KIV/OS - cvičení č. 7

Martin Úbl

12. srpna 2021

1 Obsah cvičení

- ovladače pro periferie rozšiřující desky KIV-DPP-01
  - posuvný registr
  - sedmi-segmentový displej
  - HW generátor náhodných čísel (TRNG)
  - I2C
  - OLED displej
- protokol pro ovládání displeje z uživatelského režimu

2 Posuvný registr

Na desce KIV-DPP-01 je obsažen právě jeden posuvný registr, jehož výstupy jsou přivedeny na vstup sedmi-segmentového displeje (viz dále).

Co je to ale posuvný registr? Jde o soustavu sériově propojených (v tomto případě - druhů je víc) klopných obvodů, které uchovávají stav. S každou (sestupnou nebo vzestupnou) hranou hodinového signálu se obsah těchto registrů synchronně posune o jeden registr dál, přičemž na vstup (první registr) se nasune obsah vstupního registru. Často je doprovodným signálem i latch signál, který „překlopí“ hodnoty z interních registrů do výstupních hradel. Lze tak „nasunout“ celou osmici výstupů (bitů) a pak ji propsat na výstup najednou. Tímto způsobem jsme schopni z jednoho řídicího procesoru ovládat násobný počet výstupů oproti skutečně připojeným vývodům. Posuvné registry pak jde například řadit i za sebe, a tak získat mnohem větší počet ovladatelných výstupů.

Z dokumentace 74HC595N lze vyčíst, že ovládání se provádí třemi vstupy – data in, latch a clock vstupem. Náš driver bude muset umět nasunout jak samostatný jeden bit, tak celý oktet (jelikož má celkem 8 výstupů). Naším úmyslem je pak rozsvěcet sedmi-segmentový displej, a tam je žádoucí výstup měnit naráz.

Pokud chceme měnit výstup, musíme tedy:

1. vypnout latch pin
2. na data in poslat hodnotu
3. zahýbat clock pinem s nějakým odstupem, aby se hodnota korektně nasunula
4. opakovat od bodu 2 dokud máme data na výstup
5. zapnout latch pin

Vytvořme proto třídu ovladače posuvného registru, který je inicializován konstruktorem, dá se rezervovat a uvolnit, a navíc podporuje nasunutí jednotlivého bitu a celého oktetu. Rezervace by měla zarezervovat GPIO piny, nastavit je na výstupní, a naopak uvolňovací metoda by je měla nastavit na vstupní (do výchozího stavu) a uvolnit je. Z časových a prostorových důvodů uveďme pouze nasouvací metody:

```
void CShift_Register::Shift_In(bool bit)
{
  if (!mOpened)
    return;

  volatile int i;

  sGPIO.Set_Output(mLatch_Pin, false);
  sGPIO.Set_Output(mData_Pin, bit);

  sGPIO.Set_Output(mClock_Pin, true);
  for (i = 0; i < 0x4000; i++)
    ;
  sGPIO.Set_Output(mClock_Pin, false);
  for (i = 0; i < 0x4000; i++)
    ;

  sGPIO.Set_Output(mLatch_Pin, true);
}

void CShift_Register::Shift_In(uint8_t byte)
{
  if (!mOpened)
    return;

  volatile int i;

  sGPIO.Set_Output(mLatch_Pin, false);

  for (int j = 7; j >= 0; j--)
```

```
  {
    sGPIO.Set_Output(mData_Pin, ((byte >> j) & 0x1) );

    sGPIO.Set_Output(mClock_Pin, true);
    for (i = 0; i < 0x4000; i++)
      ;
    sGPIO.Set_Output(mClock_Pin, false);
    for (i = 0; i < 0x4000; i++)
      ;
  }

  sGPIO.Set_Output(mLatch_Pin, true);
}
```

Tím bychom měli mít připravený posuvný registr. Dále je třeba připravit FS driver. To už není jádrem dnešního cvičení a proto je implementace ponechána čtenáři. Driver nahookujte na cestu `DEV:sr/0`

3 Sedmi-segmentový displej

Segmentový displej je vlastně jen uspořádání LED do vzoru, který připomíná číslici 8. V tomto vzoru můžeme zapínat a vypínat specifické části tak, že jsme schopni získat všechny cifry od 0 do 9 a možná i něco navíc. Prakticky to ponechme u klasických číslic.

