Kapitola 1

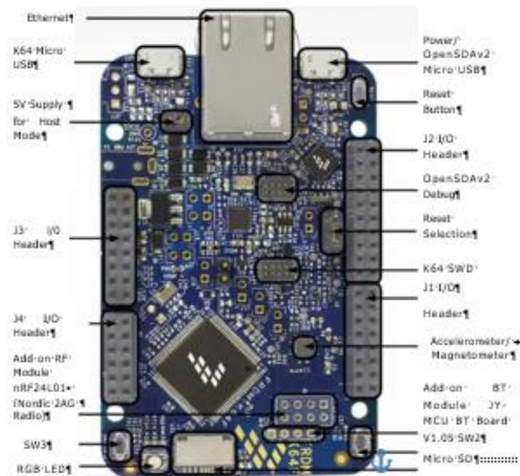
Microcomputer and Development Platform

Mikropočítač (Microcontroller, mikroprocesor, monolitický počítač) je malý počítač implementování na jediném čipu. Musí obsahovat procesor (CPU), paměť RAM a flash a periferních zařízení.

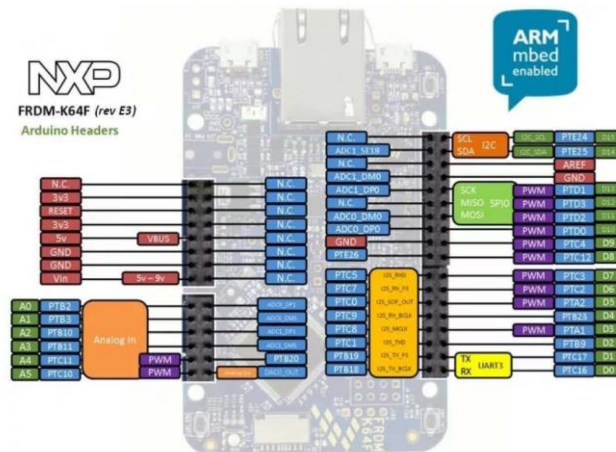
Na trhu je v současné době kromě známých výrobců procesorů pro osobní počítače jako Intel a AMD i řada známých výrobců mikropočítačů, např. Atmel, Microchip, NXP, STMicroelectronics, Texas Instruments, Fujitsu, ARM (nikoli výrobce), atd. Sortiment produktů je velmi široký a pokrývá veškeré požadavky průmyslu.

V posledních deseti letech se 32-bitové procesory ARM značně rozšířili. Jejich masivní produkce velmi snížila jejich cenu a pomalu vytlačují starší 8 a 16-bitové mikropočítače.

Proto se v první polovině laboratorních cvičení v předmětu počítačové architektury a paralelní systémy zaměříme na programování ARMů.



Obrazek 1.1: FRDM-K64F Přehled

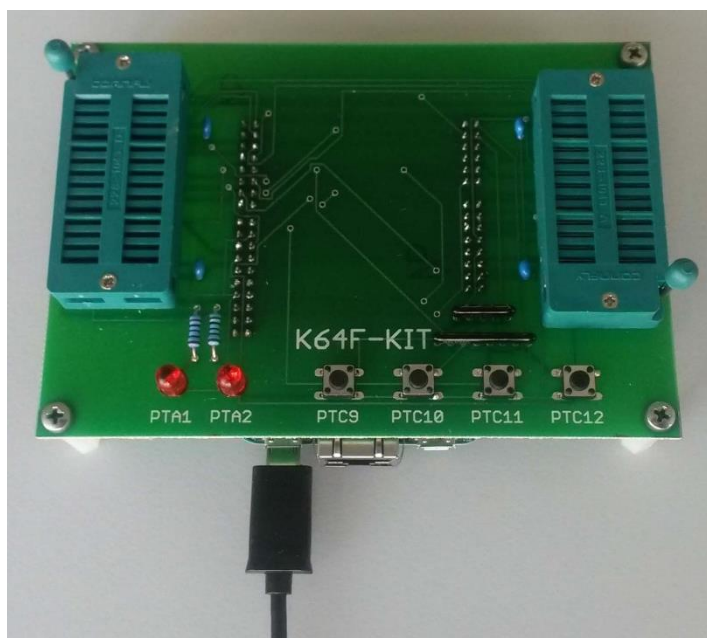


Obrázek 1.2: FRDM-K64F konektory http://development.mbed.org

1.1 Kit FRDM-K64F

Špičkový mikropočítač NXP z rodiny K64 byl vybrán pro laboratorní cvičení. Obsahuje 32-bitové jádro ARM Cortex M4 s 1 MB Flash paměti a 256 KB RAM pracující na frekvenci 120 MHz. Tento mikropočítač je součástí vývojového kitu FRDM-K64F, viz obrázek 1.1, spolu s periferními zařízeními a programovacím a ladícím rozhraním.

Kit FRDM-K64F je znázorněn na obrázku 1.1. Piny jsou znázorněny na obrázku 1.2. Názvy pinů jsou pro následné programování velmi důležité.



Obrázek 1.3: Rozšíření K64F-KIT

Nedílnou součástí tohoto textu jsou také následující katalogové listy:

- frdm-k64f-usrguide.pdf - popis FRDM-K64F
- k64f-refman.pdf - K64 Sub- Family referenční příručka

Více informací o tomto produktu je k dispozici na adrese
http://www.nxp.com

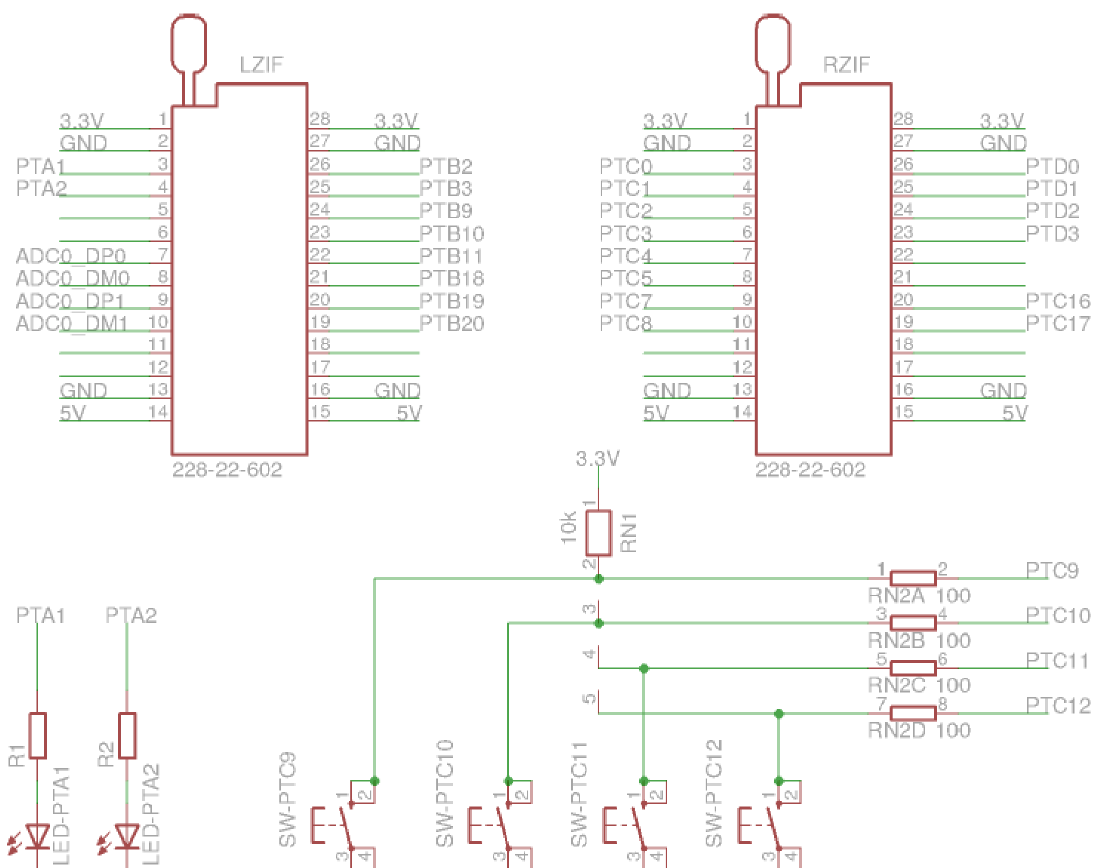
1. 2 Rozšíření K64F-KIT

FRDM-K64F Kit je určen pro laboratorní cvičení. Tato souprava je znázorněna na obrázku 1.3. FRDM-K64F je pod touto soupravou.

Tato sada obsahuje čtyři tlačítka připojená na piny PTC9, PTC10, PTC11 a PTC12. Názvy jednotlivých pinů jde také vidět K64F-KITu. Dvě přídavkem LED diody jsou spojeny s piny PTA1 a PTA2.

Pro laboratorní cvičení jsou velmi důležité dvě ZIF (Zero Insertion Force) patice na vrchu. Jsou určeny pro přídavné moduly používané pro jednotlivé laboratorní cvičení. Tyto moduly jsou popsány dále.

Úplné schéma K64F-KIT je znázorněno na obrázku 1.4.



Obrázek 1.4: K64F-KIT Schéma

1.3 FRDM-K64F Propojení s počítačem

Propojení mezi FRDM-K64F počítače je realizováno pomocí USB kabelu. Na FRDM-K64F musí být USB kabel připojený k levému micro USB konektoru, viz obrázek 1.3

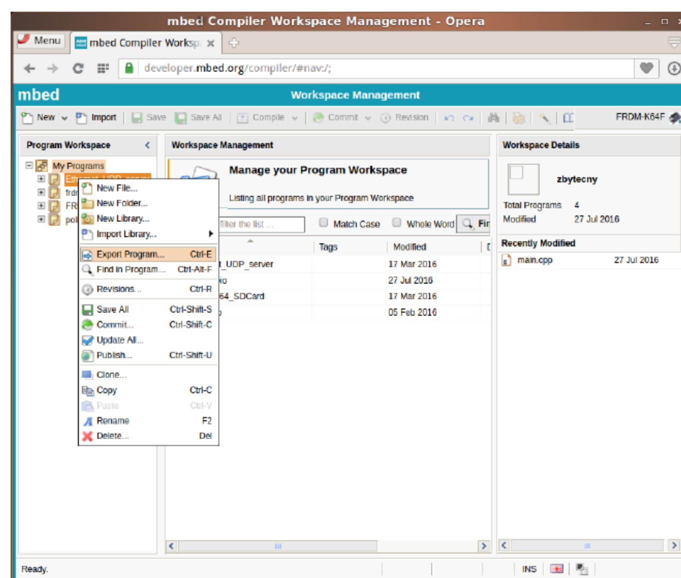
Kapitola 2

Programový rámec ARM mbed

Projekt ARM mbed https://www mbed.com je široce podporován mnoha velkými světovými výrobci. Jeho cílem je vytvořit univerzální programovací rozhraní, které může být použito na velkém počtu vývojových platforem: https://developer.mbed.org/platformy/. Mnoho z těchto desek mají konektory kompatibilní s Arduinem, aby bylo možné používat Arduino shieldy. ARM mbed také podporuje moderní technologie internetu věcí.

2.1 Online Compiler

Webová stránka ARM mbed https://developer.mbed.org/compiler/ nabízí vývojářům on-line překladač s jednoduchým integrovaným vývojovým prostředím (IDE), viz obrázek 2.1.



Obrázek 2.1: ARM mbed online compiler

Tento kompilátor, není určen pro profesionální práci ale pro vzdělávání. Může být použit pro skládání projektu z dostupných modulů, a následnému exportu tohoto projektu do další IDE.

Během laboratorních cvičení nebude použit tento on-line překladač.

2.2 ARM - API (Application Programming Interface)

Při programování v laboratorních cvičeních bude použit pouze malý zlomek celého API. Programování bude zaměřena především na programování portů, které jsou obvykle známé jako GPIO (General Purpose input and output). Také bude nutné použít funkci zpoždění.

2.2.1 GPIO (Porty)

API pro programování GPIO obsahuje tři třídy: `Digitalin, DigitalOut and DigitalinOut.` Stručný popis těchto tří skupin je uvedeno níže:

```
class Digitalin
{
  public:
    // constructor
    Digitalin( PinName pin ); //
    read pin state
    int read();
    operator int ();
};
class DigitalOut
{
  public:
    // constructor
    DigitalOut( PinName pin ); //
    read pin state
    int read();
    operator int ();
    // change output value void
    write( int value );
    operator= ( int value );
};
class DigitalinOut
{
  public:
    // constructor
    DigitalinOut( PinName pin );
```

```
DigitalinOut( PinName pin, PinDirection direction,
         PinMode mode, int value)
// set pin I/O direction
     void output();
     void input();
// read pin state
int read();
operator int ();
// change output value void write( int
value ); operator= ( int value );
};
```

```
Examples:
 DigitalOut led1( PTA1 );
 DigitalOut led2( PTA2 );
 Digitalin sw1( PTC10 );
 Digitalin sw2( PTC10 );
 if ( sw2 ) led2 = 1; // check buttons state
 else led2 = 0;
 led1 = !led1;        // invert state of led1
 while ( sw1 == 0 ); // wait while button is pressed
```

2.2.2 Delay funkce

ARM mbed API obsahuje dvě funkce pro zpoždění:

```
 void wait( float sec );// delay seconds
 void wait_ms( int msec );// delay miliseconds
```

1.2.5 Time Control

Třída Ticker Mbed API obsahuje několik tříd pro řízení časových událostí. Jednou z nich je třída Ticker zaměřená na řízení periodických událostí v programe. Krátké představení třídy Ticker :

```
class Ticker {
public: Ticker() { ... }

// pripojí callback funkciu
void attach_us(Callback func,us_timestamp_t t);

 // pripojí callback šablonu objektu
void attach_us(Callback (T *obj, M method), us_timestamp_t
t)

// pripojí callback funkciu
void attach(Callback func,float t);

 // pripojí callback šablonu objektu
void attach(Callback (T *obj, M method), float t)

// destructor virtual ~Ticker()
{ detach(); }

// odpojí callback
void detach();
...
};
```

Třída Ticker umožnuje připojit pouze jednu callback funkci nebo jeden objekt a jednu jeho metodu.
Následující kód ukazuje možné použití třídy Ticker :

```
DigitalOut g_led1( PTA1 );
DigitalOut g_led2( PTA2 );

void fun_blink() {
 g_led1 = !g_led1;
}

class Blinker {
public:
Blinker( DigitalOut &t_digout ) : m_digout( t_digout ) {}

void blink() {
 m_digout = !m_digout;
}
protected:
   DigitalOut &m_digout; };


 int main() {
   Ticker l_t1, l_t2;
   Blinker l_blinker( g_led2 );

   l_t1.attach_us( callback( fun_blink ), 100000 );
   l_t2.attach_us( callback( &l_blinker , &Blinker::blink
   ), 500000 );
   ...
   while( 1 );
}
```

Výhodou třídy Ticker je, že časovaní je nezávislé na CPU. Callbacky jsú periodicky volané pomocí přerušení.

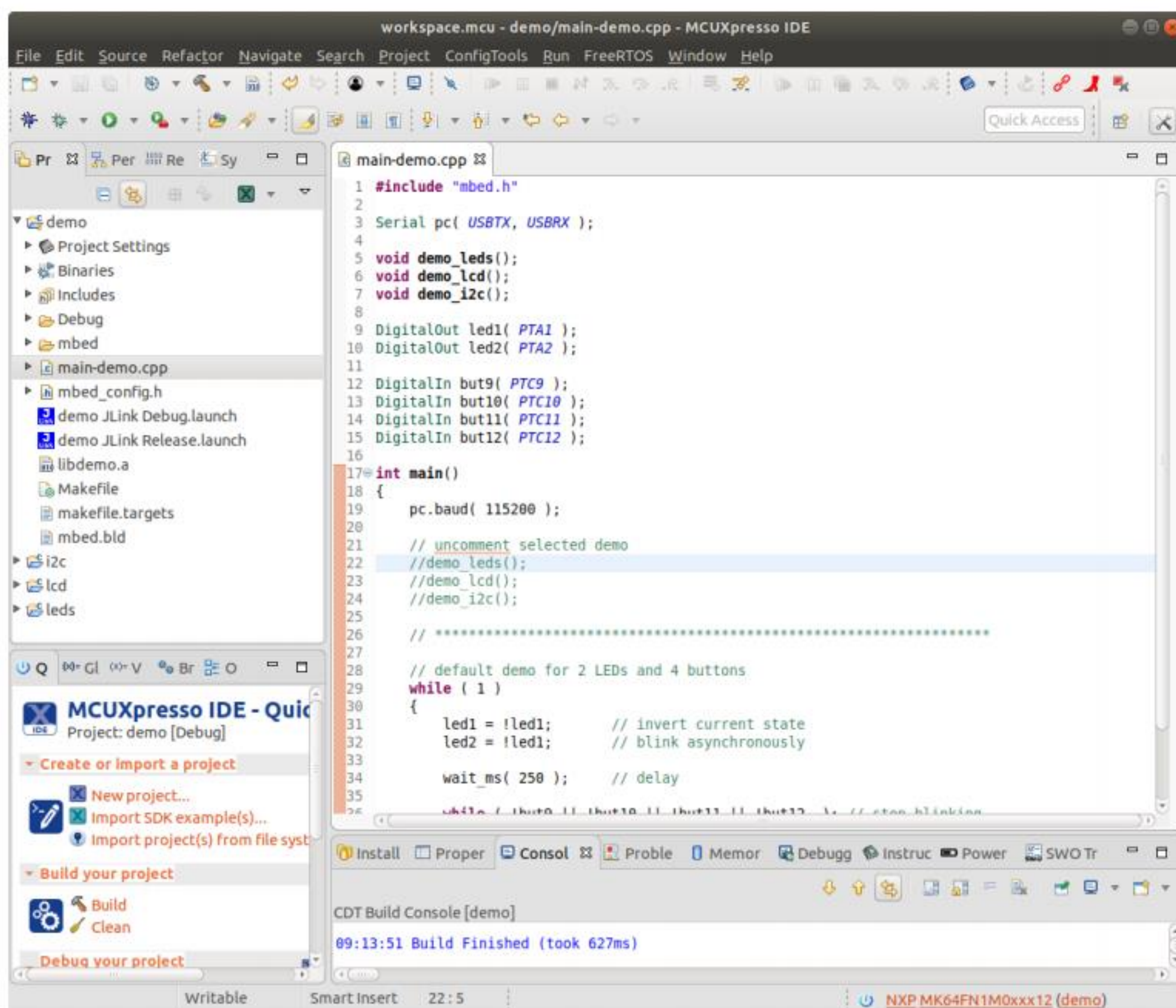
POZOR!!! Callback funkce a metody NESMÍ používat funkce Wait a printf.

Kapitola 3

IDE - MCUXpresso

NXP, výrobce FRDM-K64, podporuje pro vývoj vlastní IDE - MCUXpresso. Toto IDE je založen na platformě Eclipse a je volně k dispozici na webových stránkách http://www.nxp.com. MCUXpresso je moderní IDE splňující všechny požadavky na profesionální programování a její použití je intuitivní, jako použití mnoha dalších známých IDE.

Hlavní okno MCUXpresso je vidět na obrázku 3.1. Na levé straně je zobrazen workspace s projektem. Ve středu je vidět zdrojový kód v okně editoru.



Obrázek 3.1: Hlavní okno MCUXpresso

IDE MCUXpresso vyhledáme přes první ikonu na levém panelu (***Search your computer***). Do vyhledávacího pole zadáme „**mcuxpresso**“. Dole se

zobrazí ikona IDE MCUXpresso, pres kterou studio spustíme. IDE MCUXpresso bude chtít nastavit pracovní workspace (místo na disku, kde jsou uloženy projekty). Připraveny workspace naleznete na stránkách http://poli.cs.vsb.cz/edu/apps/ soubor *apps-labexc.zip*.

3.1 Program Examples - Archiv apps-labexc.zip

Archiv s příklady je připraven pro laboratorní cvičení. Obsahuje jednoduchý zdrojový kód pro snadné spuštění programování v jednotlivých cvičeních. Tento archiv apps-labexc.zip obsahuje celý workspace v adresáři workspace.kds. Při spuštěni IDE MCUXpresso nastavíme cestu k tomuto adresáři (např. nastavíme cestu do složky ***Downloads****,* kde se stáhl archiv s projekty – POZOR, nezapomeňte extrahovat adresář z archivu). Zaklikneme i box „***use this as the default and do not ask again***“ (při dalším spuštění IDE MCUXpresso už nemusíme znova nastavovat workspace). IDE MCUXpresso se spustí. Jelikož se zvolila nova workspace, je třeba poprvé „***obnovit – refresh***“ obsahu projektu, aby se zdrojové kódy zobrazili v editoru – použijeme pravé tlačítko myši na složku příslušného projektu (levé postranní okno) a zvolíme „***refresh***“. Nebo stiskneme klávesu F5. V editoru se zobrazí zdrojový kód příslušného projektu.

Obsah pracovního prostoru /workspace/ workspace.mcu

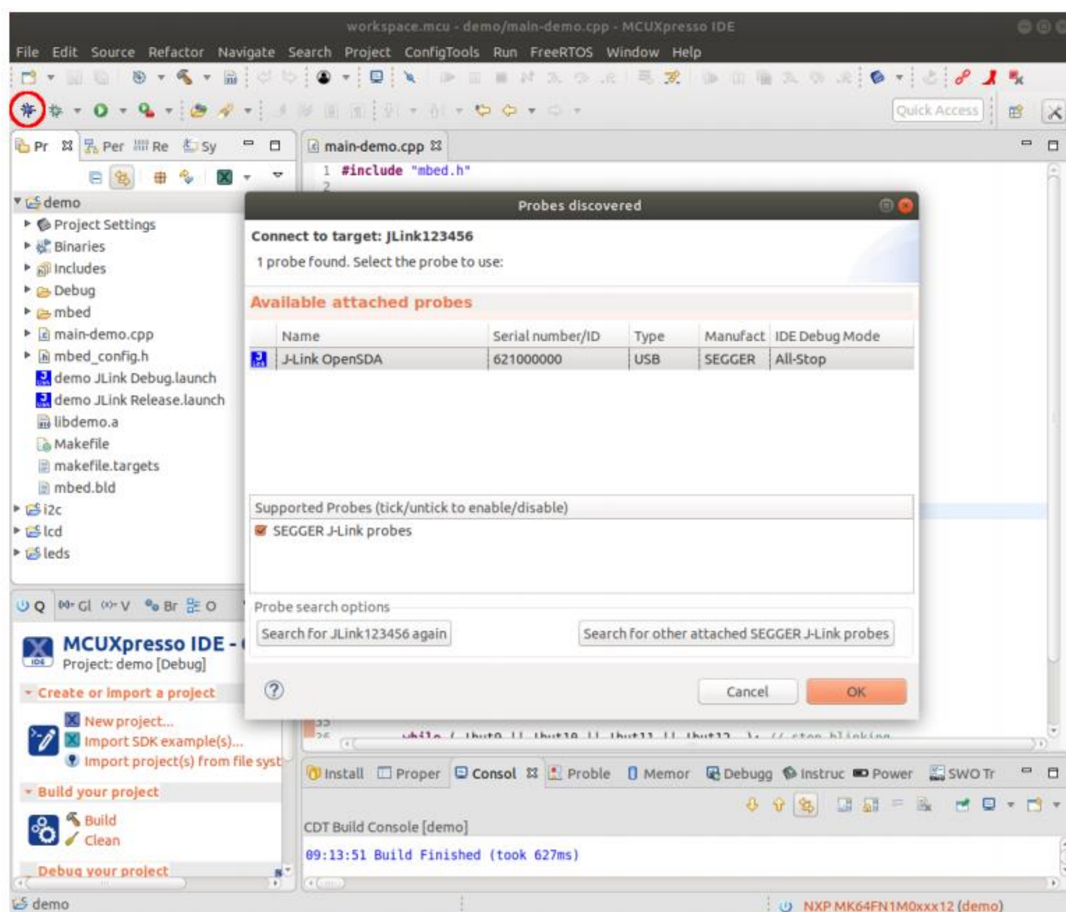
```
./workspace.mcu                -> Workspace for MCUXpresso
./workspace.mcu/leds           -> Directory with project for
LEDs
./workspace.mcu/lcd            -> Directory with project for
LCD module
./workspace.mcu/i2c            -> Directory with project for
I2C bus
./workspace.mcu/demo           -> Demo examples for individual
exercises
 Content of directory . /workspace.mcu/leds:

 ./main-leds.cpp-> Source code for LEDs

 Content of directory . /workspace.mcu/lcd:
```

```
./main-lcd.cpp-> Source code for LCD
./lcd-lib.cpp-> Source code with functions for LCD
./lcd-lib.cpp-> Header file with functions for LCD
./font8x8.cpp-> Source code with fixed size font 8x8
```

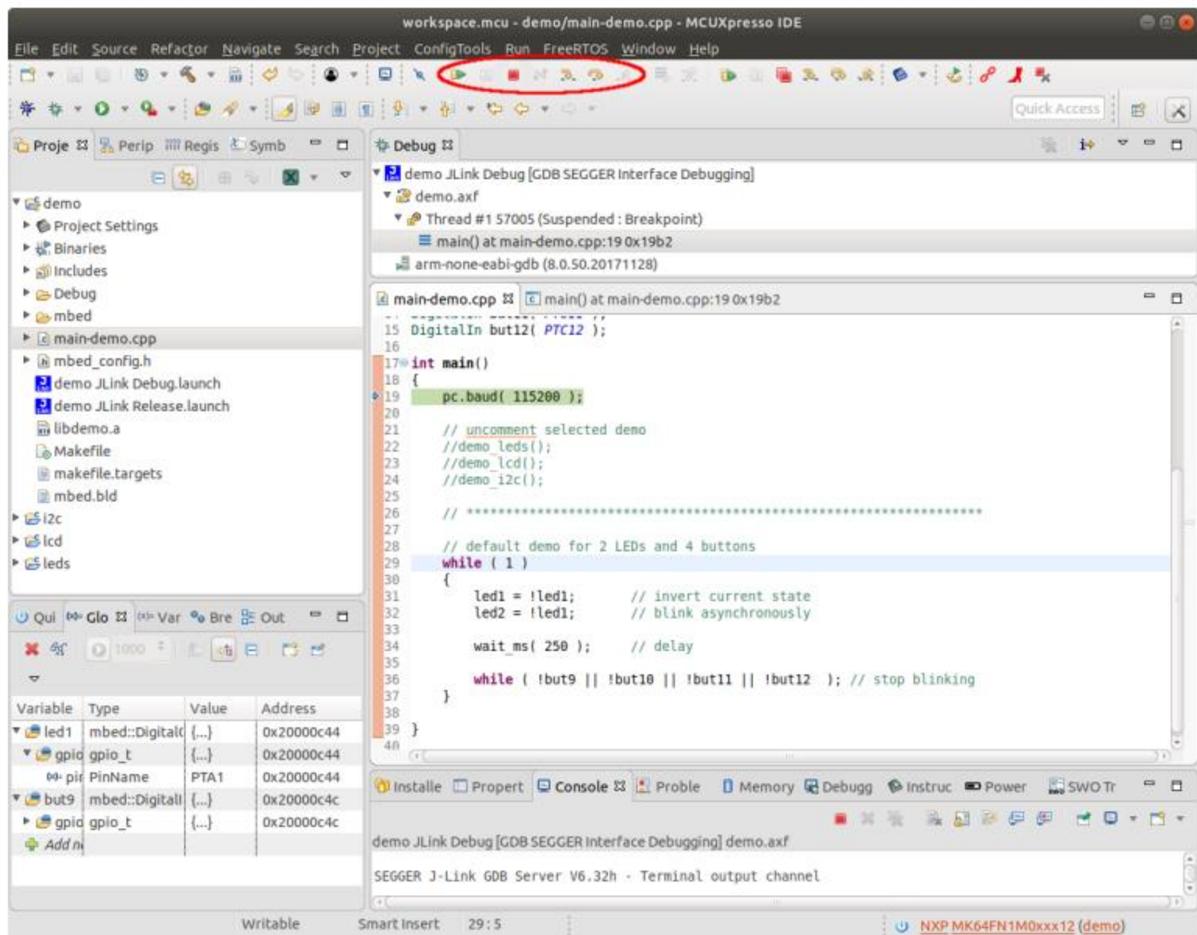
```
Content of directory ./workspace.mcu/i2c:
./main-i2c.cpp      -> Source code for I2C bus
./i2c-lib.cpp -> Source code with functions for I2C bus
./i2c-lib.h    -> Header files with functions for I2C bus
./si4735-lib.cpp    -> Function for SI4735 initialization
./si4735-lib.h      -> Header file with function for SI4735
```

```
Content of directory ./workspace.mcu/demo:

./main-demo.cpp     -> Demo for individual exercises
```

3.2 Run and Debug Programu v MCUXpresso



Obrázek 3.2: **Start debugging v MCUXpresso**



Obrázek 3.3: **Run or debug program v MCUXpresso**

Chcete-li spustit nebo ladit programy v MCUXpresso je nutné provést dva kroky.

1. krok - je vybrat jeden projekt v nabídce Debug (mala šipka vedle brouka) - na obrázku 3.2 je označen červeným kroužkem. Počátek ladění přepne C++ perspektivu (režim MCUXpresso) na ladění. Nové uspořádání MCUXpresso je vidět na obrázku 3.3. Současná perspektiva je vidět v MCUXpresso v pravém horním rohu.
2. krok – spustit, ladit nebo ukončit program. Všechny možné volby jsou vyznačeny na obrázku 3.3 červenou elipsou.

POZOR: Na začátku každého cvičeni, doporučujeme spustit příslušný demo program (main-demo.cpp) pro ověření funkčnosti připojeného kitu s modulem (LED, LCD, I2C). Zdrojový kód upravíme na:

```
int main()
{
        pc.baud( 115200 );
```

```
	// zruste koment u prislusneho dema, ktere chcete spustit...
	// ...pro 1. cviceni je zrusen koment pro demo_leds();.
	demo_leds();
	//demo_lcd();
	//demo_i2c();

	// ************************************************************

	// default demo for 2 LEDs and 4 buttons
	// nezapomente zakomentovat i nasledujici cyklus while...
	// ...ten je z hlediska overeni funkcnosti prislusneho modulu (treba LCD)...
	// ...zbytecny (pro komentovani bloku kodu pouzijte /* ... */).
/*	while ( 1 )
	{
		led1 = !led1;			// invert current state
		led2 = !led1;			// blink asynchronously

		wait_ms( 250 );			// delay

		while ( !but9 || !but10 || !but11 || !but12  ); // stop blinking
	}*/
}
```

Po ověření funkčnosti modulu se přesuneme na příslušný zdrojový kód (podle cvičeni – např. main-leds.cpp), který podle zadaní budete modifikovat (to Vám usnadni prací).

TIPY, KTERE SE MUZOU HODIT:

- před návratem do perspektivy editoru, nezapomeňte zastavit běžící program. Jinak ho znovu nespustíte. Zastaveni provedeme stiskem ikony „červeného čtverce“. Ikona je bud nahoře v perspektivě režimu laděni nebo dole v právo v perspektivě editovaní.

- **v případě, že vám program nefunguje, stáhnete si znova původní zdrojové kódy (je to rychlejší než hledat chybu).**
- **v případě, že přestane fungovat IDE MCUXpresso, program znovu spusťte.**
- **s dobrými tipy, mé neváhejte na cvičeni kontaktovat a já to zde doplním.**

4.1 Výstup na sériové lince

V program je nutné použít metodu printf objektu Serial pro výstup.

```
Serial g_pc(USBTX, USBRX);
 ...
 g_pc.baud(115200);
 g_pc.printf("Hello␣world!\r\n");
 ...
```

Výsup je přesměrován přes sériovou linku do počítače.
Zobrazení výstupu je možné pomocí minicom v oknu terminálu :

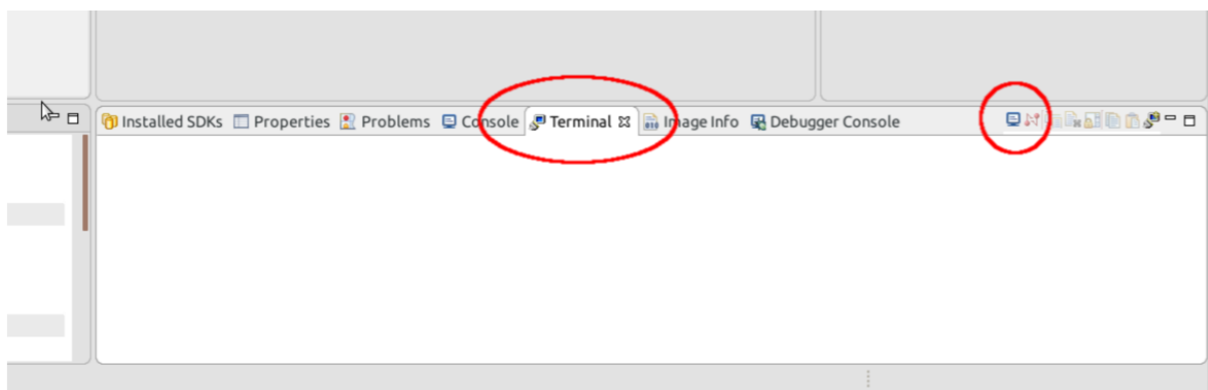
```
$ minicom -D /dev/ttyACM0
```

Pro pomoc stlačte CTRL-A-Z pro vypnutí CTRL-A-Q.
Rychlost musí být nastavená na 115200.

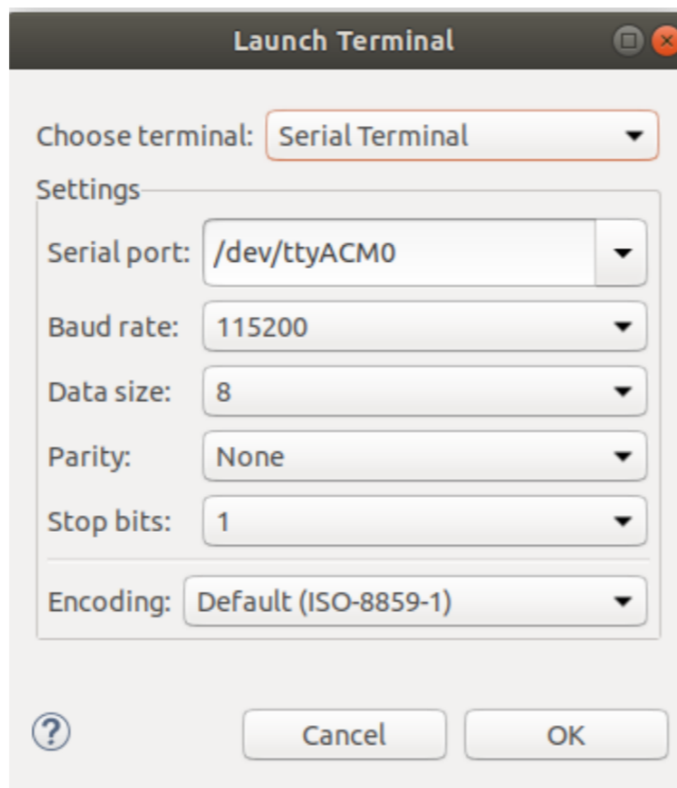
Další možností je použití stty :

```
$ stty -F /dev/ttyACM0 115200 raw -crtscts
$ cat /dev/ttyACM0
```

Třetí alternatívou je použití terminálu přímo v MCUXpresso.
Nastavení je na obrázku



Obrázek 1 : Záložka terminálu v MCUXpresso



Obrázek 2 Nastavení terminálu v MCUXpresso

Kapitola 4

GCC/G++ Compiler - krátký přehled

GNU kompilátoru GCC a G ++ je open source projekt vyvinutý pro kompilaci C a C++ zdrojových kódů pro mnoho procesorů, včetně ARMů. Tento překladač je integrována do MCUXpresso, stejně jako v mnoha dalších IDE. V této kapitole bude stručné shrnutí vlastností jazyka C.

4.1 Datové Typy

Následující datové typy mohou být použity v programech:

- signed char: -128 ÷ 127
- signed short: -32768 ÷ 32767
- signed int, signed long: -2147483648 ÷ 2147483647
- unsigned char: 0 ÷ 255
- unsigned short: 0 ÷ 65535
- unsigned int, unsigned long: 0 ÷ 4294967295

Klíčové slovo unsigned musí být použito pro všechny neznaménkové proměnné. Klíčové slovo signed je volitelné.

4.2 Formát čísel

Čísla mohou být psány ve čtyřech formátech:

Decimalní:

```
int i = 10578;
char ch = -10;
float f = 13.54;
```

Hexadecimalní:

```
int i = 0x12B7;
char ch = 0xF1;
```

Binarní:

```
int i = 0b1101001010010110;
char ch = 0b10001101;
```

Oktalový:

```
int i = 0737;
char ch = 015;
```

Jedno číslo může být psáno ve čtyřech formátech: 10 = 0xA = 0b1010 = 012.
Je to stále jedno a totéž číslo!

4.3 Aritmetické, logické a binární operátory

4.3.1 Aritmetické + - * / %

První čtyři operátoři jsou dobře známé ze základní školy. Pátým operátorem je modulo a výsledkem je zbytek dělení, například: 11 % 2 = 1, 13 % 5 = 3, 31 % 31 = 0.

Krátký zápis:

x++, ++x	x = x + 1
x-- , --x	x = x - 1
x += y	x = x + y
x -= y	x = x - y
x *= y	x = x * y
x /= y	x = x / y
x %= y	x = x % y

4.3.2 Logické && || !

Logical conjunction (AND) Logical disjunction (OR)

false && false => false false || false => false

false && true => false false || true => true

true && false => false true || false => true

true && true => true true || true => true

Logical negation

!true => false !false => true

V jazyce C je pravda jakákoliv nenulová hodnota, false je nulová.

4.3.3 Binary & | ˆ ˜ << >>

Binary AND	Binary OR
0 & 0 => 0	0 | 0 => 0
0 & 1 => 0	0 | 1 => 1
1 & 0 => 0	1 | 0 => 1
1 & 1 => 1	1 | 1 => 1

Binary XOR	Binary negation
0 ^ 0 => 0	~1 => 0
0 ^ 1 => 1	~0 => 1
1 ^ 0 => 1	
1 ^ 1 => 0	

Bit shift
0b01 << 1 => 0b10
0b10 >> 1 => 0b01

Examples:
5 & 3 = 1 because 0b101 & 0b011 = 0b001
5 | 3 = 7 because 0b101 | 0b011 = 0b111
5 ^ 3 = 6 because 0b101 ^ 0b011 = 0b110
~5 = 250 because ~0b101 = 0b11111010,
if 5 is 8-bit unsigned number

5 << 3 = 40 because 0b101 << 3 = 0b101000
5 >> 3 = 0 because 0b101 >> 3 = 0b000000

Short form of binary operations:

x >>= y	x = x >> y
x <<= y	x = x << y
x &= y	x = x & y
x |= y	x = x | y
x ^= y	x = x ^ y

The logical and binary operators are very often mistaking:

```
char x = 5;
char y = 2;

if (x && y) {...} //true
if (x & y) {...} //false
```

Kapitola 5

Laboratorní cvičení s mikropočítačem

5.1 LEDs (Light-Emitting Diodes)

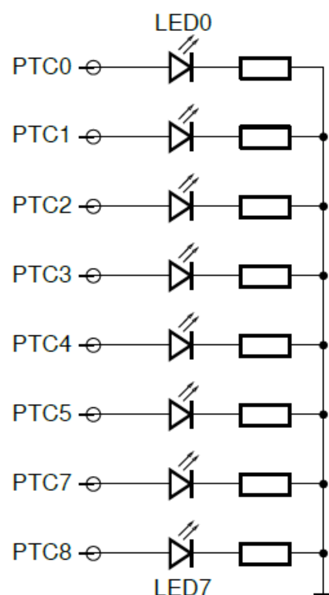
První laboratorní cvičení je zaměřeno na kontrolu LED. V tomto cvičení se používá rozšiřující modul s LED diodami. K64F-KIT s namontovaným LED modulu je znázorněn na obrázku 5.1. Tento modul obsahuje osm červených LED a dvě RGB LED.

.

Obrázek 5.1: K64F-KIT s modulem LED

5.1.1 Připojení červené led k mikropočítači

GPIO piny umožňují přímé připojení LED přes odpor. Odpor omezuje protékající proud přes LED a také zabraňuje přetížení GPIO piny.
Úplné schéma připojení červené LED je vidět na obrázku 5.2. Z tohoto schématu je zřejmé, že všechny LED diody jsou aktivovány (zapnuty) logickou úrovní 1 na každém pinu GPIO. Osm LED LED0 ÷ 7 je připojeno k osmi pinů PTC0 ÷ PTC8 (ne PTC6!).



Obrázek 5.2: Schéma připojení červených LED

5.1.2 LED Ovládání jasu PWM(Pulse Wide Modulation)

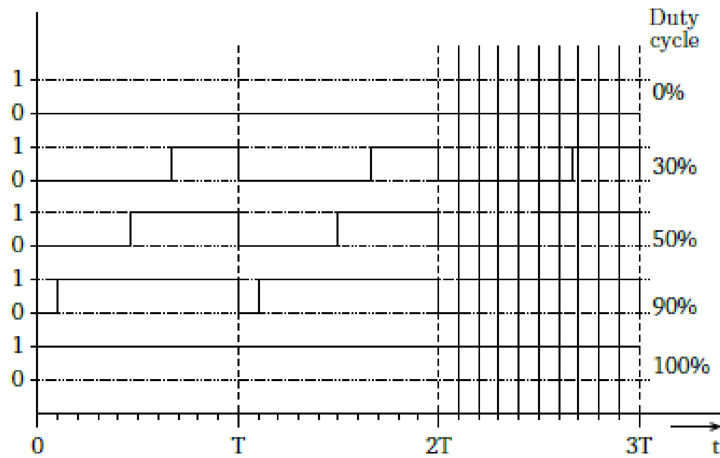
Popis

Změna šířky impulsu modulace (PWM) je nejjednodušší způsob konverze signálu z digitální do analogové formy. GPIO pin umožňuje nastavit výstupní stav pouze na logické úrovně 0 a 1. Ale pravidelné střídání nabízí možnost nastavit různé načasování obou výstupních úrovní. V neposlední řadě toto nepravidelné střídání, známé jako střída, může řídit výstupní výkon (proud nebo napětí).

Proto může být PWM také použit pro ovládání jasu LED diod. Velikost pracovní cyklus je vyjádřena jako část časové období T v procentech. Několik příkladů signálů s různými pracovními cykly je viditelné na obrázku 5.3.

Příklad úlohy

- Ovládání jasu všech LED diod, nastavit různý jas pro každou LED.
- Pomalu, jeden po druhém, zapněte všechny LED diody (pomalu zapnout prvním LED, pak druhou, atd.). Pak pomalu vypnout LED.
- 4 LED pomalu zapnout a současně 4 LED diody pomalu vypnout.
- Pomocí dvou tlačítek výběr LED a pomocí dvou tlačítek nastav její jas



Obrázek 5.3: Timing of signals with different duty cycle

Kroky řešení

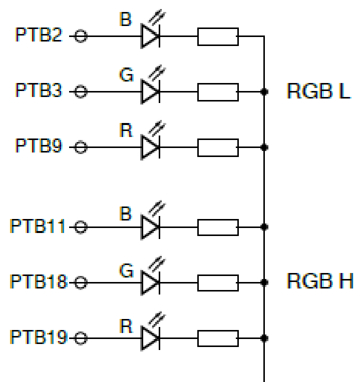
- Blikání s jednou vybranou LED.
- Blikání s vybranými LED se střídou 50% ..
- Najít časové periody $T = 1 / f$ když blikání není vidět.
- Řídit jas jediné LED, pracovní cyklus je stanoven v procentech.
- Ovládání jasu dvou LED diod paralelně, střída pro obě LED je jiná. Rozdělit časové období dle milisekund a kontrolujte pracovní cyklus všech LED diod.
- Dokončení vzorové úlohy.

5.1.3 Míchání barev - RGB LED

Popis

Počítač používá pro vizualizaci ve většině případů princip aditivního míchání barev. Tento princip je založen na směšování ze tří základních barev: červené, zelené a modré (RGB). Tyto tři barvy mohou složit jakoukoliv barvu ve viditelném spektru.

Aditivní míchání barev lze realizovat pomocí RGB LED, kde jas všech barev je ovládán PWM (princip je popsán výše). Schéma zapojení RGB LED k mikropočítači je znázorněno na obrázku 5.4.



Obrázek 5.4: Schéma připojení RGB LED

První RGB LED-L je připojena na piny PTB2, PTB3 a PTB 9.
Druhá RGB LED-H je připojena na vývody PTB 11, 18 a PTB PTB19.

Příklady úlohy

- Zapněte vybranou barvu s různým jasem.
- Pomalu přepínejte mezi dvěma barvami.
- Zapněte RGB LED s barvou dle HTML hodnotu # RRGGBB
- Pomocí tlačítka zvolíte a nastavíte jednotlivé barvy LED RGB.

Řešení

Řešení je podobné PWM.

5.1.4 Ovládání blikání frekvence

Popis

Cílem tohoto úkolu je kontrola bliká LED s danou frekvencí. Potřebná doba pro daný kmitočet je $T=1/f$. Pozor! Tato doba musí být rozdělena do dvou částí, kde bude jedna polovina periody T LED svítí a druhou polovinu T nesvítí.

Příklady úlohy

Úkol řízení frekvence je podobný jako u PWM. Ale cílem je kontrolovat frekvenci, nikoli pracovní cyklus.

Řešení

Existují dvě možnosti.
Prvním řešením je použít jeden čítač pro jednu LED. Tento čítač je zvýšen každou milisekundu a je kontrolováno přetečení.
Druhým řešením je použití jednoho čítač pro všechny LED diody. Modulo bude vypočítán pro všechny LED diody a jejich časových obdobích. V případě, že modulo je 0, pak se změní stav LED.
Druhé řešení je trochu jednodušší, první řešení rovněž umožňuje ovládat fázi blikání.

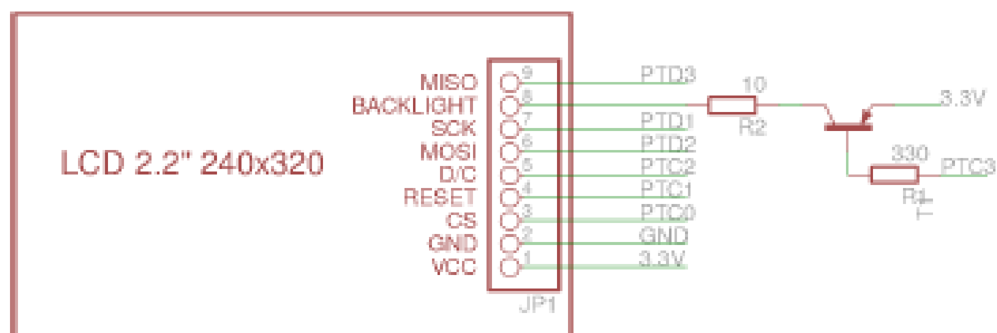
5.2 LCD - RGB Graphic Display

Grafické LCD displeje jsou v současné době používaný v mnoha zařízeních a pomalu vytlačují textový LCD displeje. Nabízejí lepší rozhraní pro uživatele, ale požaduje vyšší výkon připojeného mikropočítače.
Rozšiřující modul pro laboratorní cvičení implementuje malý grafický LCD displej s rozlišením 240 × 320 bodů a 16-bitovou barevnou hloubkou, SPI rozhraní a řídící obvod ILI9341. Tento modul je vidět na obrázku 5.5. Více informací o tomto modulu LCD lze nalézt v datasheet ili9341.pdf.

Obrázek 5.5: K64F-KIT s LCD modul

5.2.1 LCD Module Connection

Propojení mezi LCD modul a mikropočítač je znázorněno na obrázku 5.6.
Rozhraní SCK SPI / MISO / MOSI je přímo napojen na rozhraní SPI mikropočítače. Signál RESET slouží k resetování regulátoru LCD a je připojen k PTC1. Signál select čip CS, připojený k PTC0, slouží k aktivaci SPI rozhraní řadiče LCD. Signál D/C určuje mezi daty a příkazy na SPI. Podsvícení LCD modul je ovládán pinem PTC3 přes tranzistor.



Obrázek 5.6: Schéma připojení LCD modul k mikropočítači

Programovací rozhraní obsahuje pouze tři funkce:

```
// LCD controller initialization
void LCD_init();

// draw one pixel to LCD screen
void LCD_put_pixel( int x, int y, int color );

// clear screen
void LCD_clear();

// HW reset of LCD controller
void LCD_reset();
```

První funkce LCD_Init musí být volána jednou na začáteku programu. Tato funkce inicializuje obvod ILI9341. Funkce LCD_clear vymaže obrazovku.
Funkce LCD_put_pixel vykreslí pixel na pozici [x, y] s danou barvu. Barva je 16-bitová hodnota v podobě RGB 5-6-5. Tato forma je pro přehlednost popsána v tabulce 5.1.

Tabulka 5.1: 16-bit RGB color formát 5-6-5

bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
color	R	R	R	R	R	G	G	G	G	G	G	B	B	B	B	B

Funkce LCD_reset se používá pouze interně pro HW reset řadiče LCD.
Příklady programování jsou v archivu apps-labex.zip.

Matrix Font

Řadič grafického LCD není schopen ve většině případů zobrazit textové informace. Chcete-li zobrazit textové informace je třeba je naprogramovat.
Pro laboratoře zvolíme pevnou velikost písma 8x8, v souboru font8x8.cpp (z https://github.com/dhepper/font8x8), jej naleznete připraven. Ve zdrojovém souboru je písmo uloženo jako dvourozměrné pole 256×8 bajtů. Forma tohoto pole je v následujícím příkladu:

```
uint8_t font8x8[ 256 ][ 8 ] = {
        ....
        {0x3E, 0x63, 0x73, 0x7B, 0x6F, 0x67, 0x3E, 0x00},   //U+0030 (0)
        {0x0C, 0x0E, 0x0C, 0x0C, 0x0C, 0x0C, 0x3F, 0x00}, //U+0031 (1)
        {0x1E, 0x33, 0x30, 0x1C, 0x06, 0x33, 0x3F, 0x00},   //U+0032 (2)
        {0x1E, 0x33, 0x30, 0x1C, 0x30, 0x33, 0x1E, 0x00},   //U+0033 (3)
        {0x38, 0x3C, 0x36, 0x33, 0x7F, 0x30, 0x78, 0x00},   //U+0034 (4)
```

```
{0x3F, 0x03, 0x1F, 0x30, 0x30, 0x33, 0x1E, 0x00},   //U+0035 (5)
{0x1C, 0x06, 0x03, 0x1F, 0x33, 0x33, 0x1E, 0x00},   //U+0036 (6)
{0x3F, 0x33, 0x30, 0x18, 0x0C, 0x0C, 0x0C, 0x00},  //U+0037 (7)
{0x1E, 0x33, 0x33, 0x1E, 0x33, 0x33, 0x1E, 0x00},   //U+0038 (8)
{0x1E, 0x33, 0x33, 0x3E, 0x30, 0x18, 0x0E, 0x00},   //U+0039 (9)
....
};
```

Forma uložených znaky jsou znázorněny na obrázku 5.7.

Index	0 1 2 3 4 5 6 7 bit	
[K][0]		0b00000000 / 0x00
[K][1]		0b00010000 / 0x10
[K][2]		0b00111000 / 0x38
[K][3]		0b00010000 / 0x10
[K][4]		0b00010000 / 0x10
[K][5]		0b00010000 / 0x10
[K][6]		0b00110000 / 0x30
[K][7]		0b00000000 / 0x00

Obrázek 5.7: Formát matice písma

Příklady úlohy

- Program, který zobrazuje jeden znak na pozici [x, y].
- Program, který zobrazuje text horizontálně / vertikálně.
- Pomalé přesouvání textu na obrazovce.
- Čas zobrazen ve formátu HH: MM: SS s blikajícím oddělovačem ':'. Čas se nastaví pomocí tlačítek.
- Stopky.
- Odpočítávání.
- Vykresli čáru / obdélník / kruh.
- Regulovat úrovně podsvícení pomocí PWM.

Úkoly budou v laboratoři kombinované a modifikované.

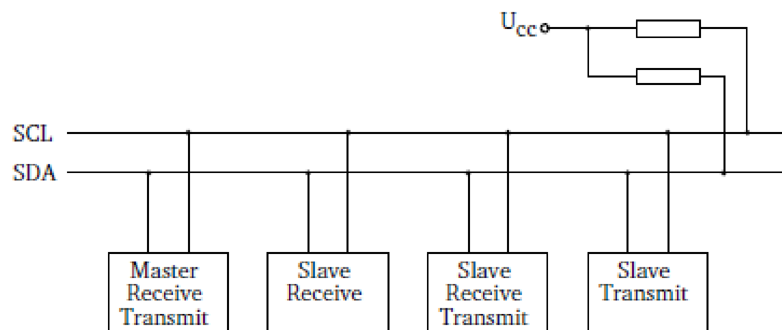
5.3 Sběrnice I2C

5.3.1 Popis sběrnice

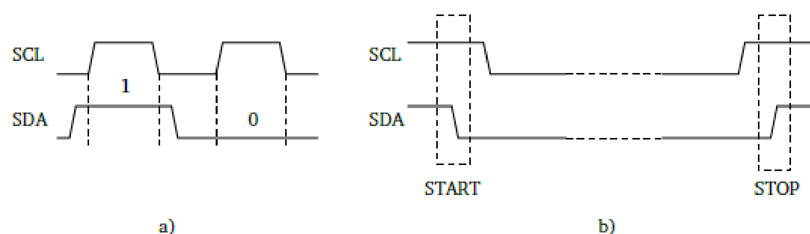
Sběrnice I2C byla navržena firmou Philips Semiconductors pro komunikaci jednočipových mikropočítačů s dalšími číslicovými obvody.
I^2C je obousměrná dvouvodičová sběrnice. Někdy bývá ale nesprávně označována jako sériová linka. Zkratka I2C označuje *Inter Integrated Circuit Bus*. Český překlad by mohl být *meziobvodová sběrnice*.

Základní charakteristika

Sběrnice se skládá ze dvou linek. **SDA** slouží pro obousměrný přenos dat a **SCL** linka slouží jako hodinový signál. Obě linky pracují v napěťových logických úrovních shodně s technologií TTL.
V klidové stavu jsou obě linky v úrovni log. 1. Tento stav je udržovány dvěma zdvihacími (pull-up) rezistory. Výstupy číslicových obvodů jsou řešeny jako výstup typu *otevřený kolektor*. Tím je zaručena obousměrnost linky.
Základní uspořádání systému můžeme vidět na obrázku 5.8. Na sběrnici může být připojeno více zařízení. Obvykle jedno zařízení označované jako *Master* řídí vysílání a příjem zpráv.
Další zařízení, označovaná jako *Slave*, po výzvě nadřízeným obvodem data přebírají (receive), nebo odesílají (transmit). Na sběrnici smí být i více zařízení typy *Master*, ale taková zapojení již nepředstavují jednoduché systémy vhodné pro výuku.



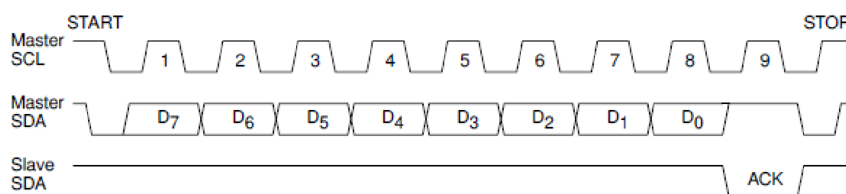
Obrázek 5.8: Uspořádaní typického systému se I^2C



Obrázek 5.9: Průběhy signálů na sběrnici I^2C

Přenos bitů
Přenos bitů probíhá na sběrnici synchronně. Synchronizace se provádí hodinovým signálem SCL. Jeden bit je přenesen vždy v jednom hodinovém cyklu. Jak vypadá přenos hodnoty 1 a 0 můžeme vidět na obrázku 5.9a. Z obrázku je patrné, že datové signály se mění vždy jen v době, kdy je signál SCL v úrovni log. 0. Pokud dochází ke změně signálu SDA v době, kdy ke úroveň signálu SCL v log. 1, jde o signály řídící.

Řídící signály START a STOP
Vzhledem k počtu vodičů sběrnice I2C, můžeme definovat jen dva řídící signály. Těmi jsou signály START (S) a STOP (P). Jak vypadá průběh těchto signálů můžeme vidět na obrázku 5.9b. Slovní popis je následující: z klidového stavu, kdy signál SDA i SCL jsou v log. 1 musíme přejít do komunikačního režimu signálem START. Tím rozumíme přechod signálu SDA z log. 1 na log. 0 při neměnné hodnotě signálu SCL v log. 1. Pak teprve musí následovat i přechod signálu SCL z log. 1 na log. 0.
Chceme-li ukončit komunikaci, musíme uvést obě linky SDA i SCL do klidového stavu. Učiníme tak signálem STOP, který představuje nejprve nastavení SCL na log. 1 a poté nastavení i SDA na log. 1.
Komunikace po bajtech
Při přenosu dat mezi signály START a STOP není množství přenesených dat nijak omezeno. Data se ovšem nepřenáší po jednotlivých bitech, ale po celých bajtech. Průběh celého přenosu je vidět na obrázku 5.10. Z průběhu signálů a jejich popisu je patrné, že významově nejvyšší bit se přenáší jako první.



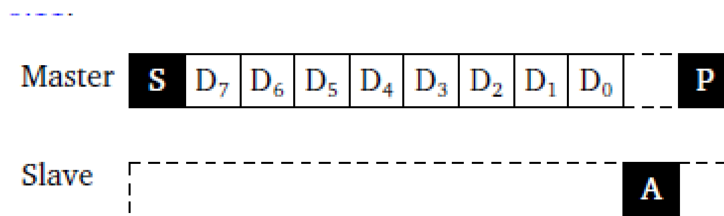
Obrázek 5.10: Odeslání jednoho bajtu i s potvrzením na sběrnici I2C

Potvrzování
Na obrázku 5.10 je zobrazen nejen přenos jednoho bajtu, ale i další důležitá součást přenosu informace. Tou je potvrzování. Každý obvod musí příjem bajtu povrdit bitem ACK, což je log. 0 linky SDA v následujícím hodinovém cyklu po odvysílání bajtu. Potvrzení může být i NACK, tedy negovaný ACK, což je hodnota log. 1 linky SDA (master tak naznačuje, že ukončuje přenos dat).
Všimněte si na obrázku 5.10 chování vysílače (Master) v devátém hodinovém cyklu, kdy se očekává signál ACK od přijímače (Slave). Vysílač nastaví SDA na

log. 1, aby jeho výstup neovlivňoval linku SDA (viz výše informace o výstupu typu otevřený kolektor). Master musí nastavit SDA do klidového stavu na log. 1 před příjmem každého bitu od podřízeného obvodu.

Zjednodušené zobrazení

Pro další popis komunikace budeme používat zjednodušené zobrazení, jako na obrázku 5.11.

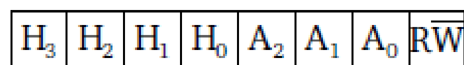


Obrázek 5.11: Zjednodušené zobrazení průběhu komunikace na lince I^2C

Adresování

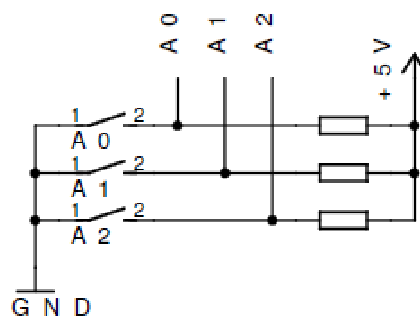
Na sběrnici I2C může být připojeno více obvodů současně, proto musí mít každý obvod svou jednoznačnou adresu, aby jej bylo možno „oslovit“.

Každá adresa má osm bitů, které jsou rozděleny podle obrázku 5.12. Bity označené $H_0 \div H_3$ jsou dány výrobcem a nalezneme je vždy v technickém



Obrázek 5.12: Formát adresy obvodu na I^2C

manuálu daného obvodu. Bity $A_0 \div A_2$ si při zapojení obvodu nastaví uživatel. Na vývojovém KITu se bity $A_0 \div A_2$ nastavují pomocí třech přepínačů dle zapojení na obrázku 5.13. Ze zapojení je patrné, že zapnutý přepínač znamená logickou úroveň 0. Na uvedený přepínač jsou připojeny oba dále popisované obvody.

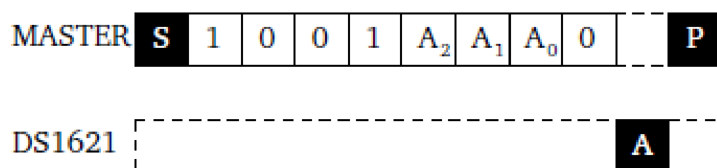


Obrázek 5.13: Určení adresových bitů A_0-A_2 pomocí přepínačů

Poslední bit adresy je $R/\overline{W}$. Tímto bitem říkáme, zda následující data budeme z pohledu řídícího obvodu do příslušného obvodu zapisovat ($R/\overline{W}$ = 0), nebo budeme data číst ($R/\overline{W}$ = 1).

5.3.2 Ověření adresy

Nejjednodušším příkladem komunikace je ověření („zaadresování“), zda daný obvod má skutečně přidělenou adresu dle dokumentace a uživatelského nastavení. Povinností obvodu, který přijme jeden bajt, je příjem potvrdit signálem ACK. Pokud má náš obvod DS1621 adresu $A_0 \div A_2$, pak musí na zaslání bajtu dle obrázku 5.14 odpovědět. Tím máme jistotu, že obvod komunikuje.



Obrázek 5.14: Příklad „zaadresování“ I^2C obvodu DS1621

5.3.3 8 bitový expandér PCF8574

Nedílnou a nezbytnou součástí následujícího textu je i technický manuál výrobce **PCF8574.pdf**.

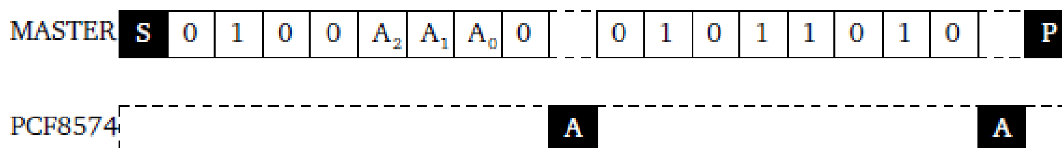
Pro výukové účely byl obvod PCF8574 vybrán záměrně. Jde o nejjednodušší dostupný obvod pro sběrnici I^2C. Neobsahuje žádné konfigurační registry a jeho chování nelze nijak měnit. Obsahuje pouze jediný 8 bitový registr, jehož každý bit je přímo vyveden na právě jeden výstupní pin pouzdra. Hodnotu tohoto registru lze jediným zápisem změnit a lze si aktuální
nastavenou hodnotu i přečíst.

Průběh komunikace s obvodem PCF8574 je znázorněn v manuálu na straně 10 a 11.

Adresa

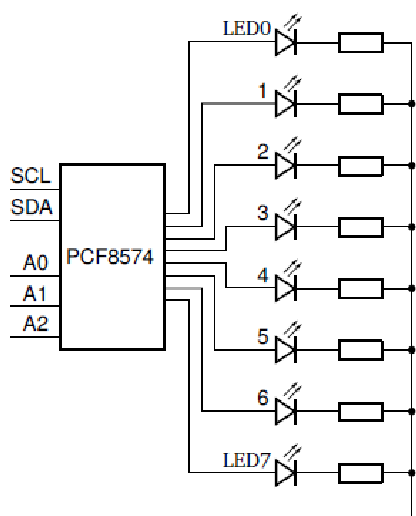
Na straně 9 manuálu PCF8574 je uvedena adresa přidělená výrobcem pro nevyšší 4 bity $H_0 \div H_3$ adresy obvodu. Je to hodnota **0100b**. Zbylé tři bity adresy $A_0 \div A_2$ se nastavují přepínači přímo u obvodu na vývojové desce.

Příklad zápisu jednoho bajtu i s určením adresy může vypadat jako na obrázku 5.15.



Obrázek 5.15: Zaslání jednohy bajtu (0b01011010) obvodu PCF8574

Zapojení
Pro možnost vizuální kontroly je na vývojové desce připojeno na vývody obvodu PCF8574 osm LED podle obrázku 5.16. Každému bitu registru odpovídá jedna LED označená 0-7. Jak je patrno z obrázku, LED se aktivují logickou úrovní 1.



Obrázek 5.16: Připojení LED k obvodu PCF8574

5.3.4 Digitální teploměr Dallas DS1621

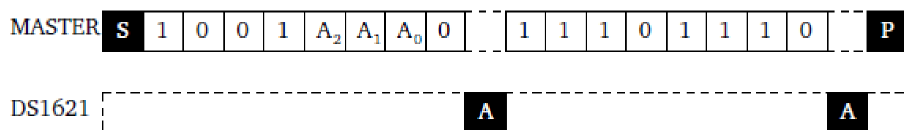
Nedílnou a nezbytnou součástí následujícího textu je i technický manuál výrobce teploměru **DS1621.pdf**.
Obvod DS1621 je digitální teploměr, který může pracovat jako teploměr pro spojité měření teploty, nebo provádět měření teploty na požádání, nebo pracovat jako termostat s nastavitelnou hysterezí.
Pro výukové účely budeme preferovat použití teploměru pro spojité měření teploty.

Formát teploty
Na straně 4 manuálu DS1621 je uveden digitální formát naměřené teploty. Jde o devítibitové znaménkové číslo, kde nejnižší bit představuje rozlišení půl stupně. Protože devíti bity je překročen formát jednoho bajtu, je naměřená teplota předávána ve dvou bajtech za sebou.

Adresa
Na straně 8 manuálu DS1621 je uvedena adresa přidělená výrobcem pro nevyšší 4 bity $H_0 \div H_3$ adresy obvodu. Je to hodnota **1001b**. Zbylé tři bity adresy se nastavují instalovanými přepínači přímo na vývojové desce.
Požadovaná hodnota bude zadána přímo na cvičení.
Příkazy pro řízení teploměru
Na straně 10 manuálu DS1621 jsou uvedeny všechny příkazy, kterými se teploměr ovládá. Stručný přehled s krátkým popisem uvádíme zde:

- Příkaz **ACh** (Access Config) slouží pro nastavení chování teploměru. Formát konfiguračního slova je na straně 5 manuálu DS1621. Pro funkci teploměru je důležitý bit označený jako *1SHOT*, kterým se určuje, zda teploměr bude měřit teplotu spojitě, nebo na požádání. Při modifikaci konfiguračního slova by se měl nejprve aktuální stav přečíst, změnit potřebné bity a poté teprve zapsat.
- Příkaz EEh (Start Convert) spustí převod teploty.
- Příkaz 22h (Stop Convert) zastaví převod teploty.
- Příkaz AAh (Read Temperature) slouží pro čtění teploty. Obdrží-li obvod tento příkaz, vyšle následně 2 bajty s naměřenou teplotou.
- Příkaz A1h (Access TH) registr horního limitu termostatu.
- Příkaz A2h (Access TL) registr dolního limitu termostatu.

Podrobnější popis a ostatní příkazy jsou v manuálu výrobce.
Příklad komunikace
Chceme-li předat obvodu nějaký příkaz, například požadavek na spuštění převodu teploty, zašleme příkaz **EEh** na požadovanou adresu. Postup je na obrázku 5.17.



Obrázek 5.17: Zaslání příkazu *Start Convert* (EEh) obvodu DS1621

Další přesný popis všech komunikačních možností s obvodem DS1621 naleznete v dokumentaci DS1621 na straně 9.

Programátorské rozhraní

Pro komunikaci s obvody na sběrnici I^2C je implementována následující množina funkcí:

void I2C_Init() - inicializace, signály SDA a SCL jsou nastaveny na 1.
void I2C_Start() - vygenerování sekvence START.
void I2C_Stop() - vygenerování sekvence STOP.
void I2C_Ack() - odeslání ACK.
void I2C_*NA*ck() - odeslání NACK.
char I2C_getAck() - přečtení ACK.
char I2C_Vystup(unsigned char value) - odeslání bajtu a převzetí ACK.
unsigned char I2C_Vstup() - příjem jednoho bajtu bez potvrzení.

Příklady úkolů ve cvičení

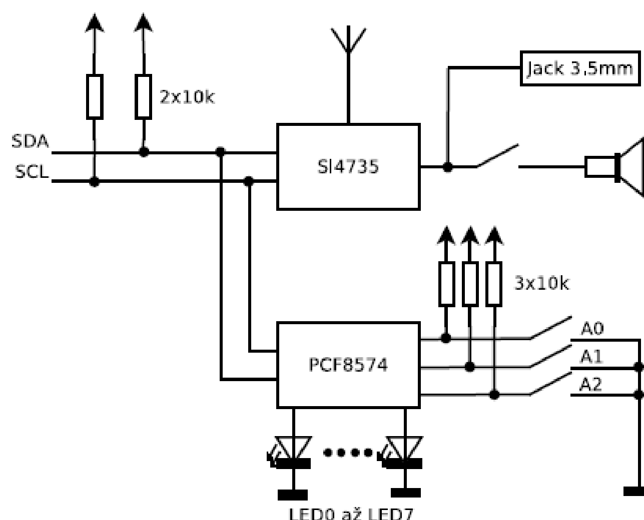
- Prostudování dokumentace a připravených programů je součástí přípravy před cvičením.
- Prvním úkolem na cvičení bude nastavit obvodu DS1621 požadovanou tříbitovou adresu.
- Programem ověřit, zda je obvod na požadované adrese. Ten musí odpovědět signálem ACK.
- Čtení a zápis dat z/do expandéru PCF8574
- Nastavit teploměr pro spojitý převod teploty.
- Spustit převod teploty, číst aktuální teplotu.
- Nastavit obvod DS1621 jako termostat

5.3.5 FM/AM radiomodul Si4735

Nedílnou součástí této dokumentace je i programátorská příručka výrobce FM radiomodulu **SI4735-PG.pdf** a text normy **EN50067.pdf** specifikující systém RDS.

Modul FM rádia obsahuje digitální rádiový přijímač Si4735, který v sobě zahrnuje vše potřebné pro příjem rádiového signálu v pásmech FM/AM. Ve cvičení budeme využívat pouze FM pásmo. Tento obvod se běžně používá v mobilních telefonech jako miniaturní FM přijímač. Součástí FM modulu je i obvod PCF8574 s osmi LED, popsaný v kapitole 5.3.3. Pro ovládání obou obvodů bude použita sběrnice I2C.

Po správném nastavení obvodu rádia je na jeho výstupech k dispozici zvukový signál, který je dále zesílen v zesilovači a po stisknutí tlačítka je možné zvuk přehrát v integrovaném reproduktoru. Je také možné připojit si vlastní sluchátka, jejichž reprodukce již není závislá na stisku tlačítka.



Obrázek 5.18: Schéma zapojení radiového modulu Si4735

Adresa obvodu Si4735 na sběrnici I^2C je 0x22. Za touto adresou se dále uvádí konfigurační bajt, kterým se obvodu sděluje, jakou akci bude dále vykonávat. Některé konfigurační bajty je možné doplnit dalšími bajty jako argumenty. Vše je podrobně popsáno v dokumentaci výrobce.

Změna hlasitosti

Pro změnu hlasitosti se využívá příkaz SET_PROPERTY 0x12, který má jako první argument 0x00 a jako další dvoubajtový argument podpříkaz RX_VOLUME 0x4000. Dále následuje nulový bajt a poslední bajt určuje svými pěti bity s nejnižší vahou hlasitost v úrovni 0-63.
Podrobnější informace jsou uvedeny v technické dokumentaci na straně 65 a 117.
Dotaz

0	1	2	3	4	5
0x12	0x00	0x40	0x00	0x00	0b000VVVVV*

*Bity V určují hlasitost

Ladění rozhlasové stanice

Pro naladění obvodu na požadovanou frekvenci je možné využít dvou příkazů. První možností je příkaz FM_TUNE_FREQ 0x20, který obsahuje další 4 bajty jako argumenty. První a čtvrtý bajt mohou zůstat nulové.
Druhý a třetí bajt obsahuje požadovanou frekvenci vynásobenou stem. Například pro frekvenci 97.3 MHz je nutné zadat hodnotu 9730 a to tak, že do druhého argumentu se zadá horní byte frekvence a do třetího spodní byte frekvence.
Bližší informace jsou uvedeny v technické dokumentaci na straně 67-68.
Dotaz

0	1	2	3	4
0x20	0x00	FreqHI	FreqLO	0x00*

*Bližší význam jednotlivých bitů je možné vyčíst z dokumentace

Druhou možností ladění stanic je proces automatického ladění pomocí příkazu FM_SEEK_START 0x21. Příkaz obsahuje pouze jeden argument, ve kterém je na místě druhého bitu uvedeno, zda se má po dosažení limitu rozsahu začne znovu na opačném konci rozsahu - log.1, nebo na konci rozsahu zastavit - log.0. Na pozici třetího bitu je log. 1, pokud se provádí ladění směrem nahoru a log.0, pokud se ladí směrem dolů.
Odpovědí na příkaz automatického ladějí je jeden byte. Bližší informace jsou uvedeny v technické dokumentaci na straně 69.
Dotaz

0	2
0x21	0b0000SW00*

* S - ladění nahoru/dolu 0/1
W - na konci rozsahu začne ladění opět z opačného konce intervalu

Detekce stavu přijímače
Příkaz FM_TUNE_STATUS 0x22 obsahuje pouze jeden argument, který je možné ponechat nulový. Po zadání tohoto příkazu je možné vyčíst osm bajtů. Ve druhém a třetím bajtu lze zjistit aktuálně naladěnou frekvenci.
Ve čtvrtém bajtu je hodnota síly signálu a v pátém bajtu kvalita signálu po naladění stanice.
Bližší informace jsou uvedeny v technické dokumentaci na straně 70-71.

Dotaz

0	1
0x22	0x00*

*Bližší význam jednotlivých bitů je možné vyčíst z dokumentace

Odpověď

0	1	2	3	4	5	6	7
S1*	S2*	FreqHI	FreqLO	RSSI	SNR	MULT*	CAP*

*Bližší význam jednotlivých bitů je možné vyčíst z dokumentace

Detekce kvality signálu
Příkaz FM_RSQ_STATUS 0x23 obsahuje pouze jeden argument, který je možné ponechat nulový. Po zadání tohoto příkazu je možné vyčíst osm bajtů. Ve druhém bajtu je v nejnižším bitu *V* informace, zda na aktuálně nastavené frekvenci byla rozpoznána nějaká stanice. Ve čtvrtém bajtu je hodnota síly signálu a v pátém bajtu kvalita signálu.
Bližší informace jsou uvedeny v technické dokumentaci na straně 72-73.
Dotaz

0	1
0x23	0x00*

*Bližší význam jednotlivých bitů je možné vyčíst z dokumentace

Odpověď

0	1	2	3	4	5	6	7
S1*	S2*	0bxxxxxxxV*	STBL*	RSSI	SNR	MULT*	FREQ*

*Bližší význam jednotlivých bitů je možné vyčíst z dokumentace

Radio Data System (RDS)

RDS je systém, který může doprovázet vysílání rádiových stanic. Slouží k přenosu datových informací do rádiového přijímače. Mezi tyto informace patří například název vysílané stanice, název právě přehrávaného pořadu, alternativní frekvence na kterých vysílá identická stanice, nebo hlášení o poloze a závažnosti dopravních nehod v okolí.

Pro získání jedné skupiny RDS dat z obvodu Si4735 je nutné zadat příkaz FM_RDS_STATUS 0x24. Tento příkaz je následován jedním bajtem, ve kterém je podstatný pouze bit I s nejnižší vahou. Bitem I=1 se povolí vnitřní přerušení pro příjem RDS dat.

Jako odpověď na tento příkaz je přijato 13 bajtů. Ve druhém bajtu je nejnižším bitem S oznamováno, zda jsou RDS informace synchronizovány. Bez tohoto bitu nemá smysl další informace zpracovávat! Pro následné zpracování jsou nejdůležitější bajty označené jako BxHI a BxLO kde x je 1 až 4. Jedná se přímo o bloky dat přenášené v rámci jedné skupiny.

Bližší informace jsou uvedeny v technické dokumentaci na straně 74-76.

Dotaz

0	1
0x24	0b00000SMI*

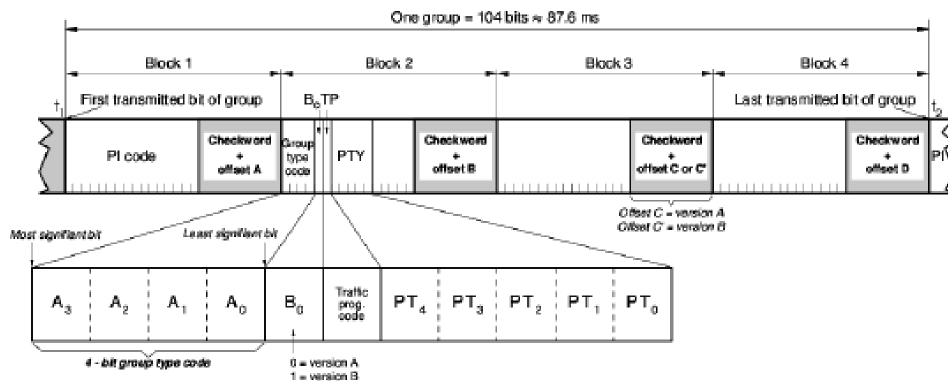
* bit I zapíná vnitřní přerušení pro příjem RDS,
význam bitů S a M je uveden v dokumantaci

Odpověď

0	1	2	3	4	5	
RDS0*	RDS1*	0bxxxxxxxS*	RDS3*	B1HI	B1LO	
6	7	8	9	10	11	12
B2HI	B2LO	B3HI	B3LO	B4HI	B4LO	0baabbccdd**

*Bližší význam jednotlivých bitů je možné vyčíst z dokumentace
** Bity aa, bb, cc a dd určují stav jednotlivých bloků

Data v systému RDS se přenášejí ve skupinách. Každá skupina se dále dělí na čtyři bloky, viz obrázek 5.19. Obsah jednotlivých bloků je dán normou EN50067. Bity Checkword jsou zpracovány přímo obvodem Si4735 a nejsou v odpovědi na dotaz 0x24 obsaženy.

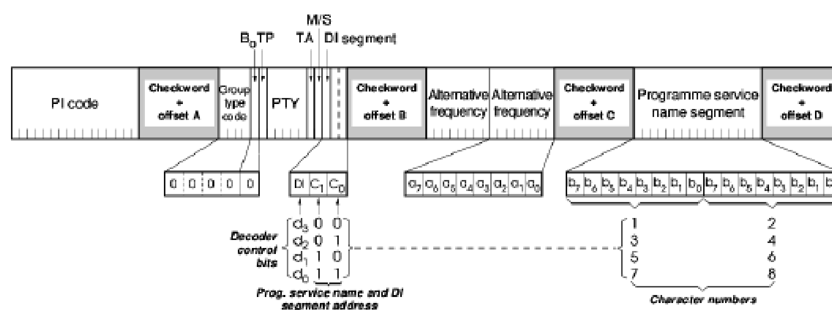


Obrázek 5.19: Obecná struktura RDS

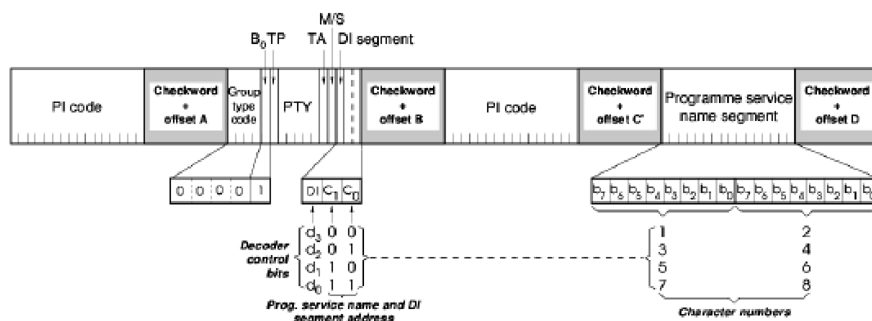
Z obrázku 5.19 je patrné, že v prvním bloku každé skupiny je vysílána informace označená jako PI – Program Identification. Jedná se o jedinečné číslo, které je specifické pro každou rozhlasovou stanici. Druhý blok obsahuje v bitech A3 až A0 identifikaci skupiny. Dále bit B0, který určuje zdali se jedná o verzi A nebo B. Bit TP určuje, zdali se aktuálně vysílá dopravní
hlášení. Bity PT4 až PT0 definují typ vysílaného programu (hudba, řeč, aj.).
Význam zbylých bitů v bloku 2 a bitů v bloku 3 a 4 jsou již závislé na typu skupiny.

Dekódování služby Program Service

Služba Program Service (název stanice) se vysílá ve skupině číslo 0 a to jak ve verzi A, tak i B, přičemž text v obou verzích se může lišit. Rozložení jednotlivých bitů ve skupinách A a B je znázorněno na obrázcích 5.20 a 5.21. Pokud je detekována skupina 0, je v bloku 2 na posledních dvou bitech příznak, které dva znaky z osmiznakového názvu stanice se přenášejí. Název programu je tedy rozdělen na čtyři dvojice. Samotné znaky jsou vysílány v bloku 4 v horním a spodním bajtu.



Obrázek 5.20: Skupina 0A



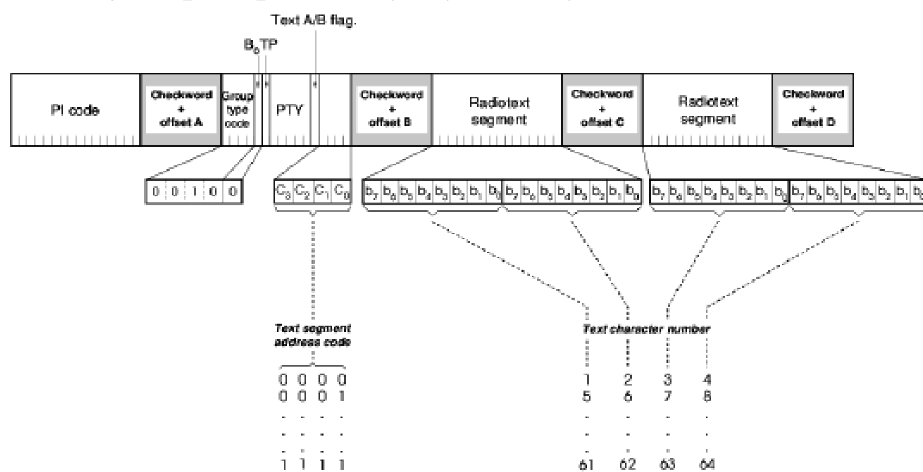
Obrázek 5.21: Skupina 0B

Dekódování služby Alternative Frequency

Služba alternativní frekvence je vysílána spolu se službou PS ve skupině 0 a to pouze ve verzi A. Verze B skupiny 0x00 se liší především ve třetím bloku, jak je patrné z obrázků 5.20 a 5.21. Ve verzi A jsou na pozici třetího bloku vysílány alternativní frekvence na pozici horního a dolního bajtu. Hodnota 0b00000001 odpovídá frekvenci 87.6MHz až hodnota 0b110011 odpovídá frekvenci 107.9MHz.

Dekódování služby Radiotext

Radiotext je obdoba služby PS s tím rozdílem, že znaků je 64. Znaky jsou rozděleny do čtveřic. Radiotext je vysílán ve skupině s označením 2A. Čtyři bity s nejnižší vahou v bloku 2 určují, která čtveřice znaků je přenášena.
V blocích 3 a 4 jsou poté přenášeny čtyři znaky.



Obrázek 5.22: Skupina 2A

Příklady úkolů ve cvičení

- Nalaďte modul FM rádia na požadovanou frekvenci.
- Pomocí dvou tlačítek nastavujte úroveň hlasitosti.
- Pomocí dvou tlačítek spusťte automatické ladění rádia (nahoru nebo dolů).
- Využijte bargraf připojený k obvodu PCF8574 k indikaci síly přijímaného signálu.

- Nalaďte stanici vysílající RDS a zobrazte data přenášená službou Program Service.
- Nalaďte stanici vysílající RDS a zobrazte data přenášená službou Radiotex.
- Nalaďte stanici vysílající RDS a zobrazte alternativní frekvence na kterých je vysílána identická stanice.

Část II

Programování paralelních systémů

použit c++eclipse nebo codeblock

Kapitola 6
Vlákna

6.1 Využití výpočetního výkonu procesorů s více jádry pomocí vláken (threads)

V posledních letech se díky rozvoji moderních technologií výroby procesorů stalo běžným standardem výrobců, umístit do jednoho pouzdra více procesorů současně. Výpočetní výkon počítače se takto navyšuje prakticky tolikrát, kolik jader je celkově v daném počítači nainstalováno. Za tímto technickým vývojem ale zaostává vývoj programového vybavení počítače. Programy i operační systémy jsou za technickým vývojem stále pozadu. Ukázalo se to v posledních 20 letech velmi viditelně několikrát. Jako příklad vezměme čas, než se 64 bitové operační systémy staly standardem. Firma AMD představila svůj první 64 bitový procesor v roce 2000. Kdy se dostaly 64 bitové OS na servery a kdy na desktopy? Odpověď snadno naleznete na internetu.

Problémem současnosti je vývoj programů pro využívání více procesorů paralelně. V roce 2010 ve svém článku „The Trouble With Multicore“ uvedl jeden z významných odborníků David Patterson, profesor na katedře informatiky Univerzity v Berkeley i následující odstavec:

> „*One of the biggest factors, though, is the degree of motivation. In the past, programmers could just wait for transistors to get smaller and faster, allowing microprocessors to become more powerful. So programs would run faster without any new programming effort, which was a big disincentive to anyone tempted to pioneer ways to write parallel code. The La-Z-Boy era of program performance is now officially over, so programmers who care about performance must get up off their recliners and start making their programs parallel.*“

Pokud tedy požadujeme, aby naše programy pracovaly rychleji, není do budoucna jiná cesta, než je psát jako paralelní!

6.1.1 Programování s vlákny

Vlákny rozumíme samostatně (a paralelně) běžící podprogramy jednoho programu (procesu). Tato technologie je v operačních systémech Windows i Unixech implementována již téměř 20 let. Přináší programátorům větší pohodlí při psaní programů a to bez ohledu na počet procesorů počítače. Teprve však při dvou a více procesorech mohou vlákna otevřít programátorovi cestu k využití vyššího výpočetního výkonu počítače.

Jak používat vlákna v OS Linux budou popsány v následným textu. Prvním krokem je funkce programu, který bude reprezentovat vlákno. Tato funkce musí být přesně definovány formát:

```
void *thread_function_name(void *argument) { /* code */ }
```

Argument lze použít k předání vláknu jakýkoli typ dat. Je-li nutné zpracovat více argumentů, pak musí být zapouzdřeny do pole, struktury nebo třídy.
Chceme-li funkci řešit pomoci vlákna, pak je možné vytvořit vlákno z něj. V Linuxu je k tomuto účelu určena funkcí:

```
int pthread_create( pthread_t *thread,
                    const pthread_attr_t *attr,
                    void *(*start_routine) (void *),
                    void *arg );
```

První parametr vlákna je **ukazatel na ID vlákna**. Toto číslo může být později použit např. pro synchronizaci vláken. Druhý parametr ***attr*** umožňuje definovat některé parametry nového vlákna. Pro běžné použití může být použit jako ukazatel ***NULL***. Třetí parametr ***start_routine*** je funkce, která bude provozován jako vlákno. Poslední parametr ***arg*** je argumentem pro nové vlákno. To se používá k předávání dat do nového vlákna.

Čekání na dokončení vlákna je nutno provést podle funkce:

```
int pthread_join(pthread_t thread, void **retval);
```

První parametr vlákno je ID stávajícího (běžícího) vlákna a druhý parametr ***retval*** může být použit pro získání návratovou hodnotu vlákna.
Více informací o programování nití je k dispozici v manuálové stránky, viz ***man pthreads, man pthread_create and man pthread_join, or the internet.***

Příklad programu, který vytváří vlákno a pak čeká, až jeho dokončení je v následujícím kódu:

```
void *simple_thread( void *str )
{
   printf( "Thread created with argument '%s'.\n", ( char *
   ) str );
   sleep( 5 );
   return NULL;
}

int main()
{
   pthread_t tid;
   pthread_create( &tid, NULL, simple_thread, ( void * )
   "Testing" ); pthread_join( tid, NULL );
   printf( "Done\n" );
```

Další příklad na využití vláken je uveden v příloze tohoto studijního materiálu – threads-demo.zip. Tento archiv obsahuje ukázku programu, kde se hledá pomocí dvou vláken maximální prvek v poli. Každé ze dvou vláken prohledává jen polovinu pole. Výsledný čas hledání je v případě spuštění na dvou a víceprocesorovém počítači poloviční. Zdrojový kód programu je velmi jednoduchý a je popsán ve vložených komentářích.

Na cvičeních se k tomuto účelu použije IDE C++ Eclipse. Spuštěni toho IDE probíhá podobné jako u IDE KDS. Jelikož jsou oba IDE na stejné platforme, použiti IDE C++ Eclipse je skoro stejné.

Po spuštěni IDE C++ Eclipse, nahoře vlevo zvolíme „***Makefile Project with Existing Code***“. Objeví se okno, kde zadáme cestu k projektu ***threads-demo*** (nezapomenout předem složku extrahovat z archivu). Ještě zaklikneme položku *Linux GCC*. Potvrdíme tlačítkem ***Finish***.

Spustíme program klasicky, jako to bylo v případě IDE KDS. Ale program nebude ještě pracovat. Jelikož demo program na vlákna vyžaduje na vstupu číslo, které reprezentuje velikost pole, musíme tuto hodnotu nastavit. To učiníme následně - pravým tlačítkem myši na projekt – zvolit „***Properties***“. V pravé nabídce zvolit „***Run/Debug Settings***“ a označit ***threads-sort*** a zvolit ***Edit*** (v případě, že v okně doposud nebude zobrazena položka ***threads-sort,*** pak ještě jednou spusťte program). Po stisku tlačítka ***Edit*** se zobrazí nove okno, tam se prekliknete do záložky ***Arguments***. Do okna napíšeme číslo reprezentující velikost pole (např. **50000**). Potvrdíme *OK, OK*. Program spusťte.

NEZAPOMENOUT před návratem do perspektivy editoru, nezapomeňte zastavit běžící program. Jinak ho znovu nespustíte. Zastaveni provedeme stiskem ikony „červeného čtverce“. Ikona je bud nahoře v perspektivě režimu laděni nebo dole v právo v perspektivě editovaní.

Příprava na cvičení

- Seznámit se ukázkovým programem a způsobem měření doby běhu částí programu.
- Zopakovat si nejjednodušší algoritmy třídění - přímý výběr, přímé vkládání a bublinkové třídění.
- Připravit si třídící funkce tak, aby bylo možno třídit sestupně i vzestupně.
- Upravit třídící funkce tak, aby bylo možno setřídit jen část pole.
- Připravit si třídící funkce pro třídění pole s libovolnými typy prvků - int, long, fload, double, ...
- Vyzkoušet si vygenerovat libovolně dlouhé pole náhodných čísel.
- Vyzkoušet si vytváření vláken.

Úkoly ve cvičení

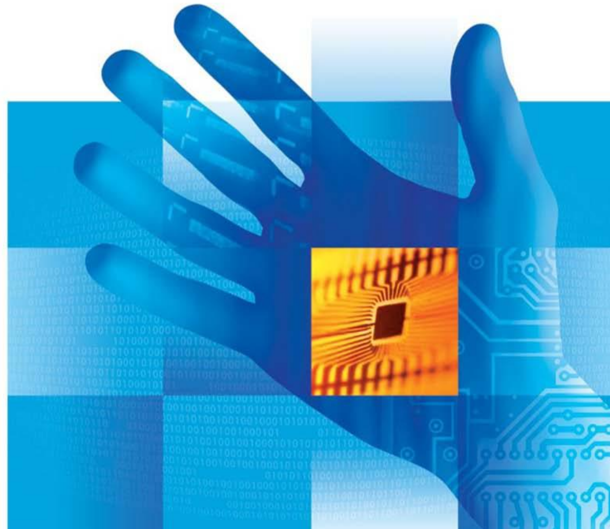
- Setřídit určeným algoritmem pole náhodně vygenerovaných čísel.
- Setřídit část pole, např. horní a dolní polovinu pole.
- Spojit dvě setříděné části pole v jednu posloupnost.
- Spustit třídění částí pole paralelně.
- Změřit čas třídění.

Kapitola 7

CUDA

Pro CUDU připravila Nvidia vlastní verzi eclipsu "**NSIGHT**".

Předmět APPS
GPGPU a CUDA



Petr Olivka
katedra informatiky

Obsah

- Historie GPU
- CUDA
- Architektura
- Komunikace CPU - GPU

Historie GPU

- 1970 – ANTIC v 8 bitovém Atari
- 1980 – IBM 8514
- 1993 – založena Nvidia Co.
- 1994 – založeno 3dfx Interactive
- 1995 – čip NV1 od Nvidia
- 1996 - 3dfx vydalo Voodoo Graphics
- 1999 - GeForce 256 by Nvidia – podpora geom. transf.
- 2000 - Nvidia kupuje 3dfx Interactive
- 2002 - GeForce 4 vybaveno pixel a vertex shadery
- 2006 - GeForce 8 - unifikovaná architektura (nerozlišuje pixel a vertex shader) (Nvidia CUDA)
- 2008 - GeForce 280 - podpora dvojité přesnosti
- 2010 - GeForce 480 (Fermi) - první GPU postavené pro obecné výpočty - GPGPU

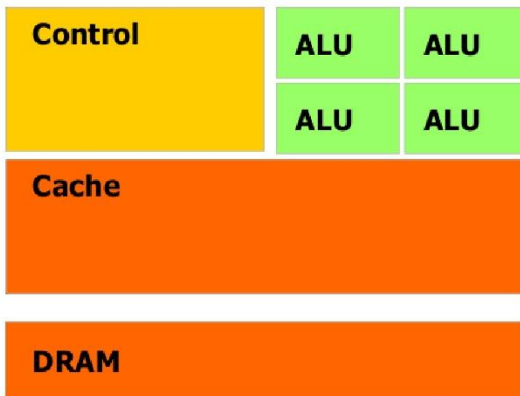
Výhody GPU

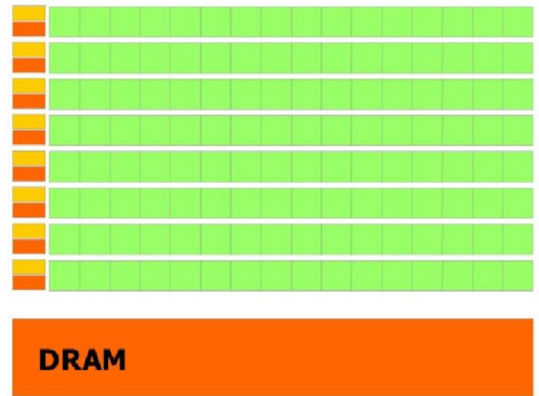
- GPU je navrženo pro současný běh stovek vláken - virtuálně až stovek tisíc vláken
- Vlákna musí být nezávislá, není zaručeno, v jakém pořadí budou provedena
- GPU je vhodné pro intenzivní výpočty s malým počtem podmínek
- Není zde podpora spekulativního zpracování
- GPU je optimalizována pro sekvenční přístup do paměti. Přenosové rychlosti stovky GB/s

Porovnání CPU - GPU

	Nvidia GeForce 580	Intel i7-960 6xCore
Tranzistory	3000 * 10^6	1170 * 10^6
Frekvence	1.5 GHz	3.5 GHz
Počet vláken	512	12
Max. výkon	1.77 Tflops	~200 GFLops
Propustnost	194 GB/s	26 GB/s
RAM	1.5 GB	~48GB
Výkon	244W	130W

Výhody GPU





Maximální možný počet tranzistorů je použit na výpočetní jednotky, ne na cache a řízení toku dat.

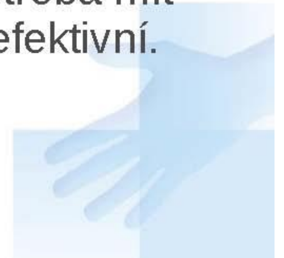
Podstata GPGPU

- Na počátku bylo nutno pro GPGPU výpočty používat rozhraní OpenGL
- Úlohy byly formulovány pomocí textur a operací s body
- Vývojáři her potřebovali univerzálnější hardware – vznikly pixel shadery
  - Jde o jednoduchý programovatelný procesor pro operace s body
  - Má podporu pro výpočty s plovoucí desetinnou tečkou jednoduché přesnosti
  - Kód je omezen na několi desítek instrukcí

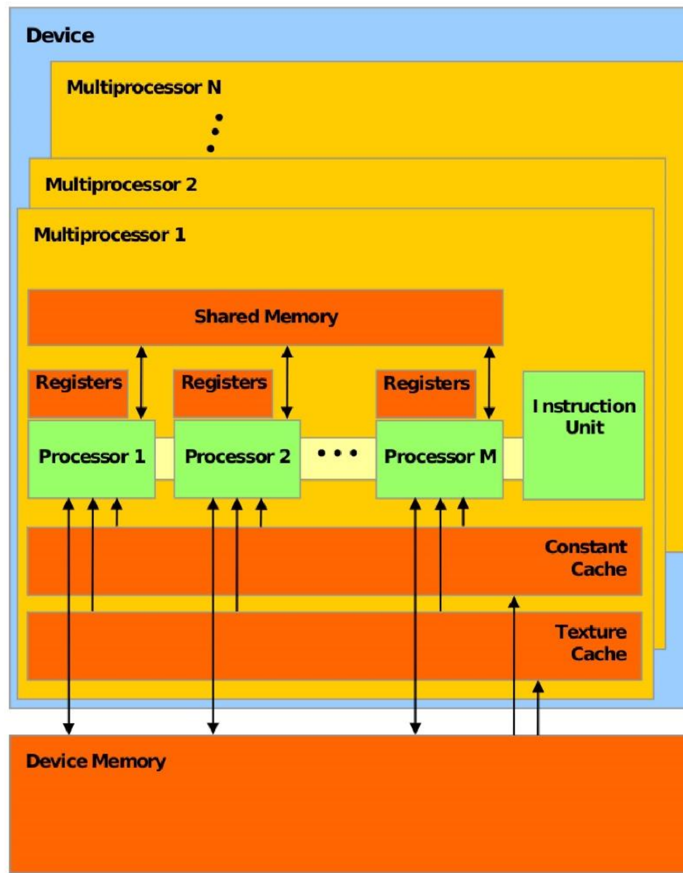
CUDA – Compute Unified Device Architecture (Nvidia 2/2007)

- Výrazně jednodušší programování GPGPU
- Zcela odstraněna nutnost pracovat s OpenGL a definování úloh pomocí textur
- Je založena na jednoduchém rozšíření jazyka C/C++
- Funguje jen s kartami Nvidia

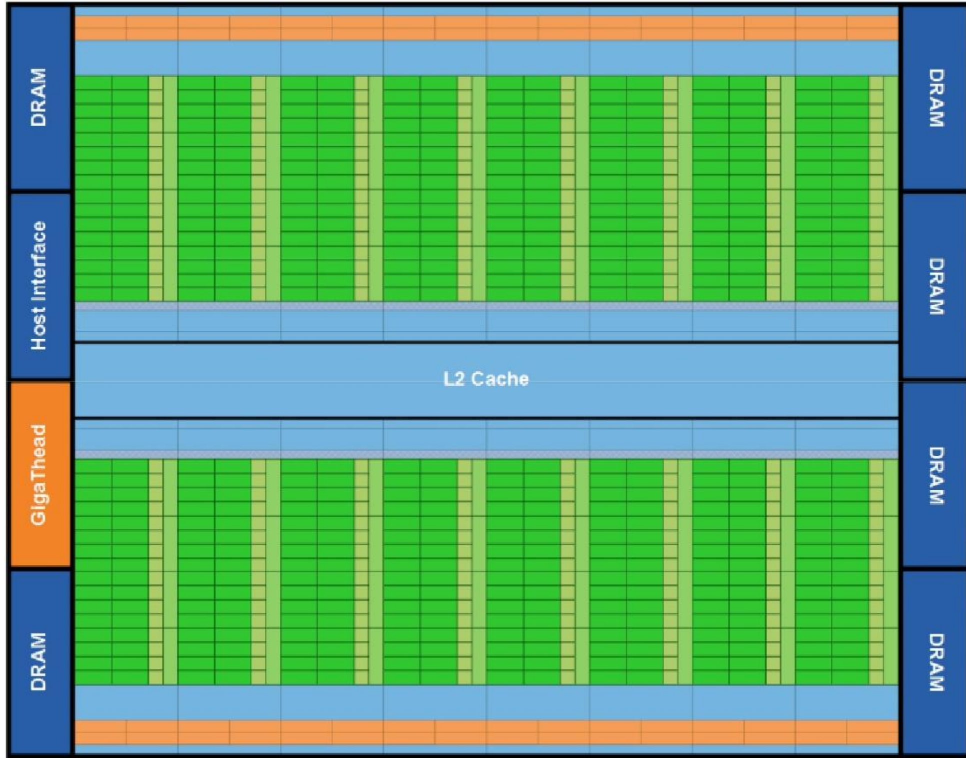
Je velmi snadné napsat kód pro CUDA, ale je potřeba mít hluboké znalosti o GPU, aby výsledný kód byl efektivní.



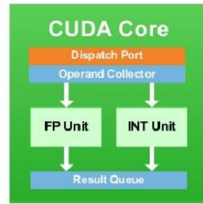
Architektura

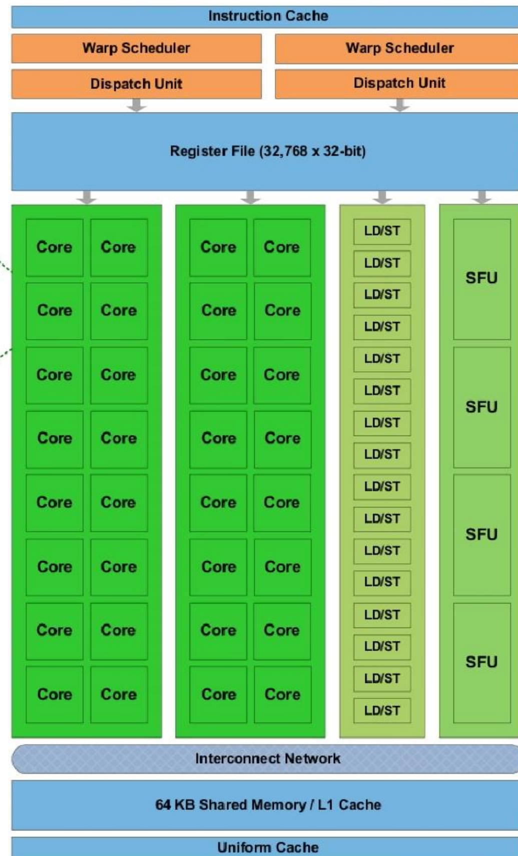


Architektura Fermi - ????

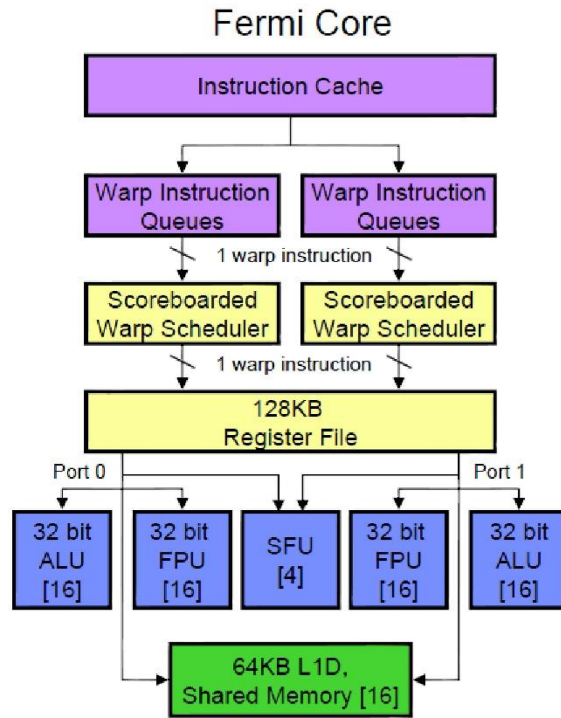


Architektura Fermi, blokové schéma

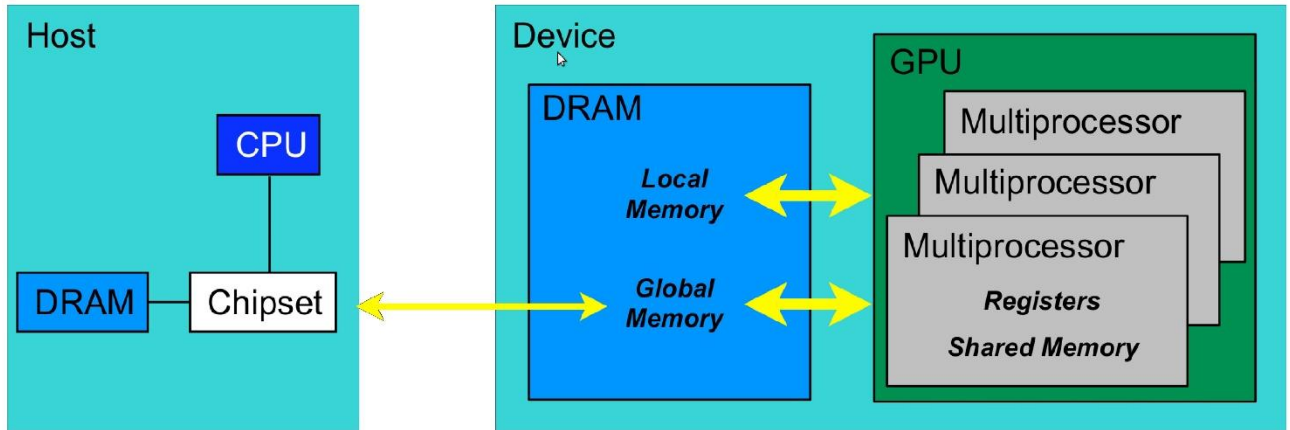




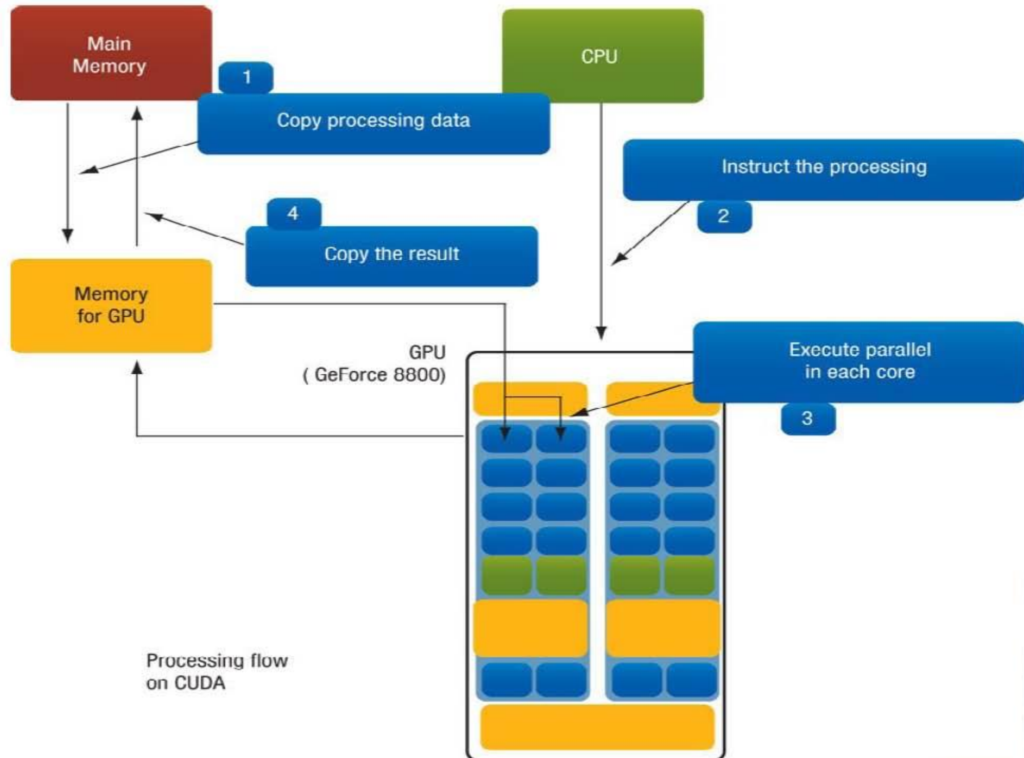
Architektura Fermi, blokové schéma



Paměťový model v PC



Postup výpočtu



Komunikace mezi CPU a GPU

- Komunikace přes PCI-Express je velmi pomalá, méně než 5GB/s
- Je nutno tuto komunikaci minimalizovat, ideální jen přesun dat na začátku a na konci výpočtu.
- GPU se nevyplatí pro úlohy s nízkou výpočetní intenzitou
- Výhodnější mohou být GPU on-board se sdílenou pamětí
- Pokud se provádí častá komunikace CPU-GPU, je výhodné použití pipeliningu
- Je možné současně provádět: výpočet na CPU, výpočet na GPU, kopírovat data do GPU, kopírovat data z GPU

Vývoj efektivního kódu

Pro vývoj efektivního kódu je potřeba dodržet následující pravidla:

- Redukovat přenos dat mezi CPU a GPU
- Optimalizovat přístup do globální paměti GPU
- Omezit divergentní vlákna
- Zvolit správnou velikost bloků vláken

JAK NA TO!

Příklady programovaní CUDA technologie v OS Linux pro předmět APPS.

Pro CUDU připravila Nvidia vlastní verzi eclipsu "**NSIGHT**". Po instalaci nenaskočila do menu, ale je v cestách, takže se to musí spustit ideálně z příkazového řádku. To učiníme následovné – vyhledáme si program ***Terminal*** (ekvivalent příkazové řádky ve Win). Tam napíšeme ***nsight*** a potvrdíme enterem. IDE NSIGHT se spustí.
Projekty opět importujte jako projekty s Makefile a v nabídce překladač Linux GCC. Projekty importujte po **jednom** (tzn. cuda1, cuda2, cuda3 a pak cuda 4). Postup je stejný jako u C++ Eclipse. Program spouštíme před ikonu ***Run***.

V případě, ze Vám to nepůjde, postupujte podle návodu p. Olivky. Viz níže:

Příklady programovaní CUDA technologie v OS Linux pro předmět APPS.
(c) Petr Olivka, katedra informatiky, FEI, VSB-TU Ostrava

V souboru makefile.vars nastavte cestu do adresáře s nainstalovaným CUDA toolkit.

Kazdy adresář cuda* obsahuje Makefile, stačí tedy pro sestaveni programu zavolat v každém adresáři příkaz "make".

Před spuštěním všech demo programu musíte mít nastavenu cestu k dynamickým knihovnám cuda*.so. Proveďte příkazem:

export LD_LIBRARY_PATH=/usr/local/cuda/lib64

Příslušná cesta musí korespondovat s udají uvedenými v souboru makefile.vars.

Demo CUDA1

```
//*****************************************************************
// Demo program pro vyuku predmetu APPS (10/2011)
// Petr Olivka, katedra informatiky, FEI, VSB-TU Ostrava
// email:petr.olivka@vsb.cz
//
// Priklad pouziti CUDA technologie.
//*****************************************************************

#include <stdio.h>

void run_cuda();

int main()
{
	// volani funkce ze souboru .cu
	run_cuda();
	return 0;
}
```

Demo CUDA2

```
//***************************************************************
// Demo program pro vyuku predmetu APPS (10/2011)
// Petr Olivka, katedra informatiky, FEI, VSB-TU Ostrava
// email:petr.olivka@vsb.cz
//
// Priklad pouziti CUDA technologie.
// Pole prvku float[] se vynasobi hodnotou float
//***************************************************************

#include <stdio.h>

void run_mult( float *pole, int L, float x );

#define N 200

int main()
{
	// inicializace pole
	float prvky[ N ];
	for ( int i = 0; i < N; i++ )
		prvky[ i ] = i;

	// volani funkce ze souboru .cu
	run_mult( prvky, N, 3.14 );

	// vypis vysledku
	for ( int i = 0; i < N; i++ )
		printf( "%8.2f", prvky[ i ] );
	printf( "\n" );
	return 0;
}
```

Demo CUDA3

```
//***************************************************************
// Demo program pro vyuku predmetu APPS (10/2011)
// Petr Olivka, katedra informatiky, FEI, VSB-TU Ostrava
// email:petr.olivka@vsb.cz
//
// Priklad pouziti CUDA technologie.
// Do prazneho obrazku se vykresli ceska vlajka
//
// Pro manimulaci s obrazkem je pouzita knihovna OpenCV.
//***************************************************************

#include <stdio.h>
#include <cuda_runtime.h>
#include <opencv/highgui.h>

void run_flag( uchar4 *bgr_pic, int sizex, int sizey, int elem );

#define RECT 16                         // rozmer nejmensiho ctverce obrazku
#define SIZEX ( RECT * 20 )             // rozmer obrazku X
#define SIZEY ( RECT * 20 )             // rozmer obrazku Y

int main()
{
	// vytvoreni pole pro ulozeni vsech bodu obrazku velikosti SIZEX * SIZEY,
ulozeni po radcich
	uchar4 *bgr_pole = new uchar4[ SIZEX * SIZEY ];

	// prenos vypoctu do souboru .cu
	run_flag( bgr_pole, SIZEX, SIZEY, RECT );

	// vytvoreni prazneho obrazku
	IplImage *img = cvCreateImage( cvSize( SIZEX, SIZEY ),
IPL_DEPTH_8U, 3 );

	// prevedeni ziskanych dat do obrazku
	for ( int y = 0; y < SIZEY; y++ )
		for ( int x  = 0; x < SIZEX; x++ )
		{
			uchar4 bgr = bgr_pole[ y * SIZEX + x ];
			CvScalar s = { bgr.x, bgr.y, bgr.z };
			cvSet2D( img, y, x, s );
		}
```

```
    // zobrazeni vysledku
    cvShowImage( "CZ Flag", img );
    cvWaitKey( 0 );
}
```

Demo CUDA4

```
//***************************************************************
// Demo program pro vyuku predmetu APPS (10/2011)
// Petr Olivka, katedra informatiky, FEI, VSB-TU Ostrava
// email:petr.olivka@vsb.cz
//
// Priklad pouziti CUDA technologie.
// Prevedeni barevneho obrazku na odstiny sede.
//
// Pro manimulaci s obrazkem je pouzita knihovna OpenCV.
//***************************************************************

#include <stdio.h>
#include <cuda_runtime.h>
#include <opencv/highgui.h>

void run_grayscale( uchar4 *bgr_pic, int sizex, int sizey );

int main( int numarg, char **arg )
{
	if ( numarg < 2 )
	{
		printf( "Enter picture filename!\n" );
		return 1;
	}

	// nacteni obrazku
	IplImage *img = cvLoadImage( arg[ 1 ] );
	cvShowImage( "Color", img );
	int sizex = img->width;
	int sizey = img->height;

	// alokace pole uchar4 pro body obrazku a jeho naplneni
	uchar4 *bgr_pole = new uchar4[ sizex * sizey ];
	for ( int y = 0; y < sizey; y++ )
		for ( int x  = 0; x < sizex; x++ )
		{
			CvScalar s = cvGet2D( img, y, x );
			uchar4 bgr = { s.val[ 0 ], s.val[ 1 ], s.val[ 2 ] };
			bgr_pole[ y * sizex + x ] = bgr;
		}

	// volani funkce ze souboru .cu
```

```
	run_grayscale( bgr_pole, sizex, sizey );

	// ziskana data ulozime zpet do obrazku
	for ( int y = 0; y < sizey; y++ )
		for ( int x  = 0; x < sizex; x++ )
		{
			uchar4 bgr = bgr_pole[ y * sizex + x ];
			CvScalar s = { bgr.x, bgr.y, bgr.z };
			cvSet2D( img, y, x, s );
		}

	// zobrazeni vysledku
	cvShowImage( "GrayScale", img );
	cvWaitKey( 0 );
}
```

Příklad: Druhá mocnina prvků vektoru

```
const int N = 10;
const int blocksize = 4;

__global__ void square_array(float *a,int N)
{
  int idx=blockIdx.x*blockDim.x+threadIdx.x;
  if (idx<N) a[idx] = a[idx] * a[idx];
}
```

- Definice CUDA kernelu
  - Realizuje paralelní výpočet druhých mocnin prvků vektoru na GPU
  - `float *a` – ukazatel do paměti GPU zařízení
  - `int N` – skutečný počet prvků

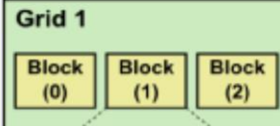
```
int main(void) {
  float *a_h;
  const size_t size = N*sizeof(float);
  a_h = (float *)malloc(size);
  for (int i=0; i<N; i++) {
    a_h[i] = (float)i;
  }

  float *a_d;
  cudaMalloc((void **) &a_d, size);

  cudaMemcpy(a_d, a_h, size,
             cudaMemcpyHostToDevice);

  int n_blocks = N/block_size +
         (N%block_size == 0 ? 0:1);

  square_array <<<n_blocks,
               block_size>>>(a_d,N);

  cudaMemcpy(a_h, a_d, sizeof(float)*N,
             cudaMemcpyDeviceToHost);

  for (int i=0; i<N; i++) {
    printf("%d %f\n", i, a_h[i]);
  }
  cudaFree(a_d);
  free(a_h);
}
```

Příklad: Druhá mocnina prvků vektoru

```
const int N = 10;
const int blocksize = 4;

__global__ void square_array(float *a,int N)
{
  int idx=blockIdx.x*blockDim.x+threadIdx.x;
  if (idx<N) a[idx] = a[idx] * a[idx];
}
```

- Definice CUDA kernelu
  - Realizuje paralelní výpočet druhých mocnin prvků vektoru na GPU
  - `float *a` – ukazatel do paměti GPU zařízení
  - `int N` – skutečný počet prvků

```
int main(void) {
  float *a_h;
  const size_t size = N*sizeof(float);
  a_h = (float *)malloc(size);
                                  {
```




```
                                  d, size);
                                  :e,
                                  'oDevice);
                                  :ize +
                                  0 ? 0:1);
                                  :,
                                  ->(a_d,N);
  cudaMemcpy(a_h, a_d, sizeof(float)*N,
             cudaMemcpyDeviceToHost);

  for (int i=0; i<N; i++) {
    printf("%d %f\n", i, a_h[i]);
  }
  cudaFree(a_d);
  free(a_h);
}
```

Příklad: Druhá mocnina prvků vektoru

```
const int N = 10;
const int blocksize = 4;

__global__ void square_array(float *a,int N)
{
  int idx=blockIdx.x*blockDim.x+threadIdx.x;
  if (idx<N) a[idx] = a[idx] * a[idx];
}
```

- Stanovení velikosti vektoru
  - Počet prvků vektoru (10)
- Nastavení počtu vláken bloku
  - Maximální počet vláken v bloku (4)
- Důsledek je popis velikosti gridu
  - Počet prvků / Počet vláken bloku

```
int main(void) {
  float *a_h;
  const size_t size = N*sizeof(float);
  a_h = (float *)malloc(size);
  for (int i=0; i<N; i++) {
    a_h[i] = (float)i;
  }

  float *a_d;
  cudaMalloc((void **) &a_d, size);

  cudaMemcpy(a_d, a_h, size,
            cudaMemcpyHostToDevice);

  int n_blocks = N/block_size +
          (N%block_size == 0 ? 0:1);

  square_array <<<n_blocks,
              block_size>>>(a_d,N);

  cudaMemcpy(a_h, a_d, sizeof(float)*N,
            cudaMemcpyDeviceToHost);

  for (int i=0; i<N; i++) {
    printf("%d %f\n", i, a_h[i]);
  }
  cudaFree(a_d);
  free(a_h);
}
```

Příklad: Druhá mocnina prvků vektoru

```
const int N = 10;
const int blocksize = 4;

__global__ void square_array(float *a,int N)
{
  int idx=blockIdx.x*blockDim.x+threadIdx.x;
  if (idx<N) a[idx] = a[idx] * a[idx];
}
```

- Alokace a inicializace vektoru v hlavní paměti („paměť CPU“)
  - Vektor a_h velikosti N *float* hodnot je k dispozici v paměti CPU
  - `size` jako celková velikost pole pro uložení N *float* hodnot bude použita i při následné alokaci GPU paměti

```
int main(void) {
  float *a_h;
  const size_t size = N*sizeof(float);
  a_h = (float *)malloc(size);
  for (int i=0; i<N; i++) {
    a_h[i] = (float)i;
  }

  float *a_d;
  cudaMalloc((void **) &a_d, size);

  cudaMemcpy(a_d, a_h, size,
          cudaMemcpyHostToDevice);

  int n_blocks = N/block_size +
         (N%block_size == 0 ? 0:1);

  square_array <<<n_blocks,
            block_size>>>(a_d,N);

  cudaMemcpy(a_h, a_d, sizeof(float)*N,
           cudaMemcpyDeviceToHost);

  for (int i=0; i<N; i++) {
    printf("%d %f\n", i, a_h[i]);
  }
  cudaFree(a_d);
  free(a_h);
}
```

Příklad: Druhá mocnina prvků vektoru

```
const int N = 10;
const int blocksize = 4;

__global__ void square_array(float *a,int N)
{
  int idx=blockIdx.x*blockDim.x+threadIdx.x;
  if (idx<N) a[idx] = a[idx] * a[idx];
}
```

- Alokace a kopírování hodnot vektoru do paměti GPU
  - Vektor a_d velikosti N *float* hodnot je alokován v paměti GPU
  - Při kopírování pamětí je specifikován a_d cíl a a_h zdroj přenosu, `size` velikost kopírované oblasti a `cudaMemcpyHostToDevice` směr přenosu

```
int main(void) {
  float *a_h;
  const size_t size = N*sizeof(float);
  a_h = (float *)malloc(size);
  for (int i=0; i<N; i++) {
    a_h[i] = (float)i;
  }

  float *a_d;
  cudaMalloc((void **) &a_d, size);

  cudaMemcpy(a_d, a_h, size,
             cudaMemcpyHostToDevice);

  int n_blocks = N/block_size +
         (N%block_size == 0 ? 0:1);

  square_array <<<n_blocks,
             block_size>>>(a_d,N);

  cudaMemcpy(a_h, a_d, sizeof(float)*N,
             cudaMemcpyDeviceToHost);

  for (int i=0; i<N; i++) {
    printf("%d %f\n", i, a_h[i]);
  }
  cudaFree(a_d);
  free(a_h);
}
```

Příklad: Druhá mocnina prvků vektoru

```
const int N = 10;
const int blocksize = 4;

__global__ void square_array(float *a,int N)
{
  int idx=blockIdx.x*blockDim.x+threadIdx.x;
  if (idx<N) a[idx] = a[idx] * a[idx];
}
```

- Výpočet počtu bloků vláken, do kterých bude provádění rozprostřeno
  - Proměnná `n_blocks` definuje počet bloků v rámci gridu v 1D uspořádání
  - Případný nenulový zbytek celočíselného podílu počtu prvků a počtu vláken v bloku vynutí alokaci bloku navíc (nebude plně využit)

```
int main(void) {
  float *a_h;
  const size_t size = N*sizeof(float);
  a_h = (float *)malloc(size);
  for (int i=0; i<N; i++) {
    a_h[i] = (float)i;
  }

  float *a_d;
  cudaMalloc((void **) &a_d, size);

  cudaMemcpy(a_d, a_h, size,
             cudaMemcpyHostToDevice);

  int n_blocks = N/block_size +
        (N%block_size == 0 ? 0:1);

  square_array <<<n_blocks,
            block_size>>>(a_d,N);

  cudaMemcpy(a_h, a_d, sizeof(float)*N,
             cudaMemcpyDeviceToHost);

  for (int i=0; i<N; i++) {
    printf("%d %f\n", i, a_h[i]);
  }
  cudaFree(a_d);
  free(a_h);
}
```

Příklad: Druhá mocnina prvků vektoru

```
const int N = 10;
const int blocksize = 4;

__global__ void square_array(float *a,int N)
{
  int idx=blockIdx.x*blockDim.x+threadIdx.x;
  if (idx<N) a[idx] = a[idx] * a[idx];
}
```

- Volání provádění CUDA kernelu
  - Za voláním funkce kernelu následují parametry direktivy volání `n_blocks` počet bloků a `block_size` jejich velikost (počet vláken v bloku)
  - Vektor a_d je předán jako ukazatel do paměti GPU, následuje konstantní `N`

```
int main(void) {
  float *a_h;
  const size_t size = N*sizeof(float);
  a_h = (float *)malloc(size);
  for (int i=0; i<N; i++) {
    a_h[i] = (float)i;
  }

  float *a_d;
  cudaMalloc((void **) &a_d, size);

  cudaMemcpy(a_d, a_h, size,
             cudaMemcpyHostToDevice);

  int n_blocks = N/block_size +
           (N%block_size == 0 ? 0:1);

  square_array <<<n_blocks,
               block_size>>>(a_d,N);

  cudaMemcpy(a_h, a_d, sizeof(float)*N,
             cudaMemcpyDeviceToHost);

  for (int i=0; i<N; i++) {
    printf("%d %f\n", i, a_h[i]);
  }
  cudaFree(a_d);
  free(a_h);
}
```

Příklad: Druhá mocnina prvků vektoru

```
const int N = 10;
const int blocksize = 4;

__global__ void square_array(float *a,int N)
{
  int idx=blockIdx.x*blockDim.x+threadIdx.x;
  if (idx<N) a[idx] = a[idx] * a[idx];
}
```

- Kopírování hodnoty vypočtem modifikovaného vektoru zpět do paměťového prostoru CPU
  - Při kopírování pamětí je opět specifikován a_h cíl a a_d zdroj přenosu, `size` velikost kopírované oblasti a `cudaMemcpyDeviceToHost` směr přenosu

```
int main(void) {
  float *a_h;
  const size_t size = N*sizeof(float);
  a_h = (float *)malloc(size);
  for (int i=0; i<N; i++) {
    a_h[i] = (float)i;
  }

  float *a_d;
  cudaMalloc((void **) &a_d, size);

  cudaMemcpy(a_d, a_h, size,
             cudaMemcpyHostToDevice);

  int n_blocks = N/block_size +
         (N%block_size == 0 ? 0:1);

  square_array <<<n_blocks,
               block_size>>>(a_d,N);

  cudaMemcpy(a_h, a_d, size,
              cudaMemcpyDeviceToHost);

  for (int i=0; i<N; i++) {
    printf("%d %f\n", i, a_h[i]);
  }
  cudaFree(a_d);
  free(a_h);
}
```

Příklad: Druhá mocnina prvků vektoru

```
const int N = 10;
const int blocksize = 4;

__global__ void square_array(float *a,int N)
{
  int idx=blockIdx.x*blockDim.x+threadIdx.x;
  if (idx<N) a[idx] = a[idx] * a[idx];
}
```

- Zobrazení výsledku

```
int main(void) {
  float *a_h;
  const size_t size = N*sizeof(float);
  a_h = (float *)malloc(size);
  for (int i=0; i<N; i++) {
    a_h[i] = (float)i;
  }

  float *a_d;
  cudaMalloc((void **) &a_d, size);

  cudaMemcpy(a_d, a_h, size,
             cudaMemcpyHostToDevice);

  int n_blocks = N/block_size +
         (N%block_size == 0 ? 0:1);

  square_array <<<n_blocks,
                block_size>>>(a_d,N);

  cudaMemcpy(a_h, a_d, size,
             cudaMemcpyDeviceToHost);

  for (int i=0; i<N; i++) {
    printf("%d %f\n", i, a_h[i]);
  }
  cudaFree(a_d);
  free(a_h);
}
```

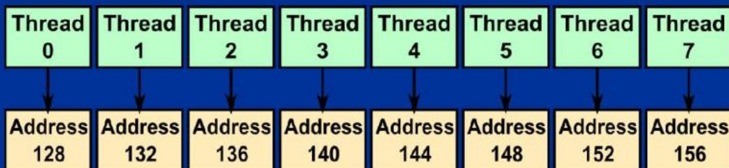
Příklad: Druhá mocnina prvků vektoru

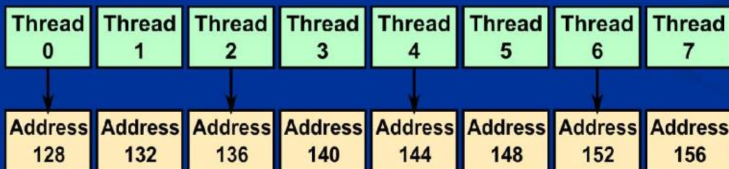
```
const int N = 10;
const int blocksize = 4;

__global__ void square_array(float *a,int N)
{
  int idx=blockIdx.x*blockDim.x+threadIdx.x;
  if (idx<N) a[idx] = a[idx] * a[idx];
}
```

- Úklid dynamicky alokovaných pamětí před ukončením provádění programu
  - Uvolnění alokované paměti v prostoru GPU pomocí `cudaFree(a_d)`
  - Uvolnění paměti v prostoru CPU pomocí `free(a_h)`

```
int main(void) {
  float *a_h;
  const size_t size = N*sizeof(float);
  a_h = (float *)malloc(size);
  for (int i=0; i<N; i++) {
    a_h[i] = (float)i;
  }

  float *a_d;
  cudaMalloc((void **) &a_d, size);

  cudaMemcpy(a_d, a_h, size,
          cudaMemcpyHostToDevice);

  int n_blocks = N/block_size +
        (N%block_size == 0 ? 0:1);

  square_array <<<n_blocks,
             block_size>>>(a_d,N);

  cudaMemcpy(a_h, a_d, size,
           cudaMemcpyDeviceToHost);

  for (int i=0; i<N; i++) {
    printf("%d %f\n", i, a_h[i]);
  }
  cudaFree(a_d);
  free(a_h);
}
```

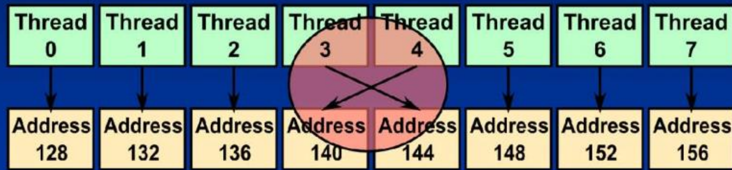
Slučitelné přístupy do paměti



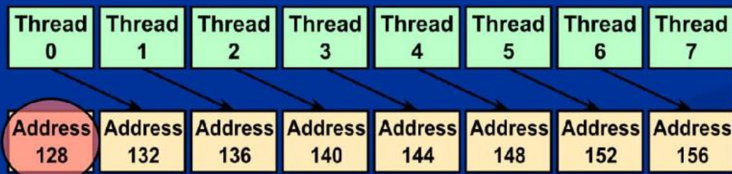


- všechna vlákna přistupují k prvkům podle vzoru k-té vlákno – k-tý prvek
- všechny vlákna warp se přístupu do paměti neúčastní
  - vzorec k-té vlákno – k-tý prvek je zachován

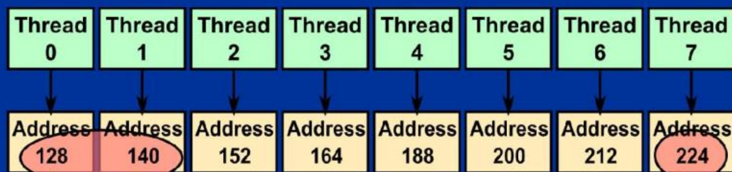
Neslučitelné přístupy do paměti



- Při přístupu není dodržen vzor k-té vlákno k-tý prvek



- Sloučení není možné kvůli nezarovnané bázové adrese



- Nezarovnaná velikost prvků
  - float3 nebo int3