



EVROPSKÁ UNIE  
Evropské strukturální a investiční fondy  
Operační program Výzkum, vývoj a vzdělávání

**MŠMT**  
MINISTERSTVO ŠKOLSTVÍ,  
MLÁDEŽE A TĚLOVÝCHOVY

# Laboratorní cvičení Embedded systémy s mikropočítači

**Petr Dostálek**



**2020**

## Informace o autorech:

Petr Dostálek, Ing, Ph.D.

Fakulta aplikované informatiky, UTB ve Zlíně

dostalek@utb.cz



EVROPSKÁ UNIE  
Evropské strukturální a investiční fondy  
Operační program Výzkum, vývoj a vzdělávání



# OBSAH

<b>OBSAH</b>	<b>3</b>
<b>ÚVOD</b>	<b>6</b>
<b>1 ZÁKLADY PROGRAMOVÁNÍ V JAZYCE SYMBOLICKÝCH ADRES MIKROPOČÍTAČE NXP RODINY HCS08</b>	<b>8</b>
1.1 JAZYK SYMBOLICKÝCH ADRES	8
1.1.1 Tvar zdrojového řádku	9
1.1.2 Direktivy překladače	10
1.1.3 Zadávání číselných hodnot	11
1.2 MIKROPOČÍTAČE NXP HCS08	12
1.2.1 Programátorský model	12
1.2.2 Způsoby adresace	15
1.3 UKÁZKOVÉ PROGRAMY	23
1.3.1 Součet dvou čísel	23
1.3.2 Porovnání dvou čísel	31
1.3.3 Vynulování pole s použitím cyklu	33
1.4 ZADÁNÍ SAMOSTATNÉ PRÁCE	36
<b>2 BINÁRNÍ VSTUPY/VÝSTUPY FRDM-KL25Z</b>	<b>37</b>
2.1 INICIALIZACE VSTUPNĚ/VÝSTUPNÍHO ROZHRANÍ	37
2.1.1 Modul řízení systému a jeho konfigurace (SIM)	37
2.1.2 Modul řízení portů a přerušení (PORT)	38
2.2 OBSLUHA VSTUPNĚ/VÝSTUPNÍHO ROZHRANÍ (GPIO)	40
2.2.1 Nastavení režimu pinů pro vstup/výstup	40
2.2.2 Zápis hodnoty na výstupní piny	40
2.2.3 Čtení stavu vstupních pinů	41
2.3 PRÁCE S JEDNOTLIVÝMI BITY - MASKOVÁNÍ	41
2.4 UKÁZKOVÉ PROGRAMY	42
2.4.1 Obsluha LED na výukovém kitu	42
2.4.2 Obsluha tlačítek na výukovém kitu	52
2.5 ZADÁNÍ SAMOSTATNÉ PRÁCE	54
<b>3 A/D PŘEVODNÍK FRDM-KL25Z</b>	<b>55</b>
3.1 INICIALIZACE SOUVISEJÍCÍCH MODULŮ	55
3.2 VYBRANÉ REGISTRY A/D PŘEVODNÍKU	56
3.3 UKÁZKOVÉ PROGRAMY	59
3.3.1 Obsluha A/D převodníku - jednorázové převody	60
3.3.2 Obsluha A/D převodníku - kontinuální převody	62
3.4 ZADÁNÍ SAMOSTATNÉ PRÁCE	64
<b>4 ČASOVACÍ SUBSYSTÉM KL25Z</b>	<b>65</b>



4.1	POPIS FUNKCE TPM MODULU.....	65
4.2	REGISTRY MODULU ČASOVAČE TPM .....	68
4.3	INICIALIZACE SOUVISEJÍCÍCH MODULŮ .....	71
4.4	INICIALIZACE MODULU ČASOVAČE .....	73
4.5	UKÁZKOVÉ PROGRAMY .....	75
4.5.1	Periodické přerušení od přetečení časovače .....	75
4.5.2	Hardwarově generovaná pulzně-široková modulace .....	78
4.6	ZADÁNÍ SAMOSTATNÉ PRÁCE.....	81
<b>5</b>	<b>SÉRIOVÉ KOMUNIKAČNÍ ROZHRANÍ .....</b>	<b>82</b>
5.1	POUŽITÍ UART ROZHRANÍ NA VÝVOJOVÉM KITU .....	83
5.2	PROGRAMOVÁ OBSLUHA UART MODULŮ .....	84
5.2.1	Popis vybraných funkcí funkčního API .....	85
5.2.2	Popis funkcí ladící konzole .....	89
5.3	UKÁZKOVÉ PROGRAMY .....	91
5.3.1	Vstup/výstup znaků s použitím funkcí ladící konzole .....	91
5.3.2	Vstup/výstup znaků s použitím funkčního API.....	96
5.4	ZADÁNÍ SAMOSTATNÉ PRÁCE.....	98
<b>6</b>	<b>SÉRIOVÁ SBĚRNICE I<sup>2</sup>C .....</b>	<b>99</b>
6.1	KOMUNIKACE NA I <sup>2</sup> C .....	100
6.2	IMPLEMENTACE I <sup>2</sup> C NA VÝUKOVÉM KITU .....	101
6.2.1	Snímač teploty LM75A.....	101
6.2.2	Snímač vlhkosti HIH6130.....	103
6.2.3	Hodiny reálného času PCF8583 .....	104
6.3	PROGRAMOVÁ OBSLUHA I <sup>2</sup> C MODULŮ .....	106
6.3.1	Popis vybraných funkcí funkčního API .....	106
6.4	UKÁZKOVÉ PROGRAMY .....	113
6.4.1	Měření teploty snímačem LM75A .....	113
6.4.2	Měření vlhkosti vzduchu snímačem HIH6130.....	117
6.4.3	Čtení časového údaje z hodin reálného času PCF8583 .....	119
6.5	ZADÁNÍ SAMOSTATNÉ PRÁCE.....	123
	ZADÁNÍ SAMOSTATNÉ PRÁCE (POKRAČOVÁNÍ).....	124
<b>7</b>	<b>SÉRIOVÉ PERIFERNÍ ROZHRANÍ SPI.....</b>	<b>125</b>
7.1	IMPLEMENTACE SPI NA VÝUKOVÉM KITU .....	126
7.1.1	EEPROM paměť 25LC640A .....	127
7.2	PROGRAMOVÁ OBSLUHA SPI MODULŮ .....	130
7.2.1	Popis vybraných funkcí funkčního API .....	130
7.3	UKÁZKOVÉ PROGRAMY .....	134
7.3.1	Čtení a zápis dat do EEPROM paměti 25LC640 .....	134



7.4	ZADÁNÍ SAMOSTATNÉ PRÁCE .....	140
<b>8</b>	<b>LCD DISPLEJ .....</b>	<b>141</b>
8.1	PROGRAMOVÁ OBSLUHA DISPLEJE .....	141
8.1.1	Popis funkcí ovladače displeje .....	141
8.2	UKÁZKOVÉ PROGRAMY .....	143
8.2.1	Výstup znaků a řetězců na LCD .....	143
8.2.2	Výstup znaků přijatých z konzole na LCD .....	146
8.3	ZADÁNÍ SAMOSTATNÉ PRÁCE .....	148
<b>9</b>	<b>OPERAČNÍ SYSTÉM FREERTOS .....</b>	<b>149</b>
9.1	VYBRANÉ FUNKCE PRO PRÁCI S ÚLOHAMI .....	150
9.2	VYBRANÉ FUNKCE PRO KOMUNIKACI A SYNCHRONIZACI .....	153
9.2.1	Funkce pro obsluhu fronty .....	153
9.2.2	Funkce pro obsluhu semaforů .....	155
9.3	UKÁZKOVÉ PROGRAMY .....	158
9.3.1	Použití fronty pro předávání dat mezi procesy .....	158
9.3.2	Použití mutexu při přístupu ke sdílenému prostředku .....	163
9.4	ZADÁNÍ SAMOSTATNÉ PRÁCE .....	166
	<b>SEZNAM POUŽITÉ LITERATURY .....</b>	<b>167</b>
	<b>SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK .....</b>	<b>169</b>
	<b>SEZNAM OBRÁZKŮ .....</b>	<b>170</b>
	<b>SEZNAM TABULEK .....</b>	<b>172</b>
	<b>SEZNAM PŘÍLOH .....</b>	<b>173</b>



EVROPSKÁ UNIE  
Evropské strukturální a investiční fondy  
Operační program Výzkum, vývoj a vzdělávání

**MŠMT**  
MINISTERSTVO ŠKOLSTVÍ,  
MLÁDEŽE A TĚLOVÝCHOVY

## ÚVOD

Současný velmi dynamický vývoj v oblasti mikropočítačové techniky související s inovacemi návrhů centrálních procesních jednotek a technologií pro výrobu křemíkových čipů přináší postupné zvyšování výpočetního výkonu a snižování spotřeby elektrické energie jednočipových mikropočítačů při zachování velmi příznivé ceny. Tímto vývojem se jejich aplikační oblast stále rozšiřuje i tam, kde bylo nutné v minulosti používat podstatně nákladnější technické vybavení na bázi průmyslové výpočetní techniky. Výkonné mikrokontroléry jsou dnes schopny zajišťovat i poměrně složité úlohy spojené se zpracováním digitálních signálů v reálném čase díky speciálním rozšířením instrukční sady o DSP instrukce nebo také realizovat pokročilé regulační úlohy typické vysokými požadavky na výpočetní výkon v plovoucí řádové čárce, kde výrobci nabízí mikrokontroléry s integrovaným matematickým koprocесorem (FPU jednotkou). Nemůžeme zde opomenout i mimořádně rozvinuté schopnosti mikropočítačů komunikovat s okolím řadou standardních rozhraní počínaje běžnou asynchronní sériovou komunikací zajišťující komunikaci mezi dvěma účastníky až po pokročilé sériové sběrnice umožňující propojení desítek zařízení a externích periférií i na větší vzdálenosti než jen lokálně v rámci desky plošného spoje. Této funkcionality je hojně využíváno v palubních sítích všech moderních dopravních prostředků od automobilů až po letadla, neboť použití komunikačních sběrnic velmi významně snižuje komplexnost kabelových svazků a zároveň zvyšuje spolehlivost. Tyto složité řídicí systémy obsahující velké množství snímačů a akčních členů jsou většinou řešeny s určitým stupněm decentralizace, protože jsou více robustní vůči poruchám – selhání dílčí jednotky nezpůsobí fatální nefunkčnost celého systému (samozřejmě vše musí být podpořeno opravdu dobře promyšleným návrhem technického i programového vybavení). Pěkným příkladem je například ABS jednotka automobilu, jejíž funkce je určitě všem dobře známa. Při selhání jednotka přes sběrnici nahlásí problém a řidič je o něm příslušnou kontrolkou informován. Automobil je ve většině případů schopen dalšího provozu, jen je třeba dbát zvýšené opatrnosti při brzdění na kluzkých cestách. Kde všude tedy v běžném moderním automobilu nalezneme mikropočítač? Bude přítomen v řídicí jednotce motoru (ECU), antiblokovacím systému brzd (ABS), systému pro zajištění jízdní stability (ESP), řídicí jednotce automatické převodovky, jednotce přístrojo-



vého panelu, imobilizéru, alarmu, zařízení infotainmentu, jednotce řízení klimatizace, systému airbagů a dalších v závislosti na výbavě automobilu. Také existuje řada mikropočítačů specializovaných na komunikační aplikace vyznačující se zejména přítomností integrovaného Ethernet MAC nebo Wi-Fi a Bluetooth bezdrátového rozhraní. K dispozici jsou i mikropočítače s ultra nízkou spotřebou a integrovaným bezdrátovým rozhraním vhodné jako základ pro zařízení Internetu věcí (IoT).

Cílem laboratorních cvičení předmětu Embedded systémy s mikropočítači je prakticky se seznámit s možnostmi použití mikropočítačové techniky zejména v aplikacích řízení technologických zařízení a procesů, spotřební elektroniky a to jak z pohledu hardwarového, tak i softwarového. Jednotlivá cvičení jsou rozčleněna tak, abychom se s mikropočítačem seznamovali postupně po jednotlivých jeho perifériích integrovaných na čipu od těch jednodušších, jakou jsou například binární vstupy a výstupy, A/D převodník až po ty složitější sloužící pro komunikaci na sériových sběrnících. V průběhu cvičení se budete setkávat s mikropočítačem MKL25Z128 s 32bitovým CPU jádrem ARM Cortex M0+ osazeného na vývojové desce FRDM-KL25Z od společnosti NXP Semiconductors. Pro rozšíření výukových možností je tato deska dále osazena do výukového kitu vyvinutého na FAI UTB ve Zlíně.



# 1 ZÁKLADY PROGRAMOVÁNÍ V JAZYCE SYMBOLICKÝCH ADRES MIKROPOČÍTAČE NXP RODINY HCS08

V úvodu k laboratorním cvičením bylo uvedeno, že se v průběhu cvičení budete setkávat s mikropočítačem NXP MKL25Z128. Je zde ale jedna výjimka – pro seznámení se s funkcí CPU a jeho programování v assembleru je z didaktického hlediska výhodnější použít mikropočítač s 8bitovým CPU, konkrétně MC9S08GB60 od společnosti NXP. Důvodů pro tuto volbu je hned několik: má jednodušší programátorský model, CPU architektura CISC je přívětivější k programování v jazyce symbolických adres, ve vývojovém prostředí Code Warrior 6.3 je k dispozici Full-Chip simulátor umožňující velmi efektivní ladění programu aniž bychom fyzicky museli program zavádět do paměti mikropočítače.

## 1.1 Jazyk symbolických adres

Jazyk symbolických adres je nízkoúrovňový programovací jazyk, který se v hierarchii úrovně abstrakce všech možných jazyků nachází pouze o jedinou úroveň výše než strojový kód, který je přímo vykonáván CPU. To sebou přináší velkou výhodu v možnosti precizní optimalizace programu pro daný typ mikroprocesoru buď na co nejvyšší rychlost zpracování anebo výslednou velikost programového kódu. Nevýhodou je, že takto napsaný zdrojový text nebude možno přeložit pro nekompatibilní mikroprocesor s odlišnou instrukční sadou, vývoj programu je časově extrémně náročný a špatně se v budoucnosti udržuje, pokud není pečlivě komentován. Z těchto důvodů se používá pro naprogramování jen těch nejkritičtějších programových rutin, kde se klade extrémní důraz na rychlost zpracování. Příkladem mohou být obsluhy přerušení, části ovladačů rychlých periférií (vysokorychlostní komunikační kanály, zobrazování na displeji), speciální algoritmy využívající specifické instrukce daného CPU (zpracování signálů pomocí DSP instrukcí) a podobně. Naprostá většina programového kódu je běžně napsána ve vyšších programovacích jazycích.

Jazyk symbolických adres programátorovi poskytuje řadu výhodných vlastností, díky kterým je zápis programu podstatně přehlednější:

- Instrukce mají svůj mnemonický název pro snadné zapamatování.
- Umožňuje práci se symbolickými jmény adres a operandů.
- Numerické hodnoty lze zadávat v různých číselných soustavách.





Vlastní překlad zdrojového textu zapsaného v jazyce symbolických adres zajišťuje překladač, který se nazývá Assembler. Proto se také můžeme běžně setkat s druhým názvem tohoto jazyka „Assembler“, který je v praxi velmi hojně používán.

### 1.1.1 Tvar zdrojového řádku

Každý zdrojový řádek zapsaný v jazyce symbolických adres, samozřejmě s výjimkou řádků obsahujících direktivy, odpovídá právě jedné instrukci CPU, která se v průběhu procesu překladu přeloží obecně na operační kód instrukce a operand. Operand ovšem nemusí být za instrukcí vždy přítomen, protože řada z nich jej nepoužívá.

Vlastní řádek zdrojového textu se skládá ze čtyř základních částí: návěští, mnemonického označení instrukce, operandu a komentáře. Jednotlivé části musí být striktně odděleny buď mezerou (nevhodné, vzniká velmi nepřehledný zdrojový text) nebo tabulátorem. V případě používání tabulátoru je kód programu velmi dobře čitelný, protože ihned na první pohled je zřejmé jeho dělení viz ukázka níže.

Návěští	Instrukce	Operand	Komentář
Start:	LDA	cislo1	; načti do akumulátoru obsah paměťové buňky číslo 1
	SUB	#10	; odečti od obsahu akumulátoru číslo 10
	RTS		; návrat z podprogramu

Návěští je nepovinnou částí zdrojového řádku a použije se, chceme-li například v programu provést podmíněný skok do příslušné části programu. Použijeme jej také při konstrukci cyklů s testem na začátku či konci a volání podprogramů. Návěští je tedy obvykle operandem instrukcí podmíněných skoků a volání podprogramů. Při překladu jej překladač nahradí příslušnou cílovou adresou skoku, která může být relativní nebo absolutní.

Instrukce je povinnou částí řádku a specifikuje elementární operaci, kterou CPU mikropočítače dokáže vykonat. Celou množinu instrukcí podporující dané CPU, nazýváme instrukční sadou. Ta je specifikována výrobcem mikropočítače a je součástí jeho dokumentace. Konkrétně pro mikropočítače rodiny HCS08 je k dispozici v [1], kapitole 8.6.

Operand specifikuje s jakými daty má daná instrukce pracovat. Může to být bezprostřední hodnota uvozená speciálním znakem # (bezprostřední adresace), adresa paměťové



buňky (přímá nebo přímá rozšířená adresace), indexový registr (indexová adresace s registry H:X, případně SP). Některé instrukce jsou bezoperandové (implicitní adresace) a v těchto případech zůstává pole operandu nevyužito.

Poslední částí zdrojového řádku je komentář, který zajišťuje čitelnost programu i po uplynutí několika let, kdy již určitě nemáme přesnou představu, jak daný program funguje. Obzvláště důležité je důsledně okomentovat funkce, tj. její vstupní argumenty a co funkce vrací. Z pohledu syntaxe je nutné doplnit informaci, že komentář musí začínat středníkem. Použití komentáře není limitováno pouze na čtvrté pole řádku, můžeme jej samozřejmě použít i na jeho začátku.

### 1.1.2 Direktivy překladače

Direktivy slouží pro řízení procesu překladu. Můžeme tedy pomocí nich překladač například informovat o tom, že určitá část programového kódu se má překládat od jiné počáteční adresy, rezervovat paměť pro proměnné našeho programu, definovat symboly a mnoho dalších využití. V tabulce 1 si uvedeme základní sadu direktiv, které budeme potřebovat při vytváření našich prvních programů.

Direktiva	Popis
INCLUDE	Zahrne do aktuálního zdrojového textu obsah specifikovaného souboru. Název souboru musí být uveden mezi apostrofy.
XDEF	Specifikuje symbol, na který je možno se odkazovat z jiných programových modulů.
XREF	Specifikuje symbol, který je definován v jiném programovém modulu a používá se v aktuálním modulu.
SECTION	Definuje programovou či datovou sekci nacházející se v souvislém úseku paměti mikropočítače od určité adresy.
ORG	Specifikuje adresu překladu programu od místa umístění direktivy dále.
DS.B DS.W DS.L	Rezervace paměti. B = Byte, W = Word, L = Long
DC.B DC.W DC.L	Definice konstanty. B = Byte, W = Word, L = Long
EQU	Definice symbolu.

Tabulka 1: Vybrané direktivy překladače.

Použití vybraných direktiv je demonstrováno v programu 1.1 za použití ukázkového programového kódu určeného pro mikropočítače rodiny HCS08.



### Program 1.1:

	INCLUDE	'derivative.inc'	
RAM_Start:	EQU	\$0080	;definice symbolu RAM_Start (počáteční adresa RAM)
FLASH_Start:	EQU	\$182C	;definice symbolu FLASH_Start (počáteční adresa FLASH)
	ORG	RAM_Start	;od tohoto místa dále se překládá od adresy RAM_Start
Prom1:	DS.B	1	;rezervace 1 bajtu paměti RAM pro Prom1
Prom2:	DS.B	1	;rezervace 1 bajtu paměti RAM pro Prom2
Pole1:	DS.B	10	;rezervace 10 bajtů paměti pro Pole1
	ORG	FLASH_Start	;od tohoto místa dále se překládá od adresy FLASH_Start
Text1:	DC.B	"HCS08"	;definice textového řetězce jako konstanty
	DC.B	0	;včetně znaku '\0' na jeho konci
Konst1:	DC.L	\$FFAD1000	;definice konstanty typu Long (zabírá 4 B paměti)
_Startup:	LDA	Prom1	;načti do akumulátoru obsah paměťové buňky Prom1
	ADD	#10	;přičti k akumulátoru číslo 10
	STA	Prom2	;ulož obsah akumulátoru do Prom2
	BRA	*	;skok na sebe sama

#### 1.1.3 Zadávání číselných hodnot

Jazyk symbolických adres podporuje zadávání numerických údajů ve čtyřech číselných soustavách, které se rozlišují pomocí prefixu uvedeného před číslem. Také je možné tuto hodnotu zadat pomocí ASCII znaku. V tomto případě je nutno příslušný znak vložit mezi apostrofy. V tabulce 2 naleznete přehled příslušných prefixů a příklad zápisu čísla 65 s využitím podporovaných číselných soustav.

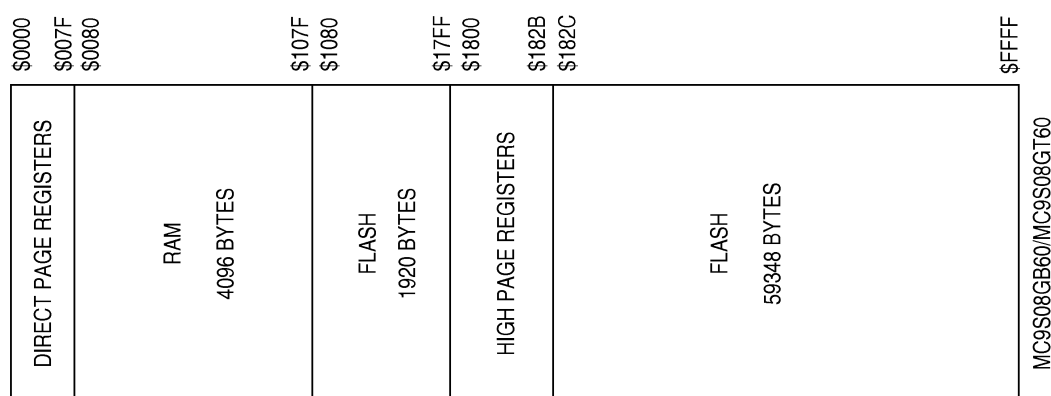
Prefix	Popis	Příklad zápisu čísla 65
%	Binární soustava	%01000001
@	Osmičková soustava	@101
	Desítková soustava	65
\$	Šestnáctková soustava	\$41
	ASCII znak	'A'

Tabulka 2: Zápis hodnot v různých číselných soustavách.

## 1.2 Mikropočítače NXP HCS08

Mikropočítače NXP z rodiny HCS08 jsou založeny na 8bitovém CPU s Von Neumannovou architekturou, který je směrem nahoru na úrovni objektového kódu plně kompatibilní s rodinami M68HC05 a M68HC08. Všechny periferní registry a paměti jsou mapovány do společného paměťového prostoru o celkové velikosti 64 KiB, viz obrázek 2 s paměťovou mapou mikropočítače MC9S08GB60. Z ní je zřejmé, že přeložený strojový kód vykonávaný mikroprocesorem bude ukládán do FLASH paměti od adresy \$182C a proměnné programu do RAM paměti od adresy \$0080. Oblast RAM paměti od adresy \$0080 do \$00FF spadá do nulté stránky paměti, což má zásadní význam pro optimalizaci uložení proměnných. Je to dáno tím, že k tomuto rozsahu adres může CPU přistupovat prostřednictvím přímé adresace (cílem je 8bitová adresa), která je rychlejší než přímá rozšířená adresace (cílem je 16bitová adresa). Z tohoto důvodu jsou také zásadní periferní registry umístěny v této stránce paměti.

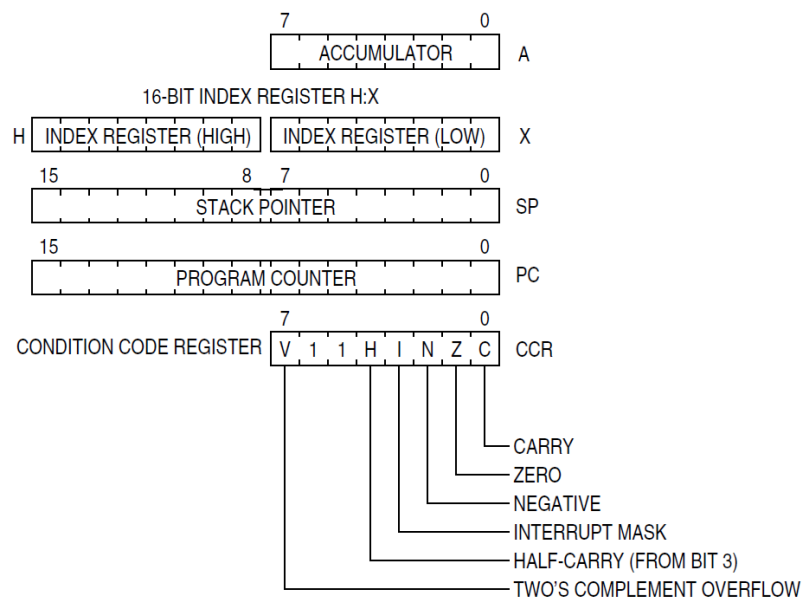
Na konci paměťového prostoru na adresách \$FFC0 až \$FFFF se nachází 64 bajtů velká oblast vektorů přerušení, kterou si lze představit jako tabulku s 32 položkami obsahující v jednotlivých záznamech 16bitovou adresu příslušné programové služby.



Obrázek 1: Paměťová mapa mikropočítače MC9S08GB60 [1].

### 1.2.1 Programátorský model

Mikroprocesor je vybaven celkem pěti vnitřními registry (nezaměňovat s registry interních periférií), které má programátor k dispozici při tvorbě programu v jazyce symbolických adres. Některé z registrů jsou univerzální (A, H:X), jiné mají svůj speciální specifický účel (SP, CCR). Přehled všech programátorovi dostupných registrů je uveden na obrázku 2.



Obrázek 2: Registry CPU mikropočítače MC9S08GB60 [1].

### Akumulátor (A)

Akumulátor (nebo také střadač) je univerzální 8bitový registr, který má mezi ostatními registry určité výsadní postavení. Pohledem do instrukční sady zjistíme, že převážná většina vykonávaných operací se provádí právě s tímto registrem. Pokud tedy například chceme sečíst obsah dvou paměťových buněk, musíme nejdříve hodnotu uloženou v první paměťové buňce načíst do akumulátoru a následně provést součet obsahu akumulátoru s hodnotou uloženou ve druhé paměťové buňce. Výsledek součtu je uložen zpět do akumulátoru. Obsah registru není resetem mikropočítače ovlivněn [1].

### Indexový registr (H:X)

Indexový registr je 16bitový registr složený ze dvou 8bitových registrů H a X. Používá se jako ukazatel na libovolnou paměťovou buňku při tzv. indexové adresaci. Z důvodu kompatibility s předchozí generací mikropočítačů 68HC05 některé instrukce pracují jen se spodní částí registru H:X. Existuje řada instrukcí, která používá registr X jako druhý 8bitový univerzální registr – lze jej nulovat, inkrementovat, dekrementovat, rotovat, provádět logický posun, jednotkový a dvojkový doplněk. Po resetu je registr H automaticky vynulován pro kompatibilitu s 68HC05, registr X není resetem ovlivněn [1].

### Ukazatel zásobníku (SP)

Ukazatel zásobníku je 16bitový registr ukazující na další volnou položku zásobníkové paměti typu LIFO (poslední dovnitř, první ven). Obsah registru je automaticky zmenšován či zvětšován podle prováděných operací se zásobníkem. Pokud na zásobník vložíme údaj pomocí instrukce typu PUSH, je hodnota v SP snížena o jedničku. Naopak při vyjmutí hodnoty ze zásobníku instrukcí typu PULL je hodnota v SP zvýšena o jedničku. Dále zásobník používají instrukce volání a návratu z podprogramu, kdy je zapotřebí uložit či načíst návratovou adresu. Zásobník se používá také při obsluze přerušení, protože CPU do něj musí před tím, než provede skok na příslušnou programovou obsluhu, uložit aktuální stav všech registrů (kromě H). Obsah registru SP je po resetu automaticky inicializován na hodnotu \$00FF (paměť RAM v nulté stránce paměti) [1].

### Programový čítač (PC)

Programový čítač je 16bitový registr obsahující adresu instrukce nebo operandu, který bude aktuálně centrální procesní jednotkou načítán z paměti. CPU zpracovává program sekvenčně. To znamená, že po zpracování instrukce je načtena následující instrukce v paměti. Pokud by program neobsahoval instrukce skoku, hodnota v registru by se neustále zvětšovala. Jakmile je procesorem zpracována instrukce skoku, je obsah PC registru nastaven na její cílovou adresu. Obdobně se CPU zachová i při požadavku na obsluhu přerušení, jen je cílová adresa skoku přečtena z příslušného vektoru přerušení. Po resetu mikropočítače je obsah programového čítače inicializován hodnotou uloženou v reset vektoru na adresách \$FFFE a \$FFFF ukazující na první instrukci programu v paměti [1].

### Příznakový registr (CCR)

Příznakový registr obsahuje aktuální stavovou informaci po provedení každé instrukce. Stav je indikován uvnitř 8bitového registru prostřednictvím šesti příznakových bitů. Zbývající bity 5 a 6 jsou nevyužity a jejich hodnota je trvale nastavená na 1 [1].

Přehled jednotlivých stavových bitů v CCR registru včetně popisu jejich významu je uveden v tabulce 3.



Příznak v CCR	Popis
C (Carry)	Příznak přenosu / výpůjčky C se nastaví do 1, pokud při zpracování instrukce aritmetického součtu došlo k přenosu do vyššího řádu, tj. ze 7. bitu akumulátoru do C nebo k výpůjčce při odečítání. Příznak C se nastaví do 1 například při provedení součtu $255 + 1$ .
Z (Zero)	Indikuje nulový výsledek provedené operace. Příznak je aktualizován po provedení aritmetických, logických operací a také po vykonání instrukcí pro přesun dat.
N (Negative)	Nastaví se na 1 při zápornosti výsledku provedené operace. Příznak je aktualizován po provedení aritmetických, logických operací a také po vykonání instrukcí pro přesun dat. Příznak je nastavován dle stavu nejvíce významného (sedmého) bitu výsledku.
I (Interrupt)	Příznak přerušení ve stavu 1 informuje, že CPU právě vykonává programovou obsluhu přerušení. V průběhu zpracování obsluhy jsou další požadavky na přerušení automaticky zakázány. Po vykonání obsluhy přerušení se příznak I automaticky nastaví zpět na 0, čímž se povolí zpracování dalších požadavků na přerušení. Příznak lze nastavit do požadovaného stavu i programově instrukcemi SEI / CLI a tím zakázat / povolit zpracování přerušení.
H (Half-Carry)	Indikuje poloviční přenos z 3. do 4. bitu výsledku operace. Používá se zejména při práci s BCD aritmetikou a dekadické korekci. Příznak je aktualizován při použití instrukcí ADD a ADC. Nastaví se do 1 v případě polovičního přenosu, v opačném případě je 0.
V (Overflow)	Příznak přetečení se nastaví do 1, pokud při zpracování instrukce došlo k přetečení dvojkového doplňku, tj. výsledek operace je větší než 127 nebo menší než -128. Přetečení nastane například při následujících aritmetických operacích: $127 + 1$ nebo $-128 - 1$ .

Tabulka 3: Význam příznakových bitů v CCR registru [1].

### 1.2.2 Způsoby adresace

Centrální procesní jednotka mikropočítačů rodiny HCS08 podporuje celkem 7 různých základních adresovacích režimů pro přístup k datům uložených v paměti:

- Implicitní (INH)
- Bezprostřední (IMM)
- Přímá (DIR)
- Rozšířená (EXT)
- Relativní (REL)
- Indexová vztažená k H:X registru (režimy IX+, IX1, IX1+, IX2)
- Indexová vztažená k SP registru (režimy SP1, SP2)





### Implicitní adresace (INH)

V tomto adresovacím režimu je operand instrukce obsažen přímo v instrukčním kódu instrukce. Tímto operandem je ve většině případů některý z registrů CPU dle použité instrukce. Pole operandu zůstává u tohoto typu adresace prázdné [1].

*Program 1.2:*

```
_Startup:    CLRA                ; vynuluj obsah akumulátoru
             CLRX                ; vynuluj obsah registru X
             CLI                 ; povol maskovatelná přerušení (vynuluje bit I v CCR)
```

### Bezprostřední adresace (IMM)

Při bezprostřední adresaci je hodnota operandu instrukce umístěna v následující paměťové buňce za instrukčním kódem (adresa instrukčního kódu + 1) dané instrukce. V případě 16bitové hodnoty je operand v paměti uložen tak, že vyšší bajt (HB) hodnoty je uložen na nižší adresu (adresa instrukčního kódu + 1) a nižší bajt (LB) hodnoty na vyšší adresu (adresa instrukčního kódu + 2). Použití bezprostřední adresace musí být překladači oznámeno zápisem prefixu # v poli operandu [1].

*Program 1.3:*

```
Pondeli:     EQU      1
_Startup:    LDA      #100        ; Načti hodnotu 100 do akumulátoru
             LDA      #Pondeli    ; Načti číslo dne v týdnu do akumulátoru
             LDHX     #$182C      ; Načti hodnotu $182C do registru H:X
             LDHX     #_Startup   ; Načti do registru H:X adresu návěští „_Startup“
```

### Přímá adresace (DIR)

U přímé adresace následuje za instrukčním kódem 8bitová adresa paměťové buňky s tím, že vyšší bajt adresy je vždy nulový. Z toho vyplývá, že cílová adresa se nachází vždy v nulté stránce paměti v paměťovém rozsahu \$0000 až \$00FF. Výhodou přímé adresace je podstatně rychlejší zpracování instrukce, protože CPU nemusí z paměti načítat vyšší bajt cílové adresy a navíc se ještě zkrátí přeložený strojový kód. Z tohoto důvodu jsou v nulté





stránce paměti mapovány nejčastěji používané registry a menší část paměti RAM – konkrétně 128 B. Zde se tedy programátorům nabízí potenciál pro optimalizaci umístění proměnných s ohledem na frekvenci jejich použití v programu [1].

*Program 1.4:*

	ORG	\$80	; Proměnné budou umístěny od adresy \$80 (RAM)
Counter:	DS.B	1	; Rezervace 1 B paměti pro proměnnou Counter v nulté stránce paměti RAM (adresa \$0080)
	ORG	\$182C	; Program bude umístěn od adresy \$182C (FLASH)
_Startup:	MOV	#5,Counter	; Ulož do proměnné Counter hodnotu 5
	LDA	Counter	; Načti obsah proměnné Counter do akumulátoru
	INCA		; Inkrementuj obsah akumulátoru
	STA	Counter	; Ulož obsah akumulátoru zpět do proměnné Counter

Po zpracování výše uvedeného programu bude v proměnné „Counter“ uložena hodnota 6. Celková délka běhu programu je 7 sběrnicevých cyklů. Inicializace proměnné „Counter“ na počáteční hodnotu 5 pomocí instrukce MOV není do počtu cyklů zahrnuta.

### Rozšířená adresace (EXT)

Pomocí rozšířené adresace lze adresovat libovolnou paměťovou buňku z celého dostupného 64 KiB velkého paměťového prostoru. Adresa operandu má v tomto případě velikost 16 bitů. Ta je uložena za instrukčním kódem v pořadí vyšší bajt (HB) adresy, nižší bajt (LB) adresy. Délka zpracování instrukce je delší než u obdobné instrukce používající přímou adresaci (DIR), protože CPU musí z paměti načíst o 1 bajt více (vyšší bajt adresy) [1].

*Program 1.5:*

	ORG	\$100	; Proměnné budou umístěny od adresy \$100 (RAM)
Counter:	DS.B	1	; Rezervace 1 B paměti pro proměnnou Counter ; Nyní se nachází mimo nultou stránku (adresa \$0100)
	ORG	\$182C	; Program bude umístěn od adresy \$182C (FLASH)
_Startup:	LDX	#5	; Načti do registru X hodnotu 5
	STX	Counter	; Ulož obsah registru X do proměnné Counter
	LDA	Counter	; Načti obsah proměnné Counter do akumulátoru
	INCA		; Inkrementuj obsah akumulátoru
	STA	Counter	; Ulož obsah akumulátoru zpět do proměnné Counter



Funkce programu je totožná s příkladem 1.4, pouze došlo ke změně umístění proměnné „Counter“ na adresu \$0100, která tímto zásahem již leží mimo oblast nulté stránky paměti. Z tohoto důvodu nebylo možné pro počáteční inicializaci obsahu proměnné použít instrukci MOV, protože nepodporuje rozšířenou adresaci. Proto musela být nahrazena delší sekvencí s instrukcemi LDX a STX. Všimněte si, že u instrukcí LDA a STA se v zápisu nic nezměnilo, překladač automaticky zjistí a použije správný typ adresace. Celková délka běhu programu je 9 sběrníkových cyklů, což je o přibližně o 29 % více než v předchozím příkladu. Inicializace proměnné „Counter“ na počáteční hodnotu 5 není opět do počtu cyklů zahrnuta.

### Relativní adresace (REL)

Relativní adresace je používána zejména u instrukcí pro větvení programu. Za instrukčním kódem následuje 8bitová relativní adresa skoku, která je vztažena k aktuálnímu umístění instrukce v paměti. Aby bylo možné provést skok směrem k vyšším i nižším adresám, je relativní offset v paměti uložen ve formě dvojkového doplňku. Z výše uvedeného vyplývá, že lze z aktuálního umístění instrukce provést skok v rozsahu od -128 až +127 adres. To může být v některých případech limitující a řeší se to částí programového kódu pro prodloužení skoků s využitím instrukcí pro nepodmíněný skok JMP [1].

#### Program 1.6:

	ORG	\$80	; Proměnné budou umístěny od adresy \$80 (RAM)
Counter:	DS.B	1	; Rezervace 1 B paměti pro proměnnou Counter v nulté stránce paměti RAM (adresa \$0080)
	ORG	\$182C	; Program bude umístěn od adresy \$182C (FLASH)
_Startup:	LDA	#5	; Načti do registru A hodnotu 5
	STA	Counter	; Inicializace počítadla cyklů na hodnotu 5
Loop:	NOP		; Tělo cyklu představuje prázdná instrukce NOP
	DEC	Counter	; Dekrementuj počítadlo cyklů Counter
	BNE	Loop	; Pokud je výsledek předchozí operace nenulový, proved' skok na návěští Loop, jinak pokračuj další instrukcí

Program je ukázkou použití instrukce podmíněného skoku BNE (jako jediná v programu používá relativní adresaci) v cyklu s testem na konci. Na začátku je inicializována proměnná počítadla cyklu „Counter“ na hodnotu 5. Následuje tělo cyklu tvořené instrukcí NOP, která nic nevykonává, jen zvýší programový čítač o 1. Dále se sníží počítadlo cyklů o

jedničku pomocí instrukce DEC. Následující instrukce BNE vyhodnotí v příznakovém registru CCR příznak Z (nulovost výsledku předchozí operace). Pokud je příznak Z roven nule, provede se relativní skok na návěští „Loop“ o -3 adresy (hodnota operandu je \$FB) od aktuálního umístění instrukce v paměti. Tělo cyklu se opakuje tak dlouho, dokud je v proměnné „Counter“ po dekrementaci nenulová hodnota. Po dosažení nulové hodnoty instrukce BNE skok na návěští „Loop“ již neprovede (příznak Z = 1) a pokračuje se další instrukcí. Tělo cyklu se v daném ukázkovém programu provede 5x.

### Indexová adresace vtažená k registru H:X

Při použití indexové adresace vztažené k registru H:X je efektivní adresa operandu uložena v indexovém registru H:X. Registry H a X jsou v tomto případě sloučeny v jeden 16bitový registr umožňující tímto přístup k libovolné paměťové buňce v celém 64 KiB adresovém rozsahu [1].

CPU mikropočítačů HCS08 podporuje 5 variant této adresace [1]:

- *Bez offsetu (IX)* – výsledná adresa operandu je 16bitová adresa obsažená v registrovém páru H:X.
- *Bez offsetu s následnou inkrementací H:X (IX+)* – výsledná adresa operandu je 16bitová adresa obsažená v registrovém páru H:X, po načtení operandu z paměti je registr H:X následně inkrementován. Tento adresovací režim podporují pouze dvě instrukce: MOV a CBEQ.
- *S 8bitovým offsetem (IX1)* – výsledná adresa operandu je dána součtem obsahu registrového páru H:X a 8bitového offsetu, který následuje za instrukčním kódem instrukce. Offset je tedy konstanta, kterou nelze za běhu programu měnit.
- *S 8bitovým offsetem s následnou inkrementací H:X (IX1+)* – funkce je obdobná jako u předchozí adresace IX1. Po načtení operandu z paměti je navíc inkrementován obsah registru H:X.
- *S 16bitovým offsetem (IX2)* – výsledná adresa operandu je dána součtem obsahu registrového páru H:X a 16bitového offsetu, který následuje za instrukčním kódem instrukce.



Program 1.7:

	ORG	\$80	; Proměnné budou umístěny od adresy \$80 (RAM)
Data:	DS.B	4	; Rezervace 4 B paměti pro pole Data (adresa \$80 až \$83)
	ORG	\$182C	; Program bude umístěn od adresy \$182C (FLASH)
_Startup:	MOV	#10,Data	; Inicializace Data[0] = 10
	MOV	#20,Data+1	; Inicializace Data[1] = 20
	MOV	#30,Data+2	; Inicializace Data[2] = 30
	MOV	#40,Data+3	; Inicializace Data[3] = 40
	LDHX	#Data	; Načti adresu prvního prvku pole do H:X
	LDA	,X	; Indexová adresace bez offsetu (IX); reg.A = 10
	LDA	1,X	; Indexová adresace s 8bitovým offsetem (IX1); reg.A = 20
	LDA	2,X	; Indexová adresace s 8bitovým offsetem (IX1); reg.A = 30
	LDA	3,X	; Indexová adresace s 8bitovým offsetem (IX1); reg.A = 40

Příklad ukazuje použití indexové adresace vztažené k registru H:X bez offsetu i s jeho použitím při přístupu k jednotlivým položkám pole „Data“. Princip spočívá v načtení adresy pole do registru H:X pomocí instrukce LDHX a poté již načítáme prvky pole pomocí indexové adresace instrukcí LDA offset,X. Nastavením hodnoty offsetu přistupujeme k požadovaným položkám pole. Je nutno mít na paměti, že offset je konstanta, kterou nelze za běhu programu měnit.

### Indexová adresace vtažená k registru SP

Při použití indexové adresace vztažené k registru SP je efektivní adresa operandu uložena v 16bitovém registru SP umožňující přístup k libovolné paměťové buňce v celém 64 KiB adresovém rozsahu. Nicméně použití vyžaduje určitou obezřetnost, protože SP je zároveň ukazatelem zásobníku. Většinou se tato adresace používá pro efektivní přístup k lokálním proměnným uložených na zásobníku [1].

CPU mikropočítačů HCS08 podporuje 2 varianty této adresace [1]:

- *S 8bitovým offsetem (SP1)* – výsledná adresa operandu je dána součtem obsahu registru SP a 8bitového offsetu.
- *S 16bitovým offsetem (SP2)* – výsledná adresa operandu je dána součtem obsahu registru SP a 16bitového offsetu.



Program 1.8:

	ORG	\$80	; Proměnné budou umístěny od adresy \$80 (RAM)
Cislo1:	DS.B	1	; Rezervace 1 B paměti pro proměnnou Cislo1
Cislo2:	DS.B	1	; Rezervace 1 B paměti pro proměnnou Cislo2
Vysledek:	DS.B	1	; Rezervace 1 B paměti pro proměnnou Vysledek
	ORG	\$182C	; Program bude umístěn od adresy \$182C (FLASH)
_Startup:	MOV	#10,Cislo1	; Inicializace obsahu proměnné Cislo1
	MOV	#20,Cislo2	; Inicializace obsahu proměnné Cislo2
	LDA	Cislo1	; Načti obsah proměnné Cislo1 do akumulátoru
	PSHA		; Ulož obsah akumulátoru do zásobníku
	LDA	Cislo2	; Načti obsah proměnné Cislo2 do akumulátoru
	PSHA		; Ulož obsah akumulátoru do zásobníku
	JSR	Soucet	; Zavolej podprogram Soucet
	AIS	#2	; Odstraň ze zásobníku argumenty funkce
	STA	Vysledek	; Ulož výsledek součtu do proměnné Vysledek
	BRA	*	; Skok na sebe sama, konec programu
Soucet:	LDA	4,SP	; Načti ze zásobníku 1. argument funkce do akumulátoru
	ADD	3,SP	; Přičti k obsahu akumulátoru 2. argument funkce
	RTS		; Návrat z podprogramu (v akumulátoru je výsledek součtu)

Příklad ukazuje použití indexové adresace vztažené k registru SP při přístupu k lokálním proměnným uložených na zásobníku při vykonávání podprogramu. Nejdříve program provede inicializaci obsahu proměnných „Cislo1“ a „Cislo2“, které budeme chtít následně sečíst pomocí podprogramu „Soucet“ očekávající dva argumenty uložené na zásobníku. Dále na zásobník pomocí instrukcí PSHA uloží dva argumenty o velikosti 1 B načtených z výše uvedených proměnných. Následně se zavolá podprogram „Soucet“ instrukcí JSR. Voláním podprogramu se na zásobník uloží návratová adresa o velikosti 2 B, s čímž je nutno počítat při přístupu k lokálním proměnným uvnitř podprogramu. Podprogram sečte předané hodnoty pomocí instrukce ADD a výsledek vrátí v akumulátoru. Návrat z podprogramu se provede instrukcí RTS, která si ze zásobníku přečte návratovou adresu a provede odpovídající skok. Zároveň zajistí odstranění návratové adresy ze zásobníku. Následně musí volající odstranit argumenty funkce ze zásobníku tak, že vrátí SP do takového stavu, v jakém byl před uložením argumentů funkce, tj. přičteme k jeho aktuálnímu obsahu hodnotu 2 instrukcí AIS

(byly uloženy 2 argumenty a velikosti 1 B). Návrátovou hodnotu funkce obsaženou v akumulátoru uložíme do proměnné „Vysledek“. Stav zásobníkové paměti a SP registru v jednotlivých fázích zpracování programu je uveden na obrázku 3.

1) Zpracovány instrukce:

MOV #10,Cislo1

MOV #20,Cislo2

SP→

Adresa	Obsah
\$17F	-
\$17E	-
\$17D	-
\$17C	-
\$17B	-

2) Zpracovány instrukce:

LDA Cislo1

PSHA

LDA Cislo2

PSHA

SP→

Adresa	Obsah
\$17F	10
\$17E	20
\$17D	-
\$17C	-
\$17B	-

3) Zpracovány instrukce:

JSR Soucet

SP→

Adresa	Obsah
\$17F	10
\$17E	20
\$17D	Ret. addr. LB
\$17C	Ret. addr. HB
\$17B	-

4) Zpracovány instrukce:

LDA 4,SP

ADD 3,SP

SP+4

SP+3

SP+2

SP+1

SP→

Adresa	Obsah
\$17F	10
\$17E	20
\$17D	Ret. addr. LB
\$17C	Ret. addr. HB
\$17B	-

5) Zpracovány instrukce:

RTS

SP→

Adresa	Obsah
\$17F	10
\$17E	20
\$17D	-
\$17C	-
\$17B	-

6) Zpracovány instrukce:

AIS #2

STA Vysledek

SP→

Adresa	Obsah
\$17F	-
\$17E	-
\$17D	-
\$17C	-
\$17B	-

Obrázek 3: Obsah zásobníkové paměti a SP registru v příkladu 1.8.

### 1.3 Ukázkové programy

V této kapitole naleznete ukázkové řešené příklady v jazyce symbolických adres. V rámci laboratorního cvičení si funkci programů důkladně vyzkoušejte jejich odkrokováním v příslušném programovém prostředí. Zadané samostatné úkoly v mnoha případech budou na ně navazovat.

#### 1.3.1 Součet dvou čísel

##### Zadání

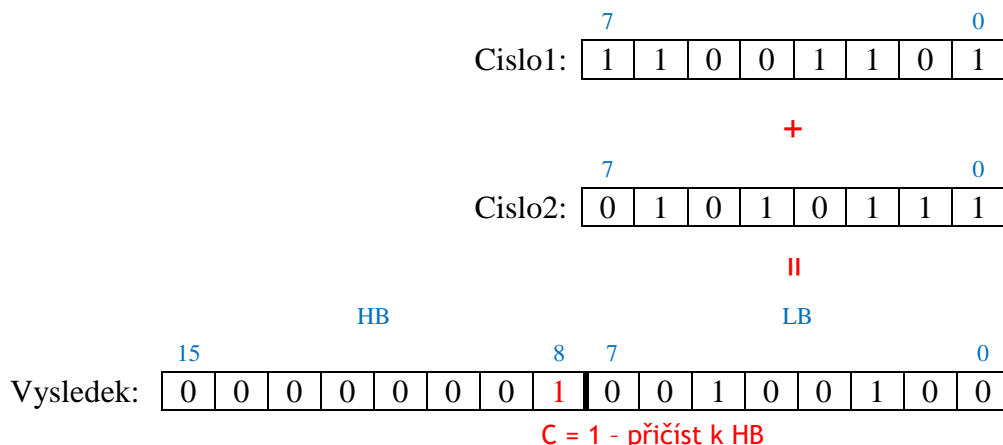
V jazyce symbolických adres mikropočítače MC9C08GB60 vytvořte program, který sečte dvě čísla bez znaménka uložená v proměnných typu byte s názvy „Cislo1“ a „Cislo2“. Výsledek součtu uložte do proměnné „Vysledek“ typu word.

##### Řešení

Z programátorského modelu mikropočítače víme, že naprostá většina aritmetických, logických a dalších operací se provádí s registrem, který se jmenuje akumulátor. CPU mikropočítačů HCS08 je vybaveno pouze jedním akumulátorem A. Pro programovou realizaci součtu tedy musíme postupovat tak, že nejprve do akumulátoru načteme obsah proměnné „Cislo1“ a teprve potom provedeme operaci součtu obsahu akumulátoru s obsahem proměnné „Cislo2“. Po nahlédnutí do instrukční sady CPU v [1] je zřejmé, že pro načtení obsahu paměťové buňky do akumulátoru bude vhodná instrukce LDA. Pro součet ale nalézáme dvě různé instrukce a to ADC a ADD. Instrukce ADC provádí operaci  $A \leftarrow (A) + (M) + (C)$ , kdežto ADD provádí  $A \leftarrow (A) + (M)$ . Rozdíl tedy spočívá u ADC v připočítání příznaku přenosu C z příznakového registru. ADC tedy použijeme v případě, když potřebujeme přičíst příznak přenosu do vyššího řádu z předchozí operace součtu. Pro vykonání první operace součtu použijeme tedy instrukci ADD, která s příznakem C nepočítá. Při výpočtu součtu dvou čísel o velikosti 1 bajt může dojít k přenosu do vyššího řádu. Proto bychom měli programově zajistit, aby hodnota příznakového bitu C byla přičtena k vyššímu bajtu proměnné „Vysledek“ viz obrázek 4. K tomu nám poslouží právě instrukce ADC. Nesmíme ale zapomenout před vlastním výpočtem vyšší bajt proměnné „Vysledek“ vynulovat. To můžeme provést elegantně instrukcí CLR, která existuje i ve variantách pro rychlé nulování registrů A a X. Výsledky součtů budeme ukládat do proměnné „Vysledek“ pomocí instrukce STA.







Obrázek 4: Ukázka sečtení dvou 1B čísel s 2B výsledkem.

Vlastní programové řešení je uvedeno níže v rámci programu 1.9. Na jeho začátku je provedena rezervace paměti pro 2 proměnné typu byte „Cislo1“ a „Cislo2“ direktivou DS.B a 1 proměnné typu word „Vysledek“ direktivou DS.W. Po inicializaci obsahu proměnných následuje vynulování vyššího bajtu proměnné „Vysledek“. Dále se provede součet obsahu proměnných a výsledek se uloží do nižšího bajtu proměnné „Vysledek“. K vyššímu bajtu výsledku se pomocí instrukce ADC přičte hodnota příznaku přenosu C.

Program 1.9:

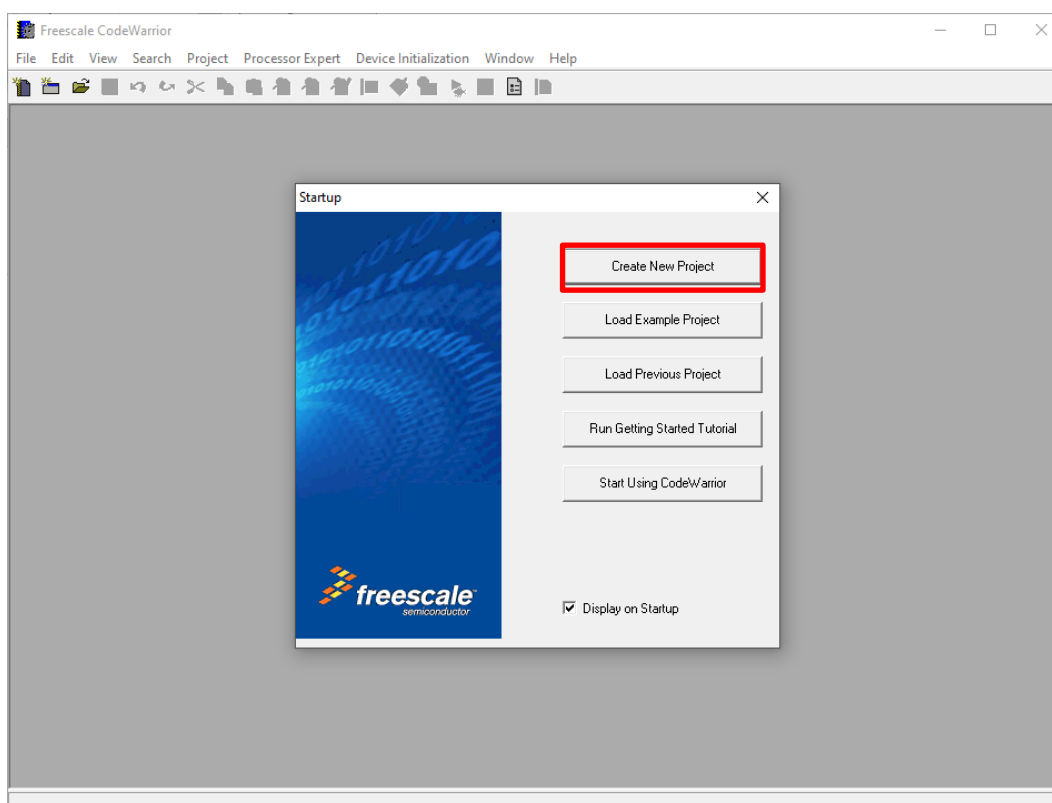
	ORG	\$80	; Proměnné budou umístěny od adresy \$80 (RAM)
Cislo1:	DS.B	1	; Rezervace 1 B paměti pro proměnnou Cislo1 (typ byte)
Cislo2:	DS.B	1	; Rezervace 1 B paměti pro proměnnou Cislo2 (typ byte)
Vysledek:	DS.W	1	; Rezervace 2 B paměti pro proměnnou Vysledek (typ word)
	ORG	\$182C	; Program bude umístěn od adresy \$182C (FLASH)
_Startup:	MOV	#128,Cislo1	; Inicializace obsahu proměnné Cislo1
	MOV	#192,Cislo2	; Inicializace obsahu proměnné Cislo2
	CLR	Vysledek	; Vynuluj vyšší bajt proměnné Vysledek
	LDA	Cislo1	; Načti obsah proměnné Cislo1 do akumulátoru
	ADD	Cislo2	; Načti obsah proměnné Cislo2 do akumulátoru
	STA	Vysledek+1	; Ulož výsledek součtu do nižšího bajtu proměnné Vysledek
	LDA	Vysledek	; Načti do akumulátoru vyšší bajt výsledku
	ADC	#0	; Přičti k obsahu akumulátoru 0 a příznakový bit C
	STA	Vysledek	; Ulož výsledek do vyššího bajtu proměnné Vysledek
	BRA	*	; Skok na sebe sama, konec programu



## Zprovoznění programu

Funkci programu ověříme ve vývojovém prostředí Freescale CodeWarrior IDE integrující editor zdrojového kódu, příslušné překladače a linkery, debugger a Full-Chip simulátor. Podporovanými programovacími jazyky jsou C, C++ a jazyk symbolických adres. Také je možno vytvářet smíšené projekty C a jazyka symbolických adres.

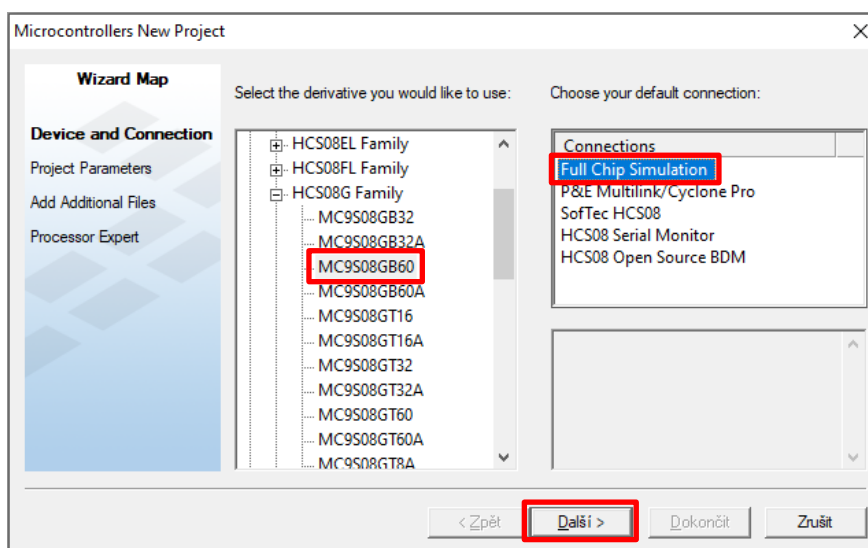
Po spuštění vývojového prostředí se nám zobrazí prázdná plocha hlavního okna aplikace, nad kterou je zobrazen dialog „Startup“ nabízející nejčastěji používané funkce po spuštění. V okně „Startup“ klikneme na tlačítko „Create New Project“ pro vytvoření nového projektu, viz obrázek 5. Touto akcí se spustí průvodce novým projektem (obrázek 6), který nás v prvním kroku nazvaném „Device and Connection“ vyzve k výběru cílového mikropočítače a způsobu jeho připojení. Ve stromu všech podporovaných mikropočítačů rozbalíme položku HCS08 kliknutím na (+) vedle názvu, dále vybereme rodinu HCS08G a pod ní již vidíme jednotlivé typy mikropočítačů. Z nabízeného seznamu zvolíme MC9S08GB60 a způsob připojení „Full Chip Simulation“, tj. běh programu bude simulován na PC.



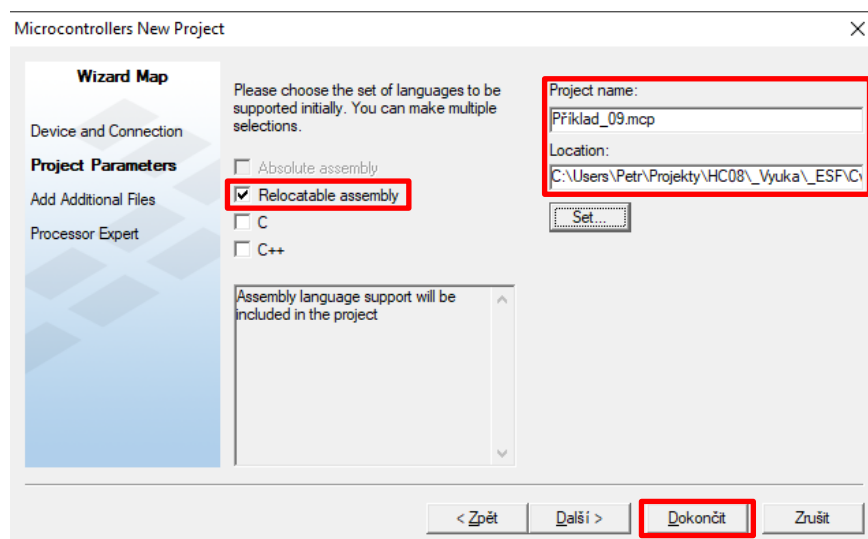
Obrázek 5: Hlavní okno aplikace CW IDE po spuštění.

Klikneme na tlačítko další, čímž se dostáváme k nastavení parametrů projektu, tj. k výběru programovacího jazyka, umístění projektu a specifikace jeho názvu. Zatrhneme pole „Relocatable assembly“, ostatní ponecháme neoznačena. V případě označení C a jazyka symbolických adres zároveň, je možné, ale vznikl by smíšený projekt, který v této chvíli nepotřebujeme. V pravé části okna zadejte do pole „Project name“ název projektu a klikneme na tlačítko „Set ...“ nastavte vhodnou složku na disku umožňující zápis (obrázek 7).

Všechna potřebná nastavení projektu jsou již pro náš program hotova, můžeme tedy průvodce ukončit kliknutím na tlačítko „Dokončit“.

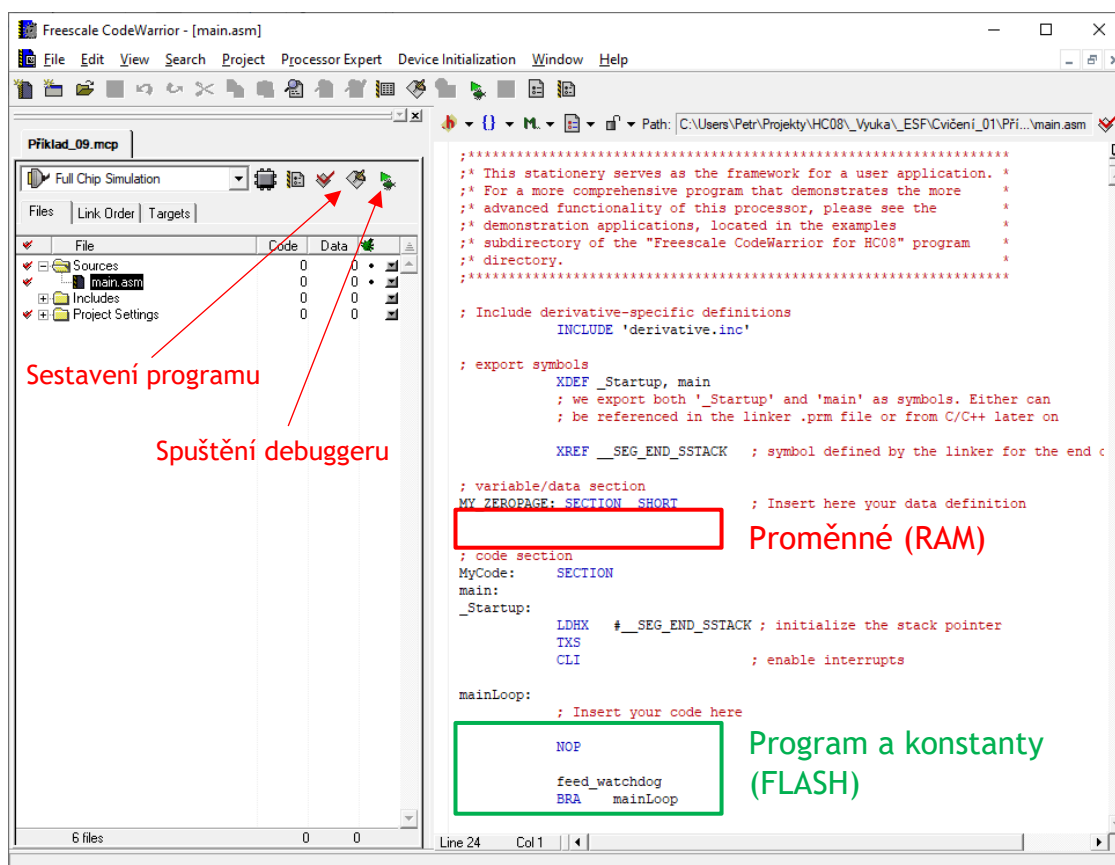


Obrázek 6: Průvodce novým projektem - výběr mikropočítače.



Obrázek 7: Průvodce novým projektem - parametry projektu.



Po dokončení průvodce novým projektem je vygenerována kostra nového programu obsahující vše nezbytné pro usnadnění tvorby nového programu. V levé části okna vývojového prostředí je umístěno podokno s aktuálně otevřeným projektem. V záložce „Files“ je k dispozici stromová struktura projektu přehledně zobrazující soubory projektu v kategoriích „Sources“ (zdrojové kódy programu), „Includes“ (obdoba hlavičkových souborů z C jazyka, zde mají příponu \*.inc) a „Project Settings“ (nastavení linkeru). Pro zobrazení zdrojového kódu, který byl připraven průvodcem novým projektem, rozklikneme ve stromové struktuře složku „Sources“ a dvojitým kliknutím levým tlačítkem myši otevřeme soubor „main.asm“. Jeho obsah se zobrazí v pravé části okna v editoru zdrojových textů (obrázek 8). Na začátku souboru je již nám známá direktiva INCLUDE, která zajistí vložení obsahu souboru „derivative.inc“ do zdrojového kódu. Tento vkládaný soubor se odkazuje na soubor „MC9S08GB60.inc“ obsahující veškeré definice periferních registrů mikropočítače MC9S08GB60 včetně masek pro přístup k jednotlivým bitům registrů. Součástí „derivative.inc“ je také definice makra „feed\_watchdog“ pro provedení resetu watchdog časovače.

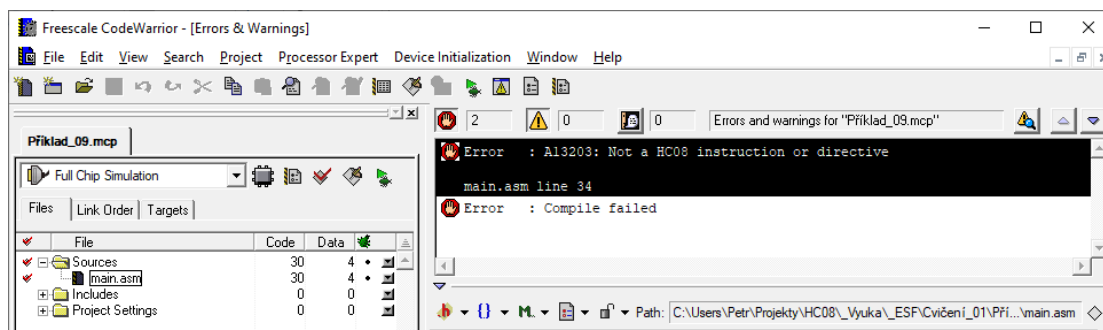


Obrázek 8: Pracovní okna Code Warrior IDE.

Dále jsou ve zdrojovém textu definovány pomocí direktiv SECTION dvě sekce, z nichž každá má své speciální určení. První z nich nazvaná „MY\_ZEROPAGE“ odkazuje do oblasti paměti RAM od adresy \$0080 a je tedy určena pro umístění proměnných programu. Druhá sekce nazvaná „MyCode“ je v paměti umístěna do oblasti paměti FLASH od adresy \$182C. Tato je vyhrazena pro uložení programu a konstantních datových struktur v nevolatilní paměti uchovávající obsah i po vypnutí napájení mikropočítače.

Programový kód začíná za návěštím „\_Startup“. Již vložený programový kód následující bezprostředně za návěštím vždy v programu ponecháváme, protože jeho účelem je inicializace ukazatele zásobníkové paměti a povolení maskovatelných přerušení. Vlastní program píšeme až za tyto tři instrukce. Makro „feed\_watchdog“ slouží pro vynulování watchdog časovače jehož cílem je hlídat správný běh programu. Pokud by program takzvaně „zabloudil“ a následně se zasekl, neprováděl by vynulování časovače a ten by po uplynutí nastavitelné prodlevy provedl automatický restart mikropočítače. Program připravený průvodcem je kompletní a lze jej tedy bez problémů přeložit a spustit. Jeho funkce je následující: provede inicializaci ukazatele zásobníku, povolí přerušení, provede prázdnou instrukci NOP, provede reset watchdog časovače a nepodmíněně skočí instrukcí BRA na návěští „main-Loop“ a sekvence od instrukce NOP se tímto neustále opakuje.

Nyní zapíšeme do sekce „MY\_ZEROPAGE“ (červeně zvýrazněná oblast na obrázku 8) direktivy pro rezervaci paměti pro proměnné programu a do sekce „MyCode“ za návěští „mainLoop“ program. Provedeme sestavení programu kliknutím na ikonu  „Make“ v projektovém okně. V případě úspěšného překladu funkce proběhne bez jakýchkoli hlášení a můžeme tudíž ihned pokračovat ve spuštění debuggeru kliknutím na ikonu  „Debug“ viz obrázek 8. Ukázka chybového hlášení překladače je ukázána na obrázku 9.



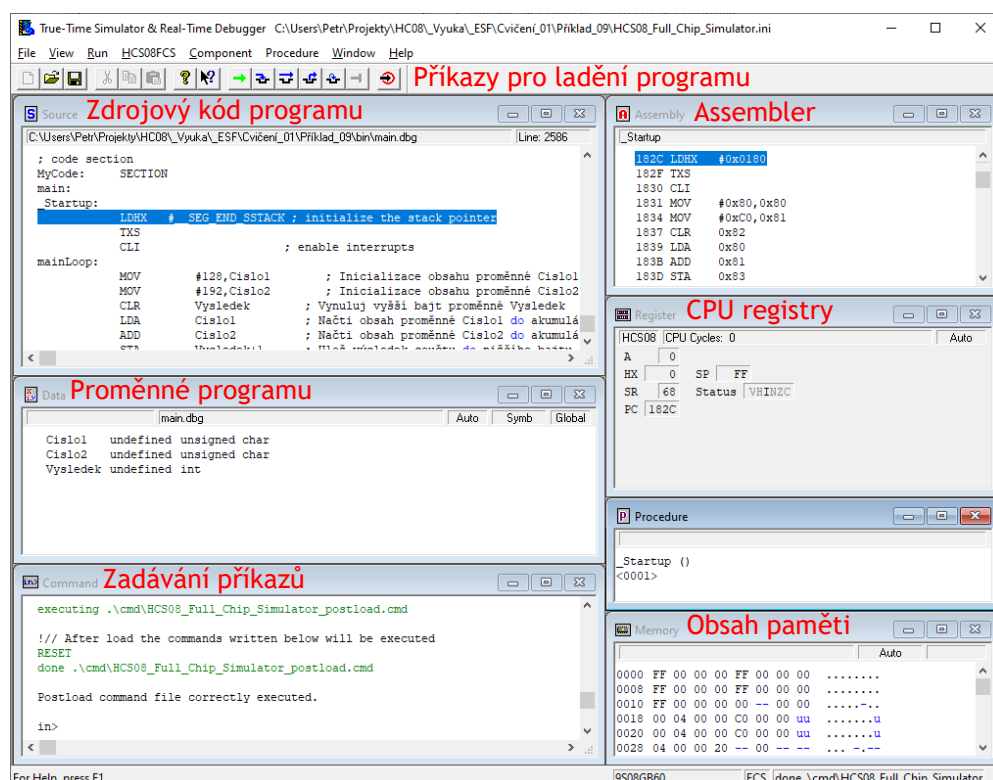
Obrázek 9: Ukázka chybového hlášení překladače.



Po spuštění debuggeru se zobrazí nové okno, jehož struktura je patrná z obrázku 10. V nástrojové liště se nachází mimo jiné velmi důležité ikony pro ovládání běhu programu, jejichž význam je následující:

- Spustí program.
- Jeden krok. Vykoná instrukci a zastaví se na následující.
- Krok přes podprogram. Provede celý podprogram a po návratu se zastaví.
- Návrat z podprogramu a zastavení po jeho provedení.
- Krok v assembleru. Má význam při krokování programu v C jazyce.
- Zastavení běhu programu.
- Restart mikropočítače.

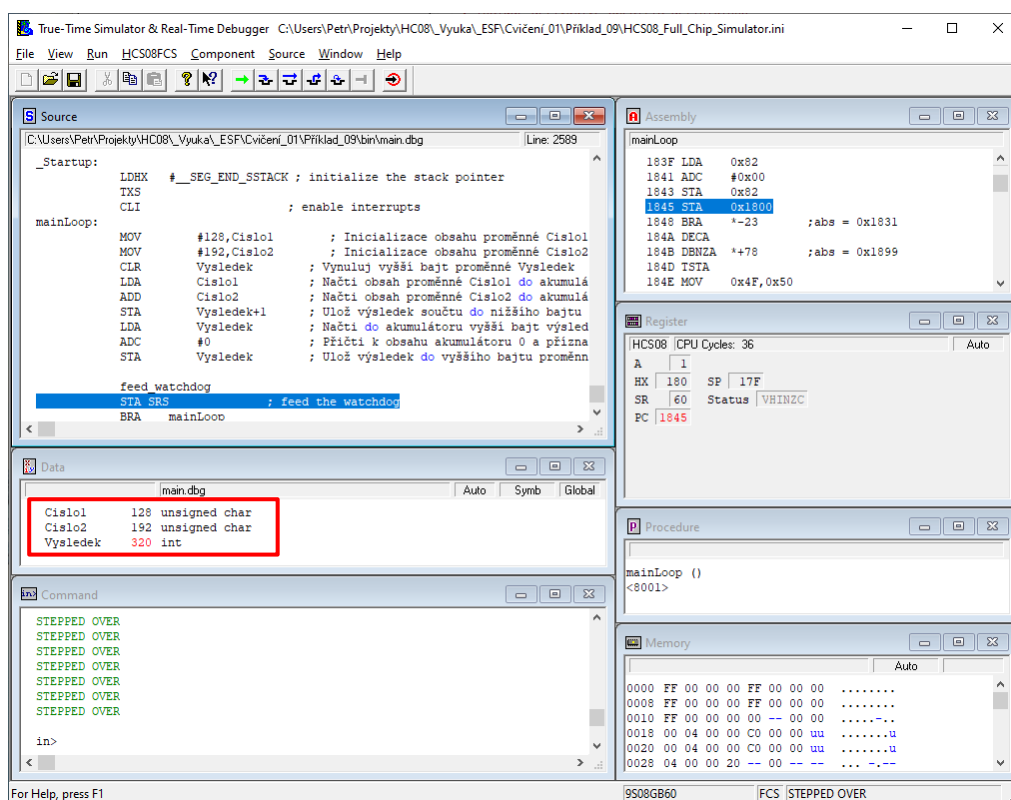
V aplikaci debuggeru se nachází 7 základních oken, ve kterých jsou přehledně prezentovány informace související s laděním programu. V okně „Source“ je zobrazován zdrojový kód programu. Modře je zvýrazněný řádek, který se bude v následujícím kroku provádět. Kliknutím pravým tlačítkem myši na vybraný řádek můžeme například nastavit zarážku, zobrazit umístění v paměti a mnoho dalších možností.



Obrázek 10: Okno debuggeru s připojeným Full-Chip simulátorem.

Obsah jednotlivých proměnných programu lze sledovat v okně „Data“. Formát zobrazených hodnot lze přizpůsobit dle požadavků kliknutím pravého tlačítka myši na vybranou položku. Také je zde možné dvojitém kliknutím na obsah proměnné změnit její hodnotu. V okně „Command“ lze zadávat příkazy debuggeru v textové podobě, zároveň jsou zde vypisována stavová hlášení programu. Hodnoty uložené v jednotlivých registrech CPU jsou zobrazovány v okně „Register“. Opět je zde možnost jejich modifikace a změny formátu zobrazení obdobně jako v okně „Data“. Obsah paměti je zobrazován v okně nazvaném „Memory“. Jsou zde k dispozici široké možnosti modifikace, jako jsou například vyplnění bloku paměti konstantou, kopírovat blok paměti, vyhledávat zadané sekvence a další. Z pohledu formátu zobrazení lze nastavit velikost slova, číselnou soustavu a zobrazování ASCII znaků.

V tento okamžik již znáte základní ovládání debuggeru a může se tedy přikročit k ověření funkce programu. Celý program odkrojujte pomocí tlačítka „Single Step“ (klávesa F11) a sledujte po každém provedení instrukce stav registrů CPU a obsah proměnných. Po provedení instrukce ADD si všimněte nastavení příznaku přenosu C do jedničky, který je následně přičten pomocí ADC k vyššímu bajtu výsledku.



Obrázek 11: Okno debuggeru po provedení programu.

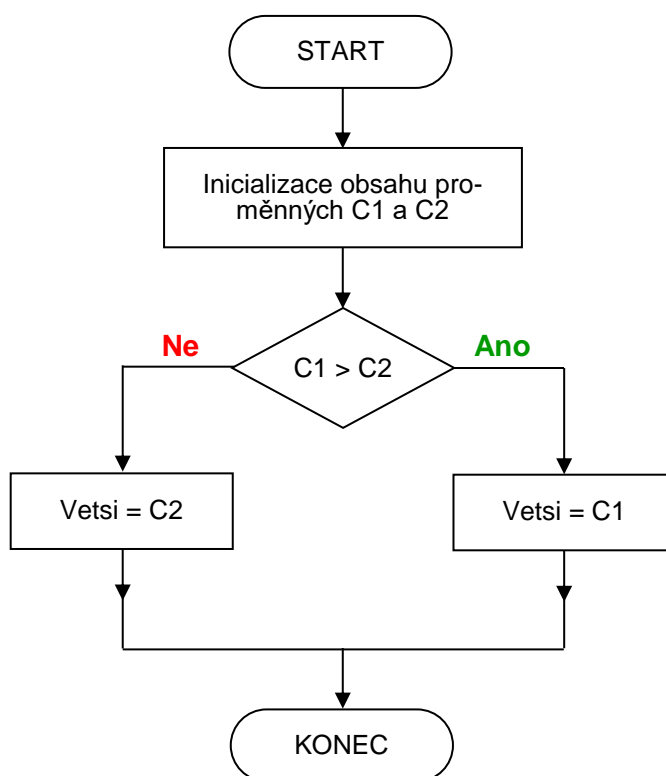
### 1.3.2 Porovnání dvou čísel

#### Zadání

V jazyce symbolických adres vytvořte program pro porovnání dvou 1B číselných hodnot bez znaménka uložených v proměnných „C1“ a „C2“. Zjištěnou větší hodnotu program uloží do proměnné „Vetsi“.

#### Řešení

Dříve než začneme se zápisem zdrojového textu programu, je vhodné nejdříve zakreslit jeho algoritmus (postup řešení dané úlohy) pomocí vývojového diagramu. Jeho použití zejména u jazyka symbolických adres velmi urychlí vývoj aplikace a vyhneme se zároveň chybám v aplikační logice. Pro tvorbu vývojových diagramů slouží standardní grafické symboly, jejichž základní výběr je uveden v příloze P I. Vývojový diagram pro program porovnání dvou čísel je uveden na obrázku 12.



Obrázek 12: Vývojový diagram programu pro porovnání dvou čísel.



Nyní převedeme algoritmus vyobrazený na vývojovém diagramu do programu v jazyce symbolických adres. Na začátku program inicializuje proměnné „C1“ a „C2“ na požadované hodnoty což lze efektivně realizovat použitím instrukcí MOV. Dále následuje blok pro rozhodování, za kterým se program podmíněně rozvětjuje do dvou větví. Pokud je hodnota uložená v „C1“ větší než „C2“, bude program pokračovat pravou větví (podmínka splněna) a do proměnné „Vetsi“ se uloží obsah „C1“. Při nesplnění podmínky program pokračuje levou větví a do proměnné „Vetsi“ uloží obsah „C2“. Pro realizaci bloku pro podmíněné větvení je nutno v jazyce symbolických adres provést nejprve porovnání pomocí instrukce CMP, která nastaví příslušné příznaky v příznakovém registru CPU. Instrukce CMP toho docílí tak, že provede interně operaci odečtení obsahu paměťové buňky od akumulátoru, přičemž výsledek rozdílu se neukládá. Nastavené příznaky následně vyhodnotí instrukce podmíněného skoku a provede či neprovede skok na dané návěští. Výběr často používaných instrukcí pro podmíněné větvení včetně příkladu použití je uveden v tabulce 4. Pro úplný přehled těchto instrukcí je nutno nahlédnout do instrukční sady v [1]. Z tabulky vyplývá, že pro realizaci podmíněného skoku při splnění podmínky  $c1 > c2$  použijeme instrukci BHI. Ta provádí otestování příznaků přenosu (C) a nulovosti (Z) a pokud jsou oba nulové, provede relativní skok na specifikované návěští.

Podmínka skoku	Stav CCR po porovnání	Ukázka programového kódu
$c1 = c2$	$(Z) = 1$	LDA c1 CMP c2 BEQ skok
$c1 \neq c2$	$(Z) = 0$	LDA c1 CMP c2 BNE skok
$c1 > c2$	$(C) \vee (Z) = 0$	LDA c1 CMP c2 BHI skok
$c1 \geq c2$	$(C) = 0$	LDA c1 CMP c2 BHS skok
$c1 < c2$	$(C) = 1$	LDA c1 CMP c2 BLO skok
$c1 \leq c2$	$(C) \vee (Z) = 1$	LDA c1 CMP c2 BLS skok

Tabulka 4: Ukázka použití vybraných instrukcí pro podmíněné skoky.



Zdrojový kód řešeného programu pro porovnání dvou číselných hodnot je uveden ve výpisu programu 1.10. Při ověřování jeho funkce v debuggeru nezapomeňte ověřit obě větve vykonávání programu modifikací obsahu proměnných „C1“ a „C2“.

*Program 1.10:*

	ORG	\$80	; Proměnné budou umístěny od adresy \$80 (RAM)
C1:	DS.B	1	; Rezervace 1 B paměti pro proměnnou C1 (typ byte)
C2:	DS.B	1	; Rezervace 1 B paměti pro proměnnou C2 (typ byte)
Vetsi:	DS.B	1	; Rezervace 1 B paměti pro proměnnou Vetsi (typ byte)
	ORG	\$182C	; Program bude umístěn od adresy \$182C (FLASH)
_Startup:	MOV	#5,C1	; Inicializace obsahu proměnné C1
	MOV	#10,C2	; Inicializace obsahu proměnné C2
	LDA	C1	; Načti obsah proměnné C1 do akumulátoru
	CMP	C2	; Načti obsah proměnné C2 do akumulátoru
	BHI	C1vetsi	; Pokud je C1>C2, skoč na návěští C1vetsi
	LDA	C2	; Skok se neprovedl -> v C2 je větší nebo stejná hodnota
	STA	Vetsi	; Ulož obsah C2 do proměnné Vetsi
	BRA	Konec	; Nepodmíněný skok na návěští Konec
C1vetsi:	STA	Vetsi	; Ulož obsah C1 do proměnné Vetsi
Konec:	BRA	*	; Skok na sebe sama, konec programu

### 1.3.3 Vynulování pole s použitím cyklu

#### Zadání

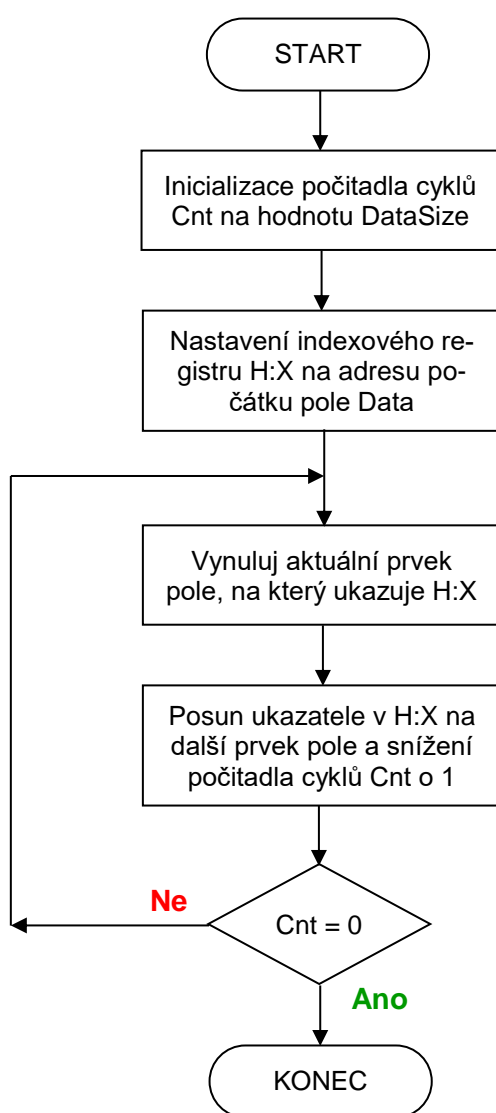
V jazyce symbolických adres vytvořte program pro vynulování všech prvků pole „Data“ datového typu byte. Pole má velikost definovanou symbolem „DataSize“. Program realizujte s použitím cyklu s testem na konci.

#### Řešení

Obdobně jako v předchozím příkladu si nejprve zakreslíme algoritmus postupu programu pomocí vývojového diagramu, který je vyobrazen na obrázku 13. Při řešení úkolů spojených s manipulací s daty v poli je výhodné použít indexovou adresaci. Na začátku algoritmu se nejprve musí inicializovat obsah indexového registru H:X na hodnotu odpovídající počáteční adrese pole „Data“ v paměti. Registr H:X bude tedy ukazovat na prvek pole,



se kterým se bude provádět příslušná operace. Dále bude zapotřebí počítadlo průchodů cyklem, aby bylo možné jej po provedení požadovaného počtu operací nulování ukončit. To bude na začátku nastaveno na hodnotu odpovídající počtu prvků pole definované symbolem „DataSize“. V těle cyklu budou prováděny následující operace: vynuluje se aktuální prvek pole, na který ukazuje H:X, sníží se počítadlo cyklů „Cnt“ o jedničku a zvýší se obsah H:X o jedničku, čímž se posune ukazatel na následující prvek pole. Na konci cyklu se provede test na nulovost proměnné počítadla cyklů „Cnt“. V případě nenulového obsahu se provede skok zpět na začátek těla cyklu, v opačném případě se skok neprovede a program skončí.



Obrázek 13: Vývojový diagram programu pro vynulování pole.

Algoritmus prezentovaný vývojovým diagramem na obrázku 13 se nyní musí převést do podoby programu v jazyce symbolických adres. Pro inicializaci obsahu počítadla cyklů „Cnt“ použijeme nám již známou instrukci MOV. Naplnění registrového páru H:X počáteční adresou pole v paměti zajistí instrukce LDHX, jejímž bezprostředním operandem je 16bitová hodnota (pole může být tedy umístěno na libovolné adrese v rámci celého adresovatelného paměťového prostoru). Tímto je inicializační část programu dokončena a následuje tělo cyklu uvozené návěštím „Loop“. Na začátku těla cyklu se provede vynulování obsahu paměťové buňky na tento účel specializovanou instrukcí CLR. Ze zápisu operandu vidíme, že bude použita indexová adresace vztažená k registru H:X bez offsetu. To znamená, že efektivní adresa operandu je obsažena v registru H:X bez přičítání offsetu. Tímto je obsah příslušné paměťové buňky vynulován a musí se provést příprava na další průchod tělem smyčky. Ta spočívá ve zvýšení obsahu indexového registru H:X o jedničku instrukcí AIX, čímž se ukazatel posune na další prvek pole. Počítadlo cyklů „Cnt“ se naopak o jedničku sníží. Následuje instrukce podmíněného skoku BNE, která provede skok na návěští „Loop“, je-li výsledek předchozí operace nenulový (příznak Z v CCR je roven nule). Všimněte si, že před instrukcí BNE nebylo zapotřebí použít instrukci pro porovnání CMP. Je to dáno tím, že testujeme pouze nulovost předchozí operace po dekrementaci.

Program 1.11:

DataSize	EQU	5	; Symbol DataSize definuje velikost pole v bajtech
	ORG	\$80	; Proměnné budou umístěny od adresy \$80 (RAM)
Data:	DS.B	DataSize	; Rezervace DataSize bajtů paměti pro pole Data (typ byte)
Cnt:	DS.B	1	; Rezervace 1 B paměti pro proměnnou Cnt (typ byte)
	ORG	\$182C	; Program bude umístěn od adresy \$182C (FLASH)
_Startup:	MOV	#DataSize,Cnt	; Do proměnné Cnt ulož velikost pole
	LDHX	#Data	; Načti adresu pole „Data“ do indexového registru H:X
Loop:	CLR	,X	; Vynuluj obsah paměťové buňky, na kterou ukazuje H:X
	AIX	#1	; Přičti k obsahu H:X jedničku = posun na další prvek pole
	DEC	Cnt	; Sniž počítadlo cyklů Cnt o jedničku (instrukce nastaví CCR)
	BNE	Loop	; Skok na Loop, pokud je příznak Z v CCR nulový
Konec:	BRA	*	; Skok na sebe sama, konec programu

## 1.4 Zadání samostatné práce

1. V jazyce symbolických adres realizujte program provádějící aritmetický součet, rozdíl, součin a podíl obsahu dvou proměnných „C1“ a „C2“ typu byte. Výsledky operací se budou ukládat do proměnných nazvaných „Soucet“, „Rozdil“, „Soucin“ a „Podil“. Proměnná „Soucin“ bude typu word (16bitový výsledek), ostatní typu byte (8 bitový výsledek). Pro realizaci programu budou zapotřebí instrukce ADD, SUB, MUL a DIV.
2. Doplňte program 1.10 z řešeného příkladu tak, aby navíc našel v uvedených proměnných minimální hodnotu a uložil ji do proměnné „Mensi“.
3. Vytvořte program pro vyhledání maximální hodnoty čísel bez znaménka uložených ve třech proměnných typu byte nazvaných „C1“, „C2“ a „C3“. Nalezenou maximální hodnotu umístěte do proměnné „Cmax“. Před vlastní tvorbou programu si nejprve nakreslete vývojový diagram prezentující algoritmus řešení úlohy. Funkci programu ověřte pro všechny alternativní větve programu zadáním vhodných vstupních údajů do proměnných.
4. Upravte program 1.11 z řešeného příkladu tak, aby místo nulování pole jej vyplňoval libovolnou konstantou typu byte definovanou symbolem „Konst“.
5. Proved'te modifikaci programu 1.11 na vyplňování pole aritmetickou posloupností definovanou vztahem  $a_{n+1} = a_n + 5$ , kde nultý člen posloupnosti  $a_0 = 0$ . Po proběhnutí programu by tedy mělo pole obsahovat hodnoty 0, 5, ..., 20.
6. Vytvořte podprogram pro vyplnění zadaného pole konstantou. Předávané argumenty budou: indexový registr H:X – adresa počátku pole, akumulátor A – počet prvků pole, zásobník – konstanta pro vyplnění pole. Vytvořený podprogram nesmí používat globální pomocné proměnné, pouze lokální na zásobníku. Před vlastním řešením si prostudujte program 1.8.

## 2 BINÁRNÍ VSTUPY/VÝSTUPY FRDM-KL25Z

Mezi základní dovednosti programování mikropočítačů patří zcela jistě obsluha binárních vstupů a výstupů (GPIO). Ty jsou ve větším počtu seskupeny do portů (označené většinou písmeny A, B, C, ..., atd.), které představují bránu pro základní komunikaci s okolím. Maximální počet vstupně / výstupních (V/V) pinů integrovaných do jednoho portu závisí na architektuře mikropočítače. U 8bitových mikropočítačů nalezneme porty s maximálně 8 V/V piny, u 32bitových, což je i případ vývojové desky FRDM-KL25Z, může být na jednom portu až 32 V/V pinů. Je to dáno efektivitou přístupu CPU na daný port, tj. CPU s 32bitovou datovou sběrnicí je schopen v rámci jediné operace zapsat údaj o šířce 32bitů, kdežto 8bitový CPU by musel provést minimálně 4 operace (ve skutečnosti dle aktuálně prováděné činnosti s V/V i mnohem více).

Na školním výukovém kitu, jehož součástí je deska NXP FRDM-KL25Z, je na binární vstupy a výstupy připojena celá řada periférií – od těch nejjednodušších, jako jsou například indikační diody (LED) a tlačítka, až po ty složitější, které reprezentuje LCD displej obsahující vlastní radič s pamětí ROM i RAM. V rámci tohoto cvičení se naučíte, jak programově V/V porty obsluhovat. Základní informace o mikropočítači a jeho propojení s perifériemi kitu jsou dostupné v doprovodné prezentaci k laboratorním cvičením.

### 2.1 Inicializace vstupně/výstupního rozhraní

Dříve než můžeme začít zapisovat či číst informace na V/V pinech portu, je nutné provést inicializaci SIM a PORT modulu dle funkčních požadavků na používané piny. Poté již můžeme v modulu GPIO nastavit směr pinů (vstupní nebo výstupní režim) a za použití odpovídajících registrů číst či zapisovat požadované hodnoty na vybrané piny.

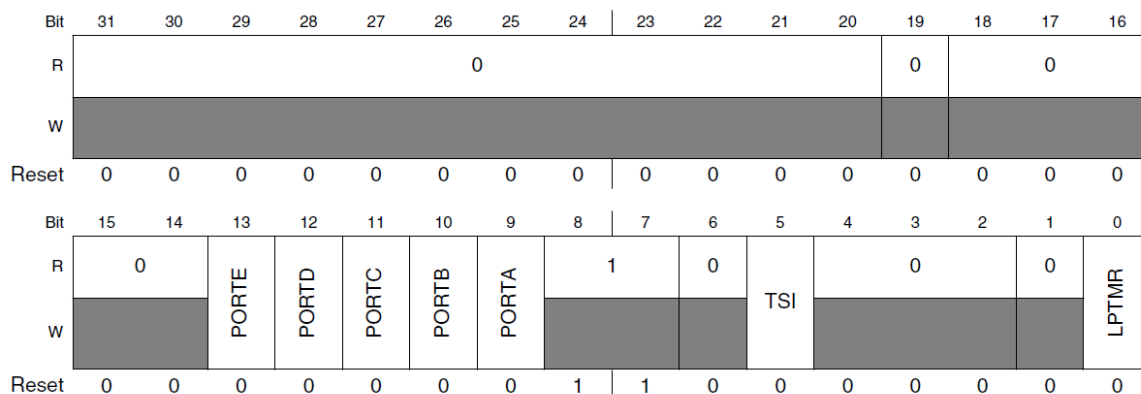
#### 2.1.1 Modul řízení systému a jeho konfigurace (SIM)

Modul SIM poskytuje funkce související s distribucí hodinového signálu do jednotlivých modulů mikropočítače, čímž do značné míry dokáže ovlivňovat spotřebu elektrické energie nutné k jeho běhu. Pro úsporu energie je po resetu mikropočítače řada modulů odpojena od zdroje hodinového signálu. V případě chybné či opomenuté inicializace a následném použití periférie s neaktivním hodinovým signálem nastane výjimka Hard Fault a provede se příslušná obsluha přerušení. Defaultně je uvnitř neinicializovaných obsluh přerušení



vložena nekonečná smyčka a tím je běh programu zastaven, tj. z přerušení se již nevrátí. Kromě distribuce hodinového signálu modul také zajišťuje konfiguraci velikosti systémové FLASH a RAM paměti a funkce napěťového regulátoru USB rozhraní [2].

Pro zajištění korektní funkce PORT modulů je zapotřebí povolit hodinový signál pro všechny používané porty v aplikaci. Toto nastavení umožňuje registr SIM\_SCGC5 na bitech 9 až 13 nazvaných PORTA až PORTE zápisem jedničky, viz obrázek 14.



Obrázek 14: Struktura registru SIM\_SCGC5 [2].

Význam jednotlivých konfiguračních bitů v SIM\_SCGC5 je následující [2]:

- *PORTE* – hodinový signál pro PORTE zakázán (0) / povolen (1)
- *PORTD* – hodinový signál pro PORTD zakázán (0) / povolen (1)
- *PORTC* – hodinový signál pro PORTC zakázán (0) / povolen (1)
- *PORTB* – hodinový signál pro PORTB zakázán (0) / povolen (1)
- *PORTA* – hodinový signál pro PORTA zakázán (0) / povolen (1)
- *TSI* – programový přístup k TSI rozhraní zakázán (0) / povolen (1)
- *LPTMR* – programový přístup k LPTMR časovači zakázán (0) / povolen (1)

### 2.1.2 Modul řízení portů a přerušení (PORT)

Modul řízení portů a přerušení (PORT) zajišťuje funkce související s řízením funkce portu a externích přerušení. Mikropočítač KL25Z128M4 je vybaven celkem pěti porty označenými písmeny A až E. Každý z těchto portů má vlastní nezávislý řídicí PORT modul, tj. PORTA, PORTB, až PORTE. Každý pin portu má vyhrazen 32bitový konfigurační registr PORTx\_PCRn, jehož struktura je vyobrazena na obrázku 15.



Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	
R	0								ISF	0				IRQC			
W									w1c								
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0					
R	0					MUX			0	DSE		0	PFE		0	SRE		PE		PS	
W																					
Reset	0	0	0	0	0	x*	x*	x*	0	x*	0	x*	0	x*	x*	x*					

\* Notes:

- x = Undefined at reset.

Obrázek 15: Struktura registru PORTx\_PCRn [2].

Význam jednotlivých konfiguračních bitů v PORTx\_PCRn je následující [2]:

- *ISF* – příznakový bit požadavku na přerušení (0 = nedetekováno)
- *IRQC* – konfigurace přerušení (detekce náběžné, sestupné hrany, atd.)
- *MUX* – výběr alternativních funkcí pinu (MUX control) s volbami:
  - ALT0 – pin zakázán nebo analogový vstup
  - ALT1 – GPIO funkce
  - ALT2 – ALT7: další speciální funkce závislé na konkrétním pinu viz prezentace snímky 58 – 64.
- *DSE* – výběr výstupního proudu pinu na normální 5 mA (0) nebo vysokou hodnotu proudu 18 mA (1). Poznámka: Vysokou hodnotu proudu podporují pouze piny PTB0, PTB1, PTD6 a PTD7.
- *PFE* – pasivní filtr zakázán (0) / povolen (1)
- *SRE* – rychlost přeběhu signálu vysoká (0) / nízká (1)
- *PE* – Interní Pull-Up nebo Pull-Down rezistor je zakázán (0) / povolen (1)
- *PS* – Výběr Pull-Down (0) nebo Pull-Up (1) rezistoru

Pro základní funkci pinu portu v režimu binární vstup/výstup postačí pouze nastavit bity MUX tak, aby byla aktivní funkce ALT1 odpovídající režimu GPIO. Při vstupním režimu může být nutné v některých případech aktivovat interní Pull-Up či Pull-Down rezistor. Stav ostatních konfiguračních bitů registru je vhodné nastavit na 0, protože jejich stav je u některých z nich po resetu mikropočítače nedefinovaný. Popis dalších registrů PORT modulu PORTx\_GPCLR, PORTx\_GPCHR a PORTx\_ISFR naleznete v [2].



## 2.2 Obsluha vstupně/výstupního rozhraní (GPIO)

Po provedení inicializace SIM a PORT modulů je již možno pracovat s vlastním GPIO rozhraním, které zprostředkovává zápis a čtení hodnot z fyzických pinů mikropočítače. GPIO modul pro tento účel poskytuje šest 32bitových registrů [2]:

- GPIOx\_PDOR – výstupní datový registr portu
- GPIOx\_PSOR – výstupní datový registr portu pro nastavení výstupu do 1
- GPIOx\_PCOR – výstupní datový registr portu pro nastavení výstupu do 0
- GPIOx\_PTOR – výstupní datový registr portu pro přepnutí stavu výstupu
- GPIOx\_PDIR – vstupní datový registr portu
- GPIOx\_PDDR – registr pro nastavení režimu pinu vstup / výstup

Za x se v názvech registrů dosadí označení portu písmeny A až E. Pro základní práci se vstupy a výstupy na příslušném portu postačí registr pro nastavení směru pinů GPIOx\_PDDR a vstupní a výstupní datové registry GPIOx\_PDIR a GPIOx\_PDOR.

### 2.2.1 Nastavení režimu pinů pro vstup/výstup

Pro nastavení vstupního či výstupního režimu pinu na daném portu slouží registr GPIOx\_PDDR, kde každý z bitů tohoto 32bitového registru nastavuje odpovídající pin. To znamená, že například 0. bit registru GPIOA\_PDDR nastavuje režim pinu PTA0, 1. bit registru GPIOA\_PDDR nastavuje režim pinu PTA1 a tak dále až do 31. bitu. Nula zapsaná do odpovídajícího bitu přepne pin do vstupního režimu, jednička do výstupního režimu [2].

### 2.2.2 Zápis hodnoty na výstupní piny

Zápis na výstupní piny portu se realizuje pomocí 32bitového výstupního datového registru GPIOx\_PDOR. Obdobně jako u předchozího registru každý bit koresponduje s odpovídajícím výstupním pinem. Například při zápisu hodnoty 1 do 5. bitu registru GPIOA\_PDOR se nastaví na výstupním pinu PTA5 logická 1, při zápisu 0 do 6. bitu registru se provede nastavení logické 0 na pinu PTA6 (za předpokladu, že je daný pin nakonfigurován do výstupního režimu). Pokud daný pin není nastaven do výstupního režimu, obsah registru se příslušně modifikuje, přičemž stav pinu a jeho funkce nebude ovlivněna [2].





### 2.2.3 Čtení stavu vstupních pinů

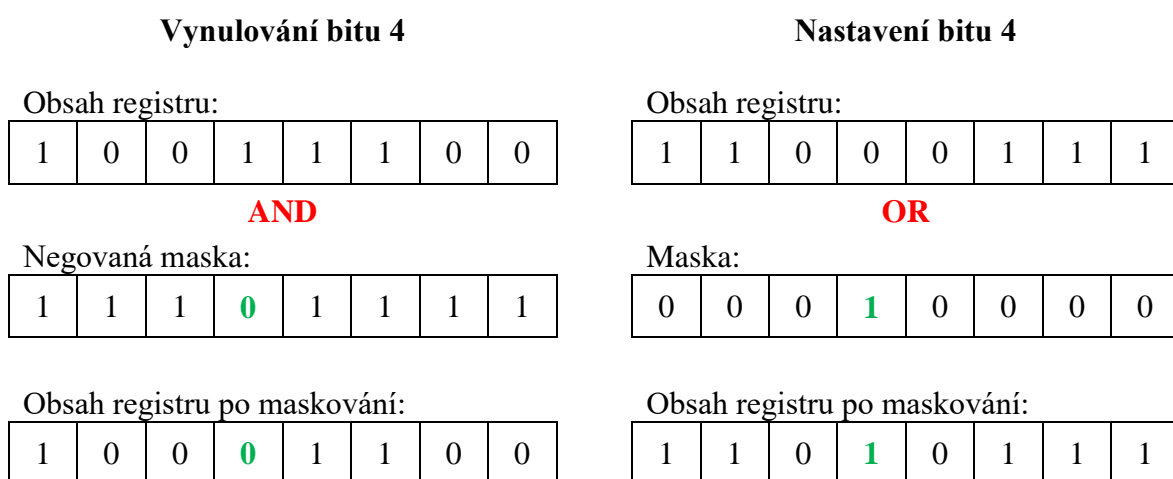
Čtení stavu vstupních pinů portu zajišťuje 32bitový vstupní datový registr GPIOx\_PDIR. Jednotlivé bity tohoto registru indikují stav odpovídajících vstupních pinů portu. Pokud není pin nakonfigurován na digitální funkci nebo není na mikropočítači implementován, čtení daného vstupu vrací vždy nulu [2].

## 2.3 Práce s jednotlivými bity - maskování

Při modifikacích obsahu registrů nebo paměťových buněk je velmi často zapotřebí provést změnu jen určitého bitu v registru se zachováním obsahu všech ostatních, případně zjistit stav určitého bitu při operaci čtení. K tomu nám slouží takzvané maskování, které lze rozčlenit do tří základních operací:

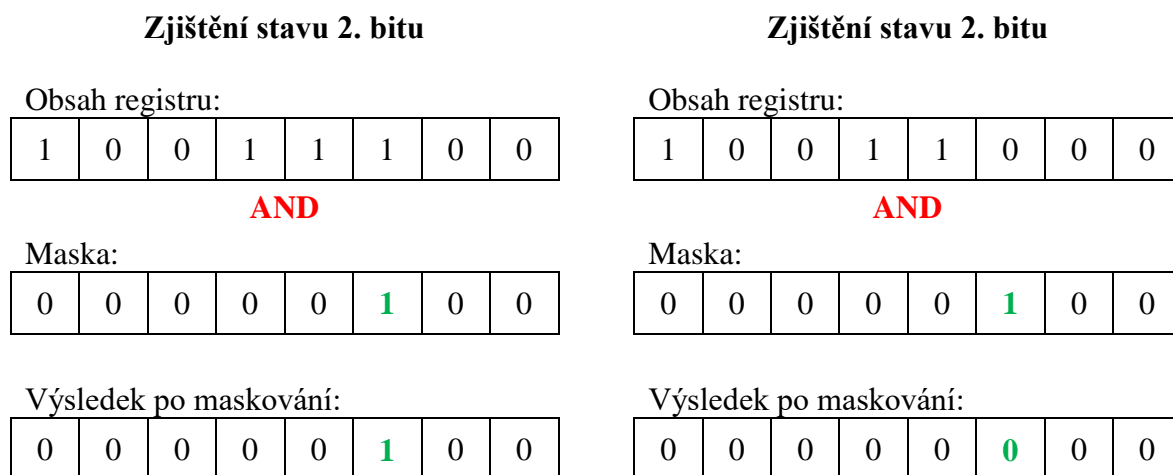
- Přechzení obsahu registru, u kterého chceme zapsat nový stav vybraného bitu.
- Provedení odpovídající logické operace mezi přečteným obsahem a maskou.
- Zápis výsledku logické operace zpět do registru.

Masku vytvoříme na základě čísla modifikovaného bitu v rámci paměťové buňky či registru. Na této pozici bitu v masce bude zapsána 1, zbytek bitů se ponechá roven nule. V jazyce C lze masku velmi efektivně vytvořit pomocí operace rotace ( $1 \ll n$ ), kde  $n$  představuje pozici bitu v masce. Další postup záleží na tom, zda se zapisuje na požadovanou pozici 0 nebo 1. Při zápisu 0 se provede logický součin obsahu registru s negovanou maskou, při zápisu 1 logický součet s maskou. Celý postup maskování je vyobrazen na obrázku 16.



Obrázek 16: Maskování - nastavení bitu na požadovanou hodnotu.

Dalším velmi častým řešeným programovým problémem je zjištění, v jakém stavu je určitý bit v registru či paměťové buňce. Opět se vytvoří maska dle postupu uvedeného v předchozím případě a provede se její logický součin s registrem nebo paměťovou buňkou, jejíž obsah se má otestovat. Výsledkem logického součinu je buď nulová hodnota indikující přítomnost 0 v testovaném bitu, anebo nenulová a potom je v daném bitu uložena 1. Způsob použití maskování pro zjištění stavu vybraného bitu je vyobrazen na obrázku 17.



Obrázek 17: Maskování - zjištění stavu vybraného bitu.

## 2.4 Ukázkové programy

V této kapitole naleznete ukázkové řešené příklady související s programovou obsluhou binárních vstupů / výstupů v C jazyce. V rámci laboratorního cvičení si funkci programů důkladně vyzkoušejte jejich odkrokováním v příslušném programovém prostředí.

### 2.4.1 Obsluha LED na výukovém kitu

#### Zadání

V jazyce C vytvořte program, který bude periodicky střídavě rozsvěcovat a zhasínat červenou a zelenou LED na výukovém kitu s periodou 500 ms. Pro realizaci časového zpoždění vytvořte podprogram *Cekej()*, jehož vstupním argumentem typu `uint32_t` bude doba čekání v milisekundách.



## Řešení

Nejprve si musíme z průvodní prezentace k laboratorním cvičením zjistit, jak a na které piny portů mikropočítače jsou fyzicky LED připojeny. Z prezentace (viz snímek 9) vyplývá, že všechny 3 LED jsou připojeny k portu B:

- Červená LED – pin PTB8
- Žlutá LED – pin PTB9
- Zelená LED – pin PTB10

Výstupy jsou přes rezistory omezujícími proud diodou na přibližně 3 mA připojeny na katody LED a jejich anody na napájecí větev Vcc (+3 V). Z výše uvedeného vyplývá, že LED svítí při logické 0 a zhasnutá je při logické 1 zapsané na odpovídající binární výstup.

Nyní se znalostí připojení obsluhované periferie může začít s tvorbou programu. Dříve než budeme zapisovat do výstupních registrů GPIO, musí se provést inicializace SIM a PORT modulu, jinak by došlo k výjimce Hard Fault a běh programu by se zastavil. Nejprve musíme povolit distribuci hodinového signálu pro PORTB modul zápisem 1 do 10. bitu registru SIM\_SCGC5. Od tohoto okamžiku již můžeme začít používat PORTB modul, tj. přistupovat k jeho registrům. V odpovídajících řídicích registrech portu B nastavíme alternativní funkci pinu ALT1 v bitovém poli MUX odpovídající GPIO funkcionalitě, do ostatních řídicích bitů zapíšeme 0. Konkrétně se tato inicializace bude týkat registrů PORTB\_PCR8 (červená LED) a PORTB\_PCR10 (zelená LED). Do zmíněných registrů nejprve zapíšeme nulu a následně nastavíme 8. bit na 1, čímž je inicializace PORTB modulu dokončena. Jako poslední nastavíme v GPIO modulu výstupní režim pinů PTB8 a PTB10. K tomuto účelu slouží registr GPIOB\_PDDR kde do 8. a 10. bitu zapíšeme 1. Tímto je inicializační část programu dokončena a dále již budeme v nekonečné smyčce v daných časových okamžicích zapisovat na příslušné výstupy 0 pro rozsvícení LED a 1 pro zhasnutí LED prostřednictvím výstupního datového registru GPIOB\_PDOR. Všechny bitové operace budeme provádět pomocí maskování. Negaci požadovaného bitu v registru lze efektivně provést pomocí logické funkce XOR a odpovídající masky. Pro lepší přehlednost programového kódu si jednotlivé masky nadefinujeme pomocí direktivy pro definici makra #define. Podprogram pro čekání zrealizujeme nejjednodušším možným způsobem – smyčkou pro vytížení procesoru na požadovaný čas. Smyčka není kalibrovaná pro přesné dodržení nastavených časů, ale pro daný



příklad je postačující. Podstatně vyšší přesnosti nastavení doby čekání by mohlo být dosaženo časovačem, který bude součástí některého z dalších cvičení. Aby smyčka trvala déle, zabráníme překladači optimalizaci řídicí proměnné cyklu, což je zajištěno klíčovým slovem `volatile` v její definici. Zdrojový kód programu naleznete níže ve výpisu programu 2.1.

Program 2.1:

```
// Definice masek pro červenou a zelenou LED
#define M_LED_R    (1<<8)
#define M_LED_G    (1<<10)

// Prototyp funkce Cekej
void Cekej (uint32_t delay_ms);

int main(void)
{
    // Inicializace SIM, PORTB a GPIO modulů
    SIM->SCGC5 |= (1<<10);           // Povolení hodinového signálu PORTB modulu
    PORTB->PCR[8] = 0;
    PORTB->PCR[8] |= (1<<8);         // Nastavení MUX PTB8 na ALT1 (GPIO)
    PORTB->PCR[10] = 0;
    PORTB->PCR[10] |= (1<<8);        // Nastavení MUX PTB10 na ALT1 (GPIO)
    GPIOB->PDDR |= M_LED_R;         // Pin s připojenou červenou LED výstupní režim
    GPIOB->PDDR |= M_LED_G;         // Pin s připojenou zelenou LED výstupní režim
    GPIOB->PDOR &= ~M_LED_R;        // Rozsvícení červené LED
    GPIOB->PDOR |= M_LED_R;         // Zhasnutí zelené LED

    while(1)
    {
        Cekej(250);                // Čekaj 250 ms

        // Ukázka použití logické funkce XOR
        GPIOB->PDOR ^= M_LED_R;    // Negace stavu červené LED

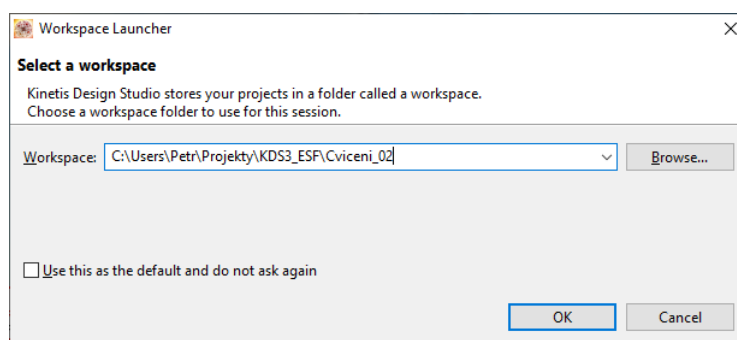
        // Ukázka použití logických funkcí AND a OR
        // Rozsvítíme / zhasneme zelenou LED na základě minulého stavu výstupu
        if (GPIOB->PDOR & M_LED_G)
            GPIOB->PDOR &= ~M_LED_G; // Rozsvícení zelené LED
        else
            GPIOB->PDOR |= M_LED_G;  // Zhasnutí zelené LED
    }
}
```

```
void Cekej(uint32_t delay_ms)
{
    static volatile uint32_t i;
    for(i = 0; i < 2333 * delay_ms; i++);
}
```

## Zprovoznění programu

Funkci programu ověříme ve vývojovém prostředí NXP Kinetis Design Studio určeného pouze pro mikrokontroléry Kinetis s ARM jádrem. Prostředí v sobě integruje editor zdrojového kódu, příslušné překladače, linkery a debugger podporující řadu hardwarových ladících a nahrávacích rozhraní. Podporovanými programovacími jazyky jsou C, C++ a jazyk symbolických adres. Vývojové prostředí je poskytováno zdarma na webových stránkách NXP v sekci Design / Software.

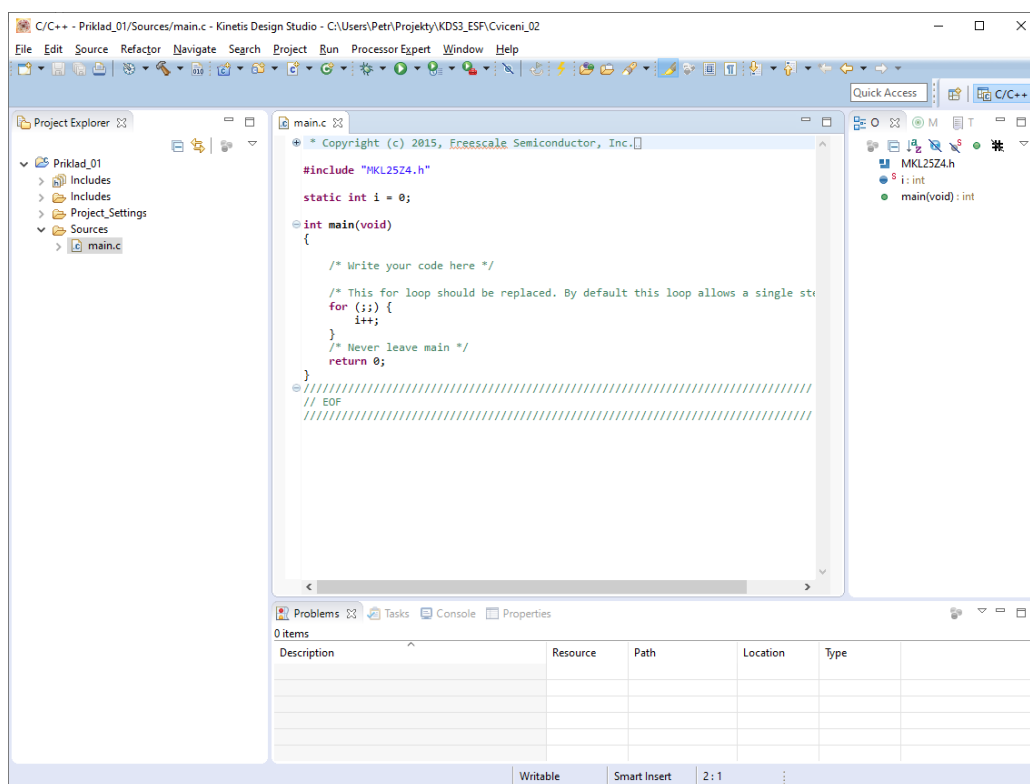
Po spuštění vývojového prostředí KDS3 se nám zobrazí okno s výzvou pro výběr složky pracovního prostoru (workspace) viz obrázek 18, kde budou ukládány jednotlivé projekty. Pomocí tlačítka „Browse“ vyvoláme dialog pro výběr složky na disku. Po výběru složky klikneme na tlačítko „OK“. Na některých počítačích se může stát, že po spuštění KDS se tento dialog nezobrazí, protože byla již dříve zatržena volba „Použít jako defaultní umístění a dále se již neptat“. V tomto případě se již rovnou zobrazí hlavní okno aplikace s automaticky otevřeným výchozím pracovním prostorem. Pokud chcete pracovat v jiném než defaultním, musí se změnit v nabídce „File / Switch Workspace“



Obrázek 18: KDS - výběr složky pracovního prostoru.

Hlavní okno aplikace vyobrazené na obrázku 19 je rozděleno do 3 základních částí. Úplně vlevo se nachází průzkumník projektů, ve kterém vidíme veškeré projekty uložené

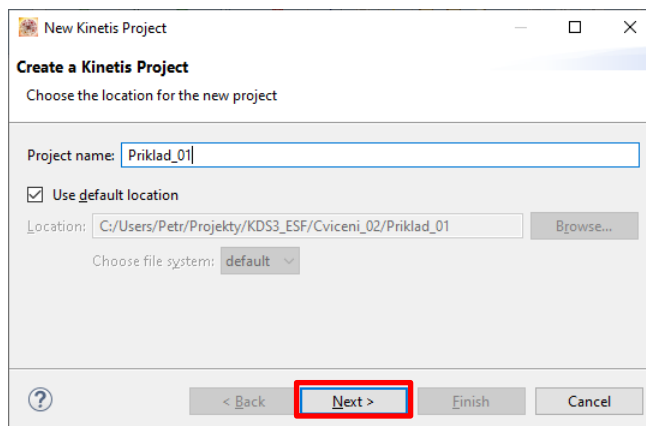
v aktuálně vybraném pracovním prostoru. Vpravo vedle něj se nachází okno editoru zobrazující aktuálně otevřený soubor. V případě více otevřených souborů je možno se mezi nimi rychle přepínat pomocí záložek v záhlaví okna. Spodní část okna aplikace je vyhrazena pro zobrazení výsledku překlada (záložka „Problems“), zobrazení konzole (záložka „Console“) a dalších informačních podoken. V nástrojové liště nalezneme ikony pro uložení programu, tisk, překlad programu, vytvoření nového projektu, přidání souboru do projektu, spuštění ladění programu, atd.



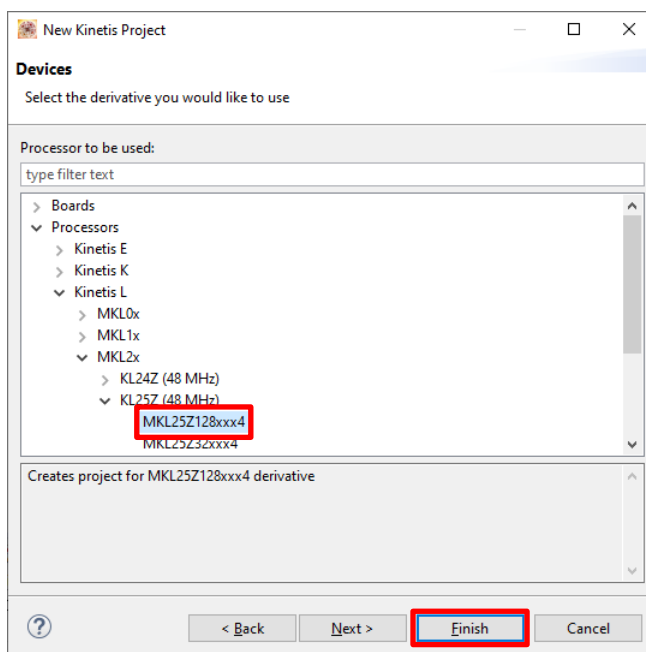
Obrázek 19: KDS - hlavní okno aplikace.

Nyní již můžeme v našem, prozatím prázdném, pracovním prostoru vytvořit nový projekt, který si nazveme „Příklad\_01“ (v názvech souborů nepoužívejte diakritiku). Klikneme v menu na „File“ a z nabídky vybereme příkaz „New / Kinetis SDK 1.x Project“. Průvodce novým projektem se nás nejprve dotáže na jméno nového projektu, viz obrázek 20. Zadáme tedy jméno projektu „Příklad\_01“ a klikneme na tlačítko „Next“. V dalším kroku vybereme cílový mikropočítač, pro který budeme vytvářet program. V seznamu rozklikneme položku „Processors“, dále ve stromu postupně vybíráme „Kinetis L / MKL2x / KL25Z (48 MHz)“ a finálně konkrétní typ použitý ve vývojovém kitu „MKL25Z128xxx4“, viz obrázek 21.

V tento okamžik již můžeme průvodce novým projektem dokončit kliknutím na tlačítko „Finish“. V okně průzkumníka nyní nalezneme nově vytvořený a již otevřený projekt „Příklad\_01“.



Obrázek 20: KDS - průvodce novým projektem krok 1.



Obrázek 21: KDS - průvodce novým projektem krok 2.

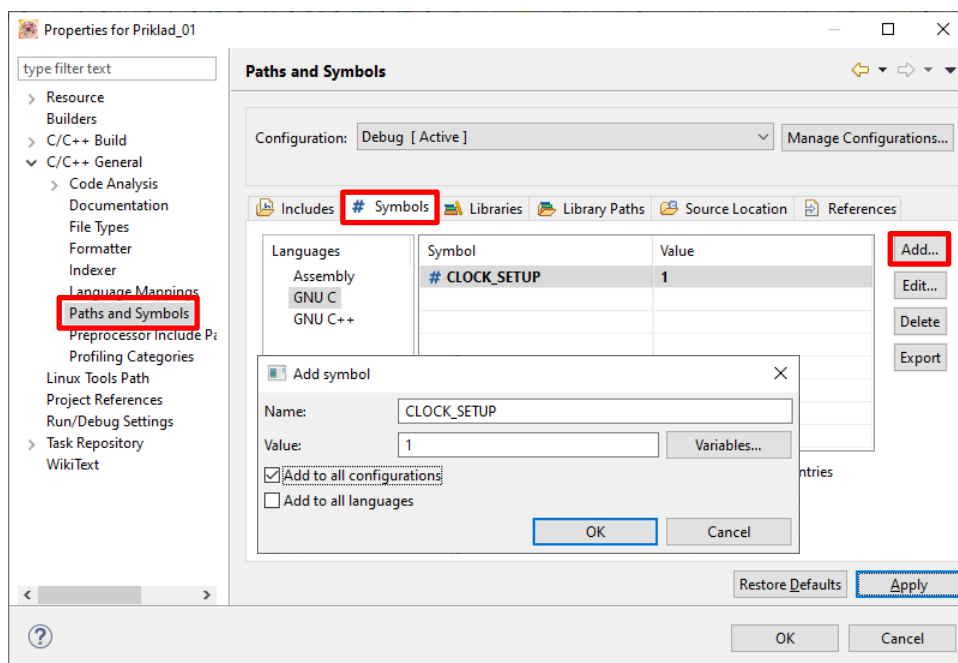
Kliknutím pravým tlačítkem myši na projekt „Příklad\_01“ se zobrazí nabídka, ve které si můžeme nastavit jeho vlastnosti výběrem položky „Properties“. Zde v zobrazené nabídce rozklikneme „C/C++ General“ a v nastavení nazvaném „Paths and Symbols“ v záložce „#




Symbols“ pro jazyk GNU C vytvoříme nový symbol `CLOCK_SETUP = 1` pro všechny konfigurace, viz obrázek 22. Hodnota tohoto symbolu může být v rozsahu 0 až 4 a slouží pro inicializaci generátoru hodinového signálu mikropočítače. Jednotlivé možnosti nastavení jsou shrnuty v tabulce 5. Námi zvolená hodnota 1 znamená použití externího 8MHz krystalu jako zdroje hodinového kmitočtu a aktivaci interního PLL pro zvýšení této frekvence na 48 MHz pro ARM jádro a 24 MHz pro sběrnici. Jedná se o maximální dosažitelné frekvence, při kterých má mikropočítač nejvyšší výpočetní výkon, tedy i rychlost zpracování programu. Nastavení potvrdíme kliknutím na tlačítko „Apply“ a celé okno vlastností potvrdíme kliknutím na „OK“.

Hodnota symbolu CLOCK_SETUP	Nastavení hodinového generátoru
0	Interně taktovaný, defaultní konfigurace. Zdroj hodinového signálu: nízkofrekvenční interní Režim MCG: FEI $f_{\text{CORE}} = 20,97152 \text{ MHz}$ $f_{\text{BUS}} = 20,97152 \text{ MHz}$
1	Externě taktovaný, maximální dosažitelná frekvence. Zdroj hodinového signálu: systémový oscilátor Režim MCG: PEE $f_{\text{CORE}} = 48 \text{ MHz}$ $f_{\text{BUS}} = 24 \text{ MHz}$
2	Interně taktovaný, velmi nízká spotřeba. Zdroj hodinového signálu: vysokofrekvenční interní Režim MCG: BLPI $f_{\text{CORE}} = 4 \text{ MHz}$ $f_{\text{BUS}} = 0,8 \text{ MHz}$
3	Externě taktovaný, velmi nízká spotřeba. Zdroj hodinového signálu: systémový oscilátor Režim MCG: BLPE $f_{\text{CORE}} = 4 \text{ MHz}$ $f_{\text{BUS}} = 1 \text{ MHz}$
4	Externě taktovaný, USB aplikace. Zdroj hodinového signálu: systémový oscilátor Režim MCG: PEE $f_{\text{CORE}} = 48 \text{ MHz}$ $f_{\text{BUS}} = 24 \text{ MHz}$


Tabulka 5: Přehled nastavení symbolu `CLOCK_SETUP`.




Obrázek 22: KDS - definice symbolu CLOCK\_SETUP.










Tímto je základní nastavení projektu dokončeno a můžeme nyní v průzkumníku projektů rozkliknout myší vytvořený projekt a ve složce „Sources“ otevřít dvojklikem soubor se zdrojovým textem „main.c“. Průvodce novým projektem vytvořil základní kostru programu obsahující hlavní programovou jednotku *main()*, do které vložíme programový kód našeho ukázkového programu a uložíme jej. Následně program přeložíme kliknutím na ikonu „Build“  v nástrojové liště. V případě neúspěšného překladu se ve spodní části okna v záložce „Problems“ zobrazí nalezené chyby, které je zapotřebí opravit.

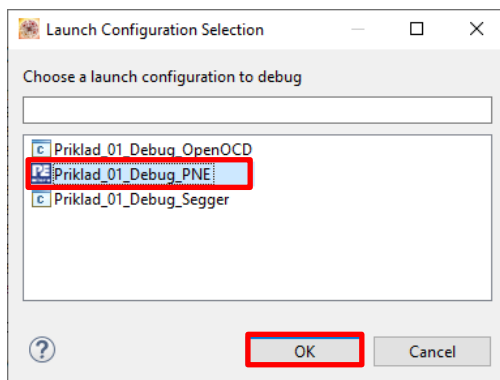
### Zavedení programu do mikropočítače

Po úspěšném překladu zavedeme do mikropočítače program a odzkoušíme jeho funkci. Ladění programu se spustí kliknutím na ikonu  v nástrojové liště. Následně se otevře okno (obrázek 23), ve kterém se musí zvolit konfigurace pro spuštění ladění. Námi používaný mikropočítač je vybaven rozhraním OpenSDA od PEmicro. Ze seznamu tedy vybereme položku nazvanou „Příklad\_01\_Debug\_PNE“ a potvrdíme tlačítkem „OK“. Poté se zobrazí okno informující o průběhu inicializace rozhraní a zavádění programu. Po dokončení této operace se přepne hlavní okno aplikace do „Debug“ pohledu, viz obrázek 24. Okno je v tomto pohledu rozčleněno do 5 základních částí. V levé horní části je okno informující o aktuálně otevřené ladící konfiguraci. Vidíme zde běžící vlákno „Thread #1“ a jeho aktuální

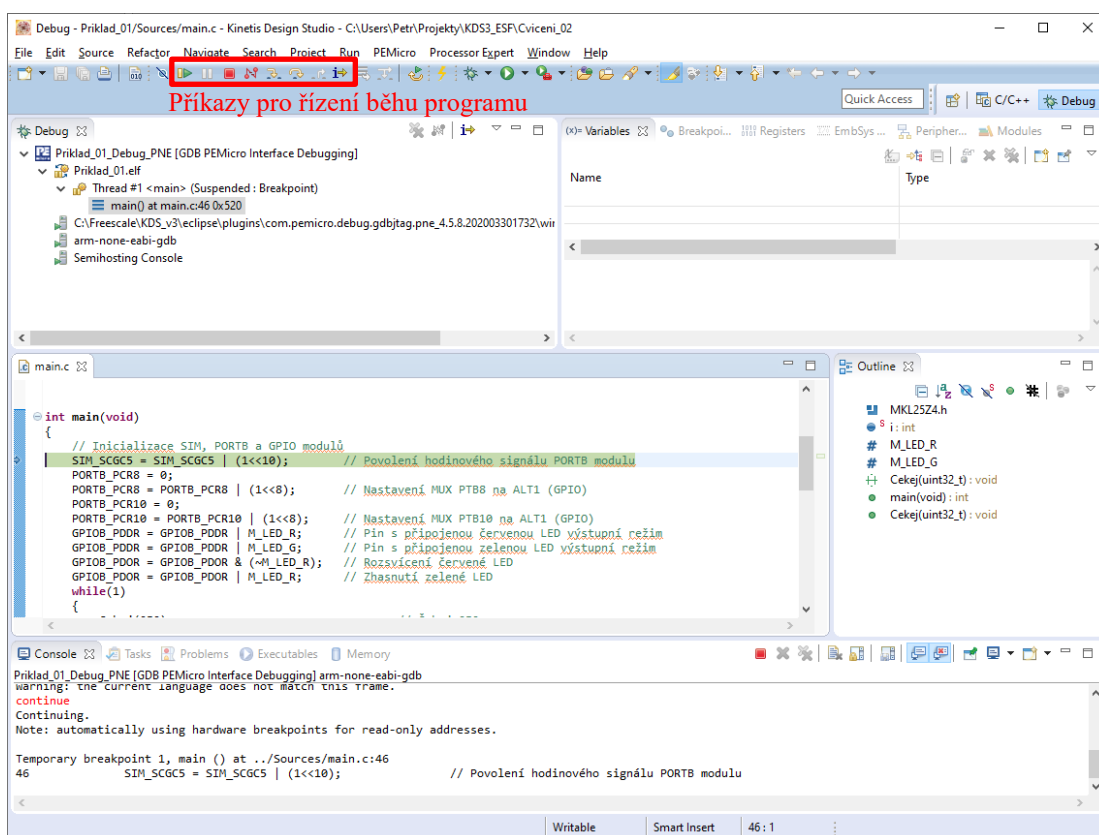
stav (běžící, pozastaveno, ukončeno, ...). Kliknutím pravým tlačítkem myši na vlákno se nám zobrazí kompletní sada příkazů pro operace s vybraným vláknem, které prakticky odpovídají příkazům v nástrojové liště. Vpravo od něj se nachází okno s více záložkami, pod kterými nalezneme možnosti zobrazení obsahu proměnných „Variables“, seznam zarážek umístěných v laděném programu „Breakpoints“ a záložku „Registers“, kde můžeme prohlížet obsah registrů CPU mikropočítače. Hodnoty uložené v proměnných a registrech lze v průběhu ladění dle potřeby modifikovat. Uprostřed hlavního okna se nachází okno editoru, ve kterém vidíme zdrojový kód aktuálně prováděného programu. Programový řádek, který bude aktuálně vykonáván, je zeleně podbarven. Na levém okraji je vyobrazen světle modrý pruh se šipkou indikující pozici v programu. Při dvojitým kliknutí na tento pruh vložíme na danou pozici v programu zarážku (symbol ). Při běhu programu se na tomto místě vykonávání zastaví a můžeme dále krokovat, případně opět program spustit plnou rychlostí. Zarážka se odstraní po opětovném dvojitým kliknutí na její místo. V okně „Outline“ vidíme přehled proměnných, definovaných maker a podprogramů. Kliknutím na příslušné položky se v okně editoru ihned vyhledají a zobrazí. Ve spodním okně se nachází opět více záložek – nalezneme zde například konzolu se stavovými informacemi a nástroj pro zobrazení obsahu paměti pod záložkou „Memory“.

V nástrojové liště se nachází sada ikon pro ovládání běhu programu v průběhu ladění, jejichž význam je následující:

-  Spustí běh programu po jeho zastavení na zarážce.
-  Pozastaví běh programu.
-  Step into. Krokuje po jednom řádku. V případě, že je na daném řádku volání podprogramu, provádí krokování jeho těla.
-  Step over. Podobně jako předchozí funkce krokuje po řádku. Při volání podprogramu provede jeho tělo a po návratu se zastaví.
-  Step return. Provede dokončení těla podprogramu, po návratu se program zastaví.
-  Zapnutí / vypnutí režimu krokování po jednotlivých instrukcích.
-  Ukončení ladění programu. Program je zastaven a debugger ukončen.
-  Odpojení od debuggeru. Program v mikropočítači běží dál.
-  Restart mikropočítače bez nutnosti nového zavádění programu. Program lze spustit nebo krokovat znovu od jeho začátku.



Obrázek 23: KDS - okno výběru konfigurace ladění.



Obrázek 24: KDS - hlavní okno v pohledu ladění programu.

Při ověřování funkce programu si vyzkoušejte rozdíl mezi krokováním pomocí funkce „Step into“ a „Single step“. Rozdíl v jejich činnosti bude zřejmý v okamžiku krokování volání funkce „Cekej“. Jakmile se budete nacházet uvnitř těla funkce, otestujte příkaz „Step return“. Umístěte do programu záložku (breakpoint), spusťte program a počkejte na jeho zastavení. Dále můžete opět krokovat program nebo jej znovu spustit plnou rychlostí.

### 2.4.2 Obsluha tlačítek na výukovém kitu

#### Zadání

V jazyce C vytvořte program, který bude ovládat červenou LED pomocí tlačítek následovně: při stisku tlačítka SW1 rozsvítí danou LED a stiskem tlačítka SW2 ji zhasne.

#### Řešení

V prezentaci k laboratorním cvičením nejprve musíme zjistit způsob připojení jednotlivých tlačítek k pinům mikropočítače. Z prezentace (viz snímek 10) vyplývá, že všechna 4 tlačítka jsou připojena k portu A následovně:

- Tlačítko SW1 – pin PTA4
- Tlačítko SW2 – pin PTA5
- Tlačítko SW3 – pin PTA16
- Tlačítko SW4 – pin PTA17

Při stisku tlačítka se příslušný pin portu propojí se signálem GND, což odpovídá logické 0, po rozpojení tlačítka zajišťuje definovanou logickou úroveň 1 pull-up rezistor připojený na napájecí větev Vcc (+3 V). Kondenzátor zapojený paralelně s tlačítkem slouží k eliminaci záskmitů na kontaktech. Z principu funkce je zřejmé, že odpovídající piny musí být přepnuty do vstupního režimu, interní pull-up rezistory není zapotřebí aktivovat, protože jsou již připojeny externě.

V tento okamžik již známe z pohledu hardware vše potřebné a můžeme začít s tvorbou programu. Obdobně jako v předchozím příkladu, musíme nejprve provést inicializaci SIM a PORT modulu. Tentokrát je zapotřebí povolit distribuci hodinového signálu pro PORTA i PORTB modul zápisem 1 do 9. a 10. bitu registru SIM\_SCGC5. V příslušných řídicích registrech portů A a B nastavíme alternativní funkci pinu ALT1 v bitovém poli MUX odpovídající GPIO funkcionalitě, do ostatních řídicích bitů zapíšeme 0. Konkrétně se tato inicializace bude týkat registrů PORTA\_PCR4 (tlačítko SW1), PORTA\_PCR5 (tlačítko SW2) a PORTB\_PCR8 (červená LED). Do zmíněných registrů nejprve zapíšeme nulu a následně nastavíme 8. bit na 1, čímž je jejich inicializace hotova. Jako poslední nastavíme v modulu GPIOA vstupní režim pro piny 4 a 5 zápisem 0 do 4. a 5. bitu registru GPIOA\_PDDR a



v modulu GPIOB výstupní režim pinu PTB8 zápisem 1 do 8. bitu GPIOB\_PDDR. Pro jednoznačně definovaný počáteční zhasnutý stav červené LED po startu programu, je zapotřebí zapsat 1 do 8. bitu výstupního datového registru GPIOB\_PDOR. Tímto je inicializační část programu dokončena. Dále budeme v nekonečné smyčce testovat stav tlačítek načtením vstupního registru GPIOA\_PDIR a po logickém součinu s odpovídající maskou testujeme výsledek této operace na 0. Při stisknutí tlačítka SW1 se provede rozsvícení červené LED zápisem 0 do 8. bitu GPIOB\_PDOR, při stisknutí SW2 do stejného bitu uložíme 1 a tím LED zhasneme. Zdrojový kód programu naleznete níže ve výpisu programu 2.2.

*Program 2.2:*

```
// Definice masek pro červenou LED a tlačítka SW1, SW2
#define M_LED_R (1<<8)
#define M_SW1 (1<<4)
#define M_SW2 (1<<5)

int main(void)
{
    SIM->SCGC5 |= (1<<9); // Povolení hodinového signálu PORTA modulu
    SIM->SCGC5 |= (1<<10); // Povolení hodinového signálu PORTB modulu
    PORTB->PCR[8] = 0;
    PORTB->PCR[8] |= (1<<8); // Nastavení MUX PTB8 na ALT1 (GPIO), červená LED
    PORTA->PCR[4] = 0;
    PORTA->PCR[4] |= (1<<8); // Nastavení MUX PTA4 na ALT1 (GPIO), tlačítko SW1
    PORTA->PCR[5] = 0;
    PORTA->PCR[5] |= (1<<8); // Nastavení MUX PTA5 na ALT1 (GPIO), tlačítko SW2
    GPIOA->PDDR &= ~M_SW1; // Pin s připojeným tlačítkem SW1 vstupní režim
    GPIOA->PDDR &= ~M_SW2; // Pin s připojeným tlačítkem SW2 vstupní režim
    GPIOB->PDDR |= M_LED_R; // Pin s připojenou červenou LED výstupní režim
    GPIOB->PDOR |= M_LED_R; // Zhasnutí červené LED

    while(1)
    {
        if ((GPIOA->PDIR & M_SW1) == 0) // Otestuj stav tlačítka SW1
            GPIOB->PDOR &= ~M_LED_R; // Rozsvícení červené LED
        if ((GPIOA->PDIR & M_SW2) == 0) // Otestuj stav tlačítka SW2
            GPIOB->PDOR |= M_LED_R; // Zhasnutí červené LED
    }
}
```

## 2.5 Zadání samostatné práce

1. Upravte ukázkový program 2.1 pro obsluhu LED tak, aby se jednotlivě rozsvěcovaly LED v sekvenci červená, žlutá, zelená a poté zpět přes žlutou k červené LED. Uvedená sekvence se opakuje v nekonečné smyčce. Mezi přepínáním stavu zvolte vhodnou časovou prodlevu s využitím funkce *Cekej()*.
2. Upravte ukázkový program 2.2 tak, aby tlačítko SW1 přepínalo stav červené LED, tlačítko SW2 žluté LED a tlačítko SW3 zelené LED. Tlačítko SW4 bude zajišťovat funkci centrálního vypnutí všech 3 LED.
3. Vytvořte si sadu obslužných podprogramů pro ovládání LED včetně RGB (viz snímek 29 prezentace) a čtení stavu tlačítek. Implementujte následující funkce:
  - *void GPIO\_LED\_Init(void)* – inicializace SIM, PORT, GPIO modulů souvisejících s funkcí LED.
  - *void GPIO\_LED\_PutVal(uint8\_t LED, uint8\_t value)* – zápis hodnoty *value* na LED specifikovanou parametrem *LED*.
  - *void GPIO\_LED\_Set(uint8\_t LED)* – zápis 1 na specifikovanou LED.
  - *void GPIO\_LED\_Clr(uint8\_t LED)* – zápis 0 na specifikovanou LED.
  - *void GPIO\_LED\_Neg(uint8\_t LED)* – negace stavu specifikované LED.
  - *void GPIO\_SW\_Init(void)* – inicializace SIM, PORT, GPIO modulů souvisejících s funkcí tlačítek.
  - *uint8\_t GPIO\_SW\_GetVal(uint8\_t SW)* – funkce vrátí stav tlačítka specifikovaného parametrem *SW*.
4. Vytvořte programové vybavení pro řízení semaforu pro automobily a chodce. Pro ovládání semaforu bude sloužit tlačítko SW1. V zapnutém stavu bude semafor vykonávat standardní sekvenci. Časy přepínání nastavte dle reálného semaforu ve vhodném časovém měřítku. Ve vypnutém stavu bude blikat žluté světlo s periodou 2 s. Po návratu do stavu zapnuto, musí sekvence začít opět od počátečního stavu. Semafor pro chodce bude představovat RGB LED na KL25Z desce. Nezapomeňte na bezpečnostní interval mezi červenou a zelenou pro automobily a chodce a naopak. Pro realizaci programu použijte funkce vytvořené v bodu 3.



### 3 A/D PŘEVODNÍK FRDM-KL25Z

Mikropočítač MKL25Z128 má integrován jeden modul A/D převodníku (ADC0) s nastavitelným rozlišením převodu 8, 10, 12 nebo 16 bitů v režimu se společnou zemí (single-ended) nebo 9, 11, 13 a 16 bitů v diferenciálním režimu. Interně převodník pracuje metodou lineární postupné aproximace. Na výstupu umožňuje hardwarové průměrování až 32 převedených vzorků. Před vlastním převodníkem je zapojen analogový multiplexor zajišťující připojení celkem 14 vstupních analogových kanálů se společnou zemí (označení ADC0\_SEn) z nichž 2 mohou pracovat jako diferenciální kanály (označení ADC0\_DPn a ADC0\_DMn) obsazující vždy dva vstupy DP a DM. Interně jsou k dispozici další přídavné kanály, na kterých jsou zapojeny signály z teplotního senzoru, bandgap napěťové reference a referenčních napětí převodníku VREFH a VREFL. Pro přenos dat bez účasti CPU má implementovanou funkci přímého přístupu do paměti (DMA) [2].

Na výukovém kitu je jeden z analogových vstupů používán pro snímání napětí na výstupu potenciometru s elektrickým odporem 10 k $\Omega$  a lineárním průběhem odporu. Jeho odporová dráha je připojena na napájecí signály Vcc (+3 V) a GND. Na běžci potenciometru lze tedy otáčením hřídele potenciometru plynule měnit napětí v rozsahu od 0 do 3 V.

Připojení periférií kitu na digitální / analogové vstupy mikropočítače je následující:

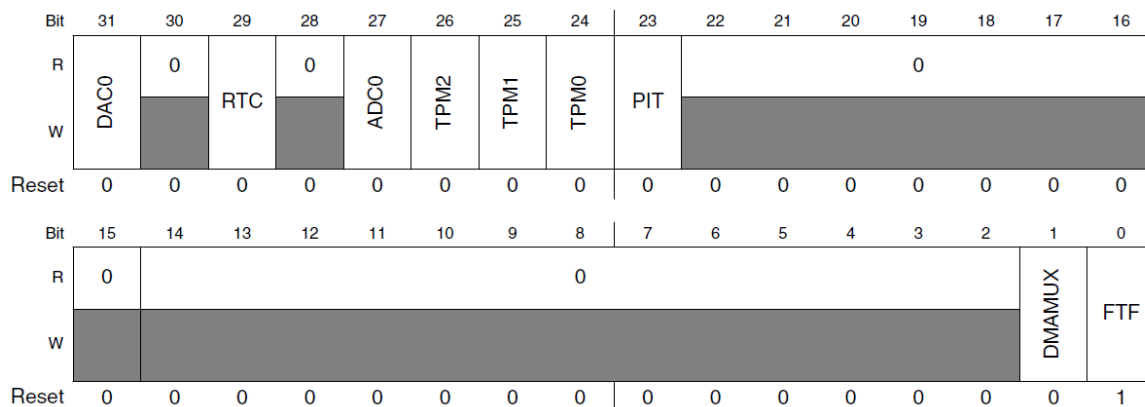
- Analog port (1), potenciometr – pin PTC2/ADC0\_SE11
- Analog port (2), rotační enkodér B – pin PTB2/ADC0\_SE12
- Analog port (3), rotační enkodér A – pin PTB3/ADC0\_SE13
- Analog port (4), EEPROM\_CS – pin PTC0/ADC0\_SE14
- Analog port (5), MCU port (12) – pin PTC1/ADC0\_SE15
- Analog port (6), MCU port (21) – pin PTB1/ADC0\_SE9

#### 3.1 Inicializace souvisejících modulů

Při inicializaci se nesmí opomenout na povolení hodinového signálu pro A/D převodník (ADC0) v SIM modulu zápisem 1 do 27. bitu registru SIM\_SCGC6, viz struktura registru na obrázku 25. Analogové vstupy jsou součástí PORT modulů, a proto je nutné jim dle aktuálně používaných vstupů povolit hodinový signál v SIM\_SCGC5. Na výukovém kitu se



to týká pouze PORTB a PORTC modulů, protože ostatní piny s funkcí analogového vstupu nejsou na kitu použity. V odpovídajícím PORT modulu je dále zapotřebí v registru pro řízení pinů  $PORTx\_PCRN$  v bitovém poli *MUX* vypnout digitální funkci pinu, čímž se povolí funkce analogového vstupu. Toho se docílí zápisem nuly do odpovídajícího registru.



Obrázek 25: Struktura registru SIM\_SCGC6 [2].

### 3.2 Vybrané registry A/D převodníku

Funkce modulu A/D převodníku (ADC0) je řízena celkem 27 registry, z nichž 17 registrů je vyhrazeno pro jeho kalibraci. Níže je uveden popis vybraných registrů pro zajištění základní funkcionality modulu v režimu softwarově spouštěných jednorázových či kontinuálních převodů.

#### ADC0\_SC1A – ADC stavový a řídicí registr 1

Význam bitů (v hranaté závorce uvedeno číslo bitu v registru) [2]:

*COCO* [7] – Příznak dokončení převodu: 0 = probíhá převod; 1 = převod dokončen

*AIEN* [6] – Povolení přerušení po dokončení převodu: 0 = přerušení zakázáno

*DIFF* [5] – Diferenciální režim: 0 = režim se společnou zemí; 1 = diferenciální režim

*ADCH* [4-0] – Výběr vstupního kanálu: při  $ADCH = 11111$  je ADC0 modul vypnut

Pozn.: Při softwarovém spouštění převodu se zápisem do tohoto registru spustí převod na vybraném vstupním analogovém kanálu.

Doporučené nastavení při inicializaci na kitu:  $ADC0\_SC1A = 0x1f$

**ADC0\_CFG1** – ADC konfigurační registr 1

Význam bitů (v hranaté závorce uvedeno číslo bitu v registru) [2]:

*ADLPC* [7] – Konfigurace pro nízkou spotřebu: 0 = normální; 1 = nízká spotřeba

*ADIV* [6-5] – Výběr dělicího poměru hodinového signálu ADC:

00 – Dělicí poměr = 1;  $f_{\text{ADCK}} = \text{input\_clock} / 1$

01 – Dělicí poměr = 2;  $f_{\text{ADCK}} = \text{input\_clock} / 2$

10 – Dělicí poměr = 4;  $f_{\text{ADCK}} = \text{input\_clock} / 4$

11 – Dělicí poměr = 8;  $f_{\text{ADCK}} = \text{input\_clock} / 8$

*ADLSMP* [4] – Vzorkovací čas: 0 = krátký; 1 = dlouhý

*MODE* [3-2] – Nastavení rozlišení pro: single-ended / diferenciální režim:

00 – 8bitové / 9bitové (výsledek se znaménkem)

01 – 12bitové / 13bitové (výsledek se znaménkem)

10 – 10bitové / 11bitové (výsledek se znaménkem)

11 – 16bitové / 16bitové (výsledek se znaménkem)

*ADICLK* [1-0] – Výběr zdroje hodinového kmitočtu *input\_clock*:

00 – Frekvence sběrnice *Bus\_clock*

01 – Frekvence sběrnice *Bus\_clock* / 2

10 – Alternativní zdroj hodinové frekvence *ALTCLK*

11 – Asynchronní zdroj hodinové frekvence *ADACK*

Doporučené nastavení při inicializaci na kitu:

- *ADC0\_CFG1* = 0x31 (8bitový režim převodníku)
- *ADC0\_CFG1* = 0x39 (10bitový režim převodníku)
- *ADC0\_CFG1* = 0x35 (12bitový režim převodníku)
- *ADC0\_CFG1* = 0x3d (16bitový režim převodníku)

Pozn.: Dělicí poměr hodinového kmitočtu musí být nastaven tak, aby frekvence A/D převodníku byla v intervalu od 1 do 18 MHz při rozlišení  $\leq 13$  bitů a od 2 do 12 MHz při rozlišení převodníku 16 bitů [3].

**ADC0\_CFG2** – ADC konfigurační registr 2

Význam bitů (v hranaté závorce uvedeno číslo bitu v registru) [2]:

*MUXSEL* [4] – Výběr ADC multiplexoru: 0 = ADxxa; 1 = ADxxb kanály

*ADACKEN* [3] – Povolení výstupu asynchronního CLK: 0 = zakázáno

*ADHSC* [2] – Vysokorychlostní konfigurace ADC: 0 = normální; 1 = vysokorychlostní

*ADLSTS* [1-0] – Výběr délky vzorkovacího času při vybraném dlouhém vzorkování:

00 – Výchozí nejdelší vzorkovací čas – 24 cyklů ADCK

01 – Celkový vzorkovací čas 16 cyklů ADCK

10 – Celkový vzorkovací čas 10 cyklů ADCK

11 – Celkový vzorkovací čas 6 cyklů ADCK

Doporučené nastavení při inicializaci na kitu: ADC0\_CFG2 = 0

**ADC0\_SC2** – ADC stavový a řídicí registr 2

Význam bitů (v hranaté závorce uvedeno číslo bitu v registru) [2]:

*ADACT* [7] – Příznak probíhajícího převodu: 0 = neprobíhá; 1 = probíhá převod

*ADTRG* [6] – Výběr spouštění převodu: 0 = softwarové; 1 = hardwarové

*ACFE* [5] – Povolení funkce porovnání: 0 = zakázána; 1 = povolena

*ACFGT* [4] – Výběr porovnávací funkce CV1 a CV2:

0 – Vybrána funkce: menší než prahová hodnota

1 – Vybrána funkce: větší než prahová hodnota

*ACREN* [3] – Funkce porovnávání rozsahu: 0 = zakázána; 1 = povolena

*DMAEN* [2] – Povolení DMA funkce: 0 = zakázána; 1 = povolena

*REFSEL* [1-0] – Výběr zdroje referenčního napětí:

00 – Výchozí zdroj referenčního napětí (signály  $V_{REFH}$  a  $V_{REFL}$ )

01 – Alternativní zdroj referenčního napětí (signály  $V_{ALTH}$  a  $V_{ALTL}$ )

10, 11 – rezervováno

Doporučené nastavení při inicializaci na kitu: ADC0\_SC2 = 0

Pozn.: Na vývojovém kitu je signál  $V_{REFH}$  připojen na Vcc (+3 V) a  $V_{REFL}$  na GND (0 V).



**ADC0\_SC3** – ADC stavový a řídicí registr 3

Význam bitů (v hranaté závorce uvedeno číslo bitu v registru) [2]:

*CAL* [7] – Spuštění kalibrace převodníku: 1 = spuštění kalibrace

*CALF* [6] – Příznak výsledku kalibrace: 0 = úspěšná; 1 = neúspěšná kalibrace

*ADCO* [3] – Povolení kontinuálního převádění: 0 = zakázáno; 1 = povoleno

*AVGE* [2] – Povolení hardwarového průměrování: 0 = zakázáno; 1 = povoleno

*AVGS* [1-0] – Nastavení hardwarového průměrování:

00 – Průměrování 4 vzorků

01 – Průměrování 8 vzorků

10 – Průměrování 16 vzorků

11 – Průměrování 32 vzorků

Doporučené nastavení při inicializaci na kitu: `ADC0_SC3 = 0x04`

**ADC0\_RA** – ADC výsledkový registr

Do výsledkového registru je po dokončení převodu uložen výsledek ve tvaru odpovídajícím nastavení převodníku. V případě diferenciálního režimu je uložen ve formátu se znaménkem (dvojkový doplněk). Nepoužité výsledkové bity jsou vynulovány a hodnota je zarovnána doprava, tj. například 8bitový výsledek bude uložen v bitech 0 až 7 [2].

Podrobný popis dalších registrů pro pokročilé funkce a nastavení ADC0 modulu souvisejících s jeho kalibrací, funkcemi porovnávání a hardwarového spouštění převodů s využitím DMA přenosů naleznete v [2].

### 3.3 Ukázkové programy

V této kapitole naleznete ukázkové řešené příklady související s programovou obsluhou A/D převodníku v C jazyce. V rámci laboratorního cvičení si funkci programů důkladně vyzkoušejte jejich odkrokováním v příslušném programovém prostředí.



### 3.3.1 Obsluha A/D převodníku - jednorázové převody

#### Zadání

Vytvořte program, který bude periodicky snímat napětí na virtuálním NiMH akumulátoru a dle stavu jeho nabití rozsvítí odpovídající indikační LED. V případě dostatečně nabitého akumulátoru (napětí  $\geq 1,15$  V) bude svítit zelená LED, při nižším napětí se rozsvítí červená LED. ADC nakonfigurujte do režimu s 8bitovým rozlišením a jednorázového programově spouštěného převádění. Napětí akumulátoru bude nastavováno ručně pomocí potenciometru, jehož výstup je připojen na vstupní kanál PTC2/ADC0\_SE11.

#### Řešení

V úvodní části programu se nejprve vykoná inicializace SIM modulu tak, aby byla zajištěna distribuce hodinového signálu do všech použitých modulů, tj. PORTB (červená a zelená LED), PORTC (analogový vstup ADC0\_SE11) a ADC0. Inicializace PORTB a GPIOB je stejná jako v předcházejících příkladech. Na portu C musí být vypnuta na pinu PTC2 digitální funkce, čímž lze daný pin používat jako analogový vstup. Pro základní inicializaci A/D převodníku na vývojovém kitu použijeme hodnoty uvedené v kapitole 3.2, které nastaví převodník následovně: přerušení od dokončení převodu zakázáno, vstupní režim se společnou zemí (single-ended), ADC0 modul vypnut (zapne se při spuštění převodu), režim úspory energie vypnut, frekvence ADCK nastavena na 6 MHz, dlouhý vzorkovací interval, 8bitové rozlišení převodu (lze nastavit 10, 12, 16bitové, viz doporučené hodnoty), analogový multiplexor nastaven na kanály ADxxa, normální sekvence převodu, nejdelší vzorkovací čas, programové spouštění převodu, funkce porovnávání a DMA vypnuty, nastaven výchozí zdroj referenčního napětí, jednorázové spouštění převodů a zapnuto hardwarové průměrování 4 převedených vzorků. Tímto je modul ADC0 připraven k činnosti. V hlavní programové smyčce se na začátku provede start A/D převodu na kanálu ADC0\_SE11 zápisem hodnoty 11 do registru ADC0\_SC1A. Protože proces převodu trvá určitý čas, je zapotřebí počkat na jeho dokončení. To se provede cyklickým dotazováním ve smyčce *while*, která se ukončí, jakmile se nastaví 7. bit registru ADC0\_SC1A na 1. Následně se přečte z výsledkového registru ADC0\_RA převedená hodnota a uloží se do proměnné „AD\_val“.

Výslednou převedenou hodnotu pro režim se společnou zemí (single-ended) lze vypočítat pomocí rovnice (1):



$$AD_{OUT} = \frac{V_{IN}}{V_{REFH}} (2^n - 1), \quad (1)$$

kde  $V_{IN}$  je napětí na vstupu převodníku,  $V_{REFH}$  je referenční hodnota napětí vysoké úrovně (3 V) a  $n$  je rozlišení A/D převodníku v bitech. Pro napětí 1,15 V představující rozhodovací úroveň pro přepínání stavu LED, je výstupní hodnota převodníku rovna 94. Pokud je aktuálně převedená hodnota v proměnné „AD\_val“ menší jak 94, rozsvítí se červená LED a zelená zhasne, jinak se rozsvítí zelená LED a červená zhasne. Vyhodnocování probíhá v nekonečné smyčce plnou rychlostí mikropočítače. Zdrojový kód programu je uveden ve výpisu programu 3.1.

*Program 3.1:*

```
// Definice masek pro červenou a zelenou LED
#define M_LED_R (1<<8)
#define M_LED_G (1<<10)

int main(void)
{
    uint16_t AD_val; // proměnná pro uložení výsledku převodu

    // Inicializace SIM modulu
    SIM->SCGC6 |= (1<<27); // Povolení hodinového signálu ADC0 modulu
    SIM->SCGC5 |= (1<<10); // Povolení hodinového signálu PORTB modulu
    SIM->SCGC5 |= (1<<11); // Povolení hodinového signálu PORTC modulu
    // Inicializace PORT modulu
    PORTB->PCR[8] = 0;
    PORTB->PCR[8] |= (1<<8); // Nastavení MUX PTB8 na ALT1 (GPIO), červená LED
    PORTB->PCR[10] |= (1<<8); // Nastavení MUX PTB10 na ALT1 (GPIO), zelená LED
    PORTC->PCR[2] = 0; // Digitální funkce na pinu PTC2 vypnuty
    // Inicializace GPIO
    GPIOB->PDDR |= M_LED_R; // Pin s připojenou červenou LED výstupní režim
    GPIOB->PDDR |= M_LED_G; // Pin s připojenou zelenou LED výstupní režim
    // Inicializace A/D převodníku
    ADC0->SC1[0] = 0x1f;
    ADC0->CFG1 = 0x31;
    ADC0->CFG2 = 0;
    ADC0->SC2 = 0;
    ADC0->SC3 = 0x04;
```



Program 3.1 (pokračování):

```
while(1)
{
    ADC0->SC1[0] = 11;                // Spust' převod na kanálu 11 (potenciometr)
    while((ADC0->SC1[0] & (1<<7))==0); // Čekaj na dokončení převodu
    AD_val = ADC0->R[0];                // Ulož výsledek převodu
    if (AD_val < 94)
    {
        GPIOB->PDOR &= ~M_LED_R;      // Rozsvícení červené LED
        GPIOB->PDOR |= M_LED_G;       // Zhasnutí zelené LED
    }
    else
    {
        GPIOB->PDOR |= M_LED_R;        // Zhasnutí červené LED
        GPIOB->PDOR &= ~M_LED_G;      // Rozsvícení zelené LED
    }
}
```

### 3.3.2 Obsluha A/D převodníku - kontinuální převody

#### Zadání

Upravte předchozí ukázkový program 3.1 tak, aby používal kontinuální režim A/D převodníku. Funkce programu musí zůstat po úpravě identická.

#### Řešení

V inicializační části programu musí být provedena změna v nastavení stavového a konfiguračního registru A/D převodníku ADC0\_SC3. Konkrétně se jedná o 3. bit *ADCO* určující, zda bude použit režim převodů jednorázový (*ADCO* = 0) nebo kontinuální (*ADCO* = 1). Kontinuální převody zajistí zápis hodnoty 0x0c do výše zmíněného registru. Převody se aktivují zápisem čísla požadovaného kanálu 11 do registru ADC0\_SC1A. Od tohoto okamžiku ADC0 modul začne převádět na daném kanálu v kontinuálním režimu, během kterého aktualizuje plnou rychlostí výsledkový registr ADC0\_RA. V hlavní programové smyčce již není zapotřebí jednotlivě spouštět převody a čekat na jejich dokončení. Stačí pouze číst výsledkový registr a provádět odpovídající vyhodnocení hodnoty. Upravený zdrojový kód programu je uveden ve výpisu programu 3.2.



Program 3.2:

```
// Definice masek pro červenou a zelenou LED
#define M_LED_R (1<<8)
#define M_LED_G (1<<10)

int main(void)
{
    uint16_t AD_val; // proměnná pro uložení výsledku převodu
    // Inicializace SIM modulu
    SIM->SCGC6 |= (1<<27); // Povolení hodinového signálu ADC0 modulu
    SIM->SCGC5 |= (1<<10); // Povolení hodinového signálu PORTB modulu
    SIM->SCGC5 |= (1<<11); // Povolení hodinového signálu PORTC modulu
    // Inicializace PORT modulu
    PORTB->PCR[8] = 0;
    PORTB->PCR[8] |= (1<<8); // Nastavení MUX PTB8 na ALT1 (GPIO), červená LED
    PORTB->PCR[10] |= (1<<8); // Nastavení MUX PTB10 na ALT1 (GPIO), zelená LED
    PORTC->PCR[2] = 0; // Digitální funkce na pinu PTC2 vypnuty
    // Inicializace GPIO
    GPIOB->PDDR |= M_LED_R; // Pin s připojenou červenou LED výstupní režim
    GPIOB->PDDR |= M_LED_G; // Pin s připojenou zelenou LED výstupní režim
    // Inicializace A/D převodníku
    ADC0->SC1[0] = 0x1f;
    ADC0->CFG1 = 0x31;
    ADC0->CFG2 = 0;
    ADC0->SC2 = 0;
    ADC0->SC3 = 0x0c; // Povolení kontinuálních převodů (bit ADC0 = 1)
    ADC0->SC1[0] = 11; // Spust' kontinuální převody na kanálu 11 (potenciometr)
    while(1)
    {
        AD_val = ADC0->R[0]; // Ulož' výsledek převodu
        if (AD_val < 94) {
            GPIOB->PDOR &= ~M_LED_R; // Rozsvícení červené LED
            GPIOB->PDOR |= M_LED_G; // Zhasnutí zelené LED
        }
        else {
            GPIOB->PDOR |= M_LED_R; // Zhasnutí červené LED
            GPIOB->PDOR &= ~M_LED_G; // Rozsvícení zelené LED
        }
    }
}
```

### 3.4 Zadání samostatné práce

1. Upravte ukázkový program 3.1 pro indikaci stavu akumulátoru tak, aby kromě stavů nabito (zelená LED) a vybito (červená LED) indikoval i stav z poloviny nabito rozsvícením žluté LED při vstupním napětí v rozsahu od 1,25 do 1,15 V. Zelená LED bude svítit při napětí větším než 1,25 V, červená při napětí menším než 1,15 V. Proved'te změnu nastavení A/D převodníku na 10bitové rozlišení.
2. Vytvořte sadu obslužných podprogramů pro ovládání modulu A/D převodníku. Implementujte následující funkce:
  - *void ADC0\_Init(uint8\_t ADres)*  
Inicializace SIM, PORT a ADC0 modulu souvisejících s funkcí A/D převodníku na výukovém kitu. Funkci se předává argument *ADres* určující požadované rozlišení převodu.
  - *uint16\_t ADC0\_GetVal(uint8\_t channel)*  
Spustí převod na kanálu specifikovaným argumentem *channel*. Po dokončení převodu funkce vrátí převedenou hodnotu.
3. Vytvořte program pro ovládání RGB LED pomocí potenciometru. Při jeho otáčení se budou na RGB LED postupně rozsvěcovat následující barvy: černá (vypnuto), červená, žlutá, zelená, purpurová, tyrkysová, bílá. Přepínání jednotlivých barev bude rovnoměrně rozděleno na celý rozsah potenciometru. Pro realizaci programu použijte funkce vytvořené v úkolu 2. Nastavte A/D převodník na 16bitové rozlišení převodu.

## 4 ČASOVACÍ SUBSYSTEM KL25Z

Mikropočítač MKL25Z128 je vybaven celkem třemi nezávislými moduly časovačů označených TPM0 až TPM2. Časovač TPM0 disponuje 6 kanály, TPM1 a TPM2 má každý po 2 kanálech. Modul časovače zajišťuje následující funkce:

- Generování přerušení od události přetečení časovače (TOF), zachycení vstupu a komparace výstupu (CHxF)
- Zachycení vstupní události na pinu kanálu (input capture)
- Změna stavu pinu kanálu v definovaný okamžik (output compare)
- Pulzně-šířková modulace (PWM) [2]

### 4.1 Popis funkce TPM modulu

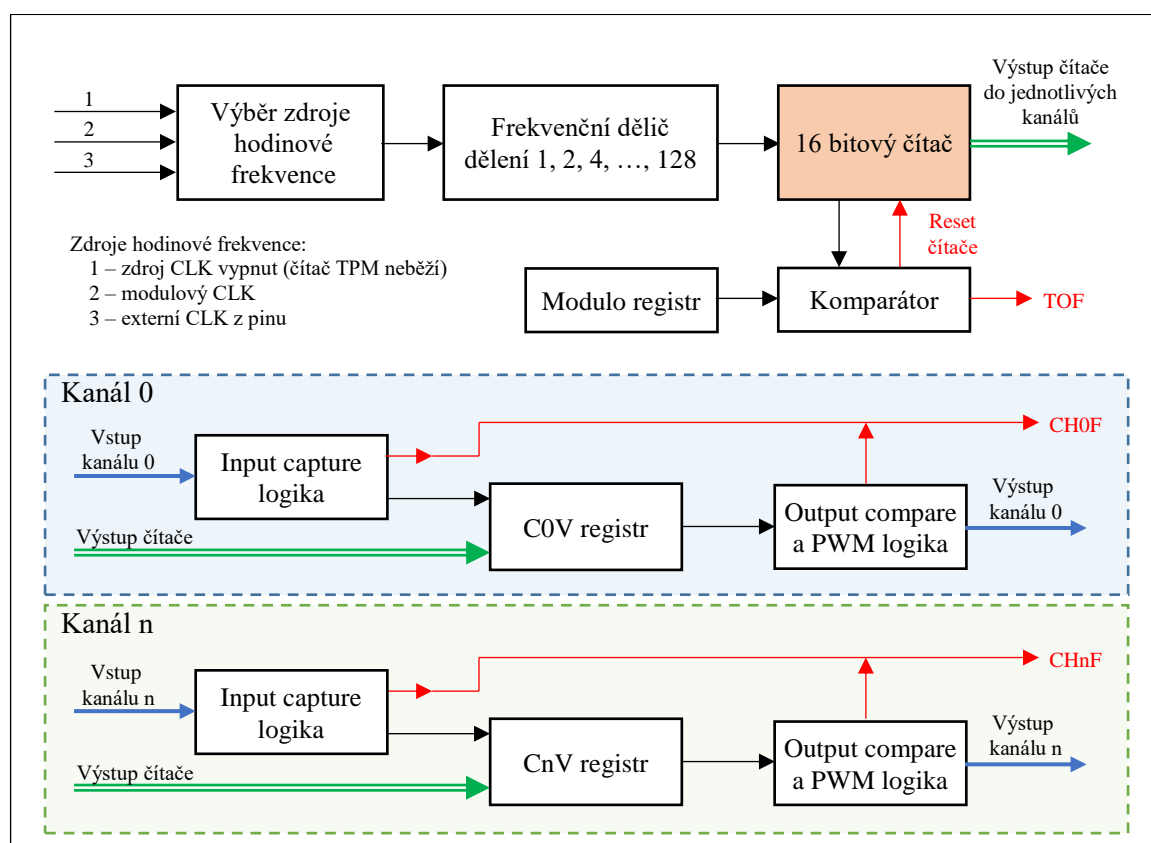
Funkce TPM modulu je zřejmá ze zjednodušené vnitřní struktury vyobrazené na obrázku 26. Na vstupu časovače se nachází blok multiplexoru pro výběr zdroje hodinové frekvence. Ten umožňuje vypnutí zdroje hodinové frekvence (čítač neběží), přepnutí na hodinový signál distribuovaný pro interní moduly nebo externí hodinový signál dostupný na příslušném vnějším pinu mikropočítače. Za ním následuje frekvenční dělič s nastavitelným dělicím poměrem pro přizpůsobení frekvence hodinového signálu na požadovanou hodnotu před vstupem do 16bitového čítače. Hodnota v čítači je v režimu čítání nahoru inkrementována při každé náběžné hraně hodinovém pulsu na jeho vstupu. Ta je porovnávána komparátorem s obsahem modulo registru a v případě jeho překročení, komparátor vygeneruje reset signál pro čítač a zároveň nastaví příznak přetečení časovače TOF, který v případě jeho povolení vygeneruje přerušení. Příznak TOF se za běhu časovače periodicky nastavuje po přesně daném čase, což lze v programu využít jako časovou základnu. Z výstupu čítače je jeho aktuální hodnota přivedena do jednotlivých kanálů, kde se dále používá pro zajištění funkcí zachycení vstupu, komparace výstupu či PWM modulace [2].

V režimu zachycení vstupu se při detekci nastavené hrany vstupního signálu na pinu TPM\_CHn uloží aktuální hodnota čítače do CnV registru a nastaví se příznakový bit CHxF. Pro správnou funkci musí být frekvence vnějšího vstupního signálu nižší nebo maximálně rovna frekvenci hodinového signálu vstupujícího do čítače vydělená čtyřmi. Kanál časovače



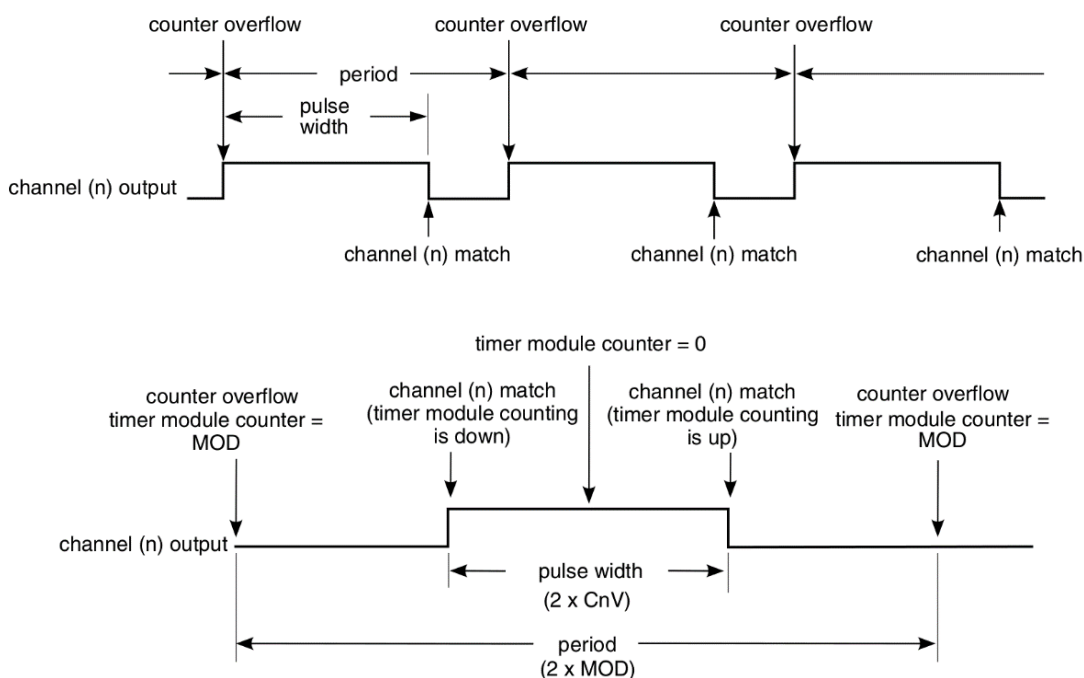
lze také použít pro generování pulsu na jeho výstupu. K tomu slouží režim porovnání výstupu, kdy při shodě hodnoty uložené v CnV registru a čítače časovače, se provede na výstupním pinu kanálu nastavená operace a zároveň se nastaví příznak CHx<sub>F</sub>. Stav výstupu může být při této události nastaven do logické 1, logické 0 nebo negovat předchozí stav [2].

V režimu pulzně-šířkové modulace (PWM) může časovač na výstupním kanálu generovat buď hranově zarovnanou PWM (EPWM) nebo středově zarovnanou PWM (CPWM). Při hranově zarovnaném režimu je začátek pulsu všech generovaných PWM signálů v rámci modulu časovače zarovnán vždy na počátek periody. Její délka je dána obsahem modulo registru zvýšeného o 1 a šířka pulsu je definována hodnotou uloženou v registru CnV příslušného kanálu. Příznakový bit CHx<sub>F</sub> je nastaven při shodě obsahu čítače s registrem CnV (na konci pulsu). V režimu středově zarovnaného PWM čítač na začátku periody čítá směrem nahoru až do okamžiku dosažení hodnoty v modulo registru, kdy začne čítat směrem dolů k nule. Délka pulsu je tedy rovna dvojnásobku hodnoty uložené v CnV a perioda dvojnásobku hodnoty v modulo registru. V okamžiku dosažení nulové hodnoty v čítači jsou



Obrázek 26: Zjednodušená vnitřní struktura TPM modulu [2].

pulsy generované na všech kanálech daného časovače právě v polovině délky svého trvání. Funkce hranově a středově zarovnané PWM je porovnána na obrázku 27. Jednotlivé kanály mohou být nakonfigurovány tak, aby generovaly požadavek na přerušení při nastavení příznaku CHxIF [2].



Obrázek 27: Hranově a středově zarovnaná PWM [2].

Přehled používaných kanálů časovačů na výukovém kitu je uveden v tabulce 6. Vybrané kanály jsou dostupné pouze na MCU portu pro externí výukové moduly.

Časovač	Kanál	Pin	Připojená periferie
TPM0	CH0	PTD0	MCU port – 22
TPM0	CH1	PTD1	<b>RGB LED – modrá</b>
TPM0	CH2	PTD2	MCU port – 23
TPM0	CH3	PTD3	MCU port – 24
TPM0	CH4	PTD4	MCU port – 26
TPM0	CH5	PTD5	MCU port – 25
TPM1	CH1	PTA13	<b>Podsvícení LCD displeje</b>
TPM2	CH0	PTB18	<b>RGB LED – červená</b>
TPM2	CH1	PTB19	<b>RGB LED – zelená</b>

Tabulka 6: Přehled použitých kanálů časovačů na kitu.

## 4.2 Registry modulu časovače TPM

Modul časovače je ovládán prostřednictvím 7 registrů. Níže je uveden popis vybraných registrů pro zajištění základní funkcionality TPM modulu.

### TPM<sub>x</sub>\_SC – TPM<sub>x</sub> stavový a řídicí registr

Význam bitů (v hranaté závorce uvedeno číslo bitu v registru) [2]:

*DMA* [8] – Povolení DMA přenosu při TOF: 0 = zakázáno

*TOF* [7] – Příznak přetečení časovače: 0 = čítač nepřetekl; 1 = čítač přetekl

*TOIE* [6] – Povolení přerušení od TOF: 0 = přerušení zakázáno

*CPWMS* [5] – Středově zarovnaná PWM: 0 = zakázána; 1 = povolena

*CMOD* [4-3] – Výběr vstupního hodinového signálu:

00 – Hodinový signál vypnut – čítač neběží

01 – Hodinový signál pro moduly

10 – Hodinový signál z externího pinu

11 – rezervováno

*PS* [2-0] – Nastavení děliče vstupního hodinového signálu:

000 – Dělení 1                      100 – Dělení 16

001 – Dělení 2                      101 – Dělení 32

010 – Dělení 4                      110 – Dělení 64

011 – Dělení 8                      111 – Dělení 128

Pozn.: Konfigurační bity *CPWMS* a *PS* lze změnit pouze při vypnutém čítači (*CMOD* = 00).

### TPM<sub>x</sub>\_CNT – TPM<sub>x</sub> čítač

Registr obsahuje aktuální 16bitovou hodnotu čítače TPM<sub>x</sub> modulu. Čítač je vynulován v případě resetu mikropočítače nebo při zápisu libovolné hodnoty do tohoto registru [2].

### TPM<sub>x</sub>\_MOD – TPM<sub>x</sub> modulo registr

Registr slouží pro uložení 16bitové modulo hodnoty čítače TPM<sub>x</sub> modulu. Jakmile čítač přesáhne nastavenou modulo hodnotu, nastaví se TOF příznak a čítač se inicializuje dle nastaveného režimu čítání [2].





### TPM<sub>x</sub>\_CnSC – TPM<sub>x</sub> stavový a řídicí registr kanálu n

Význam bitů (v hranaté závorce uvedeno číslo bitu v registru) [2]:

*CHF* [7] – Příznakový bit kanálu: 0 = bez události; 1 = nastala událost na kanálu

*CHIE* [6] – Povolení přerušení od události na kanálu: 0 = přerušení zakázáno

*MSB* [5] – Výběr režimu kanálu, nastavení viz tabulka 7

*MSA* [4] – Výběr režimu kanálu, nastavení viz tabulka 7

*ELSB* [3] – Výběr hrana / úroveň, nastavení viz tabulka 7

*ELSA* [2] – Výběr hrana / úroveň, nastavení viz tabulka 7

*DMA* [0] – Povolení DMA pro daný kanál: 0 = DMA zakázáno

<i>CPWMS</i>	<i>MSB:MSA</i>	<i>ELSB:ELSA</i>	Režim	Konfigurace
X	00	00	-	Kanál vypnut
X	01/10/11	00	SW porovnání	Pin kanálu není použit
0	00	01	Zachycení vstupu (IC)	Náběžná hrana
		10		Sestupná hrana
		11		Náběžná nebo sestupná hrana
	01	01	Porovnání výstupu (OC)	Přepnutí stavu výstupu
		10		Vynulování výstupu
		11		Nastavení výstupu
	10	10	Hranově zarovnaná PWM	Aktivní puls vysoké úrovně
		X1		Aktivní puls nízké úrovně
	11	10	Porovnání výstupu (OC)	Výstup vynulován při shodě
		X1		Výstup nastaven při shodě
1	10	10	Středově zarovnaná PWM	Aktivní puls vysoké úrovně
		X1		Aktivní puls nízké úrovně

Tabulka 7: Nastavení režimu časovače [2].

### TPM<sub>x</sub>\_CnV – TPM<sub>x</sub> datový registr kanálu n

Registr obsahuje 16bitovou hodnotu kanálu n časovače x. V režimu zachycení vstupu (IC) je zde uložena hodnota čítače v okamžiku detekce události na vstupu kanálu. Zápis do registru je v tomto režimu ignorován. V případě nastavení funkce porovnání výstupu (OC) nebo PWM se do toho registru zapisuje hodnota pro porovnání s čítačem [2].

**TPMx\_STATUS** – TPMx stavový registr

Význam bitů (v hranaté závorce uvedeno číslo bitu v registru) [2]:

*TOF* [7] – Příznak přetečení časovače: 0 = čítač nepřetekl; 1 = čítač přetekl

*CH5F* [5] – Příznakový bit kanálu: 0 = bez události; 1 = nastala událost na kanálu

*CH4F* [4] – Příznakový bit kanálu: 0 = bez události; 1 = nastala událost na kanálu

*CH3F* [3] – Příznakový bit kanálu: 0 = bez události; 1 = nastala událost na kanálu

*CH2F* [2] – Příznakový bit kanálu: 0 = bez události; 1 = nastala událost na kanálu

*CH1F* [1] – Příznakový bit kanálu: 0 = bez události; 1 = nastala událost na kanálu

*CH0F* [0] – Příznakový bit kanálu: 0 = bez události; 1 = nastala událost na kanálu

**TPMx\_CONF** – TPMx konfigurační registr

Význam bitů (v hranaté závorce uvedeno číslo bitu v registru) [2]:

*TRGSEL* [27-24] – Výběr zdroje spouštění čítače, nastavení viz tabulka 8

*CROT* [18] – Načtení obsahu čítače při spouštěcí události: 0 = není načten

*CSOO* [17] – Zastavení čítače při přetečení: 0 = není zastaven

*CSOT* [16] – Start čítače až při spouštěcí události: 0 = start ihned po povolení

*GTBEEN* [9] – Povolení globální časové základny (TPM1): 0 = zakázána

*DBGMODE* [7-6] – Funkce časovače v debug režimu:

00 – TPM čítač je pozastaven

11 – TPM čítač běží

*DOZEEN* [5] – Funkce časovače ve wait režimu: 0 = běží; 1 = zastaven

<b><i>TRGSEL</i> [27-24]</b>	<b>Zdroj spouštění</b>	<b><i>TRGSEL</i> [27-24]</b>	<b>Zdroj spouštění</b>
0000	Z externího pinu	1000	Přetečení TPM0
0001	Výstup CMP0	1001	Přetečení TPM1
0010	Rezervováno	1010	Přetečení TPM2
0011	Rezervováno	1011	Rezervováno
0100	PIT kanál 0	1100	RTC alarm
0101	PIT kanál 1	1101	RTC sekundy
0110	Rezervováno	1110	LPTMR
0111	Rezervováno	1111	Rezervováno

Tabulka 8: Nastavení zdroje spouštění čítače [2].

### 4.3 Inicializace souvisejících modulů

Pro funkci časovače TPM je nezbytné dle požadované funkce nastavit některé z dalších modulů mikropočítače. V registru `SIM_SOPT2` modulu `SIM` nejdříve vybereme zdroj hodinového signálu pro časovače nastavením bitového pole `TPMSRC` (bity 25-24). Na výběr jsou celkem 4 možnosti nastavení dle tabulky 9. Při nastavení `TPMSRC` [25-24] na 01 je vybrán zdroj `MCGFLLCLK` nebo `MCGPLLCLK/2`. Upřesnění se provede nastavením bitu 16 `PLLFLSEL`, kde 0 odpovídá `MCGFLLCLK` a 1 specifikuje `MCGPLLCLK/2`. V registru `SIM_SCGC6` dle použitého TPM časovače se zápisem 1 do konfiguračních bitů 24, 25 a 26 povolí hodinový signál pro `TPM0`, `TPM1` a `TPM2`, viz obrázek 25.

V případě použití funkcí zachycení vstupu, porovnání výstupu či PWM jsou vstupně / výstupní piny kanálů časovačů součástí `PORT` modulů, a proto je nutné jim dle aktuálně používaných pinů povolit hodinový signál v `SIM_SCGC5`. Na výukovém kitu se to týká pouze `PORTA`, `PORTB` a `PORTD` modulů, viz tabulka 10. V odpovídajícím `PORT` modulu je v registru pro řízení pinů `PORTx_PCRn` v bitovém poli `MUX` zapotřebí nastavit příslušnou funkci pinu dle tabulky 10.

<b><i>TPMSRC</i> [25-24]</b>	<b>Zdroj hodinového signálu časovačů</b>
00	Hodinový signál vypnut
01	<code>MCGFLLCLK</code> nebo <code>MCGPLLCLK/2</code>
10	<code>OSCERCLK</code>
11	<code>MCGIRCLK</code>

Tabulka 9: Nastavení `TPMSRC` bitů v `SIM_SOPT2` [2].

<b>Pin</b>	<b><i>MUX</i> [10 - 8]</b>	<b>Připojená periferie</b>
<code>PTD0</code>	100	MCU port – 22
<code>PTD1</code>	100	<b>RGB LED – modrá</b>
<code>PTD2</code>	100	MCU port – 23
<code>PTD3</code>	100	MCU port – 24
<code>PTD4</code>	100	MCU port – 26
<code>PTD5</code>	100	MCU port – 25
<code>PTA13</code>	011	<b>Podsvícení LCD displeje</b>
<code>PTB18</code>	011	<b>RGB LED – červená</b>
<code>PTB19</code>	011	<b>RGB LED – zelená</b>

Tabulka 10: Nastavení `MUX` bitů v `PORTx_PCRn` pro funkce TPM [2].

Každý modul časovače může dle nastavení konfiguračních registrů TPM generovat požadavek na přerušení při událostech přetečení časovače (TOF) nebo při vstupní či výstupní události na odpovídajícím kanále (CHnF). Všechny požadavky na přerušení v rámci TPM modulu jsou logicky sečteny a výsledný jeden přerušovací signál je přiveden do řadiče přerušení NVIC, který je součástí ARM jádra. Každému zdroji přerušení, které nejsou součástí jádra, je přiřazeno číslo přerušení IRQ0 až IRQ31 a odpovídající vektor přerušení 16 až 47. Konkrétní přiřazení vektoru přerušení a čísla přerušení pro moduly TPM je uvedeno v tabulce 11. Ve sloupci NVIC IP je uvedeno číslo IP registru, ve kterém se provede konfigurace priority konkrétního přerušení. Mikropočítač podporuje 4 prioritní úrovně pro přerušení 0 až 3, kde 0 představuje nejvyšší prioritu. Pro nastavení priority jsou v IP registru rezervovány pro každé IRQ dva bity, jejichž umístění je vyobrazeno na obrázku 28.

Adresa	Vektor	IRQ	NVIC IP	IP	Modul
0x0000_0084	33	17	4	[15-14]	TPM0
0x0000_0088	34	18	4	[23-22]	TPM1
0x0000_008C	35	19	4	[31-30]	TPM2

Tabulka 11: Konfigurace NVIC pro TPM moduly [2].

	31	30		23	22		15	14		7	6		0
IP0:	IRQ3	0	0	0	0	0	0	0	0	0	0	0	0
IP1:	IRQ7	0	0	0	0	0	0	0	0	0	0	0	0
IP2:	IRQ11	0	0	0	0	0	0	0	0	0	0	0	0
IP3:	IRQ15	0	0	0	0	0	0	0	0	0	0	0	0
IP4:	IRQ19	0	0	0	0	0	0	0	0	0	0	0	0
IP5:	IRQ23	0	0	0	0	0	0	0	0	0	0	0	0
IP6:	IRQ27	0	0	0	0	0	0	0	0	0	0	0	0
IP7:	IRQ31	0	0	0	0	0	0	0	0	0	0	0	0

Obrázek 28: Struktura registrů NVIC IP0 až IP7 [2].

Příslušné přerušení se v řadiči NVIC povolí v registru ISER, kde každý bit představuje právě jeden požadavek na přerušení IRQn [4]. Zápisem jedničky se odpovídající přerušení povolí, například při zápisu 1 do 17. bitu ISER se povolí přerušení IRQ17, viz obrázek 29.

	31	23						22	15						14	7						6	0									
ISER:	IRQ31_EN	IRQ30_EN	IRQ29_EN	IRQ28_EN	IRQ27_EN	IRQ26_EN	IRQ25_EN	IRQ24_EN	IRQ23_EN	IRQ22_EN	IRQ21_EN	IRQ20_EN	IRQ19_EN	IRQ18_EN	IRQ17_EN	IRQ16_EN	IRQ15_EN	IRQ14_EN	IRQ13_EN	IRQ12_EN	IRQ11_EN	IRQ10_EN	IRQ9_EN	IRQ8_EN	IRQ7_EN	IRQ6_EN	IRQ5_EN	IRQ4_EN	IRQ3_EN	IRQ2_EN	IRQ1_EN	IRQ0_EN

Obrázek 29: Struktura registru NVIC ISER [5].

#### 4.4 Inicializace modulu časovače

Inicializaci časovače lze rozčlenit do několika základních kroků, které předpokládají, že již byla provedena konfigurace všech funkčně souvisejících modulů, tj. v SIM modulu byl zvolen vhodný zdroj hodinového signálu pro TPM (další text předpokládá OSCERCLK s frekvencí 8 MHz) a byl povolen hodinový signál pro příslušný TPM, v případě použití přerušení od přetečení časovače TOF byl adekvátně nakonfigurován NVIC modul.

Pro nastavení časovače je zapotřebí:

1. Zastavit interní čítač zápisem 0 do registru TPMx\_SC. Tento krok zajistí, že bude možné zapisovat do konfiguračních bitů *CPWMS* a *PS*, které jsou jinak za běhu čítače chráněny proti zápisu.
2. Čekat na zastavení čítače testováním bitů *CMOD* zda jsou oba nulové.
3. Vynulovat čítač zápisem 0 do TPMx\_CNT.
4. Nastavit požadovanou hodnotu v modulo registru TPMx\_MOD dle výpočetního vztahu uvedeného níže.
5. V řídicím a konfiguračním registru TPMx\_SC zakázat DMA ( $DMA = 0$ ), nastavit požadovanou hodnotu děličky hodinového signálu bity *PS*, směr čítání nahoru ( $CPWMS = 0$ ), vynulovat TOF příznak zápisem 1 do příznakového bitu *TOF* a v případě použití přerušení od TOF, povolit jej zápisem 1 do bitu *TOIE*.
6. Spustit čítač časovače zápisem 01 do bitů *CMOD* v TPMx\_SC.
7. Pokud není použito přerušení, stav přetečení časovače TOF se musí testovat čtením stavového registru TPMx\_STATUS. Pokud je bit  $TOF = 1$ , čítač dosáhl

nastavené modulo hodnoty. Na základě této informace se provede příslušná akce v programu. Nesmí se zapomenout vynulovat příznak přetečení časovače zápisem 1 do tohoto bitu.

8. V případě použití přerušení, je při události TOF vygenerován požadavek na přerušení. Mikroprocesor dokončí aktuálně prováděnou instrukci a provede skok na obsluhu přerušení, jejíž tělo je definováno funkcí s odpovídajícím názvem (pro TPM0 je to funkce *void TPM0\_IRQHandler(void)*). V samotné obsluze se nejprve musí ověřit, zda je zdrojem přerušení TOF čtením stavového registru TPMx\_STATUS. Pokud ano, vynuluje se příslušný příznakový bit zápisem 1 a může se provést programové obsloužení události.

Hodnotu modulo registru TPMx\_MOD časovače na základě znalosti frekvence vstupního hodinového signálu  $f_{TPM}$  v [Hz] pro požadovaný čas  $t_{TOF}$  v [s] do přetečení interního čítače lze vypočítat pomocí vztahu:

$$TPMx\_MOD = \frac{t_{TOF} \cdot f_{TPM}}{2^{PS}}, \quad (2)$$

kde PS je hodnota v rozsahu 0 až 7 uložená do PS bitů v registru TPMx\_SC pro nastavení děličky hodinové frekvence.

#### **Příklad:**

Nakonfigurujte časovač TPM tak, aby byla generována událost TOF (přetečení časovače) každých 0,25 s. Frekvence vstupního hodinového signálu je 8 MHz.

#### **Řešení:**

Stanovíme hodnotu děličky hodinového signálu tak, aby údaj zapsaný do modulo registru byl co největší (pro dosažení co největšího rozlišení časovače) a zároveň nepřesáhl maximální hodnotu 65535.

Teoretická hodnota děliče pro dosažení maximální hodnoty modulo je:

$$DIV_T = \frac{t_{TOF} \cdot f_{TPM}}{65535} \quad (3)$$



Dosadíme do (3) zadané hodnoty:

$$DIV_T = \frac{0,25 \cdot 8000000}{65535} = \underline{30,518}$$

Nejbližší vyšší hodnota děliče podporovaná časovačem je 32, tj.  $2^{PS} = 32$ . Dosadíme ji společně s požadovaným časem  $t_{TOF}$  do rovnice (2) a získáváme:

$$TPMx\_MOD = \frac{0,25 \cdot 8000000}{32} = \underline{\underline{62500}}.$$

**Závěr:** Pro zadaný čas  $t_{TOF}$  a vstupní hodinový signál s frekvencí  $f_{TPM} = 8$  MHz, musí být v časovači TPM nastaven dělitel hodinové frekvence na 32, kterému odpovídá nastavení bitů  $PS = 5$  v  $TPMx\_SC$  a modulo čítače v registru  $TPMx\_MOD = 62500$ .

## 4.5 Ukázkové programy

V této kapitole naleznete ukázkové řešené příklady související s programovou obsluhou TPM modulu v C jazyce. V rámci laboratorního cvičení si funkci programů důkladně vyzkoušejte jejich odkrokováním v příslušném programovém prostředí.

### 4.5.1 Periodické přerušení od přetečení časovače

#### Zadání

Vytvořte program, který bude pravidelně každých 0,25 s přepínat stav červené LED připojené na pinu PTB8. Pro řešení použijte modul časovače TPM0 a periodické přerušení od události přetečení časovače.

#### Řešení

V úvodní části programu se nejprve vykoná inicializace SIM modulu tak, aby byla zajištěna distribuce hodinového signálu do všech modulů potřebných pro funkci programu TPM0 a PORTB. To se provede nastavením příslušných registrů SIM modulu: pro TPM0 zápisem 1 do 24. bitu  $SIM\_SCGC6$  a pro PORTB do 10. bitu  $SIM\_SCGC5$ . U časovače je nutno ještě vybrat, jaký zdroj hodinového signálu bude na jeho vstup přiveden. U vývojové desky KL25Z je vhodné vybrat zdroj přesného 8MHz hodinového signálu  $OSCERCLK$





který má frekvenci řízenou externím krystalem. Výběr se provede příslušným nastavením bitového pole  $TPMSRC = 10_{(2)}$  v registru  $SIM\_SOPT2$ . Jeho funkce je podmíněna správnou inicializací hodinového generátoru MCG, která je zajištěna v případě vytvoření symbolu  $CLOCK\_SETUP = 1$  v nastavení projektu. Dále se musí provést inicializace funkce pinu PTB8 v bitovém poli  $MUX$  registru  $PORTB\_PCR8$  na ALT1 odpovídající GPIO. Následně se pin PTB8 přepne do výstupního režimu zápisem 1 do registru  $GPIOB\_PDDR$  a nastaví se počáteční stav LED na zhasnutý zápisem odpovídající hodnoty do  $GPIOB\_PDOR$ . Z důvodu použití přerušení od události přetečení časovače TOF, bude zapotřebí nakonfigurovat modul řadiče přerušení NVIC, který je součástí jádra ARM. V zásadě jde pouze o povolení odpovídajícího čísla přerušení IRQ použitého TPM modulu v registru  $ISER$ . Náš program bude používat časovač TPM0 a jemu přísluší požadavek přerušení IRQ17. Provede se tedy nastavení 17. bitu v  $ISER$  na 1. Prioritu přerušení lze nastavit ve čtyřech úrovních od 0 do 3 v registrech NVIC IP0 až IP7, viz obrázek 28. Z obrázku je zřejmé, že nastavení priority pro IRQ17 se provede v registru IP4 v bitech 14 a 15. Při zápisu hodnoty  $11_{(2)}$  do těchto bitů se nastaví nejnižší priorita pro IRQ17. Nyní již zbývá nastavení časovače TPM0 pro generování přerušení od jeho přetečení každých 0,25 s. Postup výpočtu hodnoty děličky hodinového signálu a modulo hodnoty čítače je pro tento konkrétní čas uveden v příkladu výše. Při nastavování časovače je nutné mít na paměti, že konfigurační bity  $CPWSMS$  (výběr režimu čítání) a  $PS$  (nastavení děličky hodinového signálu) registru  $TPM0\_SC$  nelze při běžícím čítači změnit. Proto je nutné nejprve čítač časovače zastavit zápisem hodnoty  $00_{(2)}$  do bitového pole  $CMOD$  a čekat, než se změna v registru projeví. Od tohoto okamžiku je již možno provést změnu nastavení děličky v  $PS$  poli  $TPM0\_SC$  na hodnotu  $101_{(2)}$ . Dále se uloží vypočítaná modulo hodnota čítače do registru  $TPM0\_MOD = 62500$ . Poslední konfigurační kroky směřují opět do registru  $TPM0\_SC$ , kde se provede vynulování TOF příznaku ( $TOF = 1$ ), povolí se přerušení od TOF události ( $TOIE = 1$ ), nastaví směr čítání nahoru ( $CPWMS = 0$ ) a spustí se čítač ( $CMOD = 01_{(2)}$ ). V hlavní programové smyčce již může program vykonávat jakékoliv další činnosti, protože přepínání stavu LED každých 0,25 s zajistí obsluha přerušení reprezentovaná funkcí  $TPM0\_IRQHandler()$ . V obsluze přerušení se musí nejprve zjistit zdroj přerušení čtením registru  $TPM0\_STATUS$ . Pokud je to TOF, příznak se vynuluje zápisem 1 do odpovídajícího bitu a následně se provede negace stavu červené LED, čímž je obsluha ukončena. Zdrojový kód programu je uveden ve výpisu programu 4.1.



Program 4.1:

```
// Definice masky pro červenou LED
#define M_LED_R (1<<8)

int main(void)
{
    // Inicializace OSC0, SIM, PORTB a GPIO modulů
    OSC0->CR |= OSC_CR_ERCLKEN_MASK; // Povolení OSCERCLK v OSC0 modulu
    SIM->SOPT2 &= ~(SIM_SOPT2_TPMSRC_MASK); // Vynulování TPMSRC bitového pole
    SIM->SOPT2 |= SIM_SOPT2_TPMSRC(2); // Zdroj hodinového signálu pro TPM je OSCERCLK
    SIM->SCGC5 |= SIM_SCGC5_PORTB_MASK; // Povolení hodinového signálu PORTB modulu
    SIM->SCGC6 |= SIM_SCGC6_TPM0_MASK; // Povolení hodinového signálu TPM0 modulu
    PORTB->PCR[8] = 0;
    PORTB->PCR[8] |= PORT_PCR_MUX(1); // Nastavení MUX PTB8 na ALT1 (GPIO)
    GPIOB->PDDR |= M_LED_R; // Pin s připojenou červenou LED výstupní režim
    GPIOB->PDOR |= M_LED_R; // Zhasnutí červené LED

    // Inicializace řadiče přerušení NVIC
    NVIC->IP[4] |= (3<<14); // Nastavení nejnižší priority pro IRQ17
    NVIC->ISER[0] |= (1<<17); // Povolení přerušení IRQ17

    // Inicializace časovače TPM0
    TPM0->SC = 0; // Zakaž funkci čítače
    while (TPM0->SC & TPM_SC_CMOD_MASK); // Čekaj na provedení operace
    TPM0->CNT = 0; // Vynuluj čítač před zápisem do modulu registru
    TPM0->MOD = 62500; // Nastavení na TOF každých 250 ms
    TPM0->SC = 0xc0; // Vynulování TOF, TOIE = 1, čítání nahoru
    TPM0->SC |= TPM_SC_PS(5); // Nastavení děličky na 32
    TPM0->SC |= TPM_SC_CMOD(1); // Spuštění čítače, hodinový signál pro moduly

    while(1) {
        i++; // Zde může program vykonávat další činnosti...
    }
    return 0;
}

// Obsluha přerušení od časovače TPM0
void TPM0_IRQHandler(void)
{
    if ((TPM0->STATUS & TPM_STATUS_TOF_MASK) != 0) // Přerušení od TOF?
    {
        TPM0->STATUS |= TPM_STATUS_TOF_MASK; // Vynuluj příznak TOF zápisem 1 do TOF
        GPIOB->PDOR ^= M_LED_R; // Negace stavu červené LED
    }
}
```

#### 4.5.2 Hardwarově generovaná pulzně-šířková modulace

##### Zadání

Vytvořte program, který bude ovládat jas modré barevné složky RGB LED pomocí hardwarově generované pulzně-šířkové modulace. Požadovaný jas bude zvyšován tlačítkem SW1 v krocích po 10 % z maxima. Po dosažení maximální hodnoty se po následujícím stisku tlačítka LED vypne a dále se bude opět zvyšovat jas v daném kroku. Periodu PWM nastavte na 20 ms.