V zásadě je segmentový displej jen o jednu vrstvu „dál“ – je připojený za posuvný registr, a tedy vlastně driver jen „správným“ způsobem ovládá tento registr tak, aby na výstupu segmentového displeje bylo to, co po něm chceme. Každá zobrazitelná kombinace je kódovatelná do 8 bitů. Podle dokumentace KIV-DPP-01 lze odvodit, který bit odpovídá kterému segmentu. Na výstupu budeme chtít zobrazovat cifry, a tedy budeme přebírat znak v klasické ASCII podobě. Ten přes lookup tabulku transformujeme na potřebné oktety a ty pošleme na výstup posuvného registru.

Obdobným způsobem tedy vytvořme třídu, která bude představovat driver pro segmentový displej. Opět bude umět inicializaci, rezervaci, ve které si zabere posuvný registr (a ten si zabírá po své ose GPIO piny), a uvolnění. Dále bude umět poslat na výstup znak reprezentovaný jeho ASCII hodnotou. Opět uveďme jen podstatné části.

Podle ASCII tabulky vytvořme lookup tabulku sestávající jen z tisknutelných znaků (32 a výš):

```
const uint8_t CSegment_Display::mCharacter_Map[128 - 32] = {
   0b11111111, // mezera
...
   0b11011110, // 0
```

```
    0b00010100, // 1
    0b11011001, // 2
    0b10011101, // 3
    0b00010111, // 4
    0b10001111, // 5
    0b11001111, // 6
    0b10010100, // 7
    0b11011111, // 8
    0b10011111, // 9
...
};
```

Zápis na displej pak má velmi jednoduchou podobu – načte z lookup tabulky znak a jen jeho oktet předá posuvnému registru:

```
void CSegment_Display::Write(char c)
{
  if (!mOpened)
    return;

  uint8_t idx = static_cast<uint8_t>(c);

  if (idx < 32 || idx >= 128)
    return;

  sShift_Register.Shift_In(static_cast<uint8_t>(~(mCharacter_Map[
    idx - 32])));
}
```

Nutno podotknout, že na výstup posíláme negaci – LED mají společnou katodu a výstup 0 u nich znamená „zapnuto“, kdežto intuitivně pro nás je poloha „zapnuto“ hodnota 1.

Filesystem driver opět ponechme na fantazii čtenáře – nahookujte ho na cestu `DEV:segd`

4 TRNG

Mikrokontrolér BCM2835 má integrovaný hardwarový zdroj náhodných čísel. V oficiálním manuálu však není zdokumentovaný, a tak musíme sahnout po alternativních zdrojích. Těch moc není – jeden je například přímo zdrojový soubor jádra GNU/Linux v oficiálních repozitářích Raspberry Pi (`https://github.com/raspberrypi/linux/blob/204050d0eafb565b68abf512710036c10ef1bd23/drivers/char/hw_random/bcm2835-rng.c`). Skutečné důvody neveřejnosti této

dokumentace mi nejsou známé. Další dostupný kus dokumentace, jen s nejasným původem, naleznete zde ¡TODO¿.

Princip ovládání je však vcelku stejný – generátor lze povolit zápisem do řídicího registru. Dále je třeba počkat, až dosahne potřebné úrovně entropie (aby byla generovaná čísla „dostatečně náhodná“) a pak je možné si čísla zase vyzvednout z výstupního registru. Až generátor nebudeme potřebovat, je vhodné ho zápisem do řídicího registru opět vypnout, aby zbytečně nespotřebovával elektrickou energii.

Složitější architektury operačních systémů (GNU/Linux, MS Windows, ...) často používají HW zdroje skutečné náhody jako jeden ze zdrojů pro jeden sdružený zdroj. Například na GNU/Linux existuje zařízení `/dev/random`. Čtením z tohoto zařízení čtenář získá čísla, která jsou do garantované míry náhodná. Do nich přispívá jak generátor skutečné náhody na desce, tak např. nějaké systémové metriky, které jsou dostatečně nepředvídatelné. Oproti tomu GNU-/Linux dále poskytuje zařízení `/dev/urandom`, které není zdrojem „skutečné“ náhody, ale vnitřně poskytuje čísla z matematického aparátu generátoru pseudonáhodných čísel (např. LCG, Mersenne-Twister algoritmy a jím podobné).

Pro naše potřeby zjednodušme problém na jeden zdroj náhody, a to té skutečné.

Do `peripherals.h` dodefinujme registