netestujeme, zda je odvysíláno, protože celá funkce končí ve stavu "hotovo"
- **POZOR** - čekání na odvysílání dle obrázku časování
  - TXE příznak je nastaven do log. 1 v okamžiku, kdy se začne vysílat a je volný DR
    - Tj. během vysílání prvního bitu
  - BSY příznak je v log. 1 během vysílání, což je požadováno
    - Ale nastavuje se až se zpožděním po začátku vysílání a while cyklus by podmínku vyhodnotil moc brzy (než bude BSY = 1)
  - Nutná složená podmínka – "dokud není TXE v log. 1" nebo "dokud je BSY v log. 1"
    - Při prvním průchodu "zabere" text TXE, v dalších již bude BSY aktivní a stav TXE nás již "nezajímá", po skončení vysílání BSY přechází do log. 0 a cyklus končí

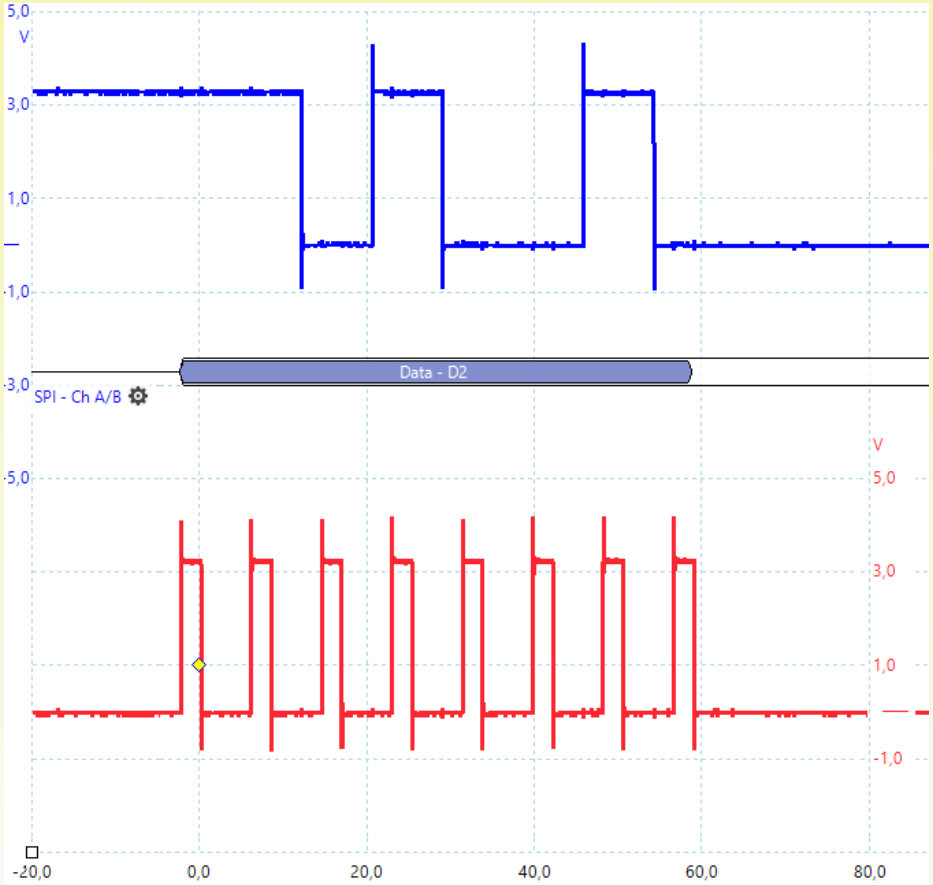
Přesun funkcí do MPP_SHIELD modulu

- Pro možnost dalšího použití umístíme do "shield" modulu
  - Těla funkcí do mpp_shield.c
    - Včetně #define USE_SPI – je to "lokální" definice tohoto modulu
  - Deklarace funkcí do mpp_shield.h

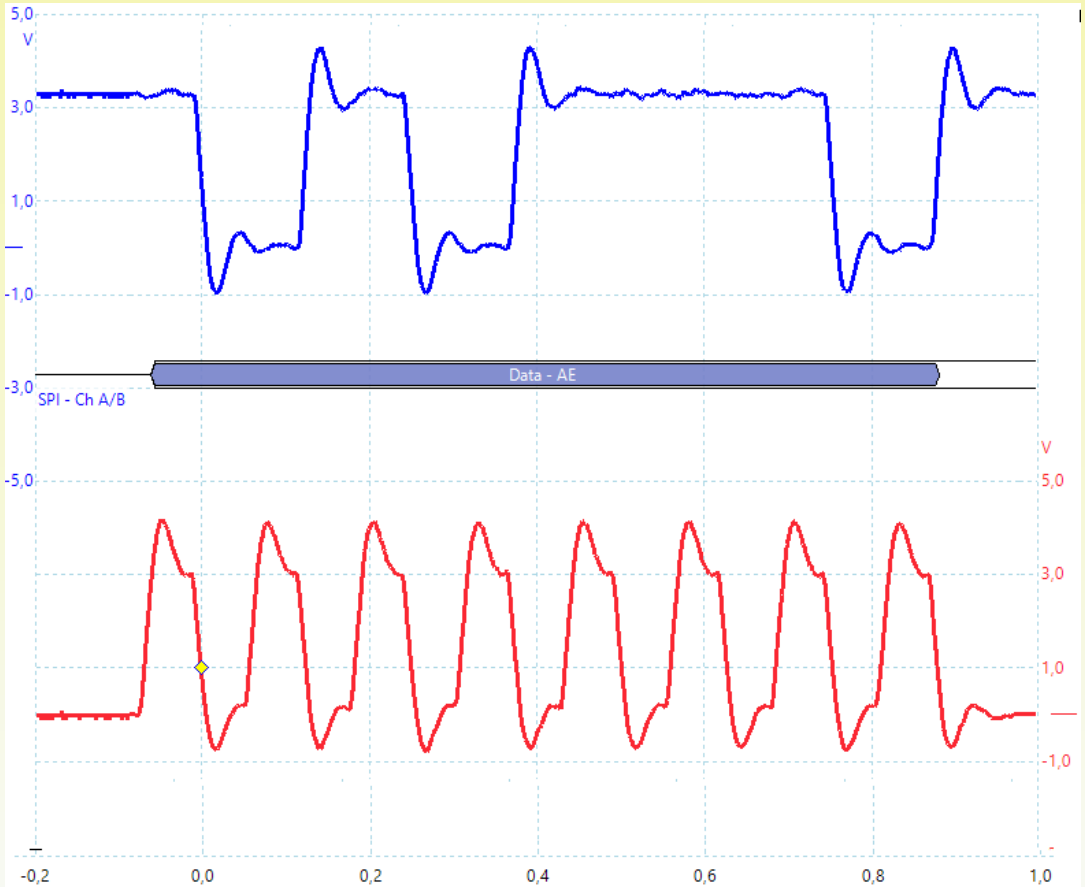
```
...
void InitSPILED(void);
void WriteSPILED(uint8_t val);
...
```

Porovnání reálného časování

GPIO – 20us/dílek, clk = 119kHz	SPI – 200ns/dílek, clk = 7,970MHz



Channel	Name	Value	Min	Max	Average	σ	Capture Count	Span
A	Frequency		---	---	---	---		Between rulers
A	Cycle Time	25,22 µs	16,8 µs	50,4 µs	30,68 µs	11,33 µs	20	Whole trace
A	Frequency	39,65 kHz	19,84 kHz	59,52 kHz	37,48 kHz	14,61 k...	20	Whole trace
B	Frequency	119 kHz	118,9 kHz	119,1 kHz	119 kHz	43,74 Hz	20	Whole trace



Channel	Name	Value	Min	Max	Average	σ	Capture Count	Span
A	Frequency		---	---	---	---		Between rulers
A	Cycle Time	251 ns	249 ns	752 ns	432,3 ns	174,9 ns	20	Whole trace
A	Frequency	3,984 MHz	1,33 MHz	4,016 MHz	2,701 MHz	1,053 M...	20	Whole trace
B	Frequency	7,955 MHz	7,955 M...	7,991 MHz	7,973 MHz	10,4 kHz	20	Whole trace

Shrnutí domácí přípravy – VII

- Prozkoumali jsme datasheet k budiči LED se sériovým výstupem
  - Dokážeme vyčíst potřebu generovaných signálů
  - Dokážeme najít časové informace o průběhu signálů včetně max. frekvence hodin
- Umíme generovat průběh podobný SPI programově pomocí GPIO operací
- Umíme odpovídající průběhy generovat pomocí bloku SPI
  - Umíme nastavit CRx registry – např. fáze a polarita
  - Umíme využívat příznaky ve stavovém registru SR
- Máme připravené funkce pro inicializaci a zápis dat s možností změnit přístup přes GPIO nebo přes SPI

Plán domácí přípravy

I. Instalace prostředí, ... První program – blikání LED na GPIO
II. Zobecnění GPIO operací
III. Systick
IV. Timer
V. UART
VI. Rezerva
VII. SPI
VIII. PWM
IX. A/D
X. I2C (využití "knihovny")
XI. Časování – HSI/HSE/PLL
XII. Rezerva

Cíle přípravy

- Použít HW blok čítače/časovače pro generování PWM průběhu
  - Využívá COMpare funkcionality CAPCOM jednotky
  - Na každý čítač/časovač je "napojeno" několik CAPCOM kanálů

- Generovat proměnnou frekvenci se střídou 50% pro zvukový výstup
- Generovat proměnnou střídu při konstantní frekvenci/periodě a řídit tak jas RGB LED
  - V rámci RGB barevného prostoru je tam možno vytvořit libovolnou barvu

CAPCOM – CAPture/COMpare blok

- Většina moderních mikrokontrolérů má k samotnému bloku čítače ještě bloky (zvané kanály) CAPCOM, které mohou pracovat v jednom z režimů
  - Capture = při nastavené vnější události vstupu se uloží stav čítače do datového registru kanálu
    - Je možno si hodnotu "později" vyzvednout (např. po interruptu)
    - Nezavádí se zpoždění při zpracování interruptu = máme přesný "timestamp" události
  - Compare = při shodě hodnoty datového registru kanálu se nastaví stav výstupu
    - Může to být log. 1, log. 0 nebo "toggle"
    - Opět jsou změny přesné na HW úrovni bez zpoždění kódem
  - Obecný pohled viz. přednášky
- Speciálním režimem COMpare bloku pak bývá možnost generování PWM

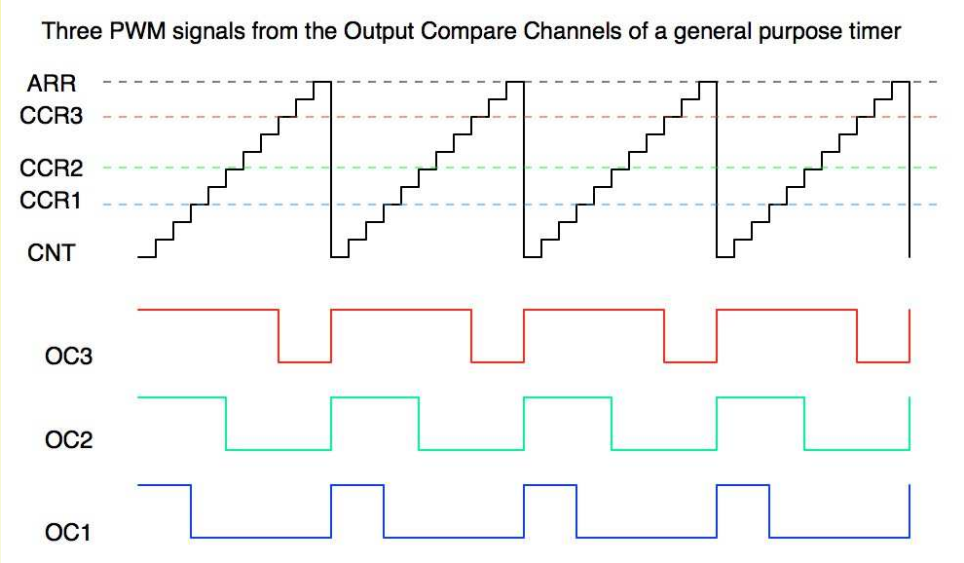
Generování PWM čítačem

- Předpokládejme čítač CNT čítající vzhůru do hodnoty v ARR registru
- Kanálové registry jsou CCRx

- Jakmile dojde k přetečení (ARR→ 0), výstupy pinů patřících ke kanálu přejdou na aktivní úroveň – zde log. 1

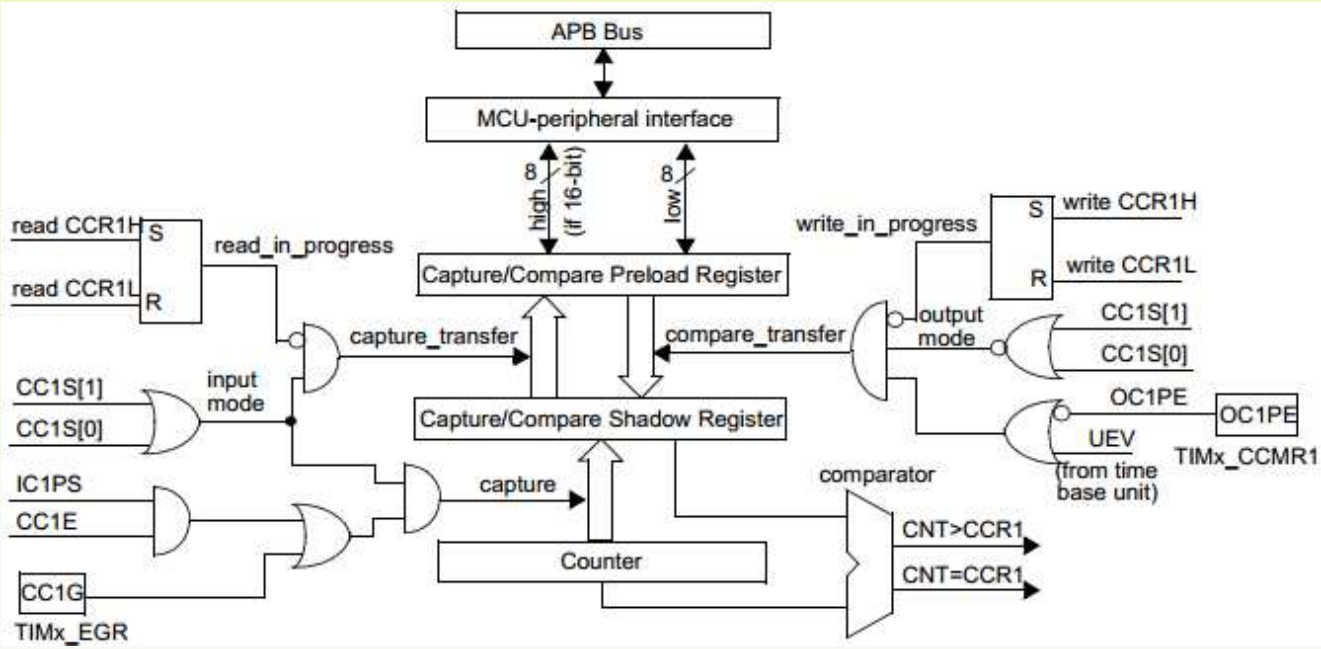
- Dokud je hodnota čítače CNT <= CCRx, zůstává v aktivní úrovni
- Když je CNT větší, překlopí se výstup do neaktivní úrovně (zde log.0)

- Čím větší hodnota CCRx, tím delší puls
  - Pro CCRx >= ARR je stále aktivní
  - Pro CCRx == 0 je stále neaktivní



Blok čítače časovače v STM32F4

- Celkové schéma viz. RM - fig. 87, str. 307/837
- Schéma jednoho kanálu – RM fig. 113, str. 321/837



Registry pro řízení kanálů

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OC2CE	OC2M[2:0]			OC2PE	OC2FE	CC2S[1:0]		OC1CE	OC1M[2:0]			OC1PE	OC1FE	CC1S[1:0]	
IC2F[3:0]				IC2PSC[1:0]				IC1F[3:0]				IC1PSC[1:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OC4CE	OC4M[2:0]			OC4PE	OC4FE	CC4S[1:0]		OC3CE	OC3M[2:0]			OC3PE	OC3FE	CC3S[1:0]	
IC4F[3:0]				IC4PSC[1:0]				IC3F[3:0]				IC3PSC[1:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

- **CCMR1** – CH1,2
- **CCMR2** – CH3,4
  - Jsou prakticky stejné, liší se pro které kanály obsahují řídící bity
  - Kompletní popis významu bitů je vždy u kanálu CH1
  - Uvedené jen bity potřebné pro PWM, zbytek viz. RM
- **CCxS** – mód – 00 = output, 01,10 a 11 režimy vstupní
- **OCxM** – režim – významy podle módu, pro výstupní:
  - 000 – frozen – výstup se neřídí, použito jen časování
  - …
  - 110 – PWM 1 – aktivní úroveň pro CCRx <= CNT
  - 111 – PWM 2 – aktivní úroveň pro CCRx > CNT (= inverzní režim)

Registry pro řízení kanálů

- **CCRx** – hodnota pro porovnání při výstupním režimu nebo stav čítače při události ve vstupním režimu

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CC4NP	Res.	CC4P	CC4E	CC3NP	Res.	CC3P	CC3E	CC2NP	Res.	CC2P	CC2E	CC1NP	Res.	CC1P	CC1E
rw		rw	rw	rw		rw	rw	rw		rw	rw	rw		rw	rw

- **CCER** – enable+control
  - **CCxP** – polarita – jaká log. úroveň je aktivní
    - 0 = active high (log. 1)
    - 1 = active low (log. 0)
  - **CCxE** – output enable – povolení činnosti kanálu

- **POZOR** – TIM1 má **CCER** rozšířený o bity **CCxNE**
  - Viz. RM kap. 12.4.9.
  - Kanály 1-3 mají "přímé" a negované výstupy a je třeba povolit ten "správný" podle zapojení HW

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved		CC4P	CC4E	CC3NP	CC3NE	CC3P	CC3E	CC2NP	CC2NE	CC2P	CC2E	CC1NP	CC1NE	CC1P	CC1E
		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Speciální rozšíření TIM1

- Časovač TIM1 má více funkcí a je určen pro motorové aplikace
  - Kanály mají komplementární výstupy
  - Je možné nastavit dead-time kvůli motorovým můstkům
  - ...

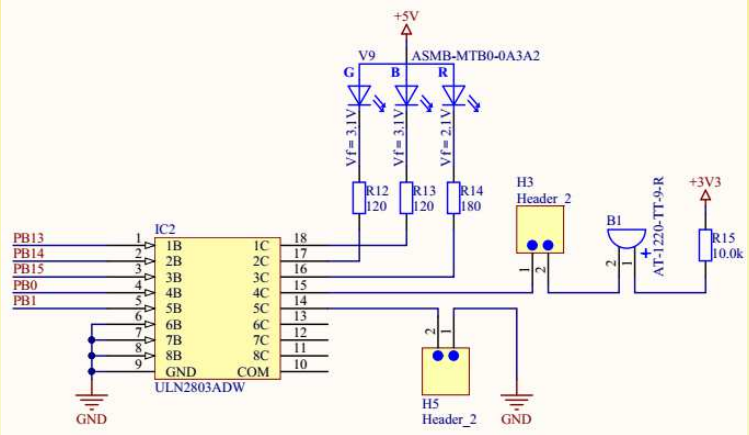
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MOE	AOE	BKP	BKE	OSSR	OSSI	LOCK[1:0]		DTG[7:0]							
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

- Kromě rozšířeného registru **CCER** má ještě speciální registr **BDTR**
  - **MOE** – log. 1 = povolení OC výstupů
    - Určeno k jednoduchému "odpojení" motorových budičů (můstků)
    - Pro běžné použití kanálů TIM1 je třeba nezapomenout bit nastavit

Využití PWM na MPP-shield

- Tabulka 9 v DS ukazuje mapování výstupů kanálů a vývodů podle GPIO portů včetně příslušné hodnoty do AFR registrů

- RGB LED
  - PB13 – modrá – TIM1, kanál CH1
  - PB14 – zelená – TIM1, kanál CH2
  - PB15 – červená – TIM1, kanál CH3
  - Všechny výstup jsou "N", takže je nutno povolit správný bit v CCER
- Pípák
  - PB0 –TIM3 a kanál CH3
  - Možná volba kanálů, ale TIM1 je pro RGB LED a zde chci měnit periodu PWM nezávisle na LEDkách



Port		AF00	AF01	AF02	AF03	
		SYS_AF	TIM1/TIM2	TIM3/ TIM4/ TIM5	TIM9/ TIM10/ TIM11	I2
Port B	PB0	-	TIM1_CH2N	TIM3_CH3	-	
	PB1	-	TIM1_CH3N	TIM3_CH4	-	
	PB2	-				
	PB13	-	TIM1_CH1N	-	-	
	PB14	-	TIM1_CH2N	-	-	
	PB15	RTC_50Hz	TIM1_CH3N	-	-	

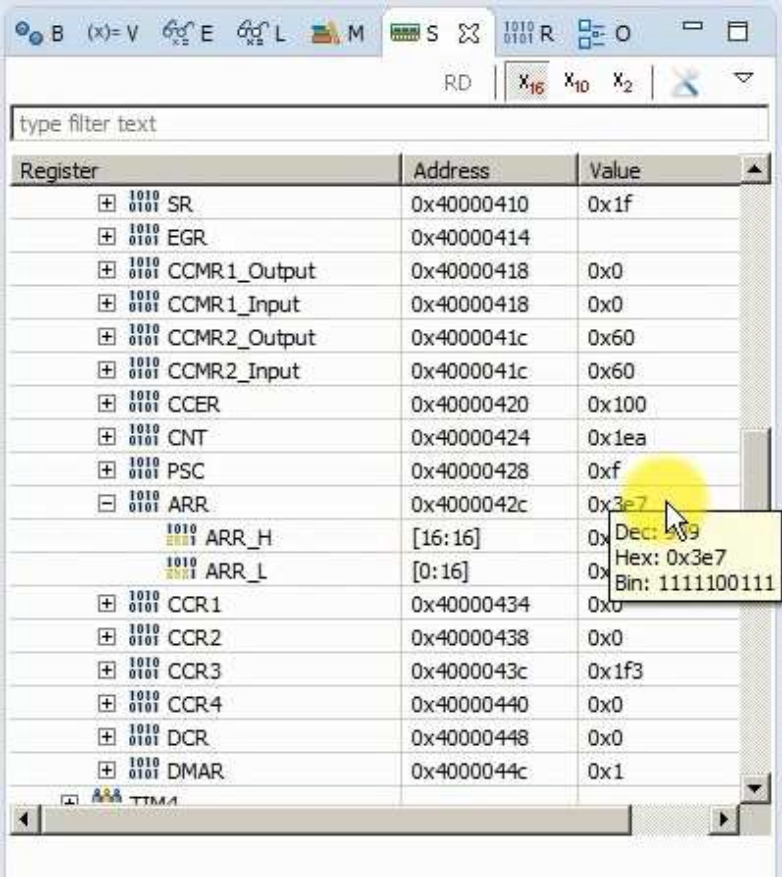
Nastavení generování tónu pomocí PWM

- Takt APB1 zatím shodný s SystemCoreClock
- Povolit periférii v APBxENR
- Bit DIR v CR1 je = 0, tj. čítač vzhůru
- Předdělička nastavena na 1MHz
- Perioda signálu (přetečení CNT) je 1000x, tj. 1ms
- Nastavit PB0 jako alternativní 2
- Nastavení bitů OCxM v CCRx
  - Napřed vynulovat všechny 3
  - Nastavit na 110 pomocí definovaných konstant
- Povolit bit CCxE
- Komparační hodnotu na stavit na ARR/2
  - Odpovídá 50% střídě

```
{
  uint32_t apb1 = SystemCoreClock;   //TODO dopln podle RCC

  if (!(RCC->APB1ENR & RCC_APB1ENR_TIM3EN))
  {
    RCC->APB1ENR |= RCC_APB1ENR_TIM3EN;
    RCC->APB1RSTR |= RCC_APB1RSTR_TIM3RST;
    RCC->APB1RSTR &= ~RCC_APB1RSTR_TIM3RST;
  }

  TIM3->CR1 = 0   // DIR = 0 - upcounter
    | TIM_CR1_ARPE;   // bufferovany zapis do ARR
  TIM3->CR2 = 0;

  TIM3->PSC = apb1 / 1E6 - 1;     // 1MHz - 1us
  TIM3->ARR = 1000 - 1;           // 1kHz - 1ms preteceni

  TIM3->CR1 = TIM_CR1_CEN;

  Nucleo_SetPinGPIO(GPIOB, 0, ioPortAlternatePP);
  Nucleo_SetAFGPIO(GPIOB, 0, 2);    // AF02 = TIM3_CH3

  TIM3->CCMR2 &= ~TIM_CCMR2_OC3M;
  TIM3->CCMR2 |= TIM_CCMR2_OC3M_2 | TIM_CCMR2_OC3M_1;
      // 110 - PWM mode 1

  TIM3->CCER |= TIM_CCER_CC3E;
  TIM3->CCR3 = TIM3->ARR / 2;
}
```

Ovládání PWM při debug

- Protože generování PWM výstupu řídí HW samostatně, je možno program zastavit a měnit "ručně" registry – změny se projeví ihned
  - Záložka **SFRs** v Debug perspektivě
    - Příp. přidat v menu View-SFRs
  - Zastavit program (pauza nebo breakpoint)
  - Rozbalit **TIM3**
  - Změnit obsah **ARR**
    - Defaultně je 0x (=999)
    - Nutno zadávat hexadecimálně
    - Po změně se ihned mění frekvence/zvuk
    - Změna střídy v CCRx registru
      - Mění jen charakter zvuku



Synchronizace změn ARR registru

- Při změně výšky tónu (frekvence) je třeba změnit 2 registry
  - **ARR** obsahuje periodu v mikrosekundách (z PSC jde 1MHz)
  - **CCRx** by měl být ARR/2 pro 50% střídu

- Pokud se změní hodnota ARR během COMpare funkcí, může dojít k chybě v jednom taktu
  - Řešením je měnit **ARR** synchronně po použití pří přetečení
  - Bit **ARPE** v registru **CR1** – nastaven v ukázkovém kódu

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved						CKD[1:0]		ARPE	CMS		DIR	OPM	URS	UDIS	CEN
						rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Generování jasu RGB LED

- Prakticky stejné jako u TIM3
- Navíc třeba nastavit bit **MOE** v registru **BDTR**
- Povolují se bity **CCxNE** v **CCER**
- V kódu je střída 0, v poznámce 50%
- Vyzkoušejte v debug perspektivě měnit hodnoty **CCRx**
  - Pro malá čísla se jas mění viditelněji
  - Možno vytvořit libovolnou barvu

```
{
  uint32_t apb2 = SystemCoreClock;   //TODO dopln podle RCC

  if (!(RCC->APB2ENR & RCC_APB2ENR_TIM1EN))
  { ... Povoleni ... }

  TIM1->CR1 = 0   // DIR = 0 - upcounter
      | TIM_CR1_ARPE;   // bufferovany zapis do ARR
  TIM1->CR2 = 0;

  TIM1->PSC = apb2 / 1E6 - 1;     // 1MHz - 1us
  TIM1->ARR = 256 - 1;            // 256us preteceni

  TIM1->CR1 = TIM_CR1_CEN;
  TIM1->BDTR |= TIM_BDTR_MOE;

  Nucleo_SetPinGPIO(GPIOB, 13, ioPortAlternatePP);  // BLUE
  Nucleo_SetAFGPIO(GPIOB, 13, 1);    // AF01 odpovida TIM1_CH1N

  TIM1->CCMR1 &= ~TIM_CCMR1_OC1M;
  TIM1->CCMR1 |= TIM_CCMR1_OC1M_2 | TIM_CCMR1_OC1M_1;  // 110 – PWM 1

  TIM1->CCER |= TIM_CCER_CC1NE;   // povol komplementarni vystup
  TIM1->CCR1 = 0; //TIM1->ARR / 2;

  Nucleo_SetPinGPIO(GPIOB, 14, ioPortAlternatePP);  // GREEN
  Nucleo_SetAFGPIO(GPIOB, 14, 1);    // AF01 odpovida TIM1_CH2N

  TIM1->CCMR1 &= ~TIM_CCMR1_OC2M;
  TIM1->CCMR1 |= TIM_CCMR1_OC2M_2 | TIM_CCMR1_OC2M_1;  // 110 - PWM 1

  TIM1->CCER |= TIM_CCER_CC2NE;   // povol komplementarni vystup
  TIM1->CCR2 = 0; //TIM1->ARR / 2;

  Nucleo_SetPinGPIO(GPIOB, 15, ioPortAlternatePP);  // RED
  Nucleo_SetAFGPIO(GPIOB, 15, 1);    // AF01 odpovida TIM1_CH3N

  TIM1->CCMR2 &= ~TIM_CCMR2_OC3M;
  TIM1->CCMR2 |= TIM_CCMR2_OC3M_2 | TIM_CCMR2_OC3M_1;  // 110 - PWM 1

  TIM1->CCER |= TIM_CCER_CC3NE;   // povol komplementarni vystup
  TIM1->CCR3 = 0; //TIM1->ARR / 2;
}
```

Pozvolné rozsvěcení LED

```
while(1)
{
  if (_ticks >= tm2)
  {
    tm2 = _ticks + 50;        // 50ms

    b += 16;

    TIM1->CCR1 = b;
...
```

- Odvozeno od 1ms "tiků" Systicku
- Proměnná b je 8-bitová – neřešíme přetékání
- Přímo lze dát do CCRx
- Hodnota ARR je 255, takže čítáme 256 kroků, tj. 8-bitové rozlišení střídy generovaného PWM průběhu

Shrnutí domácí přípravy – VIII

- Umíme použít CAPCOM jednotku čítače/časovače pro generování
  - Signálu se střídou 50% a proměnnou frekvencí/periodou
    - Ověřeno na výstupu pro "pípák"
  - Signálu s fixní periodou a proměnnou střídou
    - Ověřeno na složkách RGB LED
      - Postupné rozsvěcení
      - Nastavení libovolné barvy (např. v debug perspektivě při zastaveném programu)
- Viděli jsme, každý čítač má až kanály s CAPCOM funkcionalitou
- Viděli jsme, že TIM1 má některé speciální vlastnosti, které musíme zohlednit v programu

Plán domácí přípravy

I. Instalace prostředí, ... První program – blikání LED na GPIO
II. Zobecnění GPIO operací
III. Systick
IV. Timer
V. UART
VI. Rezerva
VII. SPI
VIII. PWM
IX. A/D
X. I2C (využití "knihovny")
XI. Časování – HSI/HSE/PLL
XII. Rezerva

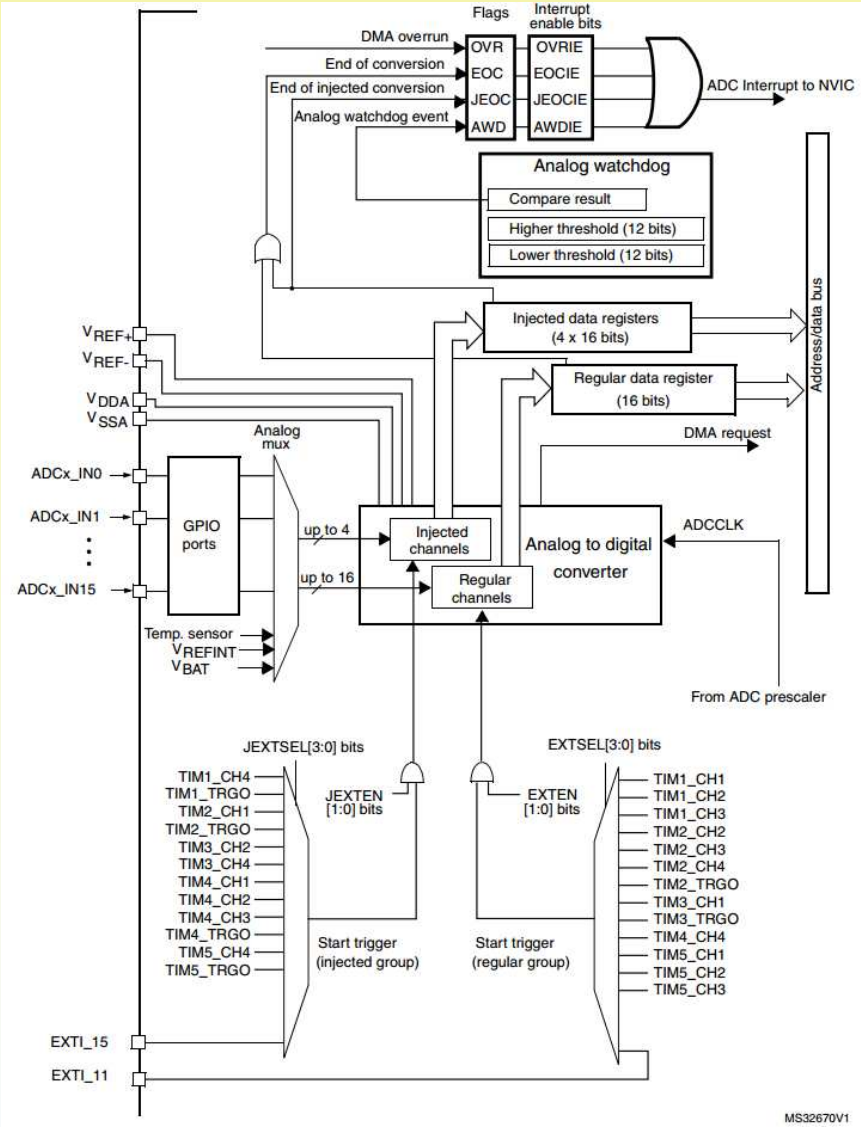
Cíle přípravy

- Prozkoumat základní funkce bloku A/D převodníku
  - Řídící registry, časování
  - Režim jednorázového měření
  - SW spuštění a detekce konce převodu
  - Další funkcionalitu zatím nebudeme využívat

- Naučit se měřit externí analogovou hodnotu
  - Ověřit na připojeném potenciometru na desce MPP shield

Blok A/D převodníku

- Základní části viz. RM – fig. 31, str. 210/836
- Převodník má až 16 vnějších vstupů (GPIO) a 3 interní
  - Výběr realizuje analogový multiplexer
- Taktování převodu jde přes předdělič z APBx sběrnice
- Spouštění převodu je možné buď SW nebo podle události
  - Výběr "start trigger"
  - Typicky odvozené od časovače nebo vstupu
- Dokončení převodu možno
  - Testovat SW bit "konec převodu"
  - Využít přerušení
  - Využít DMA – typicky automaticky ukládá měřené hodnoty do pole pro další zpracování
- Dva typy měření
  - Regular
    - Měří se právě jedna hodnota, příp. sekvence
    - Každé další měření se musí spustit
    - Využívá jeden datový registr
  - Injection – skupina hodnot
    - Změří se až 4 hodnoty jedním spuštěním
    - Samozřejmě se měří postupně, ale bez dalších prodlev
    - Má až 4 datové registry
    - Řídící bity typicky začínají J



Řídící registry A/D CR1 a CR2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved					OVRIE	RES		AWDEN	JAWDEN	Reserved					
					rw	rw	rw	rw	rw						
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DISCNUM[2:0]			JDISCEN	DISCEN	JAUTO	AWDSGL	SCAN	JEOCIE	AWDIE	EOCIE	AWDCH[4:0]				
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
reserved	SWSTART	EXTEN		EXTSEL[3:0]				reserved	JSWSTART	JEXTEN		JEXTSEL[3:0]			
	rw	rw	rw	rw	rw	rw	rw		rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
reserved				ALIGN	EOCS	DDS	DMA	Reserved						CONT	ADON
				rw	rw	rw	rw							rw	rw

- Využíváme jen některé bity
  - Začínající **J** patří k "injection" režimu – nyní nepoužíváme
  - Řada dalších pro přerušení, DMA, automatické spuštění převodu nebo analogový watchdog (= hlídání zda je vstupní hodnota v rozsahu)
- **RES** – rozlišení
  - 00 = 12bitů, 01 = 10b, 10 = 8b, 11 = 6b
  - Počet bitů ovlivňuje rychlost (více trvá déle)
- **SCAN** – povolení práce s **SQR** kanály – nutno nastavit 1
- **SWSTART** – programové spuštění převodu "regular" kanálů
- **ALIGN** – zarovnání hodnoty v 16b datovém registru
  - Log. 0 = vpravo – viz. RM kap. 11.4
- **ADON** – zapnutí A/D převodu
  - Ideálně po dokončení všech nastavení nastavit do log. 1

Další registry A/D

- **SMPR1** a **SMPR2** – určení doby převodu
  - Každý kanál řízen nezávisle "svými" 3 bity
  - Doby převodu viz. RM kap. 11.12.4
    - Nastavuje se počet taktů, skutečná doba převodu se vypočte dle RM kap. 11.5 jako T_{conv} = Sampling time + 12 cycles

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved					SMP18[2:0]			SMP17[2:0]			SMP16[2:0]			SMP15[2:1]	
					rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SMP15_0	SMP14[2:0]			SMP13[2:0]			SMP12[2:0]			SMP11[2:0]			SMP10[2:0]		
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved		SMP9[2:0]			SMP8[2:0]			SMP7[2:0]			SMP6[2:0]			SMP5[2:1]	
		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SMP 5_0	SMP4[2:0]			SMP3[2:0]			SMP2[2:0]			SMP1[2:0]			SMP0[2:0]		
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

- **SQR1-3** určují, ze kterých kanálů se bude měřit v sekvenci
  - Hodnota **L** v **SQR1** = kolik konverzí má sekvence
    - 0000 = 1 konverze
  - Hodnota **SQx** v **SQRx** určuje ze kterého kanálu se provede konverze
    - Je 5-bitová, protože je 16 externích + 3 interní kanály (0 – 18)
    - Pro jeden převod tedy použít SQ1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								L[3:0]				SQ16[4:1]			
								rw	rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SQ16_0	SQ15[4:0]					SQ14[4:0]					SQ13[4:0]				
rw	rw	rw	rw	rw	rw	rw	rw				rw	rw	rw	rw	rw

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved		SQ12[4:0]					SQ11[4:0]					SQ10[4:1]			
		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SQ10_0	SQ9[4:0]					SQ8[4:0]					SQ7[4:0]				
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved		SQ6[4:0]					SQ5[4:0]					SQ4[4:1]			
		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SQ4_0	SQ3[4:0]					SQ2[4:0]					SQ1[4:0]				
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Další A/D registry

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										OVR	STRT	JSTRT	JEOC	EOC	AWD
										rc_w0	rc_w0	rc_w0	rc_w0	rc_w0	rc_w0

- **SR** – status registr
  - Příznaky stavu převodníku
  - **EOC** – dokončení převodu
    - Nastaven HW po dokončení "regular" převod
    - Nulován programově nebo automaticky vyčtením datového registru

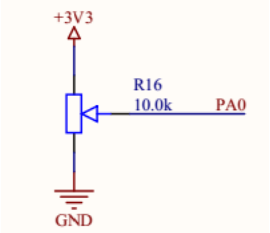
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								TSVREFE	VBATE	Reserved				ADCPRE	
								rw	rw					rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved															

- **CCR** – společné řízení všech ADC v kontroléru
  - **TSVREFE** – povolení teplotního senzoru a interní reference
    - Kanály AD16 - AD18
  - **ADCPRE** – předdělič z APBx taktu
    - 00 = clk / 2

Inicializace A/D převodníku

- Kanál AD0 odpovídá portu PA0
  - Je k němu připojen potenciometr
- ADC1 připojen na APB2
  - POZOR, v **RSTR** registru je bit **ADCRST**, protože je společný pro všechny ADC (pokud by byly na čipu)
- V **CR1** nutno nastavit jen **SCAN**
  - Bity **RES** = 00 odpovídají 12b převodu
- V **CR2** se zapnutí převodníku – bit **ADON** – provede až po komplet nastavení
  - Bit **ALIGN** = 0 odpovídá zarovnání vpravo
- V **SPMRx** nastavujeme dobu převodu pouze pro kanál 0
  - Bity **SMP0** v **SMPR2**
  - Hodnota 010 odpovídá 28 cyklů
- V **SQRx** nastavíme sekvenci
  - **L** = 0000, tj. jedna konverze
  - **SQ1** = 00000, tj. z kanálu AD0
- V registru **CCR** necháme 0
  - **ADCPRE** = 00, tedy předdělič z APBx dělí 2
  - **POZOR**, **CCR** je společný všem ADCx, proto je ve speciálním bloku ADC (bez čísla)

```
Nucleo_SetPinGPIO(GPIOA, 0, ioPortAnalog);

if (!(RCC->APB2ENR & RCC_APB2ENR_ADC1EN))
{
  RCC->APB2ENR |= RCC_APB2ENR_ADC1EN;
  RCC->APB2RSTR |= RCC_APB2RSTR_ADCRST;
  RCC->APB2RSTR &= ~RCC_APB2RSTR_ADCRST;
}

ADC1->CR1 = 0       // RES = 00 = 12bitu
        | ADC_CR1_SCAN;
ADC1->CR2 = 0;      // ALIGN = 0 = right align

ADC1->SMPR1 = 0;
ADC1->SMPR2 = 2;   // 010 pro ch0 = 28 cyklu
  // stejne jako ADC_SMPR2_SMP0_1

ADC1->SQR1 = 0;    // L = 0000 = 1 konverze
ADC1->SQR2 = 0;
ADC1->SQR3 = 0;    // SQ1 = 00000 = AD kanal 0

ADC->CCR = 0;  // neni tepl. sens, neni vbat, ADCPRE = 00 = APB2 / 2

ADC1->CR2 |= ADC_CR2_ADON;      // zapnuti AD
```



Provedení A/D převodu

```
...
  //TODO pripadne nastav jiny kanal do SQR3
  ADC1->CR2 |= ADC_CR2_SWSTART;   // spusteni prevodu

  while(!(ADC1->SR & ADC_SR_EOC))
    ;

  w = ADC1->DR;

  printf("%X %d\r\n", w, w);
...
```

- Předpokládejme inicializovaný A/D převodník
  - **POZOR**, pokud by nebyl spuštěn a neměl "hodiny", bude smyčka čekání na dokončení převodu **NEKONEČNÁ**, protože nikdy se převod ani nespustí
- Spouštění SW nastavením **SWSTART** v **CR2**
- Test dokončení převodu bit **EOC** v **SR**
- Vyčtení hodnoty z registru **DR** – efektivně 16 bitů
  - Max. hodnota pro 12bitů je 0xFFF, resp. v rozsahu 0 – 4095
  - Data zarovnána vpravo

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
DATA[15:0]															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

- Pro použití printf předpokládáme inicializaci sériové komunikace
  - Nastavení USART2 na rychlost 38400
  - Upravené funkce _read a _write v syscalls.c
  - Připojené soubory nucleo_usart.c a .h

Shrnutí domácí přípravy –IX

- Umíme nastavit A/D převodník tak, aby měřil z externího vstupu
  - Spouštění převodu programově
  - Testování dokončení převodu také programově
  - Využíváme "regular" režim a jednorázové měření
- K ověření činnosti využíváme potenciometr připojený na vstup AD0 (odpovídá pinu PA0)

Plán domácí přípravy

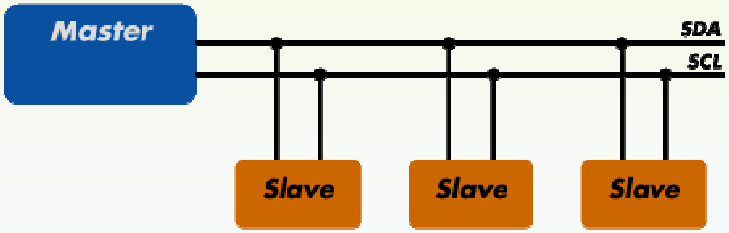
I. Instalace prostředí, ... První program – blikání LED na GPIO
II. Zobecnění GPIO operací
III. Systick
IV. Timer
V. UART
VI. Rezerva
VII. SPI
VIII. PWM
IX. A/D
X. I2C (využití "knihovny")
XI. Časování – HSI/HSE/PLL
XII. Rezerva

Cíle přípravy

- Naučit se pracovat s komunikační sběrnicí I2C
  - Využít předpřipravenou knihovnu (modul) pro naši aplikaci
  - Prozkoumat základní vlastnosti periférie I2C v mikrokontroléru
- Využít připojený vnější obvod – akcelerometr MMA7660
  - 3-osý, citlivost +- 1,5g, 6-bitová hodnota pro každou osu
- Umět přepočítat změřenou hodnotu z vnitřních registrů akcelerometru na fyzikální veličinu – zrychlení
  - Využít hodnotu zrychlení ve všech osách k identifikaci polohy zařízení

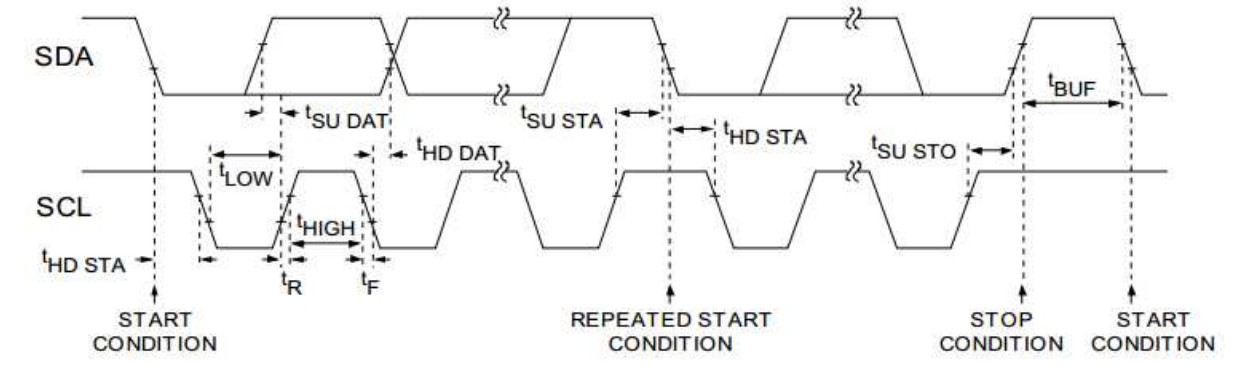
Sběrnice I2C - princip

- I2C sběrnice je synchronní se 2 signály
  - SCK – hodiny generovaném masterem (tj. jeho výstup)
  - SDA – data – obousměrná
    - Přenáší bity jak při zápisu tak při čtení z periférie
    - Typicky musí být jako otevřený kolektor (OpenDrain u STM32)
- Může být připojeno více periférií (slave obvodů)
  - Nejsou žádné výběrové signály (Select) jako u SPI
    - První bajtem přenosu je adresa zařízení, se kterým se bude komunikovat
    - Adresa je typicky 7-bitová, 8. bit je příznak R/W (čtení = log. 1)
  - Rychlost komunikace (hodin) podle nejpomalejší z periférií
    - Typické rychlosti 100kHz nebo 400kHz, většinou se uvádí maximální pro daný obvod



Akcelerometr - komunikace

- Na MPP-shieldu máme použitý akcelerometr (Freescale) MMA7660
  - Detaily viz. MMA7660FC.pdf
- Komunikace orientována registrově, v paketu musím sdělit:
  - Adresu zařízení (pevně dána výrobcem 1001100)
  - Adresu registru, který chci číst/zapisovat
  - Data zapisuji nebo čtu
  - Je možný blokový přístup – dokud generujeme SCK a posíláme nebo přijímáme data, komunikace probíhá s tím, že se interní adresa inkrementuje
- Základní registry popsány v DS včetně časování sběrnice
  - Kap. "SERIAL INTERFACE"
  - Požadované parametry odpovídají "běžné" I2C komunikaci generované HW v mikrokontroléru



Akcelerometr registry

Address	Name	Definition	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
$00	XOUT	6-bit output value X	-	Alert	XOUT[5]	XOUT[4]	XOUT[3]	XOUT[2]	XOUT[1]	XOUT[0]
$01	YOUT	6-bit output value Y	-	Alert	YOUT[5]	YOUT[4]	YOUT[3]	YOUT[2]	YOUT[1]	YOUT[0]
$02	ZOUT	6-bit output value Z	-	Alert	ZOUT[5]	ZOUT[4]	ZOUT[3]	ZOUT[2]	ZOUT[1]	ZOUT[0]
$03	TILT	Tilt Status	Shake	Alert	Tap	PoLa[2]	PoLa[1]	PoLa[0]	BaFro[1]	BaFro[0]
$04	SRST	Sampling Rate Status	0	0	0	0	0	0	AWSRS	AMSRS
$05	SPCNT	Sleep Count	SC[7]	SC[6]	SC[5]	SC[4]	SC[3]	SC[2]	SC[1]	SC[0]
$06	INTSU	Interrupt Setup	SHINTX	SHINTY	SHINTZ	GINT	ASINT	PDINT	PLINT	FBINT
$07	MODE	Mode	IAH	IPP	SCPS	ASE	AWE	TON	-	MODE
$08	SR	Auto-Wake/Sleep and Portrait/Landscape samples per seconds and Debounce Filter	FILT[2]	FILT[1]	FILT[0]	AWSR[1]	AWSR[0]	AMSR[2]	AMSR[1]	AMSR[0]
$09	PDET	Tap Detection	ZDA	YDA	XDA	PDTH[4]	PDTH[3]	PDTH[2]	PDTH[1]	PDTH[0]
$0A	PD	Tap Debounce Count	PD[7]	PD[6]	PD[5]	PD[4]	PD[3]	PD[2]	PD[1]	PD[0]
$0B-$1F	Factory	Reserved	-	-	-	-	-	-	-	-

- Viz. DS
- X, Y, ZOUT – základní datové registry
  - Zrychlení podle dané osy
  - Osa Z typicky svisle = kolmo na čip, resp. plošný spoj
  - V klidu na vše působí gravitační (tíhové) zrychlení 1g
- TILT – speciální stavy/polohy
- MODE – nutno nastavit minimálně bit MODE do log. 1 = aktivní stav (měření)
  - Další bity řídí rychlost měření, citlivost apod.

Akcelerometr datové registry

$00: 6-bits output value X (Read Only when not in Test Mode)
XOUT — X Output

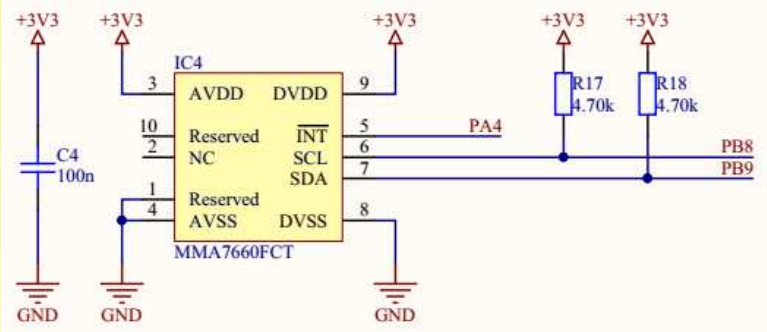
D7	D6	D5	D4	D3	D2	D1	D0
-	Alert	XOUT[5]	XOUT[4]	XOUT[3]	XOUT[2]	XOUT[1]	XOUT[0]
0	0	0	0	0	0	0	0

- Registr obsahuje hodnotu 6-bitovou
  - Max. rozlišení 64 hodnot
  - Uloženo jako dvojkový doplněk, tj. čísla v rozsahu -32 .. 0 .. 31
  - Bit 6 příznak Alert – data se mění
    - Čtení probíhá v okamžiku, kdy se aktualizuje obsah registru
    - Většinou stačí čtení zopakovat znovu pokud je nastaveno
      - Nebo jej ignorovat ("odmaskovat")
- Převodní tabulka viz. "APPENDIX C - MMA7660FC ACQUISITION CODE TABLE"
  - Ukazuje vazbu mezi získanou 6-bitovou hodnotou a skutečnou hodnotou zrychlení, resp. úhlem vůči vodorovné ose
  - Zrychlení reprezentuje výpočet:
    - **Hodnota_se_znaménkem / 21,33**

6-bit result	Binary	2's Comp	g value	Angle X or Y		Angle Z
0	0	0	0.000g	0.00°		90.00°
1	1	1	0.047g	2.69°		87.31°
2	10	2	0.094g	5.38°		84.62°
3	11	3	0.141g	8.08°		81.92°
4	100	4	0.188g	10.81°		79.19°
5	101	5	0.234g	13.55°	Z-axis must be in the range	76.45°
6	110	6	0.281g	16.33°		73.67°
7	111	7	0.328g	19.16°		70.84°
8	1000	8	0.375g	22.02°		67.98°
9	1001	9	0.422g	24.95°		65.05°
10	1010	10	0.469g	27.95°		62.05°
11	1011	11	0.516g	31.04°		58.96°
12	1100	12	0.563g	34.23°		55.77°
13	1101	13	0.609g	37.54°		52.46°
14	1110	14	0.656g	41.01°		48.99°
15	1111	15	0.703g	44.68°		45.32°
16	10000	16	0.750g	48.59°		41.41°
17	10001	17	0.797g	52.83°		37.17°
18	10010	18	0.844g	57.54°		32.46°
19	10011	19	0.891g	62.95°		27.05°
20	10100	20	0.938g	69.64°		20.36°
21	10101	21	0.984g	79.86°		10.14°

Připojení HW, stažení modulu (knihovny)

- Pro komunikaci využit PB8 a PB9
  - Dle DS je na tyto vývody namapován blok I2C1
- Na CW v bloku cvičení je ke stažení připraven soubor nucleo_f4_i2c_v1.zip obsahující
  - nucleo_i2c.c
  - nucleo_i2c.h



Port		AF00	AF01	AF02	AF03	AF04	AF05
		SYS_AF	TIM1/TIM2	TIM3/ TIM4/ TIM5	TIM9/ TIM10/ TIM11	I2C1/I2C2/ I2C3	SPI1/I2S1 PI2/ I2S2/SPI3 I2S3
Port B	PB8	-	-	TIM4_CH3	TIM10_CH1	I2C1_SCL	-
	PB9	-	-	TIM4_CH4	TIM11_CH1	I2C1_SDA	SPI2_NSS 2S2_WS

```
...
#include "stm_core.h"

#define MAX_TIMEOUT   10000

typedef enum {i2cSpeed100k = 100000, i2cSpeed400k = 400000} i2cSpeed;

bool InitI2C1(i2cSpeed);
bool I2C1_WriteByte(uint8_t devAdr, uint8_t regAdr, uint8_t val);
uint8_t I2C1_ReadByte(uint8_t devAdr, uint8_t regAdr);
bool I2C1_ReadBytes(uint8_t devAdr, uint8_t regAdr, uint8_t *pbuf, uint32_t len);
...
```

Funkce I2C modulu - inicilizace

- Předpokládáme použití I2C1 s GPIO podle Nucleo desky (PB8 a PB9)
  - Inicializační funkce z stm_core modulu
  - POZOR, SDA (PB9) musí být ioPortAlternateOC

- bool InitI2C1(i2cSpeed spd)
  - Parametr určuje rychlost hodin v Hz, tedy 100k nebo 400k, používá se k výpočtu hodnot registrů I2C
  - Některé konstrukce ve funkci jsou inspirovány příklady knihoven od ST
  - Hodiny sběrnice APB1 se vypočítávají podle skutečného stavu
    - Komunikace bude fungovat stejně i při změně taktu procesoru

Funkce I2C modulu - data

- První parametr vždy adresa
  - Dle dokumentace 1001100 představuje vyšších 7 bitů
  - Funkce si 8. bit (LSB) doplní samy (log. 1 pro čtení)
    - Předaná adresa tedy 0x98 (1001 100 0)
- Druhý parametr číslo registru
  - Kam se má zapsat hodnota třetího parametru - **WriteByte**
  - Odkud se má číst hodnota a vrátit ji - **ReadByte**
  - Odkud se má číst do bufferu (adresa jako třetí parametr) a kolik hodnot (číslo registru se inkrementuje) – **ReadBytes**
    - Pozor, nekontroluje se smysluplnost parametrů (adresa i počet)

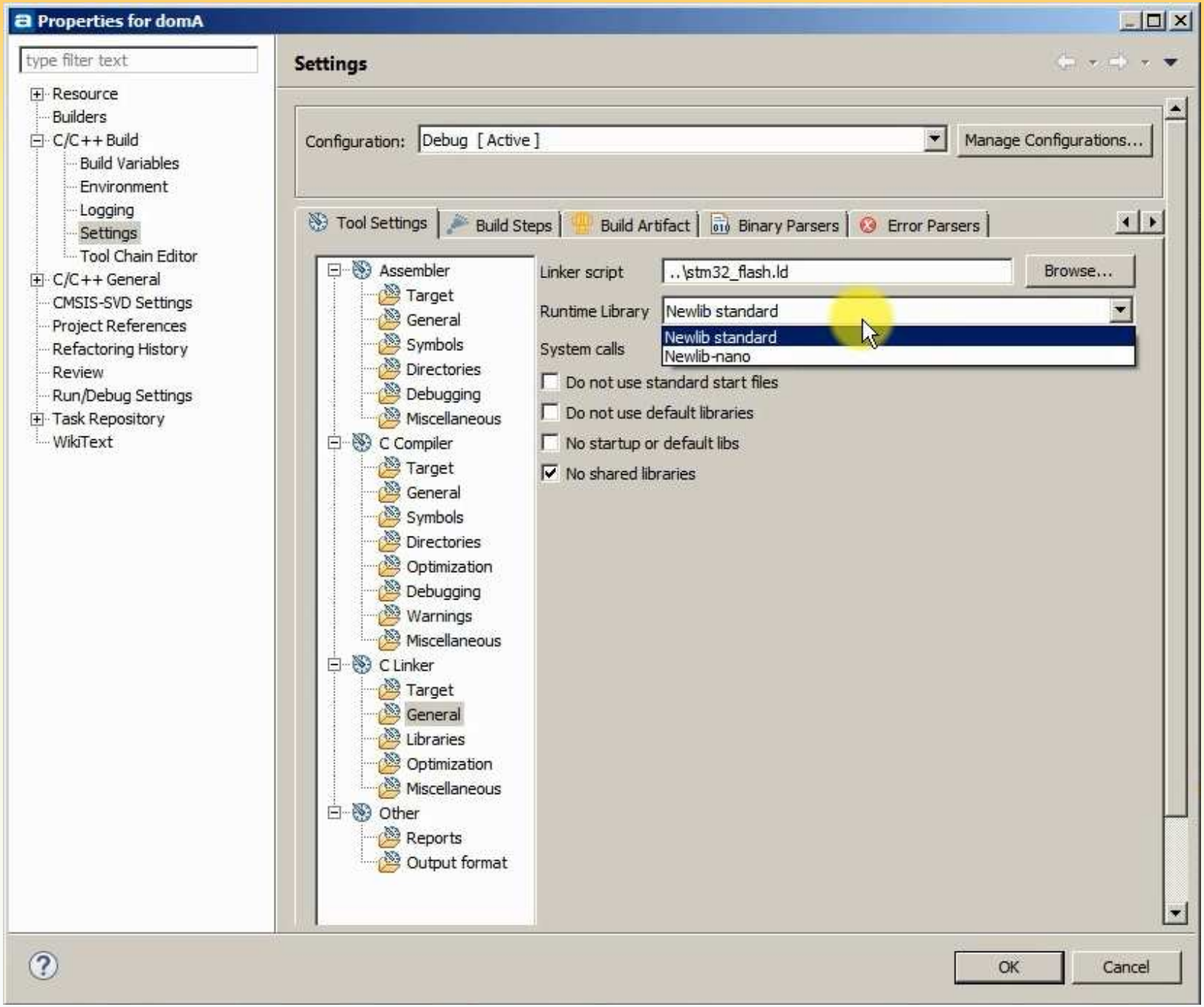
Příklad práce s akcelerometrem

- Inicializujeme rychlost na 100k
- Nutno operací je nastavení MODE bitu v MODE registru do log. 1
  - Měření aktivní
- Pravidelně vyčítám registr 0 (XOUT)
  - Výpis pomocí printf
- Případně další "osy" – YOUT a ZOUT

```
...
#include "nucleo_i2c.h"
...

#define MMA7660ADR 0x98    // 1001100 0 (1)

int main(void)
{
  uint8_t x;
...
  InitI2C1(i2cSpeed100k);
  I2C1_WriteByte(MMA7660ADR, 7, 1);
      // registr 7, hodnota 0000 0001 = MODE

  while (1)
  {
    if (_ticks >= tm2)       // casovani vysilani
    {
      tm2 = _ticks + 100;

      x = I2C1_ReadByte(MMA7660ADR, 0);   // XOUT

      printf("X=%02x\r\n", x);
    }
  }
}
```

Omezená funkce printf

- Pokud máme v projektu připojen soubor tiny_printf.c, je použita zjednodušená verze = neumí řadu běžný formátovacích povelů
  - Stačí buď smazat nebo jen přejmenovat (ne .C ani .CPP)
  - Použije se verze z knihovny
- Pokud máme nastavenu v projektu zjednodušenou knihovnu nano-lib, také ještě neumí všechny formáty a např. floaty
  - Změnit ve vlastnostech projektu
    - Pravým myšítkem na projektu – Properties ...
    - C/C++ Build – Settings – záložka Tool settings – C Linker – General
    - ... viz. následující slajd
  - Případně při vytváření nového projektu nezaškrtnout "tiny_printf" a zvolit "Newlib standard" místo "Newlib-nano"


Properties for domA
type filter text
Resource
Builders
C/C++ Build
Build Variables
Environment
Logging
Settings
Tool Chain Editor
C/C++ General
CMSIS-SVD Settings
Project References
Refactoring History
Review
Run/Debug Settings
Task Repository
WikiText
Settings
Configuration:
Debug [Active]
Manage Configurations...
Tool Settings
Build Steps
Build Artifact
Binary Parsers
Error Parsers
Assembler
Target
General
Symbols
Directories
Debugging
Miscellaneous
C Compiler
Target
General
Symbols
Directories
Optimization
Debugging
Warnings
Miscellaneous
C Linker
Target
General
Libraries
Optimization
Miscellaneous
Other
Reports
Output format
Linker script
..\stm32_flash.ld
Browse...
Runtime Library
Newlib standard
System calls
Newlib standard
Newlib-nano
Do not use standard start files
Do not use default libraries
No startup or default libs
No shared libraries
OK
Cancel

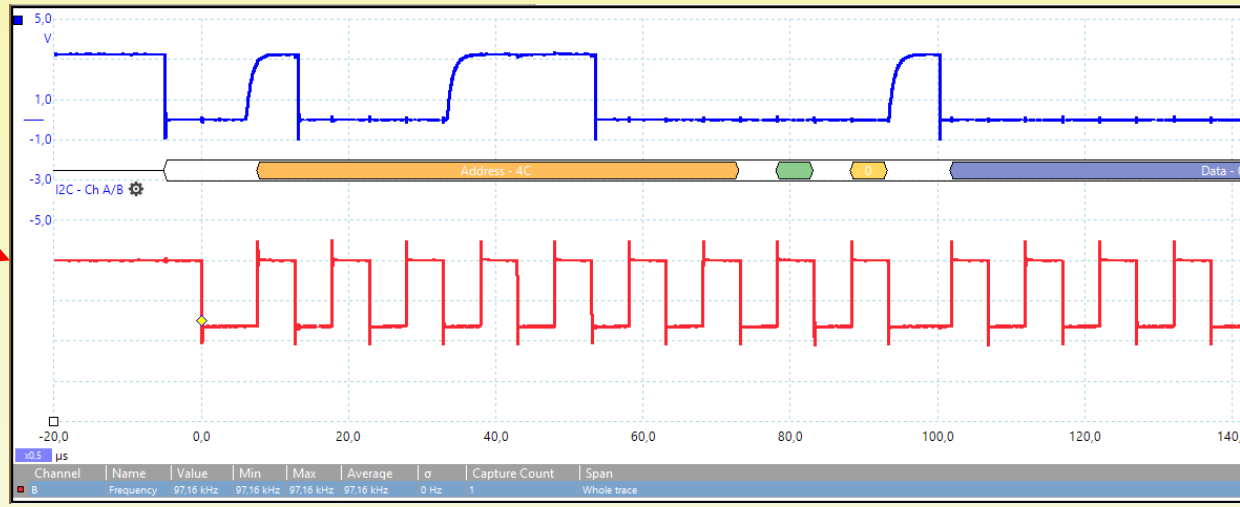
Přepočet 6-bitové hodnoty na zrychlení

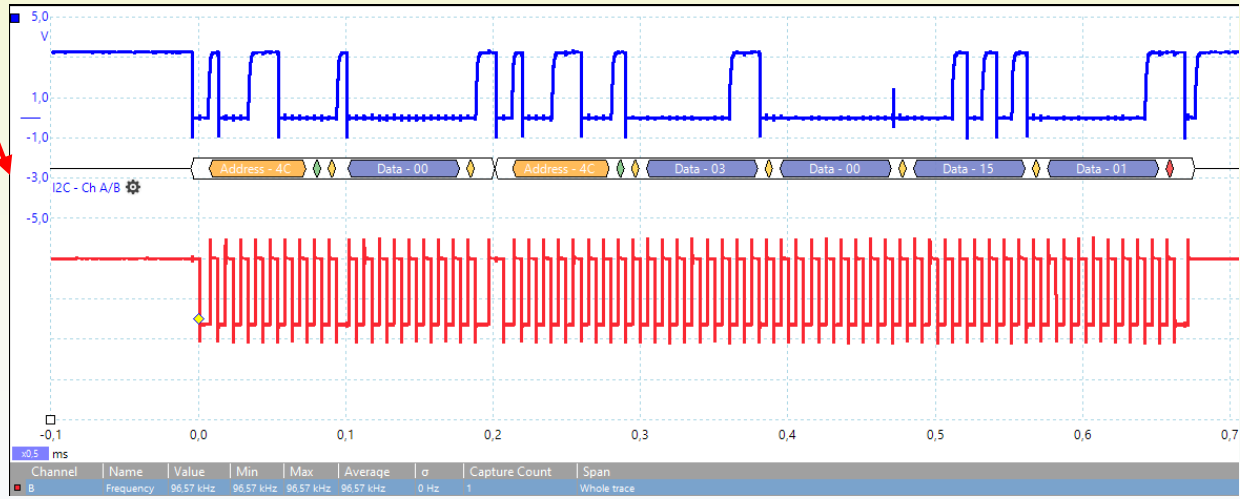
```
#define MMA7660SENS   21.33

float Acc2Float(uint8_t accVal)
{
  int val = 0;

  if (accVal & 0x20)    // nevyssi bit 6-bitove ?
    val = -(32 - (accVal & 0x1f));  // zaporne - vezmi 5bitu a odecti od 32
  else
    val = accVal & 0x1f;     // jen spodnich 5 bitu

  return val / MMA7660SENS;  // deleni je floatove - konstanta je float
}
```

- Napřed otestovat, zda je hodnota záporná
  - Maskou na xx1x xxxx – nejvyšší bit 6-bitové hodnoty
- Dvojkový doplněk (záporné číslo) musí odpovídat tabulce
  - Tj. -1 odpovídá 63 (11 1111)
  - Odečítám od 32 tu "pětibitovou" část
- Pro kladná čísla pro jistotu také vynuluji horní 3 bity
  - Resp. maskou zachovám spodních 5 jako kladné číslo 0 .. 31
- Konstanta získána v materiálech na:
  - https://developer.mbed.org/users/Sissors/code/MMA7660/file/36a163511e34/MMA7660.h

Dekódované průběhy I2C sběrnice

- Časová základna 20us a 100us na dílek
- SDA – modrá
  - OpenDrain
- SCL – červená
  - Output PushPull
  - Takt 100kHz
- Adresa 7 bitů následovaná R/W bitem (zelený kosočtverec)





Shrnutí domácí přípravy – X

- Naučili jsme se pracovat s komunikační sběrnicí I2C
  - Využili jsme předpřipravenou knihovnu (modul) pro naši aplikaci
  - Prozkoumali jsme základní registry periférie I2C v mikrokontroléru i princip samotné sběrnice
- Využili jsme připojený vnější obvod – akcelerometr MMA7660
  - Je 3-osý s citlivostí +- 1,5g, 6-bitová hodnota pro každou osu
  - Umíme přepočítat změřenou hodnotu z vnitřních registrů akcelerometru na fyzikální veličinu – zrychlení
  - Pomocí hodnoty zrychlení ve všech osách umíme určit polohu zařízení vůči zemi

Plán domácí přípravy

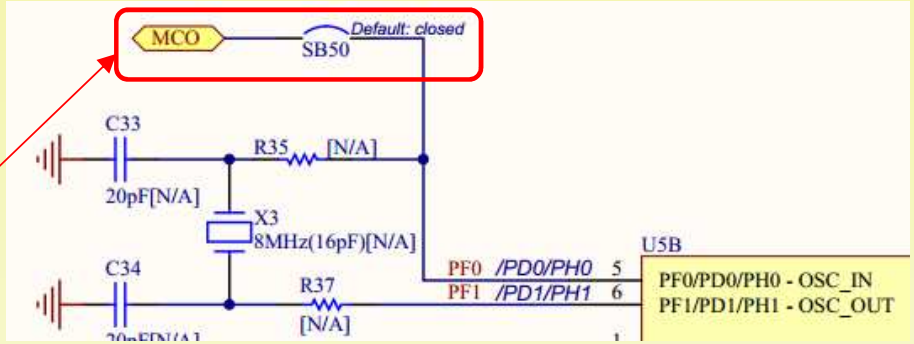
I. Instalace prostředí, ... První program – blikání LED na GPIO
II. Zobecnění GPIO operací
III. Systick
IV. Timer
V. UART
VI. Rezerva
VII. SPI
VIII. PWM
IX. A/D
X. I2C (využití "knihovny")
XI. Časování – HSI/HSE/PLL
XII. Rezerva

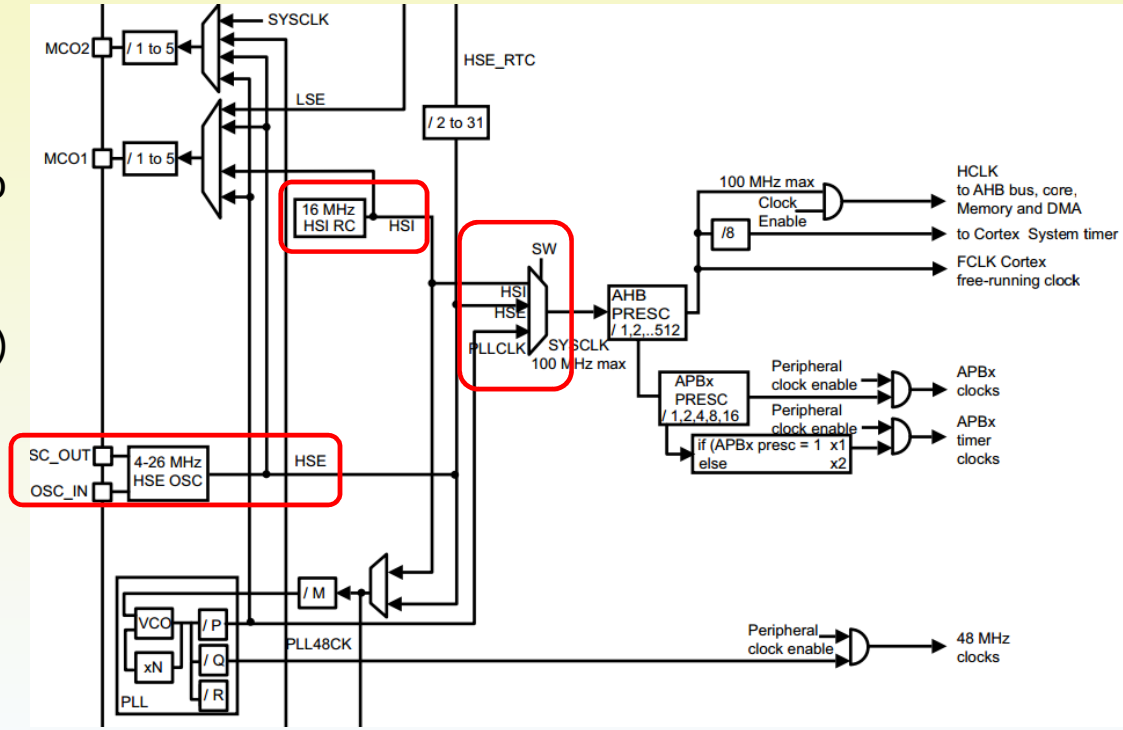
Cíle přípravy

- Prozkoumat možnosti časování v mikrokontroléru STM32F411
  - Zdroje hodin – HSI, HSE, PLL
  - Limity hodin sběrnic
- Vyzkoušet připravenou sadu funkcí fragmentů pro
  - Nastavení základní SYSCLK a děličky pro APBx
  - Zjištění taktu APBx sběrnice pro konkrétní periférie
  - Nastavení rychlosti USART podle aktuálních hodin
- Seznámit se s nastavením časování v nástroji STM32Cube (-MX nebo –F4)

Zdroje hodin v STM32F411xx

- Viz. RM – str. 90/837 – Figure 12. Clock tree
- Zdroj hodin vybírá přepínač SW
  - HSE – externí hodiny (krystal nebo oscilátor)
    - Na Nucleo připojen k 8MHz výstupu z ST-Linku (ten má vlastní krystal)
    - Neosazen X3 a C33/34
  - HSI – interní RC oscilátor kalibrovaný na 16MHz
  - PLL – dělička/násobička se zdrojem HSE nebo HSI
- Hlavní SYSCLK dělen předděličkou AHB
  - Hodiny pak vedou do jádra (CPU, SysTick, ...)
  - Každá APBx má vlastní předděličku
- Limity STM32F411:
  - SYSCLOCK max. 100MHz
  - AHB max. 100MHz
  - APB1 max. 50MHz
  - APB2 max. 100MHz





Nastavení SYSCLK

- Po RESETu defaultní stav – HSI jako zdroj hodin
- Všechny nastavovací bity mají potvrzovací bit (r/o), který indikuje úspěšnost
- Proces nastavení hodin by měl být sekvence kroků, kdy každý by měl ověřit úspěšné splnění
  - Nastav bezpečný stav – zřejmě HSI
  - Zapni příp. jiný zdroj hodin
    - HSE – zde určitě nutno testovat, že vůbec je k dispozici
    - Nastav dělící/násobící konstanty a zapni PLL, pokud je potřeba
  - Nastav děličky AHB a APBx podle požadovaného taktu
  - Nastav další HW bloky podle požadovaného taktu (FLASH, PWR)
    - Konkrétní požadavky bývají v RM celkem skryté ☹
  - Po úspěchu všech předchozích kroků nastav zdroj hodin (bity SW) a otestuj úspěšnost nastavení ve stavových bitech SWS

RCC->CR

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				PLLI2S RDY	PLLI2S ON	PLLRDY	PLLON	Reserved				CSS ON	HSE BYP	HSE RDY	HSE ON
				r	rw	r	rw					rw	rw	r	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
HSICAL[7:0]								HSITRIM[4:0]					Res.	HSI RDY	HSION
r	r	r	r	r	r	r	r	rw	rw	rw	rw	rw		r	rw

- PLLRDY – stav PLL
- PLLON – povolení PLL
- HSERDY – stav HSE = běží korektně
  - V rozsahu 4 – 25MHz
  - Korektní strmost a střída hodin (testují speciální bloky v RCC)
- HSEON – připojení HSE do RCC
- HSICAL a HSITRIM – kalibrace
  - TRIM neměnit, po RESETu 10000, tj. polovina rozsahu
  - Po kalibraci lze použít např. pro kompenzaci teploty
- HSIRDY – stav HSI
- HSION – zapnutí HSI (RESET state)

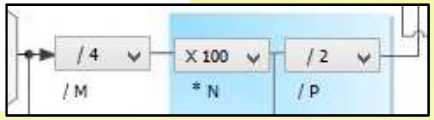
RCC->PLLCFGR

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				PLLQ3	PLLQ2	PLLQ1	PLLQ0	Reserved	PLLSRC	Reserved				PLLP1	PLLP0
				rw	rw	rw	rw		rw					rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved	PLLN									PLLM5	PLLM4	PLLM3	PLLM2	PLLM1	PLLM0
	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

This register is used to configure the PLL clock outputs according to the formulas:

- $f_{(VCO\ clock)} = f_{(PLL\ clock\ input)} \times (PLLN / PLLM)$
- $f_{(PLL\ general\ clock\ output)} = f_{(VCO\ clock)} / PLLP$
- $f_{(USB\ OTG\ FS,\ SDIO,\ RNG\ clock\ output)} = f_{(VCO\ clock)} / PLLQ$