##### Řešení

Inicializační část programu bude obdobná jako v předchozím příkladu 4.5.1. Navíc zde musí být přidána inicializace modulu PORTA, ke kterému je připojeno na pin PTA4 tlačítko SW1. To znamená: povolení hodinového signálu modulu PORTA zápisem 1 do 9. bitu SIM\_SCGC5, nastavení funkce pinu PTA4 na ALT1 (GPIO) zápisem 001<sub>(2)</sub> do bitového pole *MUX* v registru PORTA\_PCR4. Vstupní režim pinu se nakonfiguruje zápisem 0 do 4. bitu GPIOA\_PDDR. Modrá LED integrovaná v RGB LED je připojena na pin PTD1 odpovídající kanálu 1 časovače TPM0. Aby tento pin fungoval jako výstup kanálu časovače TPM0\_CH1, je nutné mu nastavit funkci ALT4 (viz snímek 62 prezentace k laboratorním cvičením), zápisem hodnoty 100<sub>(2)</sub> do bitového pole *MUX* v registru PORTD\_PCR1. Nastavení směru pinu se neprovádí v GPIO modulu, protože ve funkci ALT4 je pin řízen přímo časovačem. Základní inicializace časovače je popsána v příkladu 4.5.1, navíc se zde musí nakonfigurovat funkce kanálu 1 časovače. K tomuto účelu slouží registr TPM0\_C1SC, kde pro výběr hranově zarovnané PWM s nízkou aktivní úrovní (LED svítí při log. 0 na výstupu) je zapotřebí nastavit konfigurační bity *MSB* a *ELSA* na 1, zbytek registru se vynuluje (viz tabulka 7). Tohoto nastavení se docílí zápisem hodnoty 24<sub>(16)</sub>. Hodnoty modulo registru a děličky hodinového signálu musí být nastaveny tak, aby perioda generovaného PWM signálu byla 20 ms. Provede se výpočet dle postupu uvedeného výše. Nejdříve se vypočítá teoretická hodnota děliče pro maximální dosažitelné modulo dle (3):

$$DIV_T = \frac{t_{TOF} \cdot f_{TPM}}{65535} = \frac{0,02 \cdot 8000000}{65535} = 2,44.$$

Dělička hodinového signálu čítače se nastaví na nejbližší vyšší podporovanou hodnotu, tj. 4. Do bitového pole *PS* registru TPM0\_SC se tedy zapíše 010<sub>(2)</sub>.



Nyní se již může vypočítat výsledná hodnota modulo čítače pomocí rovnice (2):

$$\text{TPM}_x\_MOD = \frac{t_{TOF} \cdot f_{TPM}}{2^{PS}} = \frac{0,02 \cdot 8000000}{2^2} = \underline{\underline{40000}}$$

Do registru TPM0\_MOD se tedy zapíše hodnota 40000. Datový registr kanálu 1 TPM0\_C1V se inicializuje hodnotou odpovídající 10 % plného jasu, aby po spuštění programu LED již svítila. Tímto je nastavení časovače dokončeno a po jeho spuštění již stačí v registru TPM0\_C1V měnit hodnotu v rozsahu 0 až 40000 a tím nastavovat aktivní délku pulzu od 0 do 20 ms. To má za úkol hlavní programová smyčka, která na začátku testuje, zda bylo stisknuto tlačítko SW1 čtením registru GPIOA\_PDIR a testováním stavu 4. bitu (0 = tlačítko stisknuto). V následujícím kroku se ověřuje přítomnost sestupné hrany na základě porovnání s minulým stavem tlačítka uloženým v proměnné *sw1\_ps*. Pokud byla zjištěna (předchozí stav tlačítka byl různý od nuly), provede se zvětšení obsahu proměnné *ch1\_value* o hodnotu odpovídající 10 % maximální délky pulsu. Ta se před zápisem do TPM0\_C1V testuje, zda hodnota není větší než modulo čítače. Pokud ano, nastaví se v proměnné délka pulsu na 0 a teprve potom se zapíše do TPM0\_C1V. Tímto je docíleno zhasnutí LED. Při dalším stisku tlačítka SW1 se jas opět zvýší o daný krok. Pro zajištění spolehlivé funkce tlačítka bez nežádoucích zákmitů, je vyhodnocování doplněno o krátké čekání funkcí *Cekej()* z příkladu 2.1 o délce přibližně 20 ms. Zdrojový kód programu je uveden ve výpisu programu 4.2.

*Program 4.2:*

```
// Definice periody PWM v ticích časovače a masky tlačítka SW1
#define PWM_PERIOD    40000
#define M_SW1         (1<<4)

int main(void)
{
    uint8_t sw1_ps = 1;                // Minulý stav tlačítka
    uint16_t ch1_value = PWM_PERIOD / 10; // Délka pulzu na kanálu 1 časovače
    // Inicializace OSC0, SIM, PORTA a PORTD modulů
    OSC0->CR |= OSC_CR_ERCLKEN_MASK;    // Povolení OSCERCLK v OSC0 modulu
    SIM->SOPT2 &= ~(SIM_SOPT2_TPMSRC_MASK); // Vynulování TPMSRC bitového pole
    SIM->SOPT2 |= SIM_SOPT2_TPMSRC(2);    // Zdroj hodinového signálu pro TPM je OSCERCLK (2)
```



## Program 4.2 (pokračování):

```
SIM->SCGC6 |= SIM_SCGC6_TPM0_MASK; // Povolení hodinového signálu TPM0 modulu
SIM->SCGC5 |= SIM_SCGC5_PORTD_MASK; // Povolení hodinového signálu PORTD modulu
SIM->SCGC5 |= SIM_SCGC5_PORTA_MASK; // Povolení hodinového signálu PORTA modulu
PORTD->PCR1 = 0;
PORTD->PCR1 |= PORT_PCR_MUX(4); // Nastavení MUX PTD1 na ALT4 (TPM0_CH1)
PORTA->PCR4 = 0;
PORTA->PCR4 |= PORT_PCR_MUX(1); // Nastavení MUX PTA4 na ALT1 (GPIO)
GPIOA->PDDR &= ~M_SW1; // PTA4 vstupní režim - tlačítko SW1

// Inicializace časovače
TPM0->SC = 0; // Zakaž funkci čítače
while (TPM0->SC & TPM_SC_CMOD_MASK); // Čekaj na provedení operace
TPM0->CNT = 0; // Vynuluj čítač před zápisem do modulo registru
TPM0->MOD = PWM_PERIOD; // Nastavení modulo registru na TOF každých 20 ms
TPM0->CONTROLS[1].CnSC = 0x24; // PWME s nízkou aktivní úrovní: MSB = 1, ELSA = 1
TPM0->CONTROLS[1].CnV = ch1_value; // Nastavení počáteční délky pulsu
TPM0->SC |= TPM_SC_PS(2); // Nastavení děličky na 4
TPM0->SC |= TPM_SC_CMOD(1); // Spuštění čítače

while(1)
{
    if ((GPIOA->PDIR & M_SW1) == 0) // Je stisknuto tlačítko SW1?
    {
        if (sw1_ps != 0)
        {
            // Detekována sestupná hrana na SW1, prodluž délku pulsu o 10 % z max. délky
            ch1_value = ch1_value + PWM_PERIOD / 10;
            if (ch1_value > PWM_PERIOD) ch1_value = 0;
            TPM0->CONTROLS[1].CnV = ch1_value;
        }
        Cekej(20); // Čekání cca 20 ms na doznění zákmitů tlačítka
        sw1_ps = 0; // Aktualizace minulého stavu tlačítka
    }
    else sw1_ps = 1; // Aktualizace minulého stavu tlačítka
}
return 0;
}
```

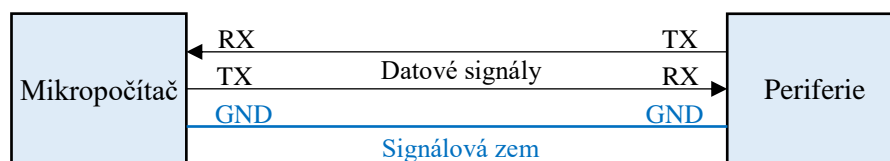
#### 4.6 Zadání samostatné práce

1. Upravte ukázkový program 4.1 tak, aby tlačítka SW1 a SW2 bylo možno nastavit periodu blikání červené LED od 100 ms do 1000 ms v kroku po 100 ms. Tlačítko SW1 bude periodu zkracovat, SW2 prodlužovat. Po dosažení krajních hodnot nastavené periody nesmí umožnit odpovídající tlačítko dále zkracovat / prodlužovat periodu. Tlačítko SW3 a SW4 bude mít funkci pro nastavení předdefinované periody: SW3 nastaví periodu 100 ms, SW4 1000 ms.
2. Upravte ukázkový program 4.2 pro možnost ovládání jasu modré barevné složky RGB LED pomocí tlačítek SW1 a SW2. Tlačítko SW1 bude zvyšovat jas v kroku 10 % až po maximální jas. Tlačítko SW2 jej bude snižovat ve stejném kroku až po úplné zhasnutí LED. Ošetřete funkci nastavování jasu tak, aby nedocházelo k nestandardnímu chování programu, tj. přetékání či podtékání řídicí proměnné uchovávající délku generovaného pulzu.
3. Rozšiřte funkci programu z bodu 2 zadání o možnost výběru ovládaného výstupního kanálu pomocí tlačítka SW3. Jeho postupnými stisky bude možno přepínat mezi nastavováním jasu červené, zelené a modré barevné složky RGB LED. Vlastní nastavení jasu pak bude probíhat pomocí tlačítek SW1 a SW2 s funkcí stejnou jako v zadání 2. Při řešení úkolu nezapomeňte, že červená a zelená barevná složka RGB LED je připojena na kanály 0 a 1 časovače TPM2, viz tabulka 6. To znamená, že je zapotřebí provést jeho celkovou inicializaci včetně souvisejících modulů.
4. Vytvořte program pro ovládání jasu modré barevné složky RGB LED hardwarovou PWM modulací se zadáváním požadovaného jasu pomocí potenciometru. Při otáčení jeho hřídelí ve směru hodinových ručiček se bude jas zvyšovat až po maximální hodnotu, v opačném směru snižovat až k úplnému zhasnutí. Modul A/D převodníku nastavte na 8bitové rozlišení převodu s hardwarovým průměrováním 16 vzorků a jednorázové programové spouštění.

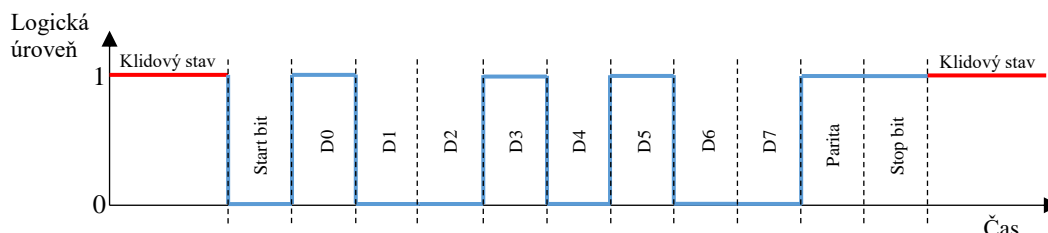


## 5 SÉRIOVÉ KOMUNIKAČNÍ ROZHRAŇÍ

Sériové komunikační rozhraní (SCI) slouží k efektivnímu přenosu informací mezi mikropočítačem a periferií za použití pouze dvou datových linek: jedna je určena pro vysílání dat (TX) a druhá pro příjem dat (RX). Obě zařízení musí mít propojenou společnou signálovou zem (GND), která představuje referenční úroveň signálu logické 0, viz obrázek 30. V případě velmi malých komunikačních vzdáleností v rámci desky plošného spoje není zapotřebí používat linkové budiče. Jádrem sériových komunikačních rozhraní je obvod nazvaný Univerzální asynchronní přijímač / vysílač (UART) zajišťující veškeré funkce související s vysíláním a příjmem dat, generováním a vyhodnocováním parity vysílaných znaků, generováním požadavků na přerušení při různých událostech na lince a vlastním rozhraní a další funkce dle vlastností konkrétního UART rozhraní. Vlastní komunikace je asynchronní, což znamená, že na sériové lince není přenášen synchronizační hodinový signál a obě zařízení musí proto mít nastavenou stejnou komunikační rychlost v Bd. Příjímač s vysílačem je synchronizován na začátku přenosu každého znaku sestupnou hranou start bitu představujícím nízkou úroveň na lince. Následuje odvysílání datových bitů reprezentujících přenášený znak (ve většině případů 8 bitů) v pořadí od nejméně významného bitu (LSB) po nejvíce významný bit (MSB), dále volitelně paritní bit (sudá nebo lichá parita). Přenos znaku je následně ukončen jedním nebo dvěma stop bity s vysokou logickou úrovní. Struktura přenosového rámce je vyobrazena na obrázku 30.



Obrázek 30: Propojení dvou zařízení pomocí SCI.



Obrázek 31: Struktura přenosového rámce SCI.



## 5.1 Použití UART rozhraní na vývojovém kitu

Mikropočítač MKL25Z128 má implementována 3 rozhraní pro asynchronní sériovou komunikaci: UART0, UART1 a UART2. Z nichž UART0 umožňuje navíc širší možnosti výběru zdroje hodinového signálu, díky čemuž dokáže pracovat i v úsporných režimech. Použití UART rozhraní na vývojovém kitu je následující:

### UART0

Rozhraní UART0 je připojeno k ladícímu rozhraní OpenSDA, kde umožňuje komunikaci s virtuálním sériovým rozhraním nainstalovaného ve vývojovém PC společně s ovladači OpenSDA. Lze jej použít s libovolným terminálovým programem pro výměnu dat mezi počítačem a mikropočítačem či jakýmkoliv jiným programovým vybavením umožňujícího práci s COM porty počítače.

### UART1

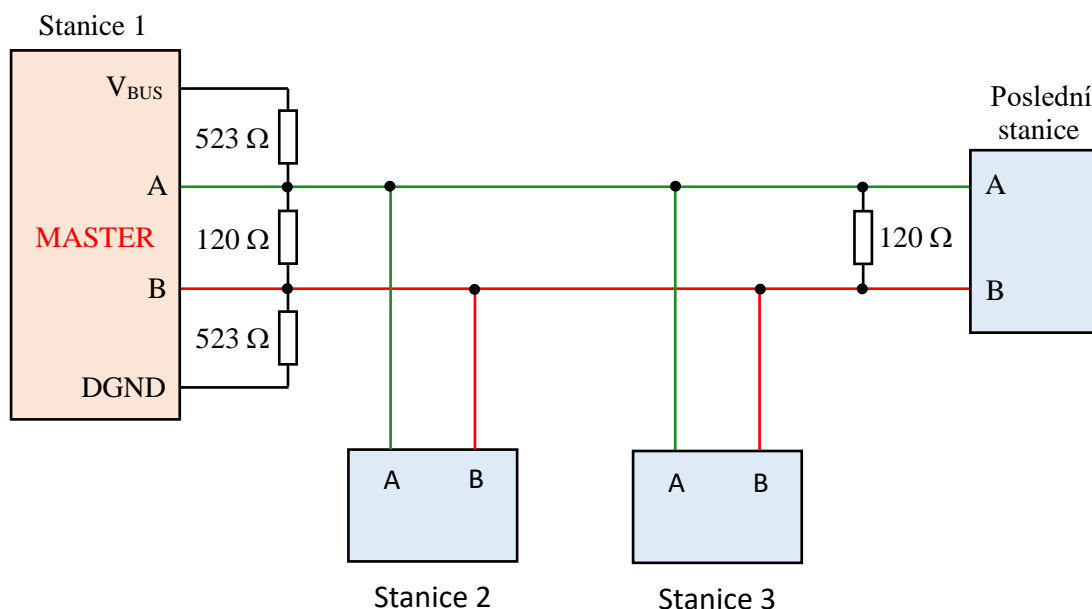
Zajišťuje funkci plně duplexního RS232 rozhraní, které je na kitu dostupné prostřednictvím standardního 9pinového konektoru CANON. Rozhraní používá linkový budič ADM3202 provádějící převod napěťových úrovní 3V logiky mikropočítače na úroveň RS232 a opačně. Standardně je maximální komunikační vzdálenost 10 m při rychlosti 9600 Bd. Při kratších vzdálenostech může dosahovat až 115200 Bd. Pomocí RS232 lze běžně propojit pouze dvě zařízení. Rozhraní na kitu nemá zapojeny signály pro řízení toku dat signály DTR, DSR, RTS, CTS a nelze tedy použít hardwarové řízení toku. Jejich propojení na kitu zajišťuje kompatibilitu s většinou běžných RS232 portů.

### UART2

Asynchronní sériové rozhraní UART2 je na kitu součástí RS485 rozhraní umožňujícího připojení až 32 zařízení na jednu komunikační sběrnici. Používá linkový budič MAX3485 s maximální přenosovou rychlostí 10 Mb/s. S rostoucí vzdáleností se rychlost musí odpovídajícím způsobem snižovat z důvodu zvyšující se kapacity vedení. Komunikace na této sběrnici je poloduplexní. Na sběrnici musí být přítomno zařízení master, které koordinuje probíhající komunikaci, aby nedocházelo ke kolizím. Ostatní zařízení jsou slave a mohou vysílat pouze na výzvu od master jednotky. Zařízení na koncích sběrnice musí být vybavena terminátory pro zamezení odrazů na lince. Režim vysílání / příjem se na RS485



budiči nastavuje signálem DE (Driver Enable), který je připojen na pin PTE29 mikropočítače. Na RS485 se velmi často používá MODBUS protokol (režimy RTU nebo ASCII) původně vyvinutý společností Modicon, nyní Schneider Electric. Propojení jednotlivých účastníků na RS485 je vyobrazeno na obrázku 32.



Obrázek 32: Připojení účastníků na RS485 sběrnici [6].

## 5.2 Programová obsluha UART modulů

Při programové obsluze komunikačních rozhraní bude použit místo nízkoúrovňového přístupu prostřednictvím periferních registrů přístup na vyšší úrovni pomocí periferních driverů, které jsou součástí MCUXpresso programového vývojového kitu (SDK) pro mikropočítače NXP. SDK obsahuje kromě periferních driverů podporu pro vícejádrové platformy, USB stack a operační systém reálného času FreeRTOS™ [7].

Periferní ovladač UART rozhraní obsahuje funkční a transakční API. První ze jmenovaných je určeno pro inicializaci, konfiguraci a obsluhu rozhraní. Každá z těchto funkcí má jako první z předávaných parametrů básovou adresu obsluhovaného modulu. Druhou možností je použití transakčního API, které je více komplexní a má tudíž i větší požadavky na obsazenou paměť programu. Druhým parametrem těchto funkcí je handle, které se získá funkcí `UART_TransferCreateHandle()`. Pro obsluhu UART bude použito funkční API.

### 5.2.1 Popis vybraných funkcí funkčního API

*status\_t* **UART\_Init** ( *UART\_Type* \*base, const *uart\_config\_t* \*config, *uint32\_t* srcClock\_Hz )

*Popis funkce:*

Funkce nakonfiguruje UART modul s bázovou adresou *base* na základě nastavení uloženého ve struktuře *config* a frekvenci zdroje hodinového signálu *srcClock\_Hz*. Strukturu lze naplnit výchozími hodnotami pomocí funkce *UART\_GetDefaultConfig()* [7].

*Parametry funkce:*

<i>base</i>	Bázová adresa UART modulu.
<i>config</i>	Ukazatel na uživatelsky definovanou strukturu s konfigurací.
<i>srcClock_Hz</i>	Frekvence zdroje hodinového signálu UART modulu v [Hz].

*Návratové hodnoty:*

<i>kStatus_UART_Baudrate-NotSupport</i>	Přenosová rychlost není podporována s aktuálním zdrojem hodinového signálu.
<i>kStatus_Success</i>	Inicializace UART proběhla v pořádku.

*void* **UART\_Deinit** ( *UART\_Type* \*base )

*Popis funkce:*

Funkce počká na dokončení operace odesílání dat, zakáže vysílání a příjem dat a vypne zdroj hodinového signálu pro UART [7].

*Parametry funkce:*

<i>base</i>	Bázová adresa UART modulu.
-------------	----------------------------

*void* **UART\_GetDefaultConfig** ( *uart\_config\_t* \*config )

*Popis funkce:*

Funkce provede inicializaci struktury s konfigurací výchozími hodnotami uvedenými v níže uvedeném seznamu jednotlivých položek [7].



*Parametry funkce:*

<i>config</i>	Ukazatel na strukturu pro uložení konfigurace.
---------------	--

Výchozí hodnoty naplněné funkcí do konfigurační struktury jsou následující:

- `baudRate_Bps = 115200U`
- `bitCountPerChar = kUART_8BitsPerChar`
- `parityMode = kUART_ParityDisabled`
- `stopBitCount = kUART_OneStopBit`
- `txFifoWatermark = 0`
- `rxFifoWatermark = 1`
- `enableTx = false`
- `enableRx = false`

*status\_t* **UART\_SetBaudRate** ( *UART\_Type* \*base, *uint32\_t* baudRate\_Bps, *uint32\_t* srcClock\_Hz )

*Popis funkce:*

Funkce nastaví požadovanou přenosovou rychlost u již inicializovaného UART rozhraní pomocí funkce *UART\_Init()* [7].

*Parametry funkce:*

<i>base</i>	Bázová adresa UART modulu.
<i>baudRate_Bps</i>	Přenosová rychlost pro nastavení.
<i>srcClock_Hz</i>	Hodinová frekvence UART rozhraní v [Hz].

*Návratové hodnoty:*

<i>kStatus_UART_Baudrate-NotSupport</i>	Přenosová rychlost není podporována s aktuálním zdrojem hodinového signálu.
<i>kStatus_Success</i>	Nastavení proběhlo v pořádku.



*uint32\_t* **UART\_GetStatusFlags** ( *UART\_Type* \*base )

*Popis funkce:*

Funkce vrací stavové příznaky specifikovaného UART rozhraní. Příznaky jsou vráceny jako logický součet enumerátorů *\_uart\_flags* [7].

*Parametry funkce:*

<i>base</i>	Bázová adresa UART modulu.
-------------	----------------------------

*static void* **UART\_EnableTx** ( *UART\_Type* \*base, *bool enable* ) [*inline*],[*static*]

*Popis funkce:*

Funkce povolí nebo zakáže vysílač specifikovaného UART rozhraní [7].

*Parametry funkce:*

<i>base</i>	Bázová adresa UART modulu.
<i>enable</i>	True hodnota pro povolení, False pro zakázání

*static void* **UART\_EnableRX** ( *UART\_Type* \*base, *bool enable* ) [*inline*],[*static*]

*Popis funkce:*

Funkce povolí nebo zakáže přijímač specifikovaného UART rozhraní [7].

*Parametry funkce:*

<i>base</i>	Bázová adresa UART modulu.
<i>enable</i>	True hodnota pro povolení, False pro zakázání

*static void* **UART\_WriteByte** ( *UART\_Type* \*base, *uint8\_t data* ) [*inline*],[*static*]

*Popis funkce:*

Funkce zapíše hodnotu přímo do TX registru specifikovaného rozhraní. Před voláním funkce musí být TX registr nebo TX FIFO prázdná [7].



*Parametry funkce:*

<i>base</i>	Bázová adresa UART modulu.
<i>data</i>	Bajt pro zápis do TX registru

```
static uint8_t UART_ReadByte ( UART_Type *base ) [inline], [static]
```

*Popis funkce:*

Funkce vrátí hodnotu přečtenou přímo z RX registru specifikovaného rozhraní. Před voláním funkce musí mít RX registr nebo RX FIFO připravená data [7].

*Parametry funkce:*

<i>base</i>	Bázová adresa UART modulu.
-------------	----------------------------

```
void UART_WriteBlocking ( UART_Type *base, const uint8_t *data, size_t length )
```

*Popis funkce:*

Funkce zapíše požadovaná data do UART rozhraní. Při zápisu do TX registru nebo TX vyrovnávací paměti čeká na uvolnění místa. Funkce netestuje fyzické odvysílání na sériovou linku. To je zapotřebí ověřit funkcí pro přečtení stavových příznaků rozhraní UART a otestovat *kUART\_TransmissionCompleteFlag* [7].

*Parametry funkce:*

<i>base</i>	Bázová adresa UART modulu.
<i>data</i>	Počáteční adresa dat pro zápis.
<i>length</i>	Velikost dat pro zápis.

```
status_t UART_ReadBlocking ( UART_Type *base, uint8_t *data, size_t length )
```

*Popis funkce:*

Funkce přečte data přijatá UART rozhraním. Pokud nejsou dostupná přijatá data v přijímacím registru nebo RX FIFO, čeká na jejich příjem [7].



*Parametry funkce:*

<i>base</i>	Bázová adresa UART modulu.
<i>data</i>	Počáteční adresa bufferu pro uložení přijatých dat.
<i>length</i>	Velikost bufferu.

*Návratové hodnoty:*

<i>kStatus_UART_Rx-HardwareOverrun</i>	Během příjmu dat nastala chyba přetečení přijímacího bufferu.
<i>kStatus_UART_Noise-Error</i>	Během příjmu dat nastala chyba z důvodu rušení/šumu na lince.
<i>kStatus_UART_FramingError</i>	Během příjmu dat nastala chyba rámce.
<i>kStatus_UART_ParityError</i>	Během příjmu dat nastala chyba parity.
<i>kStatus_Success</i>	Data přijata v pořádku.

### 5.2.2 Popis funkcí ladící konzole

MCUXpresso SDK v základním nastavení provádí inicializaci UART0 rozhraní na výchozí hodnoty (115200 Bd, 8 bitů data, 1 start a stop bit, bez parity) a poskytuje na něm funkce ladící konzole prostřednictvím standardního vstupu (stdin) a výstupu (stdout). Ve standardní konfiguraci používá vlastní SDK funkce. Na daném rozhraní lze taktéž používat i vybrané funkce z funkčního API pro UART moduly pro příjem a odeslání znaků či celého bufferu. Nelze používat funkce, které mění konfiguraci UART rozhraní.

*int PRINTF (const char \*fmt\_s, ...)*

*Popis funkce:*

Funkce provede formátovaný výstup dat na zařízení standardního výstupu.

*Parametry funkce:*

Formát prvního argumentu funkce je následující: %[flags][width][.precision][length]specifier. Podporovaná nastavení s detailním popisem jsou uvedena v [7].





*Návratová hodnota:*

Počet úspěšně zapsaných znaků na standardní výstup.

*int **SCANF** (char \*fmt\_ptr, ...)*

*Popis funkce:*

Funkce provede formátovaný vstup dat ze zařízení standardního vstupu.

*Parametry funkce:*

Formát prvního argumentu funkce je následující: %[\*][width][length]specifier. Podporovaná nastavení s detailním popisem jsou uvedena v [7].

*Návratová hodnota:*

Počet úspěšně přečtených znaků ze standardního vstupu.

*int **PUTCHAR** (int ch)*

*Popis funkce:*

Funkce provede výstup znaku na zařízení standardního výstupu [7].

*Parametry funkce:*

Znak určený pro zápis na standardní výstup.

*Návratová hodnota:*

Počet úspěšně zapsaných znaků na standardní výstup.

*int **GETCHAR** (void)*

*Popis funkce:*

Funkce přečte znak ze zařízení standardního vstupu [7].

*Návratová hodnota:*

Přečtený znak ze standardního vstupu.



## 5.3 Ukázkové programy

V této kapitole naleznete ukázkové řešené příklady související s programovou obsluhou UART modulu v C jazyce. V rámci laboratorního cvičení si funkci programů důkladně vyzkoušejte jejich odkrokováním v příslušném programovém prostředí.

### 5.3.1 Vstup/výstup znaků s použitím funkcí ladící konzole

#### Zadání

Vytvořte program, který ze standardního vstupu přečte dvě celočíselné hodnoty se znaménkem a provede jejich součet. Výsledek součtu bude odeslán na standardní výstup.

#### Řešení

Po vytvoření projektu s podporou SDK 2.x jsou do zdrojového textu programu automaticky vloženy funkce zajišťující inicializaci rozhraní UART0 jako standardní zařízení pro vstup a výstup ladící konzole. Funkce *BOARD\_InitPins()* provede povolení hodinového signálu pro PORTA modul, nastavení funkce pinů PTA1 na UART0\_RX a PTA2 na UART0\_TX a v SIM modulu nastavení zdroje signálů přijímače a vysílače. Nastavení multifunkčního hodinového generátoru provede funkce *BOARD\_BootClockRUN()* na hodnoty  $f_{\text{CORE}} = 48 \text{ MHz}$  a  $f_{\text{BUS}} = 24 \text{ MHz}$  (v nastavení projektu již nedefinujeme symbol *CLOCK\_SETUP*). Vlastní ladící konzola je inicializována funkcí *BOARD\_InitDebugConsole()*. Náš program budeme zapisovat až za tyto inicializační funkce. Program používá funkce pro standardní vstup *SCANF()* a výstup *PRINTF()*. Výpis zdrojového textu je uveden ve výpisu programu 5.1.

Program 5.1:

```
#include "board.h"
#include "pin_mux.h"
#include "clock_config.h"
#include "fsl_debug_console.h"

int main(void)
{
    int32_t c1, c2;                // Proměnné pro uložení zadaných hodnot
    int32_t soucet;                // Proměnná pro uložení výsledku
```



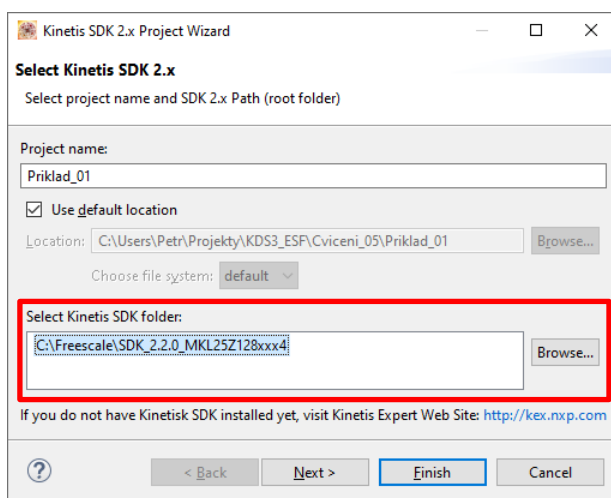
*Program 5.1 (pokračování):*

```
BOARD_InitPins();
BOARD_BootClockRUN();
BOARD_InitDebugConsole();

PRINTF("Vítejte v programu pro součet 2 čísel.\n");
PRINTF("Zadávejte celá čísla se znamenkem.\n");
while(1)
{
    PRINTF("Zadejte první číslo: ");
    SCANF("%d", &c1);
    PRINTF("Zadejte druhé číslo: ");
    SCANF("%d", &c2);
    soucet = c1 + c2;
    PRINTF("Součet zadanych čísel je: %d\n\n", soucet);
}
}
```

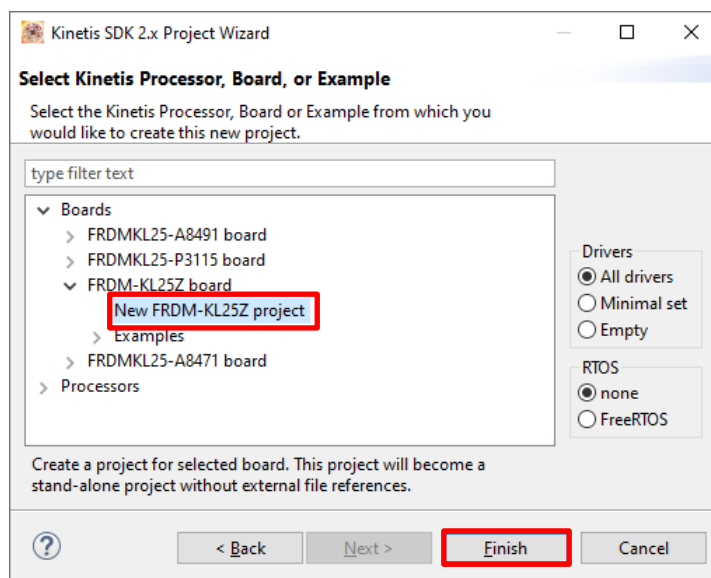
**Zprovoznění programu***a) Vytvoření projektu s podporou Kinetis SDK*

V menu KDS klikneme na „File“ a z nabídky vybereme příkaz „New / Kinetis SDK 2.x Project“. Průvodce novým projektem se nás nejprve dotáže na jméno nového projektu a umístění instalace Kinetis SDK, viz obrázek 33. Zadáme jméno projektu „Příklad\_01“ a



Obrázek 33: KDS - nový projekt s SDK krok 1.

provedeme kontrolu uvedené instalační složky Kinetis SDK. Pokud daná složka nesouhlasí, upravíme ji dle aktuálního stavu instalace kliknutím na tlačítko „Browse“. Po kliknutí na tlačítko „Next“ se zobrazí okno pro výběr mikropočítače, desky nebo ukázky. Z nabídky vybereme možnost „New FRDM-KL25Z project“ (obrázek 34), která nám zajistí programovou podporu naší vývojové desky. Průvodce dokončíme kliknutím na „Finish“.

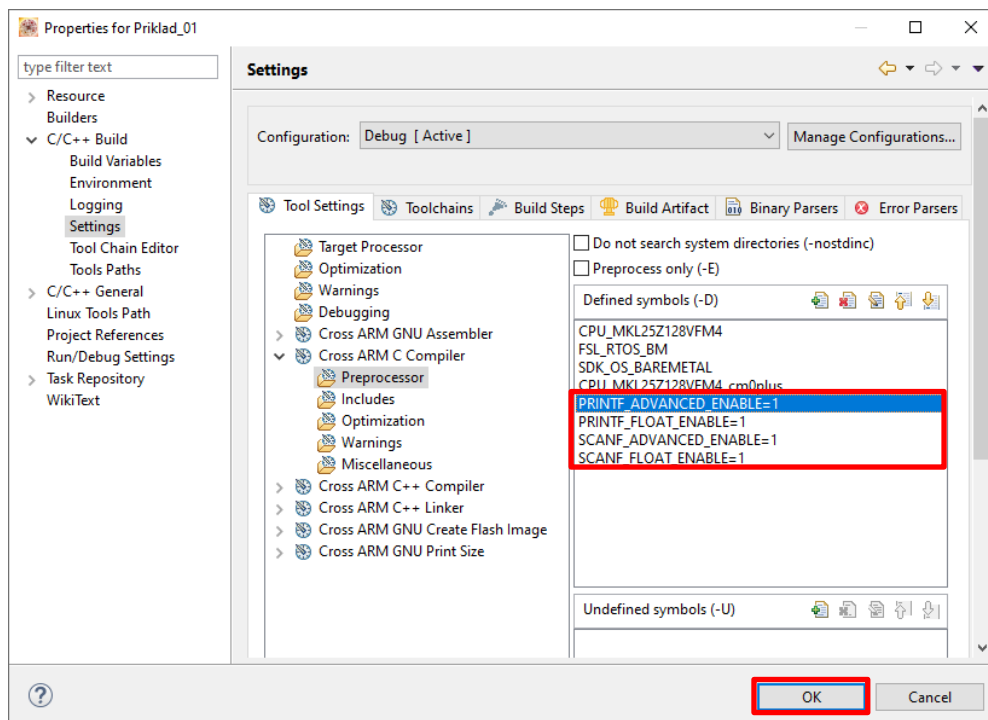


Obrázek 34: KDS - průvodce novým projektem s SDK krok 2.

Kliknutím pravým tlačítkem myši na projekt „Příklad\_01“ a výběrem položky „Properties“ upravíme vlastnosti projektu. V zobrazeném okně vlastností rozklikneme „C/C++ Build“ a vybereme položku „Settings“. V záložce „Tools Settings“ vybereme nastavení „Cross ARM C Compiler / Preprocessor“ a přidáme definici následujících symbolů:

- `PRINTF_ADVANCED_ENABLE=1`
- `PRINTF_FLOAT_ENABLE=1`
- `SCANF_ADVANCED_ENABLE=1`
- `SCANF_FLOAT_ENABLE=1`

Tímto jsou povoleny pokročilé funkce a podpora datového typu float u funkcí `PRINTF()` a `SCANF()` používaných v ukázkovém příkladu. Nastavení potvrdíme kliknutím na tlačítko „Apply“ a celé okno vlastností potvrdíme kliknutím na „OK“. Ukázka okna s příslušným nastavením je vyobrazena na obrázku 35.



Obrázek 35: KDS - definice symbolů preprocesoru překladače.

#### b) Nastavení programu Tera Term

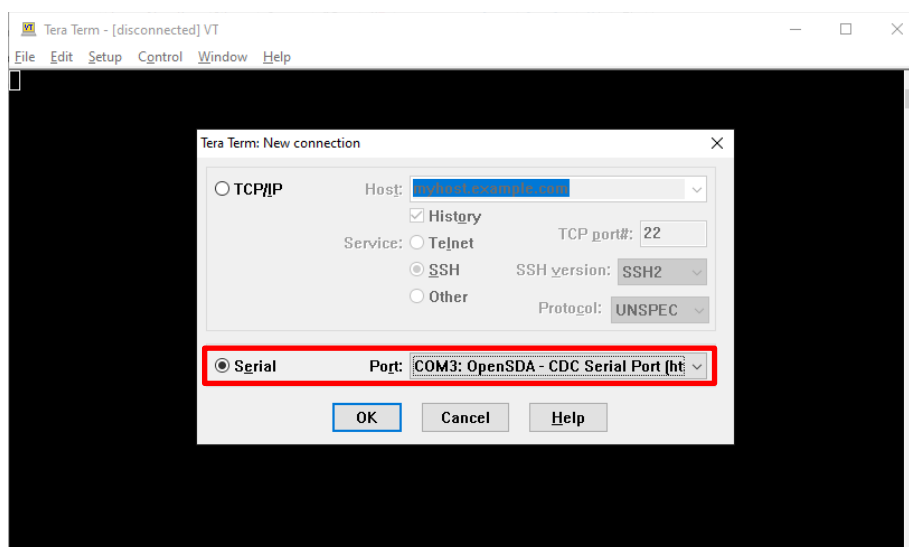
Program Tera Term je emulátor terminálu, který nám na počítači umožní komunikaci s mikropočítačem prostřednictvím sériového komunikačního rozhraní COM. Pro zajištění bezproblémové komunikace je zapotřebí nastavit komunikační port na obou zařízeních stejně. Výchozí nastavení UART rozhraní na mikropočítači je uvedeno v tabulce 12.

Parametr	Hodnota
Přenosová rychlost	115200 Bd
Počet datových bitů znaku	8
Zabezpečení paritou	Žádné
Počet stop bitů	1
Řízení datového toku	Žádné

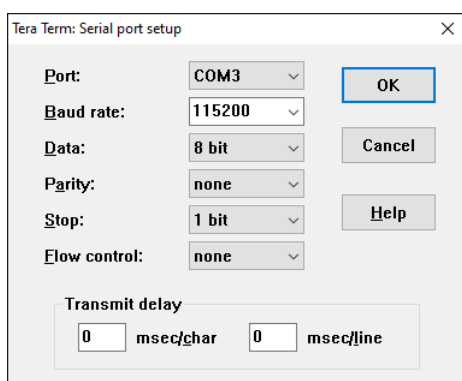
Tabulka 12: Výchozí nastavení sériového komunikačního rozhraní.

Stejně musí být nastaven komunikační port i na straně počítače v emulátoru terminálu. Po spuštění programu Tera Term klikneme v menu na „File“ a z nabídky vybereme „Disconnect“ pro případ, že je již po startu programu komunikační port otevřen. Nyní znovu otevřeme stejnou nabídku a zvolíme „New connection“, čímž otevřeme dialog pro výběr

komunikačního portu, viz obrázek 36. Z nabídky dostupných sériových portů vybereme ten, který má následující popis: „OpenSDA – CDC Serial Port“. Číslo sériového portu bude s velkou pravděpodobností jiné, než je uvedeno na obrázku. To nemá na funkci programu žádný vliv. Výběr potvrdíme tlačítkem OK a následně v menu programu klikneme na „Setup“ a z nabídky zvolíme položku „Serial port“. V zobrazeném konfiguračním okně (obrázek 37 vlevo) nastavíme parametry komunikačního portu dle tabulky 12 a potvrdíme je kliknutím na „OK“. Dále ještě upřesníme nastavení terminálu tak, aby v okně terminálu vypisoval odesílané znaky, tzv. lokální echo. To se provede opět v menu „Setup“ a položce „Terminal“, kde zatrhneme volbu „Local echo“, viz obrázek 37 vpravo. Tímto je nastavení programu dokončeno a je připraven pro komunikace s mikropočítačem.



Obrázek 36: Tera Term - nastavení nového připojení.



Obrázek 37: Tera Term - nastavení parametrů COM a terminálu.

### 5.3.2 Vstup/výstup znaků s použitím funkčního API

#### Zadání

Vytvořte program, který bude z RS232 sériového komunikačního rozhraní číst přijaté znaky a pokud se bude jednat o malé písmeno a-z, provede konverzi na velké a odešle zpět na sériovou linku. Ostatní znaky bude program posílat zpět beze změny.

#### Řešení

Na výukovém kitu je RS232 rozhraní připojeno k UART1 modulu. V inicializační části programu je nejprve povolen hodinový signál pro UART1 zápisem 1 do 11. bitu registru SIM\_SCGC4 a pro PORTC modul zápisem 1 do 11. bitu registru SIM\_SCGC5. Na portu C se dále pro pin PTC3 nastaví funkce ALT3 odpovídající UART1\_RX a na pinu PTC4 funkce UART1\_TX zápisem hodnoty 3 do bitového pole *MUX* registrů PORTC\_PCR3 a PORTC\_PCR4. Po těchto krocích je již možné přikročit k inicializaci samotného UART1 rozhraní. Nejdříve se pomocí funkce *UART\_GetDefaultConfig()* naplní konfigurační struktura *uart1\_cfg* výchozími hodnotami pro nastavení UART1 rozhraní. Kromě této struktury vyžaduje inicializační funkce UART frekvenci vstupního hodinového signálu. Ta je zjištěna funkcí *CLOCK\_GetBusClkFreq()* a uložena do proměnné *bclk\_freq*. Nyní je vše připraveno a může se zavolat funkce *UART\_Init()*, kde prvním parametrem je básová adresa UART1, druhým ukazatel na vyplněnou konfigurační strukturu a třetím frekvence hodinového signálu v Hz. Následně se pomocí funkcí *UART\_EnableRx()* a *UART\_EnableTx()* povolí přijímač a vysílač UART1 rozhraní. V hlavní programové smyčce se nejprve čeká na příjem znaku čtením stavu rozhraní UART1 funkcí *UART\_GetStatusFlags()* a testováním příznaku *kUART\_RxDataRegFullFlag*. V případě, že je výsledek logického součinu nenulový, je v rozhraní dostupný přijatý znak. Ten se následně přečte pomocí *UART\_ReadByte()* a uloží se do proměnné *znak*. Znak se zpracuje pomocí funkce *toupper()*, která převede malá písmena a-z na velká. Předtím než se zpracovaný znak může odeslat, musí se ověřit, zda je vysílač rozhraní UART1 volný. To se provede obdobně jako v předchozím případě, jen se testuje jiný příznakový bit, konkrétně *kUART\_TxDataRegEmptyFlag*. Poté se již provede zápis znaku do UART1 rozhraní pomocí funkce *UART\_WriteByte()*. Zdrojový text programu je uveden ve výpisu programu 5.2. Pozn.: Pro ověření funkce programu je nutné propojit výukový kit s počítačem pomocí RS232 kabelu.





Program 5.2:

```
#include "board.h"
#include "pin_mux.h"
#include "clock_config.h"
#include "fsl_uart.h"
#include "fsl_clock.h"
#include "ctype.h"

int main(void) {
    uart_config_t uart1_cfg;
    status_t status;
    uint32_t bclk_freq;
    uint8_t znak;
    // Inicializace KL25Z desky, vytvořeno automaticky SDK
    BOARD_InitPins();
    BOARD_BootClockRUN();
    BOARD_InitDebugConsole();
    // Inicializace SIM, PORTC
    SIM->SCGC4 |= SIM_SCGC4_UART1_MASK; // Povolení hodinového signálu UART1
    SIM->SCGC5 |= SIM_SCGC5_PORTC_MASK; // Povolení hodinového signálu PORTC
    PORTC->PCR[3] = 0; PORTC->PCR[4] = 0;
    PORTC->PCR[3] |= PORT_PCR_MUX(3); // Funkce pinu PTC3 na ALT3 = UART1_RX
    PORTC->PCR[4] |= PORT_PCR_MUX(3); // Funkce pinu PTC4 na ALT3 = UART1_TX
    // Inicializace UART1
    UART_GetDefaultConfig (&uart1_cfg); // Naplnění výchozí konfigurace do uart1_cfg
    bclk_freq = CLOCK_GetBusClkFreq(); // Zjištění frekvence sběrnice
    status = UART_Init(UART1, &uart1_cfg, bclk_freq);
    UART_EnableRx(UART1, true); // Povolení přijímače UART1
    UART_EnableTx(UART1, true); // Povolení vysílače UART1
    while(1)
    {
        // Čekání na příjem znaku z UART1
        while((UART_GetStatusFlags(UART1) & kUART_RxDataRegFullFlag) == 0);
        znak = UART_ReadByte(UART1); // Přechtení přijatého znaku
        znak = toupper(znak); // Zpracování znaku funkcí toupper()
        // Čekání na volný vysílač UART1
        while((UART_GetStatusFlags(UART1) & kUART_TxDataRegEmptyFlag) == 0);
        UART_WriteByte(UART1, znak); // Odeslání zpracovaného znaku na UART1
    }
}
```



## 5.4 Zadání samostatné práce

1. Upravte ukázkový program 4.1 tak, aby se perioda blikání LED nastavovala v rozmezí od 100 ms do 1000 ms prostřednictvím ladící konzole. Vstup od uživatele ošetřete tak, aby nebylo možné zadat neplatné hodnoty.
2. Upravte ukázkový program 4.2 pro možnost ovládání jasu modré barevné složky RGB LED pomocí ladící konzole zadáváním hodnot od 0 do 100 % jasu. Ošetřete uživatelský vstup tak, aby nebylo možné zadat neplatné hodnoty. Při každé změně jasu odešlete do konzole informaci o aktuální hodnotě nastavení modulo registru a datového registru kanálu.
3. Rozšiřte funkci programu z bodu 2 zadání o možnost výběru ovládaného výstupního kanálu pomocí uživatelského vstupu z konzole. Program uživateli nabídne na konzoli přehledné menu s popisem ovládání programu.

Například:

Vyberte kanál pro nastavení jasu:

1 – Červený barevný kanál RGB LED

2 – Zelený barevný kanál RGB LED

3 – Modrý barevný kanál RGB LED

Vaše volba:

Po zvolení funkce zadáním číselné hodnoty 1 – 3 se program dále dotáže na požadovanou hodnotu jasu daného kanálu od 0 do 100 %.

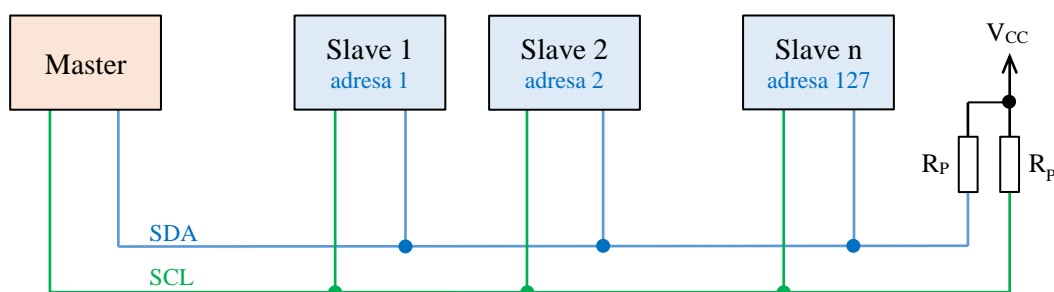
4. Upravte program 3.1 tak, aby do konzole periodicky vypisoval převedenou hodnotu přímo z A/D převodníku a hodnotu přepočítanou na napětí ve voltech. Při výpočtu předpokládejte, že velikost referenčního napětí  $V_{REFH} = 3 \text{ V}$ . Výpisy provádějte s periodou 200 ms s použitím časovače. Nemusí být použito přerušení, postačí režim dotazování na stav příznakového bitu TOF ve stavovém registru.



## 6 SÉRIOVÁ SBĚRNICE I<sup>2</sup>C

Sériová sběrnice I<sup>2</sup>C byla vyvinuta společností Philips Semiconductors pro zajištění komunikace mezi jednotlivými integrovanými obvody v aplikacích spotřební, průmyslové elektroniky a telekomunikačních zařízeních. Cílem bylo zefektivnění a zjednodušení jejich konstrukce díky zavedení jednotné komunikační sběrnice. Sběrnice I<sup>2</sup>C nalézá své uplatnění v široké škále periferních obvodů, jako jsou například paměťové obvody RAM, EEPROM a FLASH, snímače fyzikálních veličin (teplota, vlhkost, tlak, zrychlení), LCD displeje, A/D a D/A převodníky, expandéry vstupů a výstupů, které lze použít zcela univerzálně v konstrukcích zařízení, jejichž základem je většinou jednočipový mikropočítač či minipočítač. Další oblastí použití jsou aplikačně specifické obvody zahrnující digitální tunery pro televize a FM rádia, obvody zpracování zvuku, obrazu a další [8].

Komunikace je zajištěna obousměrnou 2vodičovou sběrníci přenášející sériová data (linka SDA) a hodinový signál (linka SCL), klidovou napěťovou úroveň logické 1 zajišťují pull-up rezistory. Komunikace probíhá na principu master-slave. Každý účastník na sběrnici má vlastní unikátní adresu o velikosti 7 bitů v základní verzi nebo 10 bitů v rozšířené verzi, pomocí které je oslovován master zařízením na začátku každého datového přenosu. V základní verzi je k dispozici celkem  $2^7 = 128$  adres, v rozšířené je to  $2^{10} = 1024$  různých adres. Na jedné sběrnici může být připojeno i více master jednotek, což klade důraz na detekci kolizí a předávání řízení sběrnice v případě, že více masterů přistupuje na sběrnici současně. Komunikační rychlost je ve standardním režimu 100 kb/s, rychlém režimu je to 400 kb/s a ve vysokorychlostním režimu dosahuje 3,4 Mb/s. Počet připojených účastníků na sběrnici je omezen maximální kapacitou sběrnice, která je stanovena na 400 pF [8]. Způsob připojení zařízení na sběrnici ilustruje obrázek 38.

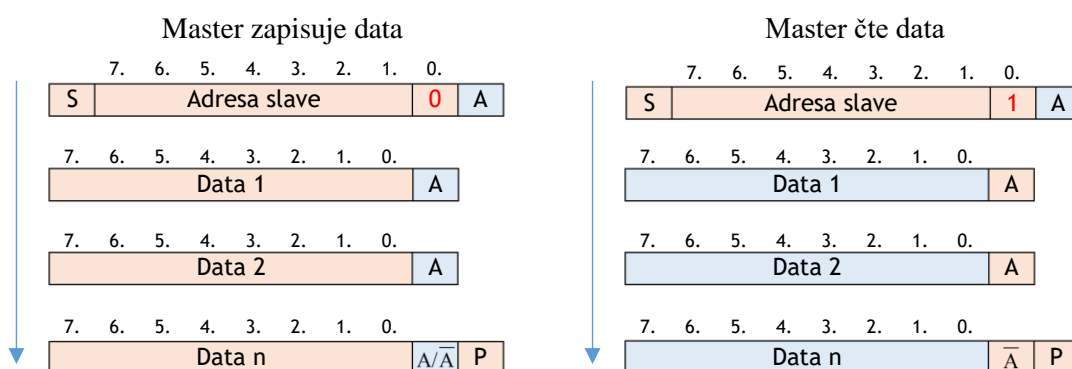


Obrázek 38: Připojení účastníků na I<sup>2</sup>C sběrnici.

## 6.1 Komunikace na I<sup>2</sup>C

Komunikaci na I<sup>2</sup>C sběrnici zahajuje vždy master jednotka v okamžiku, kdy je sběrnice volná. Průběh přenosu lze shrnout do několika základních kroků (obrázek 39):

1. Začátek datového přenosu je indikován START signálem reprezentovaný sestupnou hranou na SDA lince, zatímco na SCL je vysoká logická úroveň. Od tohoto okamžiku je sběrnice v zaneprázdněném stavu (busy) až do okamžiku vyslání signálu STOP, kdy na SDA je vzestupná hrana a SCL je v logické 1.
2. Po START signálu je odvíšována 7bitová adresa slave zařízení a R/W bit rozlišující směr komunikace v pořadí od nejvíce významného bitu (MSB) po nejméně významný bit (LSB). Pokud je v R/W bitu uložena 0, jedná se o operaci zápisu, 1 představuje operaci čtení dat.
3. Adresované slave zařízení v následujícím hodinovém cyklu potvrdí příjem potvrzovacím bitem ACK. Hodnota 0 v ACK indikuje úspěch operace, 1 její neúspěch. V případě neúspěchu master může buď zrušit přenos vygenerováním STOP signálu, anebo vyšle opakovaný START pro zahájení nového přenosu.
4. Master zapíše nebo přečte požadovaný počet bajtů dat. V případě zápisu slave po každém přijatém bajtu potvrzuje příjem ACK bitem. Při čtení dat potvrzuje přijatá data master. Operaci čtení přeruší master tak, že nepotvrdí poslední přijatý bajt od slave. Zařízení slave musí uvolnit sběrnici a umožnit tak masteru vygenerování STOP signálu nebo opakovaného START signálu.
5. Přenos dat ukončí master STOP signálem. Sběrnice tímto přejde do klidového stavu a je připravena pro další datový přenos [8].



Obrázek 39: Komunikace na I<sup>2</sup>C sběrnici.

## 6.2 Implementace I<sup>2</sup>C na výukovém kitu

Mikropočítač MKL25Z128 je vybaven dvěma moduly I<sup>2</sup>C pro zajištění komunikace s širokou škálou dostupných externích periférií. Rozhraní je navrženo tak, aby mohlo pracovat na přenosové hodinové frekvenci 100 kHz při plném zatížení sběrnice. Nicméně při nižší kapacitní zátěži sběrnice (připojeno méně účastníků, kratší komunikační vzdálenosti), může pracovat i při vyšších přenosových rychlostech. Frekvence hodinového signálu je programově nastavitelná volbou z 64 různých dostupných frekvencí. Podporována je standardní 7bitová i rozšířená 10bitová adresace slave zařízení. Pro dosažení vysoké spolehlivosti přenosu je rozhraní vybaveno nastavitelným vstupním filtrem pro potlačení šumu.

Vývojová deska KL25Z má vyhrazen modul I2C0 pro komunikaci s tříosým akcelerometrem MMA8451Q. Výukový kit používá komunikační rozhraní I2C1. Slouží pro připojení celkem tří externích periférií: teplotního snímače LM75A, vlhkoměru HIH6130 a hodin reálného času PCF8583. Přehled všech periférií kitu vybavených I<sup>2</sup>C komunikačním rozhraním a jejich slave adres je uveden v tabulce 13.

I2C rozhraní	Periferie	Adresa slave
I2C0	Akcelerometr MMA8451Q	0x1d
I2C1	Snímač teploty LM75A	0x48
	Snímač vlhkosti HIH6130	0x27
	Hodiny reálného času PCF8583	0x50

Tabulka 13: Přehled periférií kitu s I<sup>2</sup>C rozhraním.

### 6.2.1 Snímač teploty LM75A

Snímač teploty LM75A je vybaven na čipu band gap teplotním senzorem s rozsahem -55 °C až +125 °C a 11bitovým Sigma-delta analogově-digitálním převodníkem umožňujícím měření teploty s rozlišením na 0,125 °C. V teplotním rozsahu -25 °C až +100 °C je přesnost měření ±2 °C, v plném rozsahu měření je to ±3 °C. Výstup OS obvodu může pracovat ve dvou režimech: komparátoru nebo přerušení [9].

Obvod je vybaven 4 interními registry, které jsou zpřístupněny prostřednictvím Pointer registru, jehož struktura je na obrázku 40.



## Pointer registr

B7	B6	B5	B4	B3	B2	B1	B0
0	0	0	0	0	0	Pointer	

Obrázek 40: LM75A - struktura Pointer registru [9].

Spodní 2 bity B1 a B0 zajišťují výběr interního registru:

- 00 – Teplotní registr (Temp)
- 01 – Konfigurační registr (Conf)
- 10 – Hysterezní registr (Thyst)
- 11 – Registr pro nastavení vypnutí při přehřátí (Tos)

## Teplotní registr (Temp)

Temp_HB								Temp_LB							
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	X	X	X	X	X

Obrázek 41: LM75A - struktura Temp registru [9].

V teplotním registru, viz obrázek 41, je uložena aktuální hodnota naměřené teploty ve formátu dvojkového doplňku s rozlišením 0,125 °C. První se na sběrnici při čtení jeho obsahu vysílá vyšší bajt (HB), potom nižší bajt (LB). Pokud bit D10 obsahuje hodnotu 0, naměřená teplota je větší nebo rovna nule. Pro přepočtení na teplotu ve stupních Celsia se použije výpočetní vztah (4). Při D10 = 1 je teplota záporná a použije se výpočet pomocí rovnice (5).

$$TempC = Temp\_HB + \frac{Temp\_LB}{256} \quad (4)$$

$$TempC = Temp\_HB + \frac{Temp\_LB}{256} - 256 \quad (5)$$

Význam ostatních registrů a jejich podrobný popis včetně nastavení lze nalézt v [9]. Pro základní funkci měření teploty je není nutné nastavovat. Do konfiguračního registru postačí zapsat výchozí hodnotu 0.



### 6.2.2 Snímač vlhkosti HIH6130

Snímač vlhkosti HIH6130 používá laserově trimovaný polymerový kapacitní snímač s více vrstvami, který je odolný vůči kondenzaci vody, prachu, špíně, olejům a běžným chemikáliím vyskytujících se v prostředí. Integrovaný 14bitový A/D převodník umožňuje měření relativní vlhkosti v rozsahu od 10 do 90 % RH s rozlišením 0,04 %RH a teploty od -25 °C do +85 °C s rozlišením 0,025 °C. Dosahovaná přesnost měření je  $\pm 4$  %RH a  $\pm 0,5$  °C. Snímač se vyznačuje velmi nízkou spotřebou v režimu spánku (1  $\mu$ A), do kterého se automaticky přepíná, pokud aktuálně neprovádí měření [10].

Snímač nemá implementovány uživatelsky přístupné konfigurační registry. Dostupný pro čtení je společný datový registr, který uchovává celkem 4 bajty dat. První dva bajty obsahují dva stavové bity a 14 bitů s naměřenou vlhkostí, zbývající dva bajty 14 bitů s naměřenou teplotou, viz obrázek 42. Význam stavových bitů S1 a S0 je uveden v tabulce 14.

HB								LB							
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
S1	S0	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0

HB								LB							
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
T13	T12	T11	T10	T9	T8	T7	T6	T5	T4	T3	T2	T1	T0	X	X

Obrázek 42: HIH6130 - datový registr [11].

Stavové bity		Význam
S1	S0	
0	0	Data byla přečtena poprvé po provedení měření – jsou aktuální.
0	1	Data již byla přečtena dříve. Pro aktuální data je nutné počkat na dokončení měření nebo spustit nové měření.
1	0	Snímač je v příkazovém režimu. Tento režim není běžně dostupný, slouží pro programování při výrobě.
1	1	Nepoužito

Tabulka 14: HIH6130 - význam stavových bitů [11].



Naměřená hodnota vlhkosti v %RH a teploty ve stupních Celsia se vypočítá z obsahu datového registru pomocí výpočetních vztahů (6) a (7) [11].

$$Humidity = \frac{Humidity\_output[B13:B0]}{(2^{14} - 2)} \cdot 100 \quad (6)$$

$$TemperatureC = \frac{Temperature\_output[T13:T0]}{(2^{14} - 2)} \cdot 165 - 40 \quad (7)$$

Z důvodu úspory energie je snímač standardně ve stavu režimu spánku s proudovou spotřebou 1  $\mu$ A, při kterém neprobíhá měření. Proto je zapotřebí snímači poslat před čtením výsledků měření příkaz pro spuštění měření. Měření se spustí tak, že se na I<sup>2</sup>C sběrnici odešle START signál, 7bitová adresa snímače a  $R/\overline{W}$  bit nastavený na 0 (zápis). Snímač odpoví ACK a master následně vygeneruje na sběrnici STOP signál. Zda je již v datovém registru uložena aktuální změřená hodnota lze zjistit ze stavových bitů S1 a S0, viz tabulka 14.

### 6.2.3 Hodiny reálného času PCF8583

Integrovaný obvod PCF8583 obsahuje hodiny reálného času s kalendářem, jejichž registry jsou mapovány do statické paměti RAM o kapacitě 256 B. Z celkové velikosti paměti RAM je k dispozici 240 B pro ukládání libovolných dat. Hodiny mohou být řízeny buď krystalovým oscilátorem s frekvencí 32768 Hz, nebo síťovou frekvencí 50 Hz. Obvod také poskytuje funkce budíku, časovače a přerušení. Uložená data jsou uchována při minimálním napájecím napětí 1 V [12].

Funkce obvodu a základní konfigurace se nastavuje v řídicím a stavovém registru na adrese 0. Pro funkci hodin řízených krystalem s frekvencí 32768 Hz s vypnutým budíkem se zde zapíše hodnota 0 a další konfigurace již není zapotřebí. Přehled všech dostupných registrů v režimu hodin je uveden v tabulce 15.

Adresa registru	Funkce registru							
0	<i>Řídicí a stavový registr</i>							
	D7	D6	D5	D4	D3	D2	D1	D0
	Konfigurační a stavové bity, viz [12]							
1	<i>Setiny sekundy</i>							
	D7	D6	D5	D4	D3	D2	D1	D0
	1/10 s				1/100 s			

2	<i>Sekundy</i>							
	D7	D6	D5	D4	D3	D2	D1	D0
	10 s				1 s			
3	<i>Minuty</i>							
	D7	D6	D5	D4	D3	D2	D1	D0
	10 min				1 min			
4	<i>Hodiny</i>							
	D7	D6	D5	D4	D3	D2	D1	D0
	1/10 s				1/100 s			
5	<i>Rok / den</i>							
	D7	D6	D5	D4	D3	D2	D1	D0
	10 dní				1 den			
6	<i>Den v týdnu / měsíc</i>							
	D7	D6	D5	D4	D3	D2	D1	D0
	10 měsíců				1 měsíc			
7	<i>Časovač</i>							
	D7	D6	D5	D4	D3	D2	D1	D0
	10 dní				1 den			
8 – 15	<i>Registry budíku</i>							
	Podrobný popis registrů budíku viz [12].							
16 – 255	<i>Volná paměť RAM</i>							
	Zde lze zapisovat libovolná uživatelská data.							

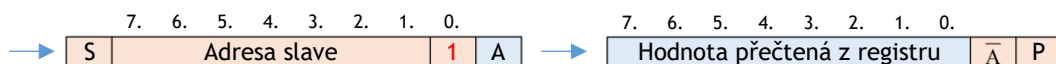
Tabulka 15: PCF8583 - přehled vybraných registrů [12].

Ukázka čtení registru je na obrázku 45. Při čtení nebo zápisu dat do registrů se automaticky zvyšuje adresa cílového registru. Nemusí se tedy na sběrnici pokaždé odesílat adresa registru po každém čtení nebo zápisu jednoho bajtu dat, čímž se významně zvýší rychlost přenosu dat při sekvenčním přístupu. Registry obsahující numerické hodnoty, jako například hodiny, minuty a sekundy, jsou uloženy v BCD formátu. To znamená, že ve spodních 4 bitech jsou uloženy jednotky v rozsahu 0 až 9 a v horních 4 bitech desítky v rozsahu 0 až 9.

Master zapisuje adresu registru, ze které bude následně číst.



Master čte hodnotu uloženou v registru.



Obrázek 43: PCF8583 - čtení obsahu registru.

### 6.3 Programová obsluha I<sup>2</sup>C modulů

Obdobně jako u modulů UART bude při programové obsluze komunikačních rozhraní I<sup>2</sup>C použit přístup na vyšší úrovni pomocí periferních driverů, které jsou součástí MCUXpresso programového vývojového kitu (SDK) pro mikropočítače NXP. Periferní driver I<sup>2</sup>C rozhraní obsahuje funkční a transakční API. Pro jeho obsluhu bude použito funkční API.

#### 6.3.1 Popis vybraných funkcí funkčního API

```
void I2C_MasterInit ( I2C_Type *base, const i2c_master_config_t *masterConfig, uint32_t  
srcClock_Hz )
```

*Popis funkce:*

Funkce nakonfiguruje I<sup>2</sup>C modul do režimu master s bázovou adresou *base* na základě nastavení uloženého ve struktuře *masterConfig* a frekvence zdroje hodinového signálu modulu *srcClock\_Hz*. Strukturu lze naplnit výchozími hodnotami pomocí funkce *I2C\_MasterGetDefaultConfig()* [7].

*Parametry funkce:*

<i>base</i>	Bázová adresa I <sup>2</sup> C modulu.
<i>masterConfig</i>	Ukazatel na uživatelsky definovanou strukturu s konfigurací master.
<i>srcClock_Hz</i>	Frekvence zdroje hodinového signálu I <sup>2</sup> C modulu v [Hz].

```
void I2C_SlaveInit ( I2C_Type *base, const i2c_slave_config_t *slaveConfig, uint32_t  
srcClock_Hz )
```

*Popis funkce:*

Funkce nakonfiguruje I<sup>2</sup>C modul do režimu slave s bázovou adresou *base* na základě nastavení uloženého ve struktuře *slaveConfig* a frekvenci zdroje hodinového signálu *srcClock\_Hz*. Strukturu lze naplnit výchozími hodnotami pomocí funkce *I2C\_SlaveGetDefaultConfig()* [7].

*Parametry funkce:*

<i>base</i>	Bázová adresa I <sup>2</sup> C modulu.
-------------	--



<i>slaveConfig</i>	Ukazatel na uživatelsky definovanou strukturu s konfigurací slave.
<i>srcClock_Hz</i>	Frekvence zdroje hodinového signálu I <sup>2</sup> C modulu v [Hz].

***void I2C\_MasterDeinit ( I2C\_Type \*base )***

*Popis funkce:*

Funkce vypne zdroj hodinového signálu pro I<sup>2</sup>C modul v režimu master [7].

*Parametry funkce:*

<i>base</i>	Bázová adresa I <sup>2</sup> C modulu.
-------------	--

***void I2C\_SlaveDeinit ( I2C\_Type \*base )***

*Popis funkce:*

Funkce vypne zdroj hodinového signálu pro I<sup>2</sup>C modul v režimu slave [7].

*Parametry funkce:*

<i>base</i>	Bázová adresa I <sup>2</sup> C modulu.
-------------	--

***void I2C\_MasterGetDefaultConfig ( i2c\_master\_config\_t \*masterConfig )***

*Popis funkce:*

Funkce provede inicializaci struktury s konfigurací pro master výchozími hodnotami uvedenými v níže uvedeném seznamu jednotlivých položek [7].

*Parametry funkce:*

<i>masterConfig</i>	Ukazatel na strukturu pro uložení konfigurace.
---------------------	--

Výchozí hodnoty naplněné funkcí do konfigurační struktury jsou následující:

- baudRate\_Bps = 100000
- enableStopHold = false
- glitchFilterWidth = 0
- enableMaster = true



```
void I2C_SlaveGetDefaultConfig ( i2c_slave_config_t *slaveConfig )
```

*Popis funkce:*

Funkce provede inicializaci struktury s konfigurací pro slave výchozími hodnotami uvedenými v níže uvedeném seznamu jednotlivých položek [7].

*Parametry funkce:*

<i>slaveConfig</i>	Ukazatel na strukturu pro uložení konfigurace.
--------------------	--

Výchozí hodnoty naplněné funkcí do konfigurační struktury jsou následující:

- addressingMode = kI2C\_Address7bit
- enableGeneralCall = false
- enableWakeUp = false
- enableBaudRateCtl = false
- sclStopHoldTime\_ns = 4000
- enableSlave = true

```
status_t I2C_MasterStart ( I2C_Type *base, uint8_t address, i2c_direction_t direction )
```

*Popis funkce:*

Funkce zahájí nový přenos dat vysláním START signálu, adresy slave zařízení a směru přenosu dat na specifikovaném I<sup>2</sup>C master rozhraní [7].

*Parametry funkce:*

<i>base</i>	Bázová adresa I <sup>2</sup> C modulu.
<i>address</i>	7bitová adresa slave zařízení.
<i>direction</i>	Směr přenosu dat: <i>kI2C_Write</i> nebo <i>kI2C_Read</i>

*Návratové hodnoty:*

<i>kStatus_Success</i>	START signál úspěšně odeslán na I <sup>2</sup> C.
<i>kStatus_I2C_Busy</i>	Sběrnice je zaneprázdněna.



*status\_t I2C\_MasterRepeatedStart* ( *I2C\_Type \*base*, *uint8\_t address*, *i2c\_direction\_t direction* )

*Popis funkce:*

Funkce odešle opakovaný START signál, adresu slave zařízení a směr přenosu dat na specifikované I<sup>2</sup>C master rozhraní [7].

*Parametry funkce:*

<i>base</i>	Bázová adresa I <sup>2</sup> C modulu.
<i>address</i>	7bitová adresa slave zařízení.
<i>direction</i>	Směr přenosu dat: <i>kI2C_Write</i> nebo <i>kI2C_Read</i>

*Návratové hodnoty:*

<i>kStatus_Success</i>	START signál úspěšně odeslán na I <sup>2</sup> C.
<i>kStatus_I2C_Busy</i>	Sběrnice je zaneprázdněna, ale není obsazena aktuálním masterem I <sup>2</sup> C.

*status\_t I2C\_MasterStop* ( *I2C\_Type \*base* )

*Popis funkce:*

Funkce ukončí přenos dat vysláním STOP signálu na master rozhraní [7].

*Parametry funkce:*

<i>base</i>	Bázová adresa I <sup>2</sup> C modulu.
-------------	--

*Návratové hodnoty:*

<i>kStatus_Success</i>	STOP signál úspěšně odeslán na I <sup>2</sup> C.
<i>kStatus_I2C_Busy</i>	Odeslání STOP signálu selhalo, nastal timeout.

*uint32\_t I2C\_MasterGetStatusFlags* ( *I2C\_Type \*base* )

*Popis funkce:*

Funkce vrací stavové příznaky specifikovaného I<sup>2</sup>C master rozhraní. Příznaky jsou vráceny jako logický součet enumerátorů *\_i2c\_flags* [7].



*Parametry funkce:*

<i>base</i>	Bázová adresa I <sup>2</sup> C modulu.
-------------	--

*uint32\_t I2C\_SlaveGetStatusFlags ( I2C\_Type \*base )*

*Popis funkce:*

Funkce vrací stavové příznaky specifikovaného I<sup>2</sup>C slave rozhraní. Příznaky jsou vráceny jako logický součet enumerátorů *\_i2c\_flags* [7].

*Parametry funkce:*

<i>base</i>	Bázová adresa I <sup>2</sup> C modulu.
-------------	--

*status\_t I2C\_MasterWriteBlocking ( I2C\_Type \*base, const uint8\_t \*txBuff, size\_t txSize, uint32\_t flags )*

*Popis funkce:*

Funkce zapíše požadovaná data na specifikované I<sup>2</sup>C master rozhraní. Při zápisu vždy čeká na uvolnění vysílacího registru [7].

*Parametry funkce:*

<i>base</i>	Bázová adresa I <sup>2</sup> C modulu.
<i>txBuff</i>	Ukazatel na data určených k přenosu.
<i>txSize</i>	Velikost dat pro zápis v bajtech.
<i>flags</i>	Určuje, zda po odeslání dat bude vygenerován STOP signál či nikoliv. <i>kI2C_TransferDefaultFlag</i> – po odeslání dat bude vygenerován STOP <i>kI2C_TransferNoStop</i> – po odeslání dat nebude vygenerován STOP

*Návratové hodnoty:*

<i>kStatus_Success</i>	Odeslání dat bylo úspěšné.
<i>kStatus_I2C_ArbitrationLost</i>	Nastala chyba při přenosu dat, ztráta řízení sběrnice.
<i>kStatus_I2C_Nak</i>	Nastala chyba při přenosu dat, přijato NAK.





*status\_t I2C\_MasterReadBlocking* ( *I2C\_Type* \*base, *uint8\_t* \*rxBuff, *size\_t* rxSize, *uint32\_t* flags )

*Popis funkce:*

Funkce přečte data ze specifikovaného I<sup>2</sup>C master rozhraní. Při čtení čeká na příjem zadaného počtu znaků [7].

*Parametry funkce:*

<i>base</i>	Bázová adresa I <sup>2</sup> C modulu.
<i>rxBuff</i>	Ukazatel na buffer pro uložení dat.
<i>rxSize</i>	Počet bajtů k přečtení z rozhraní.
<i>flags</i>	Určuje, zda po čtení dat bude vygenerován STOP signál či nikoliv. <i>kI2C_TransferDefaultFlag</i> – po čtení dat bude vygenerován STOP <i>kI2C_TransferNoStop</i> – po čtení dat nebude vygenerován STOP

*Návratové hodnoty:*

<i>kStatus_Success</i>	Příjem dat byl úspěšný.
<i>kStatus_I2C_Timeout</i>	Odeslání STOP signálu selhalo, nastal timeout.

*status\_t I2C\_SlaveWriteBlocking* ( *I2C\_Type* \*base, *const uint8\_t* \*txBuff, *size\_t* txSize )

*Popis funkce:*

Funkce zapíše požadovaná data na specifikované I<sup>2</sup>C slave rozhraní. Při zápisu vždy čeká na uvolnění vysílacího registru [7].

*Parametry funkce:*

<i>base</i>	Bázová adresa I <sup>2</sup> C modulu.
<i>txBuff</i>	Ukazatel na data určených k přenosu.
<i>txSize</i>	Velikost dat pro zápis v bajtech.

*Návratové hodnoty:*

<i>kStatus_Success</i>	Odeslání dat bylo úspěšné.
------------------------	----------------------------



<i>kStatus_I2C_ArbitrationLost</i>	Nastala chyba při přenosu dat, ztráta řízení sběrnice.
<i>kStatus_I2C_Nak</i>	Nastala chyba při přenosu dat, přijato NAK.

***void I2C\_SlaveReadBlocking ( I2C\_Type \*base, uint8\_t \*rxBuff, size\_t rxSize )***

*Popis funkce:*

Funkce přečte data ze specifikovaného I<sup>2</sup>C slave rozhraní. Při čtení čeká na příjem zadaného počtu znaků [7].

*Parametry funkce:*

<i>base</i>	Bázová adresa I <sup>2</sup> C modulu.
<i>rxBuff</i>	Ukazatel na buffer pro uložení dat.
<i>rxSize</i>	Počet bajtů k přečtení z rozhraní.

***static void I2C\_MasterClearStatusFlags ( I2C\_Type \*base, uint32\_t statusMask )***

*Popis funkce:*

Funkce vynuluje stavové příznaky na specifikovaném I<sup>2</sup>C master rozhraní [7].

*Parametry funkce:*

<i>base</i>	Bázová adresa I <sup>2</sup> C modulu.
<i>statusMask</i>	Maska stavového příznaku definovaná typem <i>_i2c_flags</i> . Parametrem mohou být následující hodnoty v jakékoliv kombinaci: <ul style="list-style-type: none"><li>• <i>kI2C_ArbitrationLostFlag</i></li><li>• <i>kI2C_IntPendingFlagFlag</i></li></ul>

***static void I2C\_SlaveClearStatusFlags ( I2C\_Type \*base, uint32\_t statusMask )***

*Popis funkce:*

Funkce vynuluje stavové příznaky na specifikovaném I<sup>2</sup>C slave rozhraní [7].

*Parametry funkce:*

<i>base</i>	Bázová adresa I <sup>2</sup> C modulu.
-------------	--



<i>statusMask</i>	Maska stavového příznaku definovaná typem <i>_i2c_flags</i> . Parametrem mohou být následující hodnoty v jakékoliv kombinaci: <ul style="list-style-type: none"><li>• <i>kI2C_ArbitrationLostFlag</i></li><li>• <i>kI2C_IntPendingFlagFlag</i></li></ul>
-------------------	--

## 6.4 Ukázkové programy

V této kapitole naleznete ukázkové řešené příklady související s programovou obsluhou I<sup>2</sup>C modulu v C jazyce. V rámci laboratorního cvičení si funkci programů důkladně vyzkoušejte jejich odkrokováním v příslušném programovém prostředí.

### 6.4.1 Měření teploty snímačem LM75A

#### Zadání

Vytvořte program, který bude na standardní výstup periodicky odesílat aktuální teplotu změřenou digitálním teplotním snímačem LM75A připojeného k sériové komunikační sběrnici I2C1 mikropočítače.

#### Řešení

Rozhraní I2C1 je na vývojovém kitu dostupné na pinech PTE0 a PTE1. V inicializační části programu proto musí být povolen hodinový signál modulu PORTE zápisem 1 do 13. bitu registru SIM\_SCGC5 a modulu I2C1 zápisem 1 do 7. bitu registru SIM\_SCGC4. Dále musí být nastavena funkce pinů PTE0 a PTE1 na ALT6 v odpovídajících registrech PORTE\_PCR v bitovém poli *MUX*. Tímto nastavením bude na PTE0 aktivována funkce I2C1\_SDA (datová linka I<sup>2</sup>C) a na PTE1 funkce I2C1\_SCL (hodinový signál I<sup>2</sup>C). Komunikační rozhraní I2C1 se inicializuje do režimu master funkcí *I2C\_MasterInit()*, která má tři vstupní argumenty: básovou adresu I2C1 rozhraní, ukazatel na strukturu s konfigurací rozhraní a frekvenci zdrojového hodinového signálu v Hz. Struktura konfigurace rozhraní je inicializována funkcí *I2C\_MasterGetDefaultConfig()* do výchozího nastavení, které je pro většinu aplikací plně vyhovující a není zapotřebí do ní zasahovat. Frekvence zdroje hodinového signálu, kterým je  $f_{\text{BUS}}$ , je zjištěna funkcí *CLOCK\_GetBusClkFreq()*. Tímto je I2C1 rozhraní připraveno k činnosti a je již možno začít používat komunikační funkce. Teplotní snímač LM75A má implementován konfigurační registr a proto je vhodné na začátku programu jej inicializovat na výchozí hodnotu pro zajištění základní funkce měření teploty. To



prakticky znamená, že bude potřeba prostřednictvím I2C1 rozhraní zapsat do LM75 celkem dva bajty dat – hodnotu pointer registru, který musí ukazovat na konfigurační registr a hodnotu pro zápis do konfiguračního registru. Zapisovat se tedy budou hodnoty 1 a 0 uložené v poli *cmd\_lm75\_init*. Komunikace na I2C1 je zahájena funkcí *I2C\_MasterStart()* vyžadující 3 parametry: bázovou adresu rozhraní, adresu slave zařízení a režim přenosu. V tomto případě se do snímače LM75A zapisuje, tudíž její třetí parametr musí být *ki2C\_Write*. Funkce vygeneruje na sběrnici START signál, odešle 7bitovou adresu LM75A a 1bitovou hodnotu  $R/\overline{W}$  nastavenou na 0 (operace zápisu). Počká se na dokončení operace testováním stavového příznaku *ki2C\_IntPendingFlag* získaného čtením stavu rozhraní funkcí *I2C\_MasterGetStatusFlags()*. Jakmile je nastaven, operace na sběrnici byla dokončena a může se přikročit k jeho vynulování použitím funkce *I2C\_MasterClearStatusFlags()*. Přenos konfiguračního příkazu do LM75A se uskuteční funkcí *I2C\_MasterWriteBlocking()*, které se předá bázová adresa I2C1, ukazatel na pole s daty *cmd\_lm75\_init*, počet zapisovaných bajtů = 2 a příznak přenosu *ki2C\_TransferDefaultFlag* určující přenos zakončený STOP signálem na sběrnici. Tímto je LM75A připraven k činnosti.

V hlavní programové smyčce se provádí periodické dotazování snímače na naměřenou teplotu. Komunikaci se senzorem lze rozčlenit do následujících základních kroků:

1. Zápis adresy teplotního registru do Pointer registru snímače:
  - Odeslání START signálu, adresy zařízení, režim zápisu.
  - Čekání na dokončení operace testováním příznaku *ki2C\_IntPendingFlag*.
  - Vynulování příznaku *ki2C\_IntPendingFlag*.
  - Zápis 1 B dat (adresa teplotního registru) na I2C1 bez ukončení přenosu signálem STOP – použít příznak přenosu *ki2C\_TransferNoStopFlag*.
2. Čtení obsahu teplotního registru:
  - Odeslání opakovaného START signálu, adresy zařízení, režim čtení.
  - Čekání na dokončení operace testováním příznaku *ki2C\_IntPendingFlag*.
  - Vynulování příznaku *ki2C\_IntPendingFlag*.
  - Čtení teploty z teplotního registru (2 B dat) do pole *buff* s ukončením přenosu STOP signálem – použít příznak přenosu *ki2C\_TransferDefaultFlag*.
  - Zpracování dat a uložení 11bitové hodnoty do proměnné *lm75a\_temp*.



Naměřená teplota uložená v proměnné *lm75a\_temp* se musí převést do tvaru vhodného pro další použití v programu. Jedná se zejména o případ, kdy je naměřená teplota záporná a je zapotřebí správně naformátovat záporné číslo v cílové proměnné *teplota*. Výsledná hodnota je celočíselná se znaménkem vynásobená tisíci. Z toho plyne, že při naměřené teplotě 25,125 °C, bude v proměnné *teplota* uložena hodnota 25125. Toto má svůj praktický význam zejména při použití mikropočítače s malou kapacitou programové paměti, protože práce s reálnými čísly (datový typ float) je podstatně náročnější na obsazenou paměť (rozsáhlejší použité knihovny) i na výpočetní čas. V našem případě na mikropočítači MKL25Z128 toto není limitující. Na konci smyčky se pomocí funkce *PRINTF()* odešle na konzoli naměřený údaj v požadovaném formátu. Zdrojový text programu je uveden ve výpisu programu 6.1.

*Program 6.1:*

```
#include "board.h"
#include "pin_mux.h"
#include "clock_config.h"
#include "fsl_debug_console.h"
#include "fsl_i2c.h"
#include "fsl_clock.h"

// Adresa teplotního snímače LM75A na I2C
#define I2C_ADR_TEMP_SENSOR (0x48)
// Vybrané registry teplotního snímače LM75A
#define LM75A_REG_TEMP      (0)
#define LM75A_REG_CONF      (1)
// Inicializační příkazy pro snímač teploty
const uint8_t cmd_lm75_init[] = {LM75A_REG_CONF, 0};

int main(void) {
    status_t status;
    i2c_master_config_t i2c1_cfg;
    uint8_t buff[2];
    uint16_t lm75a_temp;
    int32_t bclk_freq, teplota;
    // Inicializace KL25Z desky, vytvořeno automaticky SDK
    BOARD_InitPins();
    BOARD_BootClockRUN();
```



## Program 6.1 (pokračování):

```
BOARD_InitDebugConsole();
// Inicializace SIM, PORTE
SIM->SCGC5 |= SIM_SCGC5_PORTE_MASK; // Povolení hodinového signálu modulu PORTE
SIM->SCGC4 |= SIM_SCGC4_I2C1_MASK; // Povolení hodinového signálu modulu I2C1
PORTE->PCR[0] = 0;
PORTE->PCR[0] = PORT_PCR_MUX(6); // PTE0 nastavení funkce ALT6 = I2C1_SDA
PORTE->PCR[1] = 0;
PORTE->PCR[1] = PORT_PCR_MUX(6); // PTE1 nastavení funkce ALT6 = I2C1_SCL
// Inicializace I2C1
bclk_freq = CLOCK_GetBusClkFreq(); // Zjištění frekvence sběrnice
I2C_MasterGetDefaultConfig(&i2c1_cfg); // Inicializace konfigurační struktury pro master režim
I2C_MasterInit(I2C1, &i2c1_cfg, bclk_freq); // Inicializace I2C1 v master režimu
// Inicializace teplotního snímače LM75A
// 1) Odeslání Start signálu, adresy zařízení, režim zápisu
I2C_MasterStart(I2C1, I2C_ADR_TEMP_SENSOR, kI2C_Write);
// 2) Čekej na odeslání na I2C1
while((I2C_MasterGetStatusFlags(I2C1) & kI2C_IntPendingFlag)==0);
I2C_MasterClearStatusFlags(I2C1, kI2C_IntPendingFlag);
// 3) Zápis výchozí konfigurace do LM75A (Pointer, Configuration data byte)
status = I2C_MasterWriteBlocking (I2C1, cmd_lm75_init, 2, kI2C_TransferDefaultFlag);
while(1) {
    // Čtení teploty z Temp registru
    // 1) Odeslání Start signálu, adresy zařízení, režim zápisu
    I2C_MasterStart(I2C1, I2C_ADR_TEMP_SENSOR, kI2C_Write);
    // 2) Čekej na odeslání na I2C1
    while((I2C_MasterGetStatusFlags(I2C1) & kI2C_IntPendingFlag)==0);
    I2C_MasterClearStatusFlags(I2C1, kI2C_IntPendingFlag);
    // 3) Zápis adresy Temp registru do Pointer registru
    buff[0] = LM75A_REG_TEMP;
    status = I2C_MasterWriteBlocking(I2C1, buff, 1, kI2C_TransferNoStopFlag);
    // 4) Odeslání Repeated Start signálu, adresy zařízení, režim čtení
    status = I2C_MasterRepeatedStart(I2C1, I2C_ADR_TEMP_SENSOR, kI2C_Read);
    // 5) Čekej na odeslání na I2C1
    while((I2C_MasterGetStatusFlags(I2C1) & kI2C_IntPendingFlag)==0);
    I2C_MasterClearStatusFlags(I2C1, kI2C_IntPendingFlag);
    // 6) Přečti teplotu z Temp registru (přenos 2 bajtů do buff)
    status = I2C_MasterReadBlocking(I2C1, buff, 2, kI2C_TransferDefaultFlag);
    // 7) Ulož 11bitový údaj z buff[0] a buff[1] do proměnné lm75a_temp
    lm75a_temp = (256 * buff[0] + buff[1]) >> 5;
```

*Program 6.1 (pokračování):*

```
// Výpočet teploty vynásobené 1000 v celočíselném formátu se znaménkem
if ((lm75a_temp & 0x400) == 0) {
    teplota = 125 * lm75a_temp;           // Naměřená teplota je kladná
}
else {
    lm75a_temp = (-1 * lm75a_temp) & 0x7ff; // Naměřená teplota je záporná
    teplota = -125 * lm75a_temp;
}
// Výpis teploty do konzole
PRINTF("Teplota = %d.%03d degC\n", teplota / 1000, teplota % 1000);
Cekej(1000);
}
}
```

**6.4.2 Měření vlhkosti vzduchu snímačem HIH6130****Zadání**

Vytvořte program, který bude na standardní výstup periodicky odesílat aktuální vlhkost změřenou digitálním snímačem HIH6130 připojeného k sériové komunikační sběrnici I2C1 mikropočítače.

**Řešení**

Úvodní část programu zahrnující inicializaci všech potřebných modulů je shodná s příkladem z kapitoly 6.4.1, proto zde již nebude znovu popisována. Snímač vlhkosti HIH6130 nemá implementován konfigurační registr, tudíž se neprovádí jeho úvodní konfigurace. Programová obsluha je u tohoto snímače koncepčně odlišná v porovnání se snímačem teploty LM75A, protože je trvale v režimu spánku a neprovádí tudíž žádná měření. Měření se spustí pouze na příkaz z I<sup>2</sup>C rozhraní a po jeho dokončení zaktualizuje datový registr a opět přejde zpět do režimu spánku pro úsporu energie. V hlavní programové smyčce je proto na začátku sekvence pro spuštění měření, která se skládá z odeslání START signálu, adresy slave zařízení a režimu zápisu pomocí funkce *I2C\_MasterStart()*. Po čekání na dokončení operace se na sběrnici odešle STOP signál. Dále se provede čtení 2 B dat z datového registru pomocí následující sekvence:





- Odeslání START signálu, adresy zařízení, režim čtení.
- Čekání na dokončení operace testováním příznaku *ki2C\_IntPendingFlag*.
- Vynulování příznaku *ki2C\_IntPendingFlag*.
- Čtení vlhkosti (2 B dat) do pole *buff* s ukončením přenosu STOP signálem – použít příznak přenosu *ki2C\_TransferDefaultFlag*.
- Zpracování dat a uložení 14bitové hodnoty do proměnné *hih\_rh*.

Součástí přečtených dat jsou také dva stavové bity S1 a S0 indikující, zda jsou data aktuální ([S1:S0] = 00), nebo se jedná o minulá již přečtená data ([S1:S0] = 01). Čtení obsahu datového registru se ve smyčce *do-while* provádí tak dlouho, dokud je hodnota ve stavových bitech různá od nuly. Jakmile jsou k dispozici nová data, smyčka se ukončí a provede se výpočet změřené vlhkosti v celočíselném formátu a odeslání údaje do konzole pomocí funkce *PRINTF()*. Zdrojový text jádra programu je uveden ve výpisu programu 6.2. Inicializační části jsou pro přehlednost vynechány, protože jsou stejné jako v programu 6.1.

Program 6.2:

```
#include "board.h"
// ..., viz program 6.1
// Adresa snímače vlhkosti HIH6130 na I2C
#define I2C_ADR_HIH_SENSOR (0x27)

int main(void) {
    status_t status;
    i2c_master_config_t i2c1_cfg;
    uint8_t buff[2], hih_status;
    uint16_t hih_rh;
    uint32_t bclk_freq, vlhkost;
    // Inicializace KL25Z desky, vytvořeno automaticky SDK
    // ..., viz program 6.1
    // Inicializace SIM, PORTE
    // ..., viz program 6.1
    // Inicializace I2C1
    // ..., viz program 6.1
    while(1)
    {
        // Spuštění měření vlhkosti
```



*Program 6.2 (pokračování):*

```
// 1) Odeslání Start signálu, adresy zařízení, režim zápisu dat
I2C_MasterStart(I2C1, I2C_ADR_HIH_SENSOR, kI2C_Write);
// 2) Čekaj na odeslání na I2C1
while((I2C_MasterGetStatusFlags(I2C1) & kI2C_IntPendingFlag)!=0);
I2C_MasterClearStatusFlags(I2C1, kI2C_IntPendingFlag);
// 3) Odeslání Stop signálu na I2C1
status = I2C_MasterStop(I2C1);
// Čtení naměřené vlhkosti s čekáním na nová data
do {
    // 1) Odeslání Start signálu, adresy zařízení, režim čtení dat
    I2C_MasterStart(I2C1, I2C_ADR_HIH_SENSOR, kI2C_Read);
    // 2) Čekaj na odeslání na I2C1
    while((I2C_MasterGetStatusFlags(I2C1) & kI2C_IntPendingFlag)!=0);
    I2C_MasterClearStatusFlags(I2C1, kI2C_IntPendingFlag);
    // 3) Přečti vlhkost (přenos 2 bajtů do buff)
    status = I2C_MasterReadBlocking(I2C1, buff, 2, kI2C_TransferDefaultFlag);
    // 4) Ulož 14bitový údaj z buff[0] a buff[1] do proměnné hih_rh
    hih_rh = (256 * buff[0] + buff[1]) & 0x3fff;
    // 5) Ulož status snímače z horních 2 bitů buff[0]
    hih_status = buff[0] >> 6;
}
while(hih_status != 0);
// Výpočet vlhkosti vynásobené 1000 v celočíselném formátu bez znaménka
vlhkost = (uint32_t)hih_rh * 100000 / 16382;
// Výpis vlhkosti do konzole
PRINTF("Vlhkost = %d.%03d %%RH\n", vlhkost / 1000, vlhkost % 1000);
Cekej(1000);
}
}
```

**6.4.3 Čtení časového údaje z hodin reálného času PCF8583****Zadání**

Vytvořte program, který bude na standardní výstup periodicky odesílat aktuální hodnotu sekund přečtenou z obvodu reálného času PCF8583 připojeného k sériové komunikační sběrnici I2C1 mikropočítače.



## Řešení

Úvodní část programu zahrnující inicializaci všech potřebných modulů je shodná s příkladem z kapitoly 6.4.1, proto zde již nebude znovu popisována. Hodiny reálného času PCF8583 má implementován konfigurační registr, tudíž je zapotřebí provést jeho inicializaci. Ta se provede zápisem hodnoty 0 do řídicího a stavového registru na adrese 0. Do RTC obvodu se tedy zapíše 2 B dat připravené v poli *cmd\_rtc\_init*. Tímto je zajištěna standardní funkce hodin s kalendářem a budíkem, běh hodin je řízen externím krystalem s frekvencí 32768 Hz. Zápis konfigurace je stejný jako u LM75A, jen se zapisují odlišná data.

V hlavní programové smyčce se provádí periodické dotazování obvodu RTC na obsah registru sekund na adrese 2. Komunikaci se senzorem lze rozčlenit do následujících základních kroků:

1. Zápis adresy, ze které bude následně prováděno čtení:
  - Odeslání START signálu, adresy zařízení, režim zápisu.
  - Čekání na dokončení operace testováním příznaku *ki2C\_IntPendingFlag*.
  - Vynulování příznaku *ki2C\_IntPendingFlag*.
  - Zápis 1 B dat (adresa registru sekund) na I2C1 bez ukončení přenosu signálem STOP – použít příznak přenosu *ki2C\_TransferNoStopFlag*.
2. Čtení obsahu registru sekund:
  - Odeslání opakovaného START signálu, adresy zařízení, režim čtení.
  - Čekání na dokončení operace testováním příznaku *ki2C\_IntPendingFlag*.
  - Vynulování příznaku *ki2C\_IntPendingFlag*.
  - Čtení hodnoty z registru sekund (1 B dat) do pole *buff* s ukončením přenosu STOP signálem – použít příznak přenosu *ki2C\_TransferDefaultFlag*.
  - Převedení údaje z BCD kódu a uložení do proměnné *rtc\_sec*.

Dále následuje odeslání údaje aktuálních sekund do konzole pomocí funkce *PRINTF()*. Pro zamezení zbytečného vypisování stejného časového údaje, je zde podmínka umožňující výpis pouze tehdy, pokud je přečtená hodnota sekund (proměnná *rtc\_sec*) rozdílná od minule vypsané (proměnná *rtc\_sec\_last*). Po výpisu hodnoty na konzoli se provede aktualizace proměnné *rtc\_sec\_last*. Zdrojový text jádra programu je uveden ve výpisu programu 6.3. Inicializační části jsou pro přehlednost vynechány, protože jsou stejné jako v programu 6.1.



Program 6.3:

```
#include "board.h"
// ..., viz program 6.1

// Adresa RTC obvodu PCF8583 na I2C sběrnici
#define I2C_ADR_RTC    (0x50)
// Vybrané registry RTC PCF8583
#define RTC_REG_CS     (0)
#define RTC_REG_HSEC   (1)
#define RTC_REG_SEC    (2)
#define RTC_REG_MIN    (3)
#define RTC_REG_HRS    (4)

// Inicializační příkaz pro RTC: CLK 32768 Hz, RTC spuštěn
const uint8_t cmd_rtc_init[] = {RTC_REG_CS, 0};

int main(void) {
    status_t status;
    i2c_master_config_t i2c1_cfg;
    uint8_t buff[2], rtc_sec, rtc_sec_last;
    uint32_t bclk_freq;
    // Inicializace KL25Z desky, vytvořeno automaticky SDK
    // ..., viz program 6.1
    // Inicializace SIM, PORTE
    // ..., viz program 6.1
    // Inicializace I2C1
    // ..., viz program 6.1
    // Inicializace RTC PCF8583
    // 1) Odeslání Start signálu, adresy zařízení, režim zápisu
    I2C_MasterStart(I2C1, I2C_ADR_RTC, kI2C_Write);
    // 2) Čekej na odeslání na I2C1
    while((I2C_MasterGetStatusFlags(I2C1) & kI2C_IntPendingFlag)!=0);
    I2C_MasterClearStatusFlags(I2C1, kI2C_IntPendingFlag);
    // 3) Zápis výchozí konfigurace do RTC (adresa registru, hodnota)
    status = I2C_MasterWriteBlocking (I2C1, cmd_rtc_init, 2, kI2C_TransferDefaultFlag);

    while(1) {
        // Čtení registru sekund RTC
        // 1) Odeslání Start signálu, adresy zařízení, režim zápisu
        I2C_MasterStart(I2C1, I2C_ADR_RTC, kI2C_Write);
```



*Program 6.3 (pokračování):*

```
// 2) Čekaj na odeslání na I2C1
while((I2C_MasterGetStatusFlags(I2C1) & kI2C_IntPendingFlag)==0);
I2C_MasterClearStatusFlags(I2C1, kI2C_IntPendingFlag);
// 3) Zápis výchozí konfigurace do RTC (adresa registru, hodnota)
status = I2C_MasterWriteBlocking (I2C1, cmd_rtc_init, 2, kI2C_TransferDefaultFlag);

while(1)
{
    // Čtení registru sekund RTC
    // 1) Odeslání Start signálu, adresy zařízení, režim zápisu
    I2C_MasterStart(I2C1, I2C_ADR_RTC, kI2C_Write);
    // 2) Čekaj na odeslání na I2C1
    while((I2C_MasterGetStatusFlags(I2C1) & kI2C_IntPendingFlag)==0);
    I2C_MasterClearStatusFlags(I2C1, kI2C_IntPendingFlag);
    // 3) Zápis adresy registru, který se bude následně číst
    buff[0] = RTC_REG_SEC;
    status = I2C_MasterWriteBlocking(I2C1, buff, 1, kI2C_TransferNoStopFlag);
    // 4) Odeslání opakovaného Start signálu, adresy zařízení, režim čtení
    status = I2C_MasterRepeatedStart(I2C1, I2C_ADR_RTC, kI2C_Read);
    // 5) Čekaj na odeslání na I2C1
    while((I2C_MasterGetStatusFlags(I2C1) & kI2C_IntPendingFlag)==0);
    I2C_MasterClearStatusFlags(I2C1, kI2C_IntPendingFlag);
    // 6) Přečti obsah registru sekund
    status = I2C_MasterReadBlocking(I2C1, buff, 1, kI2C_TransferDefaultFlag);
    // 7) Převed' hodnotu z BCD kódu do BIN a ulož do proměnné rtc_sec
    rtc_sec = (buff[0] >> 4) * 10 + (buff[0] & 0x0f);

    // Výpis sekund do konzole s periodou 1 s
    if(rtc_sec != rtc_sec_last)
    {
        PRINTF("RTC_REG_SEC = %02d s\n", (int)rtc_sec);
        rtc_sec_last = rtc_sec;
    }
    Cekej(50);
}
}
```

## 6.5 Zadání samostatné práce

1. Vytvořte sadu funkcí pro obsluhu všech periférií připojených na komunikační rozhraní I2C1. Implementujte následující funkce:

- *void **LM75A\_Init**(I2C\_Type \*base, uint8\_t address)*  
Inicializace snímače teploty LM75A. Funkci se předá bazová adresa I2C rozhraní a adresa slave zařízení na sběrnici.
- *int32\_t **LM75A\_GetTemp**(I2C\_Type \*base, uint8\_t address)*  
Čtení teploty v celočíselném formátu.
- *float **LM75A\_GetTemp\_f**(I2C\_Type \*base, uint8\_t address)*  
Čtení teploty ve float formátu.
- *void **HIH6130\_StartMeasurement**(I2C\_Type \*base, uint8\_t address)*  
Spuštění jednoho měření vlhkosti.
- *uint8\_t **HIH6130\_GetRh**(I2C\_Type \*base, uint8\_t address, uint32\_t \*rh)*  
Čtení naměřené relativní vlhkosti. Funkci se předá ukazatel na proměnnou typu uint32\_t pro uložení vlhkosti v celočíselném formátu. Návrátovou hodnotou je stav snímače.
- *uint8\_t **HIH6130\_GetRh\_f**(I2C\_Type \*base, uint8\_t address, float \*rh)*  
Čtení naměřené relativní vlhkosti. Funkci se předá ukazatel na proměnnou typu float pro uložení vlhkosti ve formátu s pohyblivou řádovou čárkou. Návrátovou hodnotou je stav snímače.
- *void **PCF8583\_Init**(I2C\_Type \*base, uint8\_t address)*  
Inicializace hodin reálného času PCF8583.
- *void **PCF8583\_ReadTime**(I2C\_Type \*base, uint8\_t address, s\_time \*t)*  
Čtení aktuálního času z obvodu reálného času PCF8583. Funkci se předá ukazatel na strukturu obsahující prvky hod, min a sec typu uint8\_t, do kterých funkce zapíše aktuální čas.  
Příklad: 

```
typedef struct {  
    uint8_t hod;  
    uint8_t min;  
    uint8_t sec;  
} s_time;
```
- *void **PCF8583\_WriteTime**(I2C\_Type \*base, uint8\_t address, s\_time \*t)*  
Zápis času do obvodu reálného času PCF8583. Funkci se předá ukazatel na strukturu obsahující prvky hod, min a sec typu uint8\_t, ve kterých je uložen čas pro zápis do RTC.

## Zadání samostatné práce (pokračování)

2. Upravte ukázkový program 6.2 tak, aby kromě vlhkosti četl ze snímače i teplotu a posílal ji společně s relativní vlhkostí do konzole.

Tipy pro řešení:

Místo 2 bajtů přečtete ze senzoru 4 bajty (nezapomeňte přitom adekvátně zvětšit buffer pro příjem dat). První dva přijaté bajty obsahují relativní vlhkost a stavové bity S1 a S0, další dva bajty teplotu.

3. Upravte ukázkové programy 6.1, 6.2 a 6.3 tak, aby používaly pro komunikaci s periferními obvody připojenými na I<sup>2</sup>C pouze Vámi vytvořené funkce z předchozího bodu zadání. Ověřte funkci programů včetně možnosti výstupu naměřených hodnot ve formátu float.
4. Upravte ukázkový program 6.3 tak, aby bylo možné po startu programu prostřednictvím konzole zadat aktuální čas a zapsat jej do obvodu RTC. Po nastavení bude již program periodicky při každé změně sekund posílat přesný čas ve formátu hodiny:minuty:sekundy do konzole. Zadávání ošetřete proti neplatným vstupním údajům pro danou položku.

Příklad menu pro nastavení RTC:

Vyberte požadované nastavení RTC:

1 - Nastavení hodin [0-23]

2 - Nastavení minut [0-59]

3 - Nastavení sekund [0-59]

4 - Ukončení zadávání (start vypisování času)

Vaše volba:

Po zvolení funkce zadáním číselné hodnoty 1 – 3 program vyzve uživatele k zadání příslušné hodnoty. Při volbě 4 se zadávání ukončí a aktivuje se vypisování aktuálního času z RTC do konzole.



## 7 SÉRIOVÉ PERIFERNÍ ROZHRAŇÍ SPI

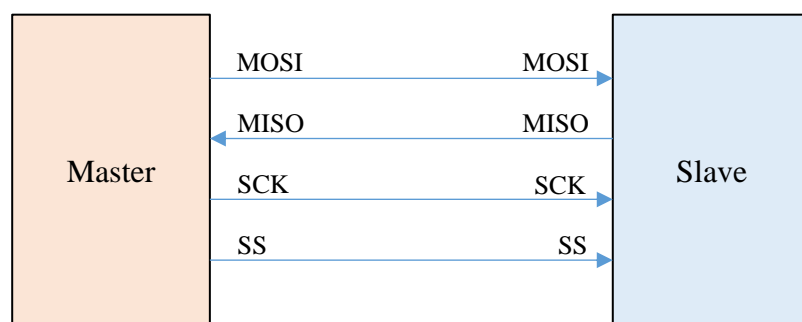
Sériové periferní rozhraní je plně duplexní vysokorychlostní synchronní sériové komunikační rozhraní sloužící pro připojení externích periférií k mikrokontroléru na malé vzdálenosti, většinou v rámci desky plošného spoje. Dostupná je široká škála periferních obvodů vybavených SPI – snímače fyzikálních veličin, paměťové obvody RAM, EEPROM a FLASH, posuvné registry, expandéry vstupů a výstupů, A/D a D/A převodníky, displeje, bezdrátová komunikační rozhraní a řada dalších.

Komunikace probíhá metodou master – slave, kdy zařízení master vždy zahajuje a řídí vlastní datový přenos. Rozhraní používá dvě jednosměrné datové linky MISO (Master In Slave Out) a MOSI (Master Out Slave In), linku SCK (Serial Clock) pro přenos hodinového signálu a řídicí signál pro výběr slave zařízení SS (Slave Select). Způsob propojení dvou zařízení prostřednictvím SPI rozhraní ukazuje obrázek 44.

Komunikace probíhá následujícím způsobem:

- Master vybere slave jednotku, se kterou bude komunikovat pomocí řídicího signálu SS změnou jeho stavu z logické 1 na logickou 0.
- Vybrané slave zařízení přejde do aktivního stavu a očekává příjem dat na pinu MOSI synchronizovaný hodinovým signálem SPSCCK. Na každou aktivní hranu SPSCCK bude zároveň odesílat data na pin MISO.
- Master zahájí vysílání dat na pinu MOSI přičemž generuje synchronizační hodinový signál SPSCCK. Dle aktuálního nastavení rozhraní se na každou aktivní hranu (může být náběžná nebo sestupná) hodinového signálu přenes 1 bit. Na pinu MISO zároveň přijímá data odesílaná slave zařízením.
- Po odeslání 1 bajtu dat masterem, je v jeho přijímači k dispozici přijatý 1 bajt od zařízení slave, který se musí vždy z rozhraní přečíst. Teprve potom se může zahájit přenos dalšího bajtu dat.
- Provede se požadovaný počet přenosů dat mezi master a slave zařízením dle aktuální potřeby aplikace.
- Komunikace se slave zařízením se ukončí změnou stavu řídicího signálu SS z logické 0 zpět do neaktivní úrovně – logické 1.

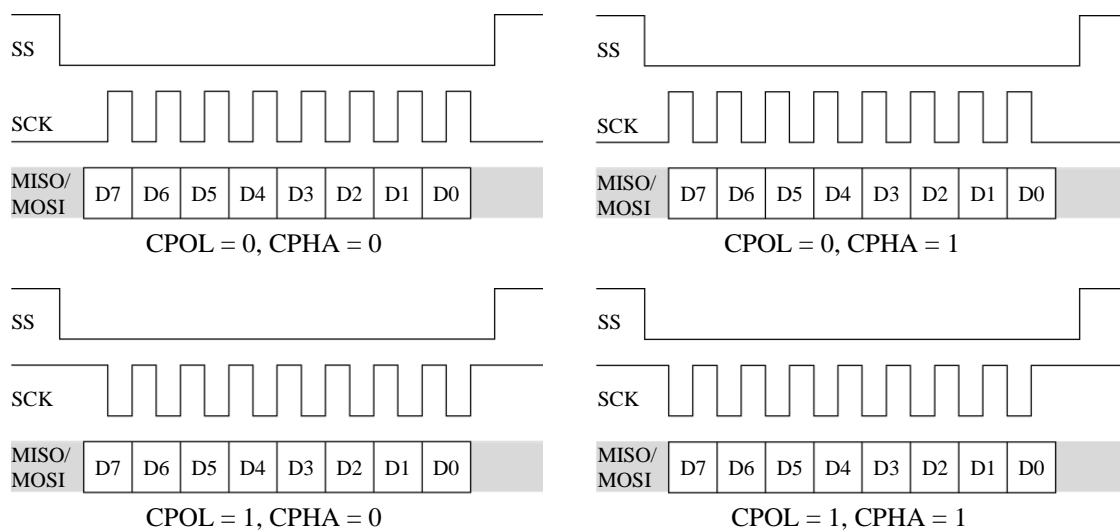




Obrázek 44: Propojení zařízení SPI rozhraním [2].

### 7.1 Implementace SPI na výukovém kitu

Mikropočítač MKL25Z128 disponuje celkem dvěma interními SPI moduly SPI0 a SPI1, které mohou být nakonfigurovány do režimu master i slave. Přenosová rychlost rozhraní je programově nastavitelná v širokém rozsahu pomocí děličky vstupní hodinové frekvence modulu. Maximální dosažitelná rychlost je u rozhraní SPI0 pro režim master  $f_{\text{BUS}} / 2$  a v režimu slave  $f_{\text{BUS}} / 4$ , u SPI1 je to z důvodu jiného zdroje hodinového signálu modulu (systémová frekvence)  $f_{\text{SYS}} / 2$  pro master a  $f_{\text{SYS}} / 4$  pro slave režim. Formát přenosu znaku lze nastavit na vysílání / příjem jako prvního nejvíce významného bitu (MSB) nebo nejméně významného bitu (LSB). Nastavitelná polarita a fáze hodinového signálu SPI modulu umožňuje přizpůsobení se komunikačním požadavkům SPI zařízení, viz obrázek 45.



Obrázek 45: Nastavení polarity a fáze hodinového signálu SPI [13].

Výukový kit používá periferní komunikační rozhraní SPI0 pro připojení externí paměti EEPROM Microchip 25LC640A o kapacitě 8 KiB. Na rozhraní SPI1 není připojena žádná periferie, jeho komunikační signály jsou dostupné na konektoru MCU port pro použití na externích výukových modulech. Přehled SPI periférií a použitých signálů rozhraní je uveden v tabulce 16.

SPI rozhraní	Periferie	Signál	Pin MCU
SPI0	EEPROM 25LC640A	MISO	PTC7
		MOSI	PTC6
		SCK	PTC5
		SS - GPIO	PTC0
SPI1	MCU port – 44	MISO	PTD7
	MCU port – 43	MOSI	PTD6
	MCU port – 25	SCK	PTD5
	MCU port – libovolný GPIO	SS - GPIO	PTXn

Tabulka 16: Přehled periférií kitu s SPI rozhraním.

#### 7.1.1 EEPROM paměť 25LC640A

Integrovaný obvod 25LC640A je nevolatilní EEPROM paměť s kapacitou 8 KiB s vnitřní organizací 8192 x 8 bitů vyrobená CMOS technologií vyznačující se nízkou spotřebou. Komunikaci s mikropočítačem zajišťuje sériové periferní rozhraní SPI s maximální frekvencí hodinového signálu 2 MHz při napájecím napětí v rozsahu od 2,5 do 5,5 V. Tato hodnota je platná i pro výukový kit, protože používá napájecí napětí pro veškeré digitální obvody o velikosti 3 V. Podporovaný režim SPI (polarita, fáze hodinového signálu) je (0, 0) nebo (1, 1). Pro zrychlení přístupu k paměťovým buňkám v režimu zápisu, pracuje se stránkami o velikosti 32 B. Zápisové a mazací cykly paměti jsou časovány interně – délka těchto cyklů činí nejvíce 5 ms. Paměť je vybavena funkcí ochrany proti zápisu, kterou lze zcela vypnout, aplikovat na čtvrtinu, polovinu nebo celý paměťový prostor. Životnost paměti je minimálně 1000000 přepisů a doba uchování uložených informací delší jak 200 let [14].

Funkce paměti je řízena prostřednictvím 6 různých instrukcí, jejichž přehled je uveden v tabulce 17. Instrukce se posílá na začátku SPI přenosu po výběru paměti signálem CS s nízkou logickou úrovní. Všechny instrukce, adresy a data jsou přenášeny s prvním odvyšláným nejvíce významným bitem. Při přenosu 16bitového údaje se první vysílá vyšší bajt.



Název instrukce	Číselná reprezentace	Popis
READ	0000 0011 <sub>2</sub>	Čtení paměti od specifikované adresy
WRITE	0000 0010 <sub>2</sub>	Zápis do paměti od specifikované adresy
WRDI	0000 0100 <sub>2</sub>	Vynulování zámku pro povolení zápisu (zápis zakázán)
WREN	0000 0110 <sub>2</sub>	Nastavení zámku pro povolení zápisu (zápis povolen)
RDSR	0000 0101 <sub>2</sub>	Čtení STATUS registru
WRSR	0000 0001 <sub>2</sub>	Zápis do STATUS registru

Tabulka 17: Instrukce EEPROM paměti 25LC640A [13].

Čtení obsahu paměťové buňky EEPROM probíhá v následujících krocích:

1. Výběr paměti přechodem řídicího signálu CS do logické úrovně 0.
2. Odeslání 8 bitové instrukce READ následované 16bitovou adresou paměťové buňky k přečtení. Během vysílání těchto 3 bajtů je výstup SO paměti ve stavu vysoké impedance. Přesto je nutné vždy na SPI0 rozhraní počkat po odeslání každého bajtu dat na příjem dat a přečíst datový registr, jinak dojde k chybě přetečení přijímače, která není indikována.
3. Odeslání 1 bajtu dat s libovolnou hodnotou, aby byly vygenerovány hodinové pulsy pro příjem hodnoty přečtené z paměťové buňky.
4. Čekání na příjem znaku na SPI rozhraní a následné přečtení hodnoty z datového registru SPI.
5. V případě požadavku na sekvenční čtení více po sobě jdoucích adres paměti lze s výhodou použít funkce automatické inkrementace adresy. V tomto případě se pokračuje bodem 3.
6. Ukončení komunikace s pamětí přechodem řídicího signálu CS do logické 1.

Při zápisu do paměti je nutné dodržet následující sekvenci:

1. Výběr paměti přechodem řídicího signálu CS do logické úrovně 0.
2. Odeslání 8 bitové instrukce WREN, která zajistí odemknutí paměti pro zápis.
3. Ukončení komunikace s pamětí přechodem řídicího signálu do logické 1.
4. Výběr paměti přechodem řídicího signálu CS do logické úrovně 0.
5. Odeslání 8 bitové instrukce WRITE následované 16bitovou adresou paměťové buňky, do které se bude zapisovat.

6. Odeslání 1 bajtové hodnoty určené pro zápis.
7. V případě požadavku na sekvenční zápis po sobě jdoucích paměťových buněk je nutné mít na paměti, že se zapisuje pouze v rámci jedné 32 bajtové stránky. Zápis přes hranici stránky není možný a dojde při něm k přepisu hodnoty paměťové buňky na začátku stránky. Opakováním bodu 6 lze postupně zapisovat až na konec stránky paměti.
8. Ukončení komunikace s pamětí přechodem řídicího signálu do logické 1.
9. Čekání na dokončení zápisu do paměťových buněk čtením STATUS registru (viz obrázek 46) příkazem RDSR a testováním příznakového bitu *WIP*.
  - Jakmile je příznak *WIP* = 0, je zápis do paměti ukončen.
10. Ukončení komunikace s pamětí přechodem řídicího signálu CS do logické 1.

### STATUS registr

Čtení obsahu status registru paměti se provádí pomocí instrukce RDSR. Jeho obsah může být čten kdykoliv i v průběhu operace zápisu dat do paměťových buněk. Zápis hodnoty do STATUS registru pomocí instrukce WRSR se provádí pouze za účelem modifikace bitů BP1, BP0 a WPEN jejichž obsah je uložen v nevolatilní části paměti. Význam jednotlivých bitů registru je následující:

WIP – Indikace probíhající operace zápisu (pouze čtení):

0 = neprobíhá zápis

1 = paměť je zaneprázdněna probíhající operací zápisu

WEL – Indikace stavu zámku pro povolení zápisu (pouze čtení):

0 = zápis do paměti je zakázán

1 = zápis do paměti je povolen

BP1, BP0 – Zjištění/nastavení stavu ochrany paměti proti zápisu (čtení i zápis):

00 = ochrana vypnuta

01 = ochrana aktivní v horní ¼ paměti (adresy 0x1800 – 0x1FFF)

10 = ochrana aktivní v horní ½ paměti (adresy 0x1000 – 0x1FFF)

11 = ochrana aktivní v celé paměti (adresy 0x0000 – 0x1FFF)



WPEN – Povolení funkce WP pinu (čtení i zápis).

0 = funkce pinu WP je zakázána

1 = funkce pinu WP je povolena

7	6	5	4	3	2	1	0
WPEN	X	X	X	BP1	BP0	WEL	WIP



- bit pouze pro čtení

- bit pro čtení i zápis (hodnota bitu uložena v nevolatilní paměti)

Obrázek 46: STATUS registr EEPROM 25LC640 [14].

## 7.2 Programová obsluha SPI modulů

Obdobně jako u modulů UART bude při programové obsluze komunikačních rozhraní SPI použit přístup na vyšší úrovni pomocí periferních driverů, které jsou součástí MCUXpresso programového vývojového kitu (SDK) pro mikropočítače NXP. Periferní driver SPI rozhraní obsahuje funkční a transakční API. Pro jeho obsluhu bude použito funkční API.

### 7.2.1 Popis vybraných funkcí funkčního API

`void SPI_MasterGetDefaultConfig ( spi_master_config_t *config )`

*Popis funkce:*

Funkce provede inicializaci struktury s konfigurací pro master režim výchozími hodnotami uvedenými v níže uvedeném seznamu jednotlivých položek [7].

*Parametry funkce:*

<code>config</code>	Ukazatel na strukturu pro uložení konfigurace.
---------------------	--

Výchozí hodnoty naplněné funkcí do konfigurační struktury jsou následující:

- `enableMaster = true`
- `enableStopInWaitMode = false`
- `polarity = kSPI_ClockPolarityActiveHigh`



- `phase = kSPI_ClockPhaseFirstEdge`
- `direction = kSPI_MsbFirst;`
- `pinMode = kSPI_PinModeNormal`
- `outputMode = kSPI_SlaveSelectAutomaticOutput`
- `baudRate_Bps = 500000`

`void SPI_MasterInit ( SPI_Type *base, const spi_master_config_t *config, uint32_t srcClock_Hz )`

*Popis funkce:*

Funkce nakonfiguruje SPI modul do režimu master s bázovou adresou *base* na základě nastavení uloženého ve struktuře *config* a frekvence zdroje hodinového signálu modulu *srcClock\_Hz*. Strukturu lze naplnit výchozími hodnotami pomocí funkce *SPI\_MasterGetDefaultConfig ()* [7].

*Parametry funkce:*

<i>base</i>	Bázová adresa SPI modulu.
<i>config</i>	Ukazatel na uživatelsky definovanou strukturu s konfigurací master.
<i>srcClock_Hz</i>	Frekvence zdroje hodinového signálu SPI modulu v [Hz].

`void SPI_SlaveGetDefaultConfig ( spi_slave_config_t *config )`

*Popis funkce:*

Funkce provede inicializaci struktury s konfigurací pro slave režim výchozími hodnotami uvedenými v níže uvedeném seznamu jednotlivých položek [7].

*Parametry funkce:*

<i>config</i>	Ukazatel na strukturu pro uložení konfigurace.
---------------	--

Výchozí hodnoty naplněné funkcí do konfigurační struktury jsou následující:

- `enableSlave = true`
- `polarity = kSPI_ClockPolarityActiveHigh`





- phase = kSPI\_ClockPhaseFirstEdge
- direction = kSPI\_MsbFirst
- pinMode = kSPI\_PinModeNormal
- enableStopInWaitMode = false

*void SPI\_SlaveInit ( SPI\_Type \*base, const spi\_slave\_config\_t \*config )*

*Popis funkce:*

Funkce nakonfiguruje SPI modul do režimu slave s bázovou adresou *base* na základě nastavení uloženého ve struktuře *config*. Strukturu lze naplnit výchozími hodnotami pomocí funkce *SPI\_SlaveGetDefaultConfig()* [7].

*Parametry funkce:*

<i>base</i>	Bázová adresa SPI modulu.
<i>config</i>	Ukazatel na uživatelsky definovanou strukturu s konfigurací slave.

*void SPI\_Deinit ( SPI\_Type \*base )*

*Popis funkce:*

Funkce vypne zdroj hodinového signálu pro SPI modul [7].

*Parametry funkce:*

<i>base</i>	Bázová adresa SPI modulu.
-------------	---------------------------

*static void SPI\_Enable ( SPI\_Type \*base, bool enable ) [inline], [static]*

*Popis funkce:*

Funkce povolí nebo zakáže funkci SPI modulu [7].

*Parametry funkce:*

<i>base</i>	Bázová adresa SPI modulu.
<i>enable</i>	true = povolení funkce; false = zakázání funkce



*uint32\_t SPI\_GetStatusFlags ( SPI\_Type \*base )*

*Popis funkce:*

Funkce vrací stavové příznaky specifikovaného SPI rozhraní. Příznaky jsou vráceny jako logický součet enumerátorů `_spi_flags` [7].

*Parametry funkce:*

<i>base</i>	Bázová adresa SPI modulu.
-------------	---------------------------

*void SPI\_WriteBlocking ( SPI\_Type \*base, uint8\_t \*buffer, size\_t size )*

*Popis funkce:*

Funkce zapíše požadovaná data na specifikované SPI rozhraní. Při zápisu vždy čeká na uvolnění vysílacího registru [7].

*Parametry funkce:*

<i>base</i>	Bázová adresa SPI modulu.
<i>buffer</i>	Ukazatel na data určených k přenosu.
<i>size</i>	Velikost dat pro zápis v bajtech.

*void SPI\_WriteData ( SPI\_Type \*base, uint16\_t data )*

*Popis funkce:*

Funkce zapíše jeden znak na specifikované SPI rozhraní. Při zápisu nečeká na uvolnění vysílacího registru. Před použitím funkce je nutné nejdříve zjistit stav rozhraní funkcí `SPI_GetStatusFlags()` [7].

*Parametry funkce:*

<i>base</i>	Bázová adresa SPI modulu.
<i>data</i>	Znak pro odeslání na rozhraní SPI.



`uint16_t SPI_ReadData ( SPI_Type *base )`

*Popis funkce:*

Funkce přečte jeden znak ze specifikovaného SPI rozhraní. Při čtení nečeká na příjem znaku do registru přijímače. Před použitím funkce je nutné nejdříve zjistit stav rozhraní funkcí `SPI_GetStatusFlags()` [7].

*Parametry funkce:*

<i>base</i>	Bázová adresa SPI modulu.
-------------	---------------------------

### 7.3 Ukázkové programy

V této kapitole naleznete ukázkové řešené příklady související s programovou obsluhou SPI modulu v C jazyce. V rámci laboratorního cvičení si funkci programů důkladně vyzkoušejte jejich odkrokováním v příslušném programovém prostředí.

#### 7.3.1 Čtení a zápis dat do EEPROM paměti 25LC640

##### Zadání

Vytvořte program, který zapíše do EEPROM paměti 25LC640 na adresu specifikovanou proměnnou *address* hodnotu obsaženou v proměnné *value*. Zápis do paměti provedte pouze v případě, že zapisovaná hodnota je rozdílná od aktuálního údaje uloženého ve specifikované EEPROM paměťové buňce. Tímto lze do velké míry prodloužit životnost paměti snížením počtu provedených přepisů. Výsledky jednotlivých operací prováděných s pamětí odesílejte na konzoli.

##### Řešení

Inicializační část programu nejprve povolí distribuci hodinového signálu do modulu PORTC zápisem 1 do 11. bitu registru `SIM_SCGC5` a modulu SPI0 zápisem 1 do 22. bitu `SIM_SCGC4`. Dále nastaví funkci pinu `PTC5` na `SPI0_SCK` (hodinový signál SPI), `PTC6` na `SPI0_MOSI` (výstup vysílače master rozhraní), `PTC7` na `SPI0_MISO` (vstup přijímače master rozhraní) zápisem hodnoty 2 (funkce pinu `ALT2`) do bitového pole *MUX* příslušných `PORTC_PCRx` registrů, kde  $x = \{5, 6, 7\}$ . Pin `PTC0` zajišťuje funkci výběrového signálu



CS EEPROM paměti a bude proto nakonfigurován do běžného GPIO režimu nastavením bitového pole *MUX* na hodnotu 1 (funkce pinu ALT1). Daný pin musí být nastaven do výstupního režimu zápisem 1 do 0. bitu *GPIOC\_PDDR* registru. Počáteční výstupní hodnota na tomto pinu je inicializována na logickou 1, což odpovídá neaktivnímu stavu EEPROM paměti. Při každé změně logické úrovně na tomto pinu musí následovat krátké čekání (minimálně 500 ns) funkcí *CS\_SetupTime()*, které paměť vyžaduje pro vykonání interních operací. Dále následuje inicializace SPI0 rozhraní do režimu master pomocí funkce *SPI\_MasterInit()*. Dříve než je funkce zavolána, musí se zjistit frekvence sběrnice mikropočítače funkcí *CLOCK\_GetBusClkFreq()* a naplnit konfigurační struktura SPI0 master rozhraní výchozími hodnotami funkcí *SPI\_MasterGetDefaultConfig()*. Po inicializaci datové struktury *spi0\_config* je provedeno přenastavení položky *outputMode* na *kSPI\_SlaveSelectAsGpio* zajišťující deaktivaci hardwarového řízení signálu SS na pinu PTC4 SPI0 rozhraním, protože výběr slave zařízení (signál CS EEPROM) se provádí prostřednictvím pinu PTC0, jak již bylo uvedeno výše. Tímto je inicializační část programu dokončena a následuje vlastní výkonná část programu, která provede zápis hodnoty uložené v proměnné *value* na adresu *address* EEPROM paměti. S ohledem na kapacitu paměti může být zadaná adresa v rozsahu od 0 do 8191 (1FFF<sub>16</sub>).

Před zápisem do paměti se nejprve provede čtení obsahu paměťové buňky na zadané adrese. Každá operace s EEPROM začíná zápisem logické 0 na pin PTC0 ovládající CS signál paměti. Tímto se paměť aktivuje a očekává komunikační aktivitu na SPI rozhraní. Na SPI0 rozhraní se postupně odešlou funkcí *SPI\_WriteBlocking()* 4 bajty dat obsahující: příkaz READ, vyšší bajt adresy, nižší bajt adresy a „dummy“ bajt, který může obsahovat libovolnou hodnotu. Během zápisu na rozhraní se po každém zapsaném bajtu musí počkat na příjem znaku z SPI0 čtením stavových příznaků rozhraní funkcí *SPI\_GetStatusFlags()* a testováním příznaku *kSPI\_RxBufferFullFlag* a poté tento znak z rozhraní přečíst, i když neobsahuje žádnou validní hodnotu. Teprve poslední čtení, po odvysílání „dummy“ bajtu, poskytne hodnotu uloženou v paměti. Následně se CS signál EEPROM nastaví na logickou 1 a komunikace je tímto ukončena. Na konzoli se odešle pomocí funkce *PRINTF()* informační text s aktuálně přečtenou hodnotou z paměti. Ta je porovnána s hodnotou pro zápis uloženou v proměnné *value*. V případě, že jsou rozdílné, provede se její zápis do paměti EEPROM. Před samotným



zápisem se musí nejprve provést povolení zápisu do paměti pomocí instrukce WREN. Průběh odeslání instrukce je obdobný jako u READ, jen se posílá pouze 1 bajt obsahující příkaz. Dále následuje vlastní zápis nové hodnoty spočívající v odeslání celkem 4 bajtů na SPI0 rozhraní: příkaz WRITE, vyšší bajt adresy, nižší bajt adresy a hodnota pro zápis. Opět se musí po odeslání každého bajtu čekat na příjem znaku z SPI0 a přečíst jej příslušnou funkcí. Po odvysílání všech 4 bajtů dat a následném nastavení logické 1 na CS pinu, začne EEPROM paměť údaj ukládat do paměťové buňky. Tato operace může trvat až 5 ms a během ní nelze provádět čtení obsahu paměti. Proto následuje čekání na provedení interních operací čtením obsahu STATUS registru EEPROM s testováním stavu 0. bitu *WIP*. To se provede periodickým odesláním 2 bajtů na SPI rozhraní – příkazu RDSR a jednoho bajtu s libovolnou hodnotou následované čtením přijatého znaku. Po odeslání 2. bajtu je z rozhraní přečten obsah STATUS registru. Jakmile je v bitu *WIP* přítomna 0, je operace zápisu ukončena a EEPROM po nastavení CS na logickou 1 přejde do neaktivního stavu. Zdrojový text programu je uveden ve výpisu programu 7.1.

*Program 7.1:*

```
#include "board.h"
#include "pin_mux.h"
#include "clock_config.h"
#include "fsl_debug_console.h"
#include "fsl_clock.h"
#include "fsl_spi.h"

// Maska EEPROM CS řídicího signálu
#define M_EEPROM_CS      (1)
// Maska WIP bitu ve STATUS registru
#define M_STATUS_WIP     (1)

// Instrukční sada EEPROM
#define EEPROM_READ      (3)
#define EEPROM_WRITE     (2)
#define EEPROM_WRD1     (4)
#define EEPROM_WREN      (6)
#define EEPROM_RDSR      (5)
#define EEPROM_WRSR      (1)
```



*Program 7.1 (pokračování):*

```
// Funkce pro čekání 500 ns pro ustálení CS signálu EEPROM
static inline void CS_SetupTime(void) {
    uint8_t i;
    for (i=0; i<5; i++) __asm("NOP");
}

int main(void) {
    spi_master_config_t spi0_config;
    uint32_t bclk_freq;
    uint16_t address;
    uint8_t i, buff[4], value, r_value, status_reg;

    // Inicializace KL25Z desky, vytvořeno automaticky SDK
    BOARD_InitPins();
    BOARD_BootClockRUN();
    BOARD_InitDebugConsole();

    // Specifikace hodnoty pro zápis a adresy paměťové buňky EEPROM
    value = 0x11;
    address = 0x100;

    // Inicializace SIM, PORTC
    SIM->SCGC5 |= SIM_SCGC5_PORTC_MASK;
    SIM->SCGC4 |= SIM_SCGC4_SPI0_MASK;
    PORTC->PCR[5] = 0; PORTC->PCR[6] = 0;
    PORTC->PCR[7] = 0; PORTC->PCR[0] = 0;
    PORTC->PCR[5] = PORT_PCR_MUX(2);
    PORTC->PCR[6] = PORT_PCR_MUX(2);
    PORTC->PCR[7] = PORT_PCR_MUX(2);
    PORTC->PCR[0] = PORT_PCR_MUX(1);
    GPIOC->PDDR |= M_EEPROM_CS;
    GPIOC->PDOR |= M_EEPROM_CS;
    CS_SetupTime();

    // Inicializace SPI0 rozhraní
    bclk_freq = CLOCK_GetBusClkFreq();
    SPI_MasterGetDefaultConfig(&spi0_config);
    spi0_config.outputMode = kSPI_SlaveSelectAsGpio;
    SPI_MasterInit(SPI0, &spi0_config, bclk_freq);

    // Povolení hodinového signálu PORTC
    // Povolení hodinového signálu SPI0
    // PTC5 nastavení funkce ALT2 = SPI0_SCK
    // PTC6 nastavení funkce ALT2 = SPI0_MOSI
    // PTC7 nastavení funkce ALT2 = SPI0_MISO
    // PTC0 nastavení funkce ALT1 = GPIO
    // PTC0 výstupní režim
    // EEPROM_CS = 1 (EEPROM neaktivní)
    // Čekaj 500 ns
    // Zjištění frekvence sběrnice
    // Inicializace konf. struktury pro master režim
    // PTC4/SPI0_SS v režimu GPIO
    // Inicializace SPI0 v master režimu
```



*Program 7.1 (pokračování):*

```
// Přečtení obsahu paměťové buňky, do které se bude zapisovat
// 1) Příprava dat pro odeslání
buff[0] = EEPROM_READ; // příkaz READ
buff[1] = address / 256; // vyšší bajt adresy
buff[2] = address % 256; // nižší bajt adresy
buff[3] = 0; // dummy bajt

// 2) Zápis instrukce EEPROM_READ, 16bitové adresy, dummy bajtu a čtení přijatých dat z SPI
GPIOC->PDOR &= ~M_EEPROM_CS; // EEPROM CS = 0
CS_SetupTime(); // Čekaj 500 ns
for (i=0; i<4; i++) {
    SPI_WriteBlocking(SPI0, &buff[i], 1); // Zápis 1 B na SPI0
    // Čekaj na příjem bajtu z SPI0
    while((SPI_GetStatusFlags(SPI0) & kSPI_RxBufferFullFlag)==0);
    r_value = SPI_ReadData(SPI0); // Přečtení přijatého bajtu z SPI0
}
GPIOC->PDOR |= M_EEPROM_CS; // EEPROM CS = 1
CS_SetupTime(); // Čekaj 500 ns
PRINTF("\nZ adresy 0x%04X precteno: 0x%02X\n", (int)address, (int)r_value);

// 3) Porovnání přečtené hodnoty r_value s hodnotou value určené pro zápis
if(value != r_value) {
    // Hodnoty jsou různé => provedení zápisu do EEPROM
    PRINTF("Zapisuji novou hodnotu: 0x%02X\n", (int)value);
    // a) Zápis instrukce WREN pro povolení zápisu
    GPIOC->PDOR &= ~M_EEPROM_CS; // EEPROM CS = 0
    CS_SetupTime(); // Čekaj 500 ns
    buff[0] = EEPROM_WREN; // příkaz WREN
    SPI_WriteBlocking(SPI0, &buff[0], 1); // Zápis příkazu na SPI0
    // Čekaj na příjem bajtu z SPI0
    while((SPI_GetStatusFlags(SPI0) & kSPI_RxBufferFullFlag)==0);
    (void)SPI_ReadData(SPI0); // Vyprázdnění přijímače
    GPIOC->PDOR |= M_EEPROM_CS; // EEPROM CS = 1
    CS_SetupTime(); // Čekaj 500 ns
    // b) Zápis nové hodnoty do EEPROM
    GPIOC->PDOR &= ~M_EEPROM_CS; // EEPROM CS = 0
    CS_SetupTime(); // Čekaj 500 ns
    buff[0] = EEPROM_WRITE; // příkaz WRITE
    buff[1] = address / 256; // vyšší bajt adresy
```



*Program 7.1 (pokračování):*

```
buff[2] = address % 256;           // nižší bajt adresy
buff[3] = value;                   // hodnota pro zápis
for (i=0; i<4; i++)
{
    SPI_WriteBlocking(SPI0, &buff[i], 1);    // Zápis 1 B na SPI0
    // Čekej na příjem bajtu z SPI0
    while((SPI_GetStatusFlags(SPI0) & kSPI_RxBufferFullFlag)==0);
    (void)SPI_ReadData(SPI0);                // Vyprázdnění přijímače
}
GPIOC->PDOR |= M_EEPROM_CS;           // EEPROM CS = 1
CS_SetupTime();                         // Čekej 500 ns
// c) Čekání na dokončení zápisu nové hodnoty do EEPROM
do
{
    GPIOC->PDOR &= ~M_EEPROM_CS;          // EEPROM CS = 0
    CS_SetupTime();                       // Čekej 500 ns
    buff[0] = EEPROM_RDSR;                // Příkaz pro čtení STATUS registru
    for (i=0; i<2; i++)
    {
        SPI_WriteBlocking(SPI0, &buff[0], 1);    // Zápis 1 B na SPI0
        // Čekej na příjem bajtu z SPI0
        while((SPI_GetStatusFlags(SPI0) & kSPI_RxBufferFullFlag)==0);
        status_reg = SPI_ReadData(SPI0);        // Čtení přijatého znaku
    }
    GPIOC->PDOR |= M_EEPROM_CS;           // EEPROM CS = 1
    CS_SetupTime();                       // Čekej 500 ns
} while ((status_reg & M_STATUS_WIP) != 0);
}
PRINTF("Hotovo.\n");

for(;;) {                             /* Infinite loop to avoid leaving the main function */
    __asm("NOP");                       /* something to use as a breakpoint stop while looping */
}
}
```



## 7.4 Zadání samostatné práce

1. Vytvořte sadu funkcí pro obsluhu paměti EEPROM 25LC640 připojenou na SPI rozhraní. Implementujte následující funkce:

- *uint8\_t 25LC640\_GetStatus (SPI\_Type \*base)*  
Funkce vrátí přečtenou hodnotu uloženou ve STATUS registru EEPROM paměti. Předávané parametry: bázeová adresa SPI rozhraní.
- *void 25LC640\_WriteEnable (SPI\_Type \*base)*  
Funkce odešle příkaz pro povolení zápisu do EEPROM paměti. Předávané parametry: bázeová adresa SPI rozhraní.
- *uint8\_t 25LC640\_Read (SPI\_Type \*base, uint16\_t addr)*  
Funkce vrátí hodnotu uloženou v paměti na adrese *addr*. Předávané parametry: bázeová adresa SPI rozhraní, adresa paměťové buňky pro čtení.
- *void 25LC640\_Update (SPI\_Type \*base, uint16\_t addr, uint8\_t value)*  
Funkce zaktualizuje obsah paměťové buňky na adrese *addr* hodnotou *value*. Pokud je hodnota uložená v paměti stejná jako zapisovaná, zápis se neprovede. Před návratem počká na dokončení operace zápisu. Předávané parametry: bázeová adresa SPI rozhraní, adresa paměťové buňky, hodnota pro zápis.

2. Upravte ukázkový program 7.1 tak, aby používal funkce vytvořené v předchozím bodu zadání a doplňte možnost zadání adresy a zapisované hodnoty prostřednictvím konzole. Uživatelsky zadávané vstupní údaje ošetřete na neplatné hodnoty.
3. Vytvořte program pro kompletní správu paměti EEPROM 25LC640 ovládaného pomocí konzole. Program bude umožňovat čtení a zápis hodnoty na zadanou adresu paměti, vyplnění zadaného adresového rozsahu paměti konstantou a výpis obsahu celé paměti v hexadecimálním formátu. V jednom řádku výpisu bude obsažena vždy adresa a 8 přečtených bajtů paměti od dané adresy paměti.

Příklad menu programu:

Vyberte požadovanou funkci:

- 1 - Čtení obsahu paměťové buňky
- 2 - Zápis obsahu do paměťové buňky
- 3 - Vyplnění zadaného rozsahu paměti konstantou
- 4 - Výpis obsahu celé paměti

Vaše volba:



## 8 LCD DISPLEJ

Výukový kit je vybaven alfanumerickým monochromatickým LCD displejem FDCC 2004C s pasivní STN zobrazovací maticí se 4 řádky po 20 znacích. Podsvícení displeje zajišťují LED žlutozelené barvy. Interní generátor znakové sady obsahuje 192 různých znaků včetně symbolů a speciálních znaků uložených v paměti ROM řadiče. Uživatelsky lze nadefinovat až 8 vlastních znaků v paměti CGRAM generátoru znaků [15].

Komunikace mikrokontroléru s displejem na výukovém kitu probíhá prostřednictvím 4bitového paralelního rozhraní s obousměrnými datovými linkami DB4 – DB7 a třemi řídicími signály E, RS a RW. Pro řízení podsvětlení displeje je použit pin mikropočítače s funkcí výstupu časovače pro možnost ovládání jasu hardwarovou PWM. Vzhledem k vyšší proudové spotřebě LED je funkce podsvitu bez připojeného externího napájecího adaptéru blokována. Vzájemné propojení vstupně / výstupních signálů je uvedeno v tabulce 18.

LCD signál	MCU pin	Popis
DB4	PTC8	Datový signál paralelního rozhraní DB4
DB5	PTC9	Datový signál paralelního rozhraní DB5
DB6	PTC10	Datový signál paralelního rozhraní DB6
DB7	PTC11	Datový signál paralelního rozhraní DB7
E	PTC12	Řídicí signál pro výběr LCD displeje (1 = aktivní)
RS	PTC13	Řídicí signál pro přístup k registrům displeje
RW	PTA12	Řídicí signál čtení / zápis dat
LCD_BL	PTA13	Ovládání podsvětlení displeje

Tabulka 18: Propojení LCD s mikropočítačem.

### 8.1 Programová obsluha displeje

Pro zajištění programové obsluhy displeje byl vytvořen periferní ovladač obsahující základní funkce pro jeho inicializaci, výstup znaku, textového řetězce a několik dalších podpůrných funkcí popsanych v následující kapitole.

#### 8.1.1 Popis funkcí ovladače displeje

*void LCD\_Init (void)*



*Popis funkce:*

Funkce provede inicializaci modulů SIM, PORTA, PORTC a vlastního LCD displeje. Nastaví 4bitový komunikační režim, font 5 x 8, vypne zobrazení kurzoru a blikání znaku, zapne displej a vymaže jeho obsah. Funkci se nepředávají žádné parametry. Musí se v programu zavolat dříve, než se použijí ostatní funkce ovladače displeje. Funkci se nepředávají žádné parametry.

*void **LCD\_Clear** (void)*

*Popis funkce:*

Smaže obsah displeje a nastaví kurzor na pozici 1. řádku a 1. sloupce. Funkci se nepředávají žádné parametry.

*void **LCD\_SetCursor** (uint8\_t row, uint8\_t col)*

*Popis funkce:*

Provede nastavení kurzoru na požadovaný řádek a sloupec. Číslování řádků a sloupců začíná od 1, souřadný systém má počátek v levém horním rohu. V případě předání neplatných hodnot mimo povolený rozsah, funkce provede jejich korekci do platného rozsahu.

*Parametry funkce:*

<i>row</i>	Specifikuje požadovaný řádek v rozsahu 1 až 4.
<i>col</i>	Specifikuje požadovaný sloupec v rozsahu 1 až 20.

*void **LCD\_Backlight** (bool state)*

*Popis funkce:*

Funkce rozsvítí či zhasne LED podsvětlení displeje na základě předané hodnoty.

*Parametry funkce:*

<i>state</i>	true = zapnutí podsvětlení displeje false = vypnutí podsvětlení displeje
--------------	---



*void LCD\_PutChar (char character)*

*Popis funkce:*

Provede výstup specifikovaného znaku na pozici kurzoru.

*Parametry funkce:*

<i>character</i>	Znak pro zobrazení.
------------------	---------------------

*void LCD\_Print (char \*string)*

*Popis funkce:*

Provede výstup specifikovaného textového řetězce na pozici kurzoru. Funkce neošetřuje případné přetečení řetězce přes hranici řádku.

*Parametry funkce:*

<i>string</i>	Ukazatel na textový řetězec pro zobrazení.
---------------	--

## 8.2 Ukázkové programy

V této kapitole naleznete ukázkové řešené příklady související s programovou obsluhou LCD displeje v C jazyce. V rámci laboratorního cvičení si funkci programů důkladně vyzkoušejte jejich odkrokováním v příslušném programovém prostředí.

### 8.2.1 Výstup znaků a řetězců na LCD

#### Zadání

Vytvořte program, který na první řádek displeje a pátý sloupec vypíše text „UTB ve Zline“ a na druhý řádek na pozice sloupců 8, 10 a 12 znaky F, A, I. Třetí řádek displeje bude celý vyplněn pomlčkami a na čtvrtém řádku bude vypsán obsah proměnné *teplota* datového typu float zaokrouhlený na dvě desetinná místa.

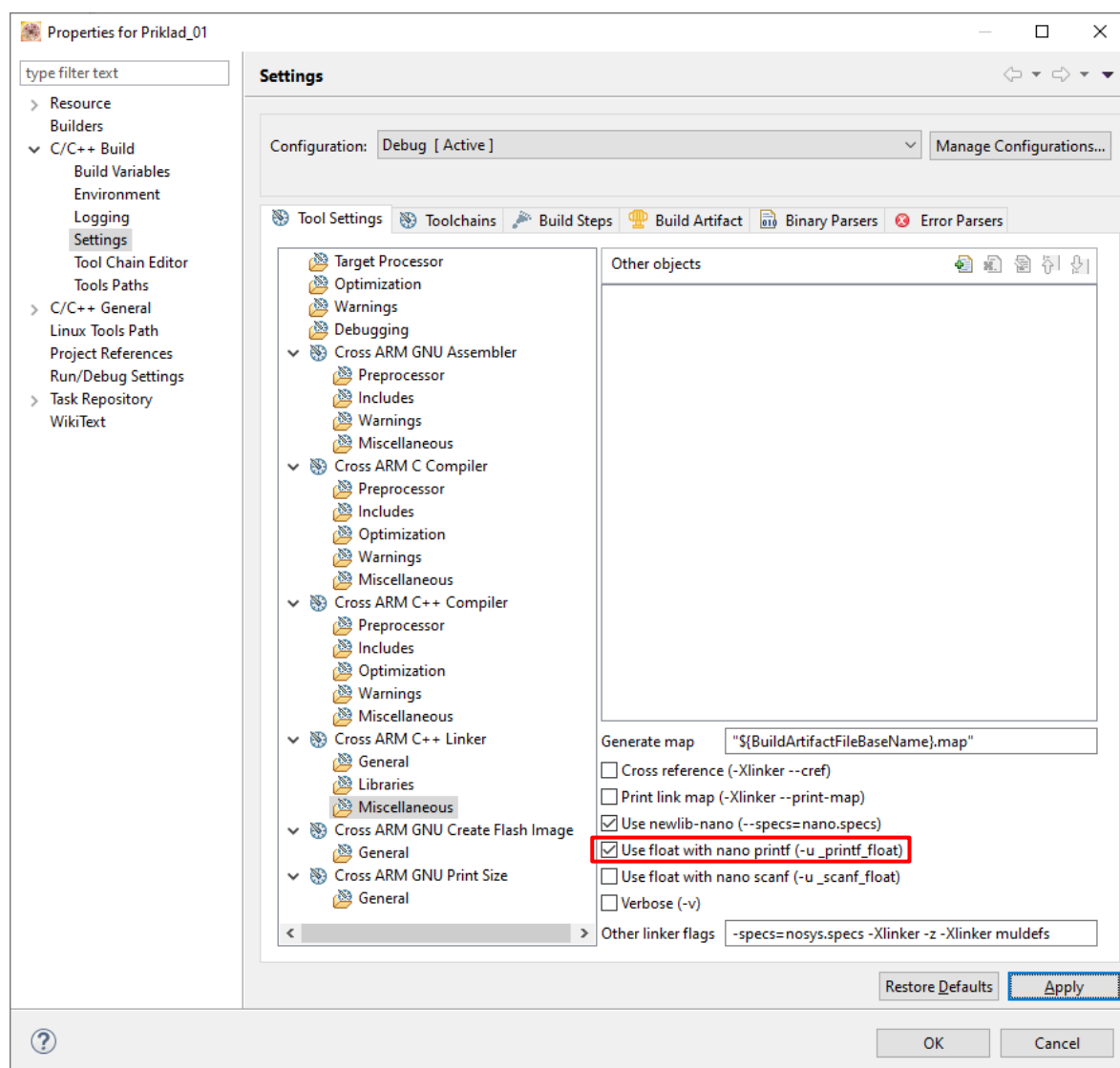
#### Řešení

Inicializační část programu následující po automaticky generovaném programovém kódu SDK sestává pouze z volání funkce pro inicializaci displeje. Od toho okamžiku lze již



používat libovolné funkce ovladače displeje. Použití jednotlivých funkcí je zřejmé z ukázkového programu. V případě, že je zapotřebí vypsát obsah proměnné, je nutno nejprve použít funkci pro formátovaný výstup do textového řetězce *sprintf()*. Její výstup se následně vypíše na displej pomocí funkce *LCD\_Print()*. Zdrojový kód programu je uveden ve výpisu programu 8.1.

Pozn.: V základním nastavení projektu z důvodu šetření paměti, nepodporuje funkce *sprintf()* práci s datovým typem *float*. Pro správnou funkci programu se proto musí v nastavení „C/C++ Build / Settings“ v záložce „Tool Settings“ v položce „Cross ARM C++ Linker / Miscellaneous“ zatrhnout volba „Use float with nano printf“, viz obrázek 47.



Obrázek 47: Zapnutí podpory float pro funkci printf.

*Program 8.1:*

```
#include "board.h"
#include "pin_mux.h"
#include "clock_config.h"
#include "LCD_driver.h"
#include <stdio.h>

int main(void) {

    float teplota = 20.145;
    char tbuff[21];

    // Inicializace KL25Z desky, vytvořeno automaticky SDK
    BOARD_InitPins();
    BOARD_BootClockRUN();
    BOARD_InitDebugConsole();

    LCD_Init(); // Inicializace displeje
    LCD_Clear(); // Vymazání displeje
    LCD_SetCursor(1,5); // Nastavení kurzoru na 1. řádek, 5. sloupec
    LCD_Print("UTB ve Zline"); // Výpis textového řetězce na pozici kurzoru
    LCD_SetCursor(2,8); // Nastavení kurzoru na 2. řádek, 8. sloupec
    LCD_PutChar('F'); // Výpis znaku na displej
    LCD_SetCursor(2,10); // Nastavení kurzoru na 2. řádek, 10. sloupec
    LCD_PutChar('A');
    LCD_SetCursor(2,12); // Nastavení kurzoru na 2. řádek, 12. sloupec
    LCD_PutChar('I');
    LCD_SetCursor(3,1); // Nastavení kurzoru na 3. řádek, 1. sloupec
    LCD_Print("-----");
    sprintf(tbuff, "teplota = %.2f ", teplota); // Formátovaný výstup do řetězce
    LCD_SetCursor(4,1); // Nastavení kurzoru na 4. řádek, 1. sloupec
    LCD_Print(tbuff);
    LCD_PutChar(223); // Výpis znaku s kódem 223 "stupně"
    LCD_PutChar('C');

    for(;;) { // Infinite loop to avoid leaving the main function */
        __asm("NOP"); // something to use as a breakpoint stop while looping */
    }
}
```





### 8.2.2 Výstup znaků přijatých z konzole na LCD

#### Zadání

Vytvořte program, který bude přijímat znaky z konzole a vypisovat je na LCD displeji výukového kitu. Ošetřete přechod na další řádek displeje po překročení konce řádku. Po zaplnění obsahu celého displeje a příchodu dalšího znaku, se provede vymazání displeje a jeho výpis od začátku prvního řádku.

#### Řešení

Obdobně jako v předchozím příkladu se inicializační část programu následující po automaticky generovaném programovém kódu SDK, opět sestává pouze z volání funkce pro inicializaci displeje. Pro přečtení přijatého znaku z konzole je použita funkce *GETCHAR()*, která čeká na příjem znaku. Po jeho příjmu je tento znak uložen do proměnné *znak* a zároveň se zvýší počítadlo přijatých znaků *cnt* o jedničku. Následuje rozhodovací blok, ve kterém se testuje, zda se již znak zapíše za hranici odpovídajícího řádku. Pokud ano, provede se nastavení kurzoru na patřičný nový řádek. V případě posledního řádku se provede vymazání displeje a inicializace počítadla na hodnotu jedna. Zdrojový kód programu je uveden ve výpisu programu 8.2.

Program 8.2:

```
#include "board.h"
#include "pin_mux.h"
#include "clock_config.h"
#include "fsl_debug_console.h"
#include "LCD_driver.h"

int main(void) {
    char znak, cnt = 0;

    // Inicializace KL25Z desky, vytvořeno automaticky SDK
    BOARD_InitPins();
    BOARD_BootClockRUN();
    BOARD_InitDebugConsole();

    LCD_Init(); // Inicializace displeje
    LCD_Clear(); // Vymazání displeje
```



*Program 8.2 (pokračování):*

```
while(1)
{
    znak = GETCHAR();           // Přijem znaku z konzole
    cnt++;                      // Zvýšení počítadla přijatých znaků
    switch(cnt)                 // Testuj hodnotu v cnt
    {
        case 21:                // Znak by se tiskl za koncem prvního řádku
            LCD_SetCursor(2,1);
            break;
        case 41:                // Znak by se tiskl za koncem druhého řádku
            LCD_SetCursor(3,1);
            break;
        case 61:                // Znak by se tiskl za koncem třetího řádku
            LCD_SetCursor(4,1);
            break;
        case 81:                // Znak by se tiskl za koncem čtvrtého řádku
            LCD_Clear();
            cnt = 1;
            break;
    }
    LCD_PutChar(znak);          // Výpis znaku na displej
}

for(;;) {                      /* Infinite loop to avoid leaving the main function */
    __asm("NOP");               /* something to use as a breakpoint stop while looping */
}
```



### 8.3 Zadání samostatné práce

1. Vytvořte program pro měření napětí na výstupu potenciometru, který je připojený na vstupní analogový kanál ADC0\_SE11. Modul A/D převodníku nakonfigurujte na 12bitové rozlišení a jednorázové spouštění převodů. Na displeji s periodou 100 ms zobrazujte na prvním řádku hodnotu přečtenou z výsledkového registru převodníku a na druhém řádku hodnotu napětí ve voltech zaokrouhlenou na dvě desetinná místa. Na třetím a čtvrtém řádku bude zobrazena minimální a maximální hodnota naměřeného napětí taktéž zaokrouhlená na dvě desetinná místa. Při přepočtu výstupní hodnoty A/D převodníku na napětí předpokládejte velikost  $V_{REFH} = 3 \text{ V}$ .

Ukázka zobrazení na displeji:

A	D	C	r	e	g	:	4	0	9	5									
N	a	p	e	t	i	:	3	.	0	0	V								
V	m	i	n			:	0	.	5	2	V								
V	m	a	x			:	3	.	0	0	V								

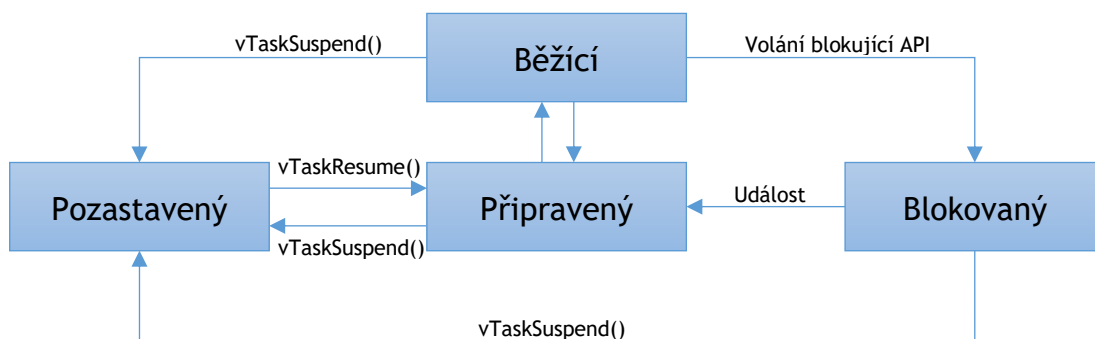
2. Vytvořte program pro jednoduchou meteorologickou stanici pro měření teploty a vlhkosti vzduchu v místnosti. Na displeji bude na prvním řádku zobrazován přesný čas řízený hodinami reálného času PCF8583. Jejich nastavení bude možné tlačítky SW1 a SW2. Tlačítko SW1 bude mít funkci pro zvyšování hodin a tlačítko SW2 pro zvyšování minut. Ošetřete nastavování tak, aby hodnoty nemohly přesáhnout platné meze. Při překročení maxima dané položky ji zpět vynulujte. Na třetím a čtvrtém řádku bude zobrazena naměřená hodnota teploty snímačem LM75A a vlhkosti snímačem HIH6130 zaokrouhlená na 1 desetinné místo. Perioda aktualizace údajů bude 1 s řízená obvodem reálného času.

Ukázka zobrazení na displeji:

P	r	e	s	n	y		c	a	s	:	2	3	:	5	8	:	1	0	
T	e	p	l	o	t	a	:	2	4	.	3		°	C					
V	l	h	k	o	s	t	:	4	0	.	5		%	R	H				

## 9 OPERAČNÍ SYSTÉM FREERTOS

FreeRTOS je operační systém reálného času určený pro embedded systémy na bázi mikropočítačů nebo malých mikroprocesorů vyvíjený a udržovaný společností Real Time Engineers Ltd. V současné době je již implementován na 35 mikropočítačových platformech. Jádro operačního systému a knihovny jsou distribuovány zdarma pod MIT open source licenci. Operační systém podporuje preemptivní multitasking umožňující mít v jednom časovém okamžiku úlohy v různém stavu zpracování. To znamená, že aktuálně zpracovávaná úloha běžící na mikroprocesoru může být přerušena v důsledku příchodu úlohy s vyšší prioritou dříve než je dokončena. Stavy, ve kterých se může úloha (proces) nacházet, jsou vyobrazeny na obrázku 48. Po vytvoření procesů a spuštění plánovače jsou všechny procesy ve stavu „Připravený“, kde jsou dle své priority zařazeny do fronty. Plánovač je z této fronty jednotlivě vybírá a přiděluje jim procesor. Proces aktuálně zpracovávaný procesorem má stav „Běžící“. V tomto stavu může být na jednoprocessorovém systému pouze jeden proces, a proto existuje řada plánovacích strategií, aby u prioritního systému nedocházelo k takzvanému stárnutí procesů (proces díky nízké prioritě není vůbec přidělován na procesor). Ze stavu „Běžící“ se po příchodu úlohy s vyšší prioritou proces vrátí zpět do stavu „Připravený“ a zařadí se do fronty nebo může být převeden do stavu „Blokovaný“ v případě, že zavolal blokující funkci API, v rámci které čeká na přidělení prostředku, událost nebo také na uplynutí časového intervalu. Po obdržení objektu, na který musel čekat, jej operační systém převede do stavu „Připravený“. Do stavu „Pozastavený“ přejde proces po zavolání funkce *vTaskSuspend()* operačního systému a je tím vyřazen z dalšího plánování. Proces ve stavu „Pozastavený“ lze znovu spustit funkcí *vTaskResume()* [16].



Obrázek 48: Stavy procesů ve FreeRTOS [16].

## 9.1 Vybrané funkce pro práci s úlohami

Tělo úlohy je reprezentováno funkcí napsané v C jazyce, která musí vracet void a mít jeden argument typu ukazatel na void, například *void task1(void \*par)*. Velmi důležitý je fakt, že daná funkce nesmí skončit, což se zajistí vnitřní nekonečnou smyčkou. Každá úloha má svůj vlastní zásobník, jehož velikost musí být rovna nebo větší než minimální. Nízká hodnota velikosti zásobníku může vést za běhu programu k velmi těžce laditelným chybám. Priorita úlohy je reprezentována celým číslem, kde 0 vyjadřuje nejnižší prioritu.

```
BaseType_t xTaskCreate ( TaskFunction_t pvTaskCode,  
                        const char * const pcName,  
                        unsigned short usStackDepth,  
                        void *pvParameters,  
                        UBaseType_t uxPriority,  
                        TaskHandle_t *pxCreatedTask )
```

*Popis funkce:*

Funkce vytvoří novou instanci úlohy. Vytvořený proces je automaticky uveden do stavu „Připraven“. V případě, že je jeho priorita ze všech vytvořených procesů nejvyšší, přejde ihned do stavu „Běžící“ [17].

*Parametry funkce:*

<i>pvTaskCode</i>	Název funkce reprezentující proces.
<i>pcName</i>	Jméno procesu pro snadnější ladění programu (uváděno v debuggeru).
<i>usStackDepth</i>	Velikost zásobníku procesu. Minimální velikost je dána konstantou <i>configMINIMAL_STACK_SIZE</i> .
<i>pvParameters</i>	Parametr předávaný procesu. Pokud není použit, zadá se NULL.
<i>uxPriority</i>	Priorita procesu vyjádřená celým číslem v rozsahu od 0 (nejnižší priorita) do <i>configMAX_PRIORITIES - 1</i> (nejvyšší priorita).
<i>pxCreatedTask</i>	Ukazatel na handle procesu, který je funkcí při vytváření procesu inicializován. Pomocí tohoto handle lze pak proces ovládat. Při zadání NULL se handle nebude inicializovat.



**void vTaskDelay** ( TickType\_t xTicksToDelay )

*Popis funkce:*

Přepne proces, který danou funkci zavolal do stavu „Blokovaný“ na čas specifikovaný parametrem *xTicksToDelay*. Po uběhnutí času přejde proces do stavu „Připravený“ [17].

*Parametry funkce:*

<i>xTicksToDelay</i>	Čas vyjádřený v ticích operačního systému. Pro převod časového údaje z milisekund na tiky lze použít makro <i>pdMS_TO_TICKS()</i> .
----------------------	---

**void vTaskDelayUntil** ( TickType\_t \*pxPreviousWakeTime, TickType\_t xTimeIncrement )

*Popis funkce:*

Funkce slouží pro realizaci periodicky spouštěných procesů s pevně danou periodou. Přepne proces, který danou funkci zavolal do stavu „Blokovaný“, dokud není dosaženo nastaveného absolutního času. Následně přejde proces do stavu „Připravený“ [17].

*Parametry funkce:*

<i>pxPreviousWakeTime</i>	Ukazatel na proměnnou s časem vyjádřeným v ticích operačního systému, kdy byl proces probuzen do stavu „Běžící“. Aktuální hodnotu tiku operačního systému lze zjistit funkcí <i>xTaskGetTickCount()</i> . Při dalších volání funkce je tento parametr aktualizován automaticky.
<i>xTimeIncrement</i>	Požadovaná perioda spouštění procesu v ticích operačního systému.

**void vTaskDelete** ( TaskHandle\_t pxTask )

*Popis funkce:*

Odstraní instanci procesu specifikovaného odkazem (handle) získaného při vytváření procesu funkcí *xTaskCreate()*. Odstraněný proces již nemůže být znovu spuštěn, paměť alokovaná procesu operačním systémem je uvolněna [17].

*Parametry funkce:*

<i>pxTask</i>	Handle procesu, který má být odstraněn. Při předání NULL proces odstraní sám sebe.
---------------	--



***void vTaskSuspend ( TaskHandle\_t pxTaskToSuspend )***

*Popis funkce:*

Funkce provede změnu stavu procesu s daným handle na „Pozastavený“. Pozastavený proces již není operačním systémem dále plánován. Z tohoto stavu lze proces přepnout do připraveného pouze funkcí *vTaskResume()* [17].

*Parametry funkce:*

<i>pxTaskToSuspend</i>	Handle procesu, který má být přepnut do stavu „Pozastavený“. Při předání NULL proces pozastaví sám sebe.
------------------------	--

***void vTaskResume ( TaskHandle\_t pxTaskToResume )***

*Popis funkce:*

Funkce změní stav procesu s daným handle z „Pozastavený“ na „Připravený“ [17].

*Parametry funkce:*

<i>pxTaskToResume</i>	Handle procesu, který má být přepnut do stavu „Připravený“.
-----------------------	---

***void vTaskStartScheduler ( void )***

*Popis funkce:*

Spustí plánovač operačního systému. Předtím než je spuštěn, musí být nejdříve vytvořeny procesy funkcí *xTaskCreate()*. Funkce *main()* se po zavolání *vTaskStartScheduler()* již dále neprovádí, protože funkce se nevrátí. V případě návratu z funkce nastal problém s nedostatkem volné paměti pro běh operačního systému [17].

***void taskENTER\_CRITICAL ( void )***

*Popis funkce:*

Funkce pro vstup do kritické sekce. Po jejím zavolání je globálně zakázáno přerušení a nemůže tudíž během provádění kritické sekce nastat změna kontextu [17].





*void taskEXIT\_CRITICAL ( void )*

*Popis funkce:*

Funkce pro vystoupení z kritické sekce. Po jejím zavolání může opět operační systém provádět změny kontextu [17].

## 9.2 Vybrané funkce pro komunikaci a synchronizaci

Operační systém FreeRTOS poskytuje funkce jak pro meziprocesní komunikaci, tak i pro jejich synchronizaci. Pro posílání informací mezi procesy slouží fronta (queue), která může obsahovat více položek. Její obsluha potom probíhá na principu zásobníku typu FIFO (první dovnitř, první ven). Při práci s frontou může nastat případ, že ve frontě není volné místo pro zápis nové položky nebo naopak není k dispozici položka pro čtení. Operační systém daný proces přepne do stavu „Čekající“ na dobu, než se fronta uvolní nebo je k dispozici položka k přečtení. Pro synchronizaci běhu procesů poskytuje FreeRTOS semaforey a mutexy. Mutex je speciální typ binárního semaforu sloužící pro řízení přístupu k prostředku sdíleného více procesy. Může se jednat o přístup na displej, ke globální proměnné, sériovému rozhraní a podobně. Princip spočívá v předávání tokenu, který dává oprávnění k práci se sdíleným prostředkem. Proces, který nezískal token, přejde do stavu „Čekající“, kde setrvá po dobu, než mu bude token předán [16].

### 9.2.1 Funkce pro obsluhu fronty

*QueueHandle\_t xQueueCreate ( UBaseType\_t uxQueueLength, UBaseType\_t uxItemSize )*

*Popis funkce:*

Funkce vytvoří novou frontu a vrátí odpovídající handle. V případě, že funkce vrátí NULL, nebyla z důvodu nedostatku paměti fronta vytvořena. Požadovaná paměť je alokována automaticky operačním systémem [17].

*Parametry funkce:*

<i>uxQueueLength</i>	Udává maximální počet položek, které lze do fronty uložit.
<i>uxItemSize</i>	Velikost položky v bajtech.



*void vQueueDelete ( TaskHandle\_t pxQueueToDelete )*

*Popis funkce:*

Odstraní z paměti frontu specifikovanou příslušnou handle. Funkci lze použít i pro odstranění semaforu [17].

*Parametry funkce:*

<i>pxQueueToDelete</i>	Handle fronty pro odstranění.
------------------------	-------------------------------

*UBaseType\_t uxQueueMessagesWaiting ( const QueueHandle\_t xQueue )*

*Popis funkce:*

Funkce vrátí počet položek aktuálně uložených ve frontě [17].

*Parametry funkce:*

<i>xQueue</i>	Handle fronty.
---------------	----------------

*BaseType\_t xQueueReceive( QueueHandle\_t xQueue,  
void \*pvBuffer,  
TickType\_t xTicksToWait )*

*Popis funkce:*

Funkce přečte položku z fronty a zapíše ji do specifikovaného bufferu. V případě, že se ve frontě položka k přečtení nenachází, přejde proces do stavu „Blokovaný“. V něm setrvá po dobu specifikovanou parametrem *xTicksToWait* nebo do příchodu zprávy [17].

*Parametry funkce:*

<i>xQueue</i>	Handle fronty, ze které se bude číst.
<i>pvBuffer</i>	Ukazatel na buffer pro zápis přečtených dat.
<i>xTicksToWait</i>	Specifikace doby čekání na příjem položky v ticích operačního systému v případě, že je fronta prázdná. Pro časově neomezené čekání se předá funkci konstanta <i>portMAX_DELAY</i> . Pro návrat bez čekání na položku ve frontě se předá hodnota 0.

*BaseType\_t* **xQueueSendToBack** ( *QueueHandle\_t* xQueue,  
const void \*pvItemToQueue,  
*TickType\_t* xTicksToWait )

*Popis funkce:*

Funkce zapíše položku na konec fronty. V případě, že je fronta plná a nelze tudíž do ní položku zapsat, přejde proces do stavu „Blokovaný“. V něm setrvá po dobu specifikovanou parametrem *xTicksToWait* nebo do uvolnění místa pro zapisovanou položku [17].

*Parametry funkce:*

<i>xQueue</i>	Handle fronty, do které se bude zapisovat.
<i>pvItemToQueue</i>	Ukazatel na proměnnou obsahující položku pro zápis do fronty.
<i>xTicksToWait</i>	Specifikuje dobu čekání na uvolnění místa ve frontě pro zápis položky v ticích operačního systému. Pro časově neomezené čekání se předá funkci konstanta <i>portMAX_DELAY</i> . Pro návrat bez čekání na uvolnění místa ve frontě se předá hodnota 0.

*BaseType\_t* **xQueueReset** ( *QueueHandle\_t* xQueue )

*Popis funkce:*

Odstraní veškerá data z fronty a uvede ji do výchozího stavu [17].

*Parametry funkce:*

<i>xQueue</i>	Handle fronty.
---------------	----------------

### 9.2.2 Funkce pro obsluhu semaforů

*SemaphoreHandle\_t* **xSemaphoreCreateBinary** ( void )

*Popis funkce:*

Vytvoří binární semafor a vrátí jeho handle. Potřebná paměť je automaticky alokována operačním systémem. Při nedostatku paměti funkce vrátí NULL [17].

*SemaphoreHandle\_t* **xSemaphoreCreateCounting** ( *UBaseType\_t* uxMaxCount,  
*UBaseType\_t* uxInitialCount )



*Popis funkce:*

Vytvoří obecný semafor a vrátí jeho handle. Potřebná paměť je automaticky alokována operačním systémem. Při nedostatku paměti funkce vrátí NULL [17].

*Parametry funkce:*

<i>uxMaxCount</i>	Maximální dosažitelná hodnota semaforu. Po dosažení této hodnoty nemůže být semafor předán.
<i>uxInitialCount</i>	Počáteční hodnota semaforu.

*SemaphoreHandle\_t xSemaphoreCreateMutex ( void )**Popis funkce:*

Vytvoří semafor typu mutex a vrátí jeho handle. Potřebná paměť je automaticky alokována operačním systémem. Při nedostatku paměti funkce vrátí NULL [17].

*void vSemaphoreDelete ( SemaphoreHandle\_t xSemaphore )**Popis funkce:*

Odstraní z paměti semafor specifikovaný příslušným handle. Funkci lze použít pro odstranění všech typů semaforu. Semafor nesmí být odstraněn v případě, že je některý z procesů na něm v blokováném stavu [17].

*Parametry funkce:*

<i>xSemaphore</i>	Handle semaforu pro odstranění.
-------------------	---------------------------------

*UBaseType\_t uxSemaphoreGetCount ( SemaphoreHandle\_t xSemaphore )**Popis funkce:*

Vrací hodnotu semaforu specifikovaného příslušným handle. U binárních semaforů je to hodnota 0 nebo 1, u obecných v rozsahu 0 až maximální hodnota specifikovaná při jeho vytváření funkcí *xSemaphoreCreateCounting()* [17].



*Parametry funkce:*

<i>xSemaphore</i>	Handle semaforu, ze kterého bude čtena hodnota.
-------------------	---

*BaseType\_t xSemaphoreGive ( SemaphoreHandle\_t xSemaphore )*

*Popis funkce:*

Předá semafor specifikovaný příslušným handle, který byl předtím úspěšně převzat funkcí *xSemaphoreTake()* [17].

*Parametry funkce:*

<i>xSemaphore</i>	Handle semaforu, který má být předán.
-------------------	---------------------------------------

*BaseType\_t xSemaphoreTake ( SemaphoreHandle\_t xSemaphore,  
TickType\_t xTicksToWait )*

*Popis funkce:*

Převezme semafor specifikovaný příslušným handle. V případě nedostupnosti semaforu přejde proces do blokováného stavu, ve kterém setrvá po dobu určenou parametrem *xTicksToWait* nebo do získání semaforu [17].

*Parametry funkce:*

<i>xSemaphore</i>	Handle semaforu, který má být předán.
<i>xTicksToWait</i>	Specifikuje dobu čekání na semafor. Pro časově neomezené čekání se předá funkci konstanta <i>portMAX_DELAY</i> . Pro návrat bez čekání na semafor se předá hodnota 0.

*Návratové hodnoty:*

<i>pdPASS</i>	Semafor byl úspěšně převzat ve specifikovaném časovém limitu.
<i>pdFAIL</i>	Semafor nebyl úspěšně převzat ve specifikovaném časovém limitu.



### 9.3 Ukázkové programy

V této kapitole naleznete ukázkové řešené příklady používající pro svůj běh operační systém reálného času FreeRTOS. V rámci laboratorního cvičení si funkci programů důkladně vyzkoušejte v příslušném programovém prostředí.

#### 9.3.1 Použití fronty pro předávání dat mezi procesy

##### Zadání

Vytvořte program s použitím operačního systému reálného času FreeRTOS zajišťující funkci hodin se zobrazením času ve formátu hod:min:sec na prvním řádku displeje s aktualizací údaje jednou za sekundu a zobrazování převedené hodnoty A/D převodníkem z analogového kanálu ADC0\_SE11 každých 100 ms na druhém řádku displeje. Činnosti programu rozdělte na 3 dílčí procesy, které budou komunikovat prostřednictvím fronty.

##### Řešení

Na základě analýzy zadání musí být nejprve navržena struktura programu z pohledu činností jednotlivých procesů a jejich vzájemné komunikace. Ze zadání vyplývá, že program bude možno rozdělit na 3 základní činnosti: běh hodin, provádění A/D převodů na zadaném analogovém kanálu a zobrazování odpovídajících údajů. Program bude tedy realizován pomocí tří procesů. První proces bude zajišťovat funkci hodin periodickou inkrementací příslušné proměnné každou jednu sekundu a tento časový údaj odesílat do fronty. Druhý proces zajistí provádění periodického spouštění A/D převodu s periodou 100 ms a odesílání převedeného údaje do fronty. Třetí proces bude zobrazovat na LCD displeji potřebné údaje, které bude číst z fronty. Díky použití fronty pro předávání údajů nemůže nastat kolize při přístupu ke sdílenému prostředku, protože s displejem pracuje pouze jeden proces.

V úvodní části programu je provedena inicializace vývojové desky automaticky vygenerovanými funkcemi SDK. Za nimi následuje povolení hodinového signálu modulům ADC0 a PORTC, které jsou nezbytné pro funkci A/D převodníku. Vlastní A/D převodník je nastaven dle doporučených hodnot řídicích registrů uvedených v kapitole 3 na jednorázové spouštění převodů a 10bitové rozlišení. Následuje inicializace LCD, jeho vymazání a výpis statických textů na 1. a 2. řádek displeje. Tímto je základní inicializace programu dokončena



a mohou se již začít vytvářet úlohy (procesy) operačního systému. Procesy se vytvářejí pomocí funkce *xTaskCreate()*. Význam jednotlivých parametrů je popsán v kapitole 9.1. Největší pozornost je třeba věnovat třetímu předávanému parametru, kterým je velikost zásobníku procesu. Jeho minimální velikost je dána konstantou *configMINIMAL\_STACK\_SIZE*, ke které se dále přičte hodnota s ohledem na množství lokálních proměnných procesu a náročnosti použitých funkcí na zásobníkovou paměť. Obecně platí, že nejvíce náročné na zásobník jsou funkce pro formátovaný výstup znaků, jako jsou například *sprintf()* a *PRINTF()*. Proto se u procesu pro zobrazování údajů na displeji přičítá k minimální velikosti zásobníku největší hodnota. Dále je vytvořena fronta umožňující uložení až dvou zpráv, každá o velikosti odpovídající struktuře pro předávání údajů mezi procesy. Jako poslední se spustí plánovač operačního systému funkcí *vTaskStartScheduler()*. Od tohoto okamžiku je funkce *main()* zastavena a operační systém již sám řídí přidělování procesoru úlohám.

Proces *Task1* v nekonečné smyčce inkrementuje každých 1000 ms proměnou *cas* a odesílá ji bez čekání pomocí funkce *xQueueSendToBack()* do fronty. Po překročení maximálního počtu sekund za 24 hodinový cyklus, je proměnná vynulována. Zobrazovací proces rozpozná typ dat prostřednictvím identifikátoru obsaženého v zasílané zprávě. Časový údaj má identifikátor 1, hodnota z A/D převodníku má identifikátor 2. Přesného časování úlohy je dosaženo použitím funkce *vTaskDelayUntil()*, která převede proces do blokováného stavu na specifikovanou dobu v ticích od okamžiku přepnutí procesu do stavu „Běžící“. Tím je kompenzován vliv délky zpracování těla smyčky.

Proces *Task2* funguje na stejném principu jako proces *Task1*, jen je nastavena kratší perioda provádění těla cyklu na 100 ms. V těle cyklu se provádí jednorázové spuštění A/D převodu na kanálu ADC0\_SE11, čekání na dokončení převodu a odeslání získaného údaje do fronty. V tomto procesu je použito odesílání do fronty s časově neomezeným čekáním na její uvolnění. Příslušná odesílaná zpráva má identifikátor roven 2.

Proces *Task3* v nekonečné smyčce čeká pomocí funkce *xQueueReceive()* s časově neomezeným čekáním na příjem zprávy do fronty. Po jejím příjmu otestuje hodnotu identifikátoru zprávy a podle něj aktualizuje přijatou hodnotou časový údaj na prvním řádku displeje nebo výsledek A/D převodu na druhém řádku displeje.

Zdrojový kód programu je uveden ve výpisu programu 9.1.





*Program 9.1:*

```
#include <string.h>
#include "board.h"
#include "pin_mux.h"
#include "clock_config.h"
#include "fsl_debug_console.h"
#include "LCD_driver.h"
#include <stdio.h>
/* FreeRTOS kernel includes. */
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "timers.h"

// Priority jednotlivých procesů
#define Task1_PRIORITY (configMAX_PRIORITIES - 1)
#define Task2_PRIORITY (configMAX_PRIORITIES - 2)
#define Task3_PRIORITY (tskIDLE_PRIORITY)

typedef struct {
    uint8_t msg_id;
    uint32_t msg_val;
} qMsg;

// Handle procesů a fronty
TaskHandle_t h_task1, h_task2, h_task3;
QueueHandle_t h_queue1;

// Proces Task1 inkrementuje čas každou 1 s a posílá do fronty
// - nejvyšší priorita
static void Task1(void *pvParameters)
{
    TickType_t wake_time;
    uint32_t cas = 0;
    qMsg s_msg;
    s_msg.msg_id = 1; // identifikátor zprávy pro zobrazovací proces
    wake_time = xTaskGetTickCount();
    while(1) {
        cas = cas + 1; // 24 hodin má 86400 s
        if (cas > 86399) cas = 0;
    }
}
```



*Program 9.1 (pokračování):*

```
s_msg.msg_val = cas;
// Zápis zprávy do fronty bez čekání
(void)xQueueSendToBack(h_queue1, &s_msg, 0);
// Proces bude periodicky probouzen každých 1000 ms
vTaskDelayUntil(&wake_time, pdMS_TO_TICKS(1000));
}
}

// Proces Task2 provádí A/D převod a výsledky posílá do fronty
// - druhá nejvyšší priorita
static void Task2(void *pvParameters)
{
    TickType_t wake_time;
    qMsg s_msg;
    s_msg.msg_id = 2; // identifikátor zprávy pro zobrazovací proces
    wake_time = xTaskGetTickCount();
    while(1) {
        ADC0->SC1[0] = 11; // Spust' převod na kanálu 11 (potenciometr)
        while((ADC0->SC1[0] & (1<<7))==0); // Čekej na dokončení převodu
        s_msg.msg_val = ADC0->R[0]; // Ulož výsledek převodu
        // Zápis zprávy do fronty s časově neomezeným čekáním
        (void)xQueueSendToBack(h_queue1, &s_msg, portMAX_DELAY);
        // Proces bude periodicky probouzen každých 100 ms
        vTaskDelayUntil(&wake_time, pdMS_TO_TICKS(100));
    }
}

// Proces Task3 zobrazuje obsah zpráv přečtených z fronty na LCD
// - nejnižší priorita
static void Task3(void *pvParameters)
{
    qMsg s_msg;
    uint8_t hod, min, sec;
    char tbuff[21];
    while(1) {
        // Časově neomezené čekání na příjem zprávy
        (void)xQueueReceive(h_queue1, &s_msg, portMAX_DELAY);
        switch(s_msg.msg_id)
        {
```



*Program 9.1 (pokračování):*

```
case 1:
    // Zobrazení aktuálního času
    hod = s_msg.msg_val / 3600;
    min = (s_msg.msg_val % 3600) / 60;
    sec = (s_msg.msg_val % 3600) % 60;
    sprintf(tbuff, "%2d:%02d:%02d", (int)hod, (int)min, (int)sec);
    LCD_SetCursor(1,13); LCD_Print(tbuff);
    break;
case 2:
    // Zobrazení převedené hodnoty A/D převodníkem
    sprintf(tbuff, "%04d", (int)s_msg.msg_val);
    LCD_SetCursor(2,13); LCD_Print(tbuff);
    break;
}
}
}

int main(void) {
    // Inicializace KL25Z vývojové desky
    BOARD_InitPins();
    BOARD_BootClockRUN();
    BOARD_InitDebugConsole();
    // Inicializace periférií používaných programem
    // Inicializace SIM modulu
    SIM->SCGC6 |= SIM_SCGC6_ADC0_MASK; // Povolení hodinového signálu ADC0 modulu
    SIM->SCGC5 |= SIM_SCGC5_PORTC_MASK; // Povolení hodinového signálu PORTC modulu
    PORTC->PCR[2] = 0; // Digitální funkce na pinu PTC2 vypnuty
    // Inicializace A/D převodníku - jednorázové převody, 10 bitové rozlišení
    ADC0->SC1[0] = 0x1f;
    ADC0->CFG1 = 0x39;
    ADC0->CFG2 = 0;
    ADC0->SC2 = 0;
    ADC0->SC3 = 0x04;
    // Inicializace LCD a výpis statických textů na displej
    LCD_Init(); LCD_Clear();
    LCD_Print("Presny cas:");
    LCD_SetCursor(2,1); LCD_Print("AD hodnota:");
    // Vytvoření úloh
    xTaskCreate(Task1, "Hodiny", configMINIMAL_STACK_SIZE + 20, NULL, Task1_PRIORITY, &h_task1);
```



*Program 9.1 (pokračování):*

```
xTaskCreate(Task2, "Snimani", configMINIMAL_STACK_SIZE + 20, NULL, Task2_PRIORITY, &h_task2);
xTaskCreate(Task3, "Zobraz", configMINIMAL_STACK_SIZE + 100, NULL, Task3_PRIORITY, &h_task3);
// Vytvoření fronty
h_queue1 = xQueueCreate(2, sizeof(qMsg));
// Spuštění plánovače operačního systému
vTaskStartScheduler();

for(;;) {           /* Infinite loop to avoid leaving the main function */
    __asm("NOP");   /* something to use as a breakpoint stop while looping */
}
}
```

**9.3.2 Použití mutexu při přístupu ke sdílenému prostředku****Zadání**

Vytvořte program s použitím operačního systému reálného času FreeRTOS zajišťující funkci hodin se zobrazením času ve formátu hod:min:sec na prvním řádku displeje s aktualizací údaje jednou za sekundu a zobrazování převedené hodnoty A/D převodníkem z analogového kanálu ADC0\_SE11 každých 100 ms na druhém řádku displeje. Činnosti programu rozdělte na 2 procesy, z nichž každý bude aktualizovat dílčí část displeje údaji. Bezkolizní přístup na displej vyřešte pomocí mutexu.

**Řešení**

Zadaná funkce programu je identická jako u příkladu 9.3.1, jen má být stejného výsledku dosaženo odlišným způsobem. Program již nebude obsahovat proces určený speciálně pro výstup údajů na displej. Jeho funkce se rozdělí mezi proces aktualizace hodin a provádění A/D převodů. Tělo těchto procesů zůstává prakticky stejné, jen se místo zápisu údaje do fronty provede výpis na displej. Vzhledem k faktu, že displej se stal sdíleným prostředkem obou procesů, musí před zahájením výpisu, požádat o mutex voláním funkce *xSemaphoreTake()* s časově neomezeným čekáním. Po získání mutexu proces provede veškeré nezbytné operace s displejem a následně mutex vrátí pomocí funkce *xSemaphoreGive()*, čímž přístup opět odblokuje.



Zdrojový kód programu je uveden ve výpisu programu 9.2. Společné části kódu s programem 9.1 jsou pro přehlednost odstraněny.

Program 9.2:

```
#include <string.h>
// ..., viz program 9.1
/* FreeRTOS kernel includes. */
// ..., viz program 9.1
#include "semphr.h"

// Priority jednotlivých procesů
#define Task1_PRIORITY (tskIDLE_PRIORITY - 1)
#define Task2_PRIORITY (configMAX_PRIORITIES - 1)

// Handle procesů a fronty
TaskHandle_t h_task1, h_task2;
SemaphoreHandle_t h_mutex;

// Proces Task1 zobrazuje na 1. řádku LCD aktuální čas, nejnižší priorita
static void Task1(void *pvParameters)
{
    TickType_t wake_time;
    uint8_t hod, min, sec;
    uint32_t cas = 0;
    char tbuff[21];
    wake_time = xTaskGetTickCount();
    while(1) {
        cas = cas + 1; // 24 hodin má 86400 s
        if (cas > 86399) cas = 0;
        hod = cas / 3600;
        min = (cas % 3600) / 60;
        sec = (cas % 3600) % 60;
        sprintf(tbuff, "%2d:%02d:%02d", (int)hod, (int)min, (int)sec);
        // LCD je sdílený prostředek -> žádost o mutex s časově neomezeným čekáním
        xSemaphoreTake(h_mutex, portMAX_DELAY);
        LCD_SetCursor(1,13); LCD_Print(tbuff); // Zobrazení na displeji
        // Práce se sdíleným prostředkem je dokončena -> vrácení mutexu
        xSemaphoreGive(h_mutex);
        // Proces bude periodicky probouzen každých 1000 ms
        vTaskDelayUntil(&wake_time, pdMS_TO_TICKS(1000));
    }
}
```



*Program 9.2 (pokračování):*

```
// Proces Task2 provádí A/D převody a výsledky zobrazuje na 2. řádku LCD, nejvyšší priorita
static void Task2(void *pvParameters)
{
    TickType_t wake_time;
    uint32_t adc_val;
    char tbuff[21];
    wake_time = xTaskGetTickCount();
    while(1) {
        ADC0->SC1[0] = 11; // Spust' převod na kanálu 11 (potenciometr)
        while((ADC0->SC1[0] & (1<<7))==0); // Čekaj na dokončení převodu
        adc_val = ADC0->R[0]; // Ulož výsledek převodu
        // Formátovaný výstup do řetězce
        sprintf(tbuff, "%04d", (int)adc_val);
        // LCD je sdílený prostředek -> žádost o mutex s časově neomezeným čekáním
        xSemaphoreTake(h_mutex, portMAX_DELAY);
        LCD_SetCursor(2,13); LCD_Print(tbuff);
        // Práce se sdíleným prostředkem je dokončena -> vrácení mutexu
        xSemaphoreGive(h_mutex);
        // Proces bude periodicky probouzen každých 100 ms
        vTaskDelayUntil(&wake_time, pdMS_TO_TICKS(100));
    }
}

int main(void) {
    // Inicializace KL25Z vývojové desky
    // ..., viz program 9.1
    // Inicializace periférií používaných programem
    // ..., viz program 9.1
    // Inicializace LCD a výpis statických textů
    // ..., viz program 9.1
    // Vytvoření úloh
    xTaskCreate(Task1, "Hodiny", configMINIMAL_STACK_SIZE + 100, NULL, Task1_PRIORITY, &h_task1);
    xTaskCreate(Task2, "Snimani", configMINIMAL_STACK_SIZE + 100, NULL, Task2_PRIORITY, &h_task2);
    // Vytvoření semaforu typu mutex
    h_mutex = xSemaphoreCreateMutex();
    // Spuštění plánovače operačního systému
    vTaskStartScheduler();

    for(;;) { /* Infinite loop to avoid leaving the main function */
        __asm__("NOP"); /* something to use as a breakpoint stop while looping */
    }
}
```



## 9.4 Zadání samostatné práce

1. Rozšiřte funkcionalitu ukázkového programu 9.1 o možnost zadání aktuálního času pomocí tlačítek SW1 a SW2 implementací dalšího procesu pro zpracování uživatelského vstupu. Tlačítko SW1 bude mít funkci pro zvyšování hodin a tlačítko SW2 pro zvyšování minut. Ošetřete nastavování tak, aby hodnoty nemohly přesáhnout platné meze. Při překročení maxima dané položky ji zpět vynulujte.

2. Přeprogramujte program meteorologické stanice zadaný v rámci minulé samostatné práce tak, aby používal operační systém FreeRTOS. Funkce jednotlivých procesů bude následující:

*Proces 1:* Obsluha hodin reálného času PCF8583 a odesílání aktuálního času s periodou 1 s do fronty.

*Proces 2:* Snímání teploty vzduchu pomocí LM75A a odesílání aktuální hodnoty s periodou 2 s do fronty.

*Proces 3:* Snímání vlhkosti vzduchu pomocí HIH6130 a odesílání aktuální hodnoty s periodou 5 s do fronty.

*Proces 4:* Obsluha uživatelského vstupu (tlačítka SW1 a SW2).

*Proces 5:* Příjem dat z fronty a jejich zobrazování na displeji.

*Proces 6:* Periodické odesílání časové značky, naměřené teploty a vlhkosti na terminál s periodou 10 s.

3. Vytvořte program stopky s rozlišením měřeného času 0,01 s používající operační systém FreeRTOS. Navrhněte vhodnou strukturu procesů a meziprocení komunikaci. Při realizaci použijte nejméně 3 procesy.

Ovládání stopek pomocí tlačítek:

SW1 – Spuštění / zastavení stopek

SW2 – Mezičas 1

SW3 – Mezičas 2

SW4 – Vynulování stopek (aktivní pouze při zastavených stopkách)



## SEZNAM POUŽITÉ LITERATURY

- [1] Freescale Semiconductor. HCS08 Microcontrollers MC9S08GB60 Data Sheet [online]. Rev. 2.3. 2004 [cit. 2019-06-04]. Dostupné z: <http://www.nxp.com>
- [2] Freescale Semiconductor. KL25 Sub-Family Reference Manual [online]. Rev. 3. 2012 [cit. 2019-07-01]. Dostupné z: <http://www.nxp.com>
- [3] Freescale Semiconductor. Kinetis KL25 Sub-Family: 48 MHz Cortex-M0+ Based Microcontroller with USB [online]. Rev. 5. 2014 [cit. 2019-07-04]. Dostupné z: <http://www.nxp.com>
- [4] Freescale Semiconductor. FRDM-KL25Z User's Manual [online]. Rev. 2. 2013 [cit. 2019-07-04]. Dostupné z: <http://www.nxp.com>
- [5] ARM. Cortex™-M0+ Devices: Generic User Guide [online]. 2012 [cit. 2019-09-19]. Dostupné z: <http://www.arm.com>
- [6] Texas Instruments. The RS-485 Design Guide: Application Report [online]. 2016 [cit. 2019-10-03]. Dostupné z: <http://www.ti.com>
- [7] NXP Semiconductors. MCUXpresso SDK API Reference Manual [online]. 2017 [cit. 2019-10-03]. Dostupné z: <http://www.nxp.com>
- [8] Philips Semiconductors. The I2C-bus specification [online]. Version 2.1. 2000 [cit. 2020-01-06]. Dostupné z: <http://www.semiconductors.philips.com>
- [9] NXP Semiconductors. LM75A: Digital temperature sensor and thermal watchdog [online]. 2007 [cit. 2020-01-06]. Dostupné z: <http://www.nxp.com>
- [10] Honeywell. Honeywell HumidIcon™ Digital Humidity/Temperature Sensors: HIH6100 Series [online]. 2015 [cit. 2020-01-14]. Dostupné z: <http://www.honeywell.com>
- [11] Honeywell. I<sup>2</sup>C Communication with the Honeywell HumidIcon™ Digital Humidity/Temperature Sensors [online]. 2012 [cit. 2020-01-14]. Dostupné z: <http://www.honeywell.com>
- [12] NXP Semiconductors. PCF8583 Clock and calendar with 240 x 8-bit RAM [online]. Rev. 06. 2010 [cit. 2020-01-14]. Dostupné z: <http://www.nxp.com>





- [13] Freescale Semiconductor. Using the Serial Peripheral Interface to Communicate Between Multiple Microcomputers [online]. 2004 [cit. 2020-02-03]. Dostupné z: <http://www.nxp.com>
- [14] Microchip Technology. 25AA640A/25LC640A 64K SPI Bus Serial EEPROM [online]. 2012 [cit. 2020-02-03]. Dostupné z: <http://www.microchip.com>
- [15] Fordata Electronics. FDCC2004C Character Type Dot Matrix LCD Module [online]. Rev. 5. 2008 [cit. 2020-03-05]. Dostupné z: <http://www.fordata.cn>
- [16] BARRY, Richard. Mastering the FreeRTOS™ Real Time Kernel: A Hands-On Tutorial Guide [online]. Rev. 5. Real Time Engineers, 2016 [cit. 2020-04-02]. Dostupné z: <http://www.freertos.org>
- [17] Amazon. The FreeRTOS™ Reference Manual: API Functions and Configuration Options [online]. Amazon Web Services, 2017 [cit. 2020-04-02]. Dostupné z: <http://www.freertos.org>



## SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

CPU	Centrální procesní jednotka.
DMA	Přímý přístup do paměti.
EEPROM	Nevolatilní, elektricky mazatelná polovodičová paměť pro čtení dat. Zápis se provádí výrobcem specifikovaným algoritmem a je podstatně pomalejší než čtení. V porovnání s FLASH pamětí umožňuje podstatně vyšší počet přepisů, má menší kapacitu a rychlost zápisu.
FLASH	Nevolatilní, elektricky mazatelná polovodičová paměť pro čtení dat. Zápis se provádí výrobcem specifikovaným algoritmem a je podstatně pomalejší než čtení. Počet zápisů do paměťové buňky je limitován dle výrobní technologie. Používá se pro uložení programu a konstantních datových struktur.
I <sup>2</sup> C	Sériová synchronní komunikační sběrnice pro komunikaci s externími periferiemi na malé vzdálenosti. Slave zařízení jsou na sběrnici adresovány 7 nebo 10bitovou adresou.
LCD	Zobrazovací jednotka na bázi tekutých krystalů.
LED	Dioda vyzařující světlo. Používá se pro indikační i osvětlovací účely.
LSB	Nejméně významný bit.
MSB	Nejvíce významný bit.
RAM	Volatilní polovodičová paměť pro čtení / zápis dat s libovolným přístupem. Používá se pro uložení proměnných programu.
RTOS	Operační systém reálného času.
SDK	Sada softwarových nástrojů pro vývoj aplikací.
SPI	Sériové periferní rozhraní. Slouží pro rychlou synchronní sériovou komunikaci s periferiemi na malé vzdálenosti, většinou v rámci plošného spoje zařízení.
UART	Univerzální asynchronní přijímač / vysílač. Je základem standardních sériových rozhraní jako jsou RS232, RS485 a další.



## SEZNAM OBRÁZKŮ

Obrázek 1: Paměťová mapa mikropočítače MC9S08GB60 [1]. .....	12
Obrázek 2: Registry CPU mikropočítače MC9S08GB60 [1]. .....	13
Obrázek 3: Obsah zásobníkové paměti a SP registru v příkladu 1.8.....	22
Obrázek 4: Ukázka sečtení dvou 1B čísel s 2B výsledkem. ....	24
Obrázek 5: Hlavní okno aplikace CW IDE po spuštění. ....	25
Obrázek 6: Průvodce novým projektem - výběr mikropočítače.....	26
Obrázek 7: Průvodce novým projektem - parametry projektu.....	26
Obrázek 8: Pracovní okna Code Warrior IDE. ....	27
Obrázek 9: Ukázka chybového hlášení překladače. ....	28
Obrázek 10: Okno debuggeru s připojeným Full-Chip simulátorem. ....	29
Obrázek 11: Okno debuggeru po provedení programu. ....	30
Obrázek 12: Vývojový diagram programu pro porovnání dvou čísel.....	31
Obrázek 13: Vývojový diagram programu pro vynulování pole.....	34
Obrázek 14: Struktura registru SIM_SCGC5 [2]. ....	38
Obrázek 15: Struktura registru PORTx_PCRn [2]. ....	39
Obrázek 16: Maskování - nastavení bitu na požadovanou hodnotu.....	41
Obrázek 17: Maskování - zjištění stavu vybraného bitu. ....	42
Obrázek 18: KDS - výběr složky pracovního prostoru.....	45
Obrázek 19: KDS - hlavní okno aplikace. ....	46
Obrázek 20: KDS - průvodce novým projektem krok 1. ....	47
Obrázek 21: KDS - průvodce novým projektem krok 2. ....	47
Obrázek 22: KDS - definice symbolu CLOCK_SETUP.....	49
Obrázek 23: KDS - okno výběru konfigurace ladění.....	51
Obrázek 24: KDS - hlavní okno v pohledu ladění programu.....	51
Obrázek 25: Struktura registru SIM_SCGC6 [2]. ....	56
Obrázek 26: Zjednodušená vnitřní struktura TPM modulu [2].....	66
Obrázek 27: Hranově a středově zarovnaná PWM [2]. ....	67
Obrázek 28: Struktura registrů NVIC IP0 až IP7 [2]. ....	72
Obrázek 29: Struktura registru NVIC ISER [5]. ....	73



Obrázek 30: Propojení dvou zařízení pomocí SCI. ....	82
Obrázek 31: Struktura přenosového rámce SCI.....	82
Obrázek 32: Připojení účastníků na RS485 sběrnici [6]. ....	84
Obrázek 33: KDS - nový projekt s SDK krok 1. ....	92
Obrázek 34: KDS - průvodce novým projektem s SDK krok 2. ....	93
Obrázek 35: KDS - definice symbolů preprocesoru překladače. ....	94
Obrázek 36: Tera Term - nastavení nového připojení.....	95
Obrázek 37: Tera Term - nastavení parametrů COM a terminálu. ....	95
Obrázek 38: Připojení účastníků na I <sup>2</sup> C sběrnici. ....	99
Obrázek 39: Komunikace na I <sup>2</sup> C sběrnici. ....	100
Obrázek 40: LM75A - struktura Pointer registru [9]. ....	102
Obrázek 41: LM75A - struktura Temp registru [9]. ....	102
Obrázek 42: HIH6130 - datový registr [11].....	103
Obrázek 43: PCF8583 - čtení obsahu registru. ....	105
Obrázek 44: Propojení zařízení SPI rozhraním [2]. ....	126
Obrázek 45: Nastavení polarity a fáze hodinového signálu SPI [13].....	126
Obrázek 46: STATUS registr EEPROM 25LC640 [14]. ....	130
Obrázek 47: Zapnutí podpory float pro funkci printf.....	144
Obrázek 48: Stavy procesů ve FreeRTOS [16]. ....	149





## SEZNAM TABULEK

Tabulka 1: Vybrané direktivy překladače. ....	10
Tabulka 2: Zápis hodnot v různých číselných soustavách. ....	11
Tabulka 3: Význam příznakových bitů v CCR registru [1]. ....	15
Tabulka 4: Ukázka použití vybraných instrukcí pro podmíněné skoky. ....	32
Tabulka 5: Přehled nastavení symbolu CLOCK_SETUP. ....	48
Tabulka 6: Přehled použitých kanálů časovačů na kitu. ....	67
Tabulka 7: Nastavení režimu časovače [2]. ....	69
Tabulka 8: Nastavení zdroje spouštění čítače [2]. ....	70
Tabulka 9: Nastavení TPMSRC bitů v SIM_SOPT2 [2]. ....	71
Tabulka 10: Nastavení MUX bitů v PORTx_PCRn pro funkce TPM [2]. ....	71
Tabulka 11: Konfigurace NVIC pro TPM moduly [2]. ....	72
Tabulka 12: Výchozí nastavení sériového komunikačního rozhraní. ....	94
Tabulka 13: Přehled periférií kitu s I <sup>2</sup> C rozhráním. ....	101
Tabulka 14: HIH6130 - význam stavových bitů [11]. ....	103
Tabulka 15: PCF8583 - přehled vybraných registrů [12]. ....	105
Tabulka 16: Přehled periférií kitu s SPI rozhráním. ....	127
Tabulka 17: Instrukce EEPROM paměti 25LC640A [13]. ....	128
Tabulka 18: Propojení LCD s mikropočítačem. ....	141





## SEZNAM PŘÍLOH

Příloha P I: Grafické symboly vývojových diagramů

Příloha P II: Schéma zapojení výukového kitu



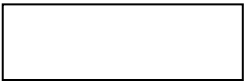
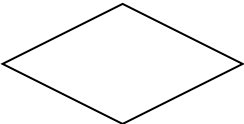
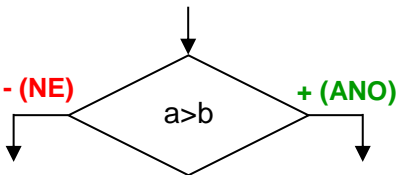
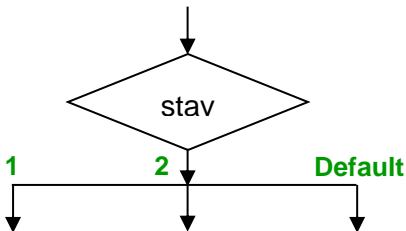
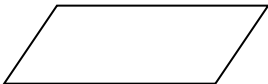

EVROPSKÁ UNIE  
Evropské strukturální a investiční fondy  
Operační program Výzkum, vývoj a vzdělávání

**MŠMT**  
MINISTERSTVO ŠKOLSTVÍ,  
MLÁDEŽE A TĚLOVÝCHOVY



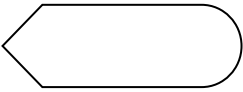


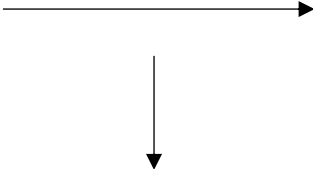


## PŘÍLOHA P I: GRAFICKÉ SYMBOLY VÝVOJOVÝCH DIAGRAMŮ

Značka	Význam
	<b>Postup</b> Může se jednat o matematický výpočet, zapnutí topení, vypnutí čerpadla, odeslání znaku na sériovou linku, atd.
	<b>Rozhodování</b> Program se na tomto místě dle vyhodnocení podmínky rozvětví minimálně do dvou větví.
	Ukázka větvení programu do dvou větví.
	Ukázka větvení programu do více větví. V případě, že není splněna žádná z uvedených podmínek, pokračuje se větví „Default“.
	<b>Automatizovaný vstup/výstup dat</b> Může se jednat o čtení dat například z diskové paměti, čtečky čárových kódů, atd.
	<b>Ruční vstup dat</b> Například z klávesnice, světelného pera, polohovacího zařízení, atd.





	<b>Zobrazení</b>
	<b>Mezní značka</b> Vymezuje například začátek či konec programu nebo podprogramu.
	<b>Spojka</b> Označuje přechod z jedné části vývojového diagramu do jiné. Nutné použít u rozlehlých diagramů na více stranách.
	<b>Spojnice</b> Vyjadřují směr postupu vykonávání algoritmu. Při kreslení se pro přehlednost preferují pravé úhly.





## PŘÍLOHA P II: SCHÉMA ZAPOJENÍ VÝUKOVÉHO KITU