- PLL má několik nastavení – vstup dělí M, následně násobí N pomocí VCO a opět dělí P, čímž lze získat frekvenci ve velmi jemném kroku
- **PLLSRC** – 0 = HSI (po RESETu), 1 = HSE
- **PLLQ** – platné hodnoty 2-15 (0010 – 1111)
  - Pro speciální periférie typu USB, ...
- **PLLP** – 00 = /2, 01 = /4, 10 = /6, 11 = /8
- **PLLN** – platné 50 – 432 (000110010 – 1100110000)
  - POZOR - výstup VCO musí být v rozsahu 100 a 432MHz
- **PLLM** – platné 2 - 63

RCC->CFGR

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MCO2		MCO2 PRE[2:0]			MCO1 PRE[2:0]			I2SSCR	MCO1		RTCPRE[4:0]				
rw		rw	rw	rw	rw	rw	rw	rw	rw		rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PPRE2[2:0]			PPRE1[2:0]			Reserved		HPRE[3:0]				SWS1	SWS0	SW1	SW0
rw	rw	rw	rw	rw	rw			rw	rw	rw	rw	r	r	rw	rw

- **MCOx** bity řídí výstup hodin pro další externí obvody
- **SWS** – stav přepínače zdrojů hodin, bity odpovídají SW
- **SW** – přepínač zdrojů hodin
  - 00 = HSI, 01 = HSE, 10 = PLL, (11 zakázáno)

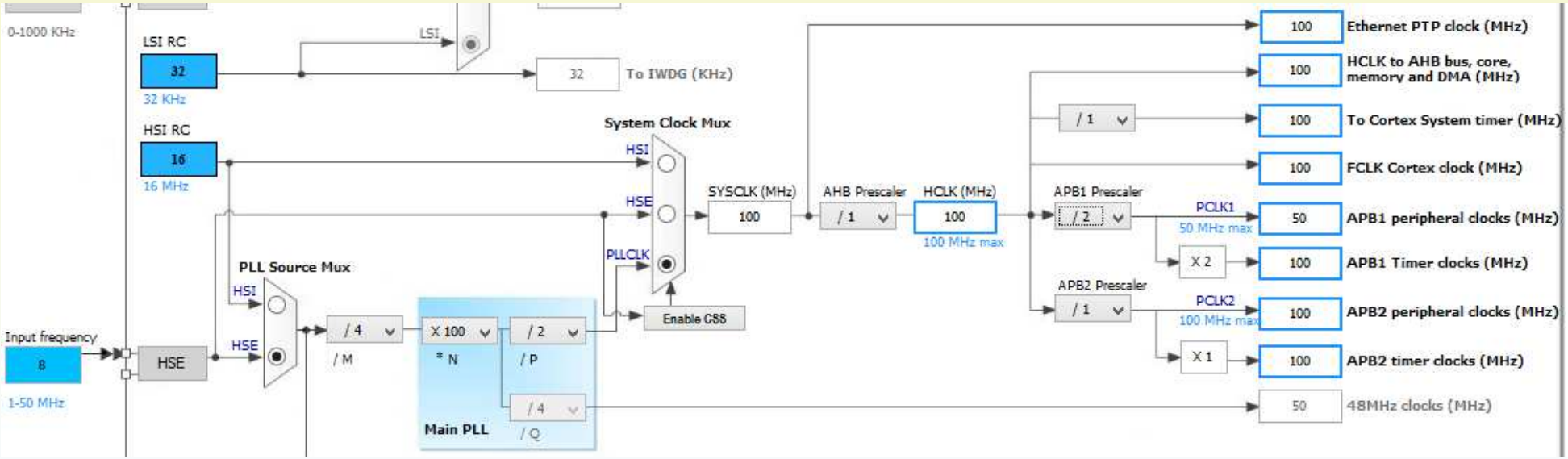
- **PPRE2** – dělička pro APB2 z AHB
- **PPRE1** – dělička pro APB1 z AHB
  - 0xx = nedělí se
  - 100 = /2, 101 = /4, 110 = /8, 111 = /16
- **HPRE** – dělička pro AHB ze SYSCLK
  - 0xxx: system clock se nedělí
  - 1000 = /2, 1001 = /4, 1010 = /8, 1011 = /16
  - 1100 = /64, 1101 = /128, 1110 = /256, 1111 = /512

```
... Pro AHB
  if (bitval & 0x8)            // 1xxx
    divider = 1 << ((bitval & 0x07) + 1);   // 0 = /2, 1 = /4
  else
    divider = 1;               // 0xxx = not divided

... Pro APB1
  bitval = (RCC->CFGR & (0x07 << 10)) >> 10; // PPRE1 [12:10] to Right
  if (bitval & 0x4)            // 1xx
    divider = 1 << ((bitval & 0x03) + 1);   // 0 = /2, 1 = /4
  else
    divider = 1;               // 0xx = not divided
```

- Hodnoty dělení z předděličky lze elegantně testovat pro MSB = 1

Nastavení hodin pomocí STM32Cube

- Zvolit nový projekt pro STM32F411RETx
- Povolit v bloku RCC zdroj hodin HSE a volbu BYPASS...
- Nastavit Input Frequenci HSE na 8 (MHz)
- V PLL source zvolit HSE
- Zadat požadovaný HCLK a nechat spočítat PLL konstanty
  - Ukáže i potřebné dělení pro APBx a výsledné takty



Bloky FLASH a PWR

- Flash je pomalejší než jádro a sběrnice, proto se musí přidávat čekací stavy při přístupu v taktu nad 30MHz
  - Příp. dokonce méně u nižšího napětí
- RM kap. 3.4 table 5
  - Nutno nastavit 3WS
  - **LATENCY** v registru **FLASH->ACR**
- V bloku řízení napájení v registru **PWR->CR** je třeba nastavit bity **VOS**
  - Viz. poznámka u FLASH nad Table 5

Table 5. Number of wait states according to CPU clock (HCLK) frequency

Wait states (WS) (LATENCY)	HCLK (MHz)			
	Voltage range 2.7 V - 3.6 V	Voltage range 2.4 V - 2.7 V	Voltage range 2.1 V - 2.4 V	Voltage range 1.71 V - 2.1 V
0 WS (1 CPU cycle)	0 < HCLK≤30	0 < HCLK ≤24	0 < HCLK ≤18	0 < HCLK ≤16
1 WS (2 CPU cycles)	30 < HCLK ≤64	24 < HCLK ≤48	18 < HCLK ≤36	16 <HCLK ≤32
2 WS (3 CPU cycles)	64 < HCLK ≤90	48 < HCLK ≤72	36 < HCLK ≤54	32 < HCLK ≤48
3 WS (4 CPU cycles)	90 < HCLK ≤100	72 < HCLK ≤96	54 < HCLK ≤72	48 < HCLK ≤64
4 WS (5 CPU cycles)	-	96 < HCLK ≤100	72 < HCLK ≤90	64 < HCLK ≤80
5 WS (6 CPU cycles)	-	-	90 < HCLK ≤100	80 < HCLK ≤96
6 WS (7 CPU cycles)	-	-	-	96 < HCLK ≤100

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			DCRST	ICRST	DCEN	ICEN	PRFTEN	Reserved				LATENCY			
			rw	w	rw	rw	rw					rw	rw	rw	rw

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved										FISSR	FMSSR	Reserved			
										rw	rw				
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
VOS		ADCDC1	Res	MRLV DS	LPLV DS	FPDS	DBP	PLS[2:0]			PVDE	CSBF	CWUF	PDDS	LPDS
rw	rw	rw		rw	rw	rw	rw	rw	rw	rw	rw	w	w	rw	rw

The prefetch buffer must be disabled when the supply voltage is below 2.1 V. The correspondence between wait states and CPU clock frequency is given in *Table 5*.

- when VOS[1:0] = 0x01, the maximum value of f_{HCLK} = 64 MHz.

- when VOS[1:0] = 0x10, the maximum value of f_{HCLK} = 84 MHz.

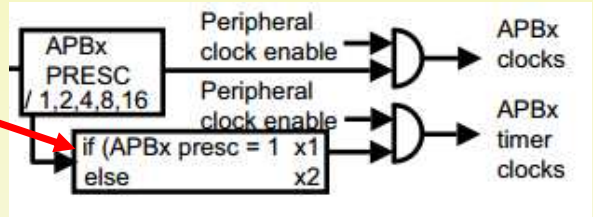
- when VOS[1:0] = 0x11, the maximum value of f_{HCLK} = 100 MHz.

```
// RM - 3.4 Read interface
 FLASH->ACR &= ~(0x0f << 0);      // LATENCY [3:0] = 0000
 FLASH->ACR |= (3 << 0);          // 3 WS

 PWR->CR |= PWR_CR_VOS_0 | PWR_CR_VOS_1;   // scale mode 1 - req. for 100MHz
```

Funkce a doplnění kódu

- Do **STM_CORE.H** doplnit typy a hlavičky
- Do **STM_CORE.C** přidat funkce
  - **SetClock100MHz** má nastavitelný zdroj HSI nebo HSE
  - **GetBusClock** má jako parametr "jakou sběrnici"
    - POZOR - pro časovače je dodatečný blok násobící 2x, pokud je **PRESC** == 1, proto mají vlastní konstanty
- V konstantách preprocesoru nastavit HSE na 8MHz
  - Project/Properties/C/C++ Build/Settings/Tool Settings/C-Compiler/Symbols
  - "Přidat" hodnotu **HSE_VALUE=8000000**
  - V knihovně je 25MHz a těch 8M je "hlídáno" #if ve funkci SetClock
- V nastavení **TIMx->PSC** je nutno vzít skutečný takt sběrnice
- Před nastavení SysTick je třeba mít aktuální SystemCoreClock
- V nastavení **USART2->BRR** je nutno počítat také s reálným taktem sběrnice APBx
  - V příkladech se počítalo s násobením setin (100x) pro větší přesnost, ale při takto 100MHz to překračuje rozsah uint32, proto pracujeme s 16-tinami

```
typedef enum {
  busClockAHB,
  busClockAPB1,
  busClockAPB2,
  timersClockAPB1,
  timersClockAPB2
} eBusClocks;

uint32_t GetBusClock(eBusClocks clk);

typedef enum {
  clockSourceHSI,
  clockSourceHSE
} eClockSources;

bool SetClock100MHz(eClockSources clkSrc);
bool SetClockHSI(void);
```




```
...
TIM3->PSC = GetBusClock(timersClockAPB1) / 1e6 - 1;
        // 1us = 1MHz
...
```

```
{
  uint sampling = (USART2->CR1 & USART_CR1_OVER8) ? 8 : 16;
  uint32_t apb1, mant, tmp, frac;
  apb1 = GetBusClock(busClockAPB1);

  mant = apb1 * 16 / (sampling * baudRate);  // v 16-tinach
  tmp = mant / 16;

  frac = mant - (tmp * 16);        // zbytek po deleni 16
  USART2->BRR = (tmp << 4) | (frac & 0x0f);
}
```

Shrnutí domácí přípravy – XI

- Máme připravené funkce pro nastavení základních hodin
  - Varianta 100MHz z externího HSE
  - Varianta 16MHz z HSI (defaultní stav po RESETu)
- Máme připravené funkce pro zjištění taktu sběrnice AHB a APBx a umíme je použít v nastavení periférií – např. TIMx
- Máme upravenou funkci inicializace USART tak, aby "si spočítala" hodnotu děliče BRR podle aktuálních hodin APBx a požadované komunikační rychlosti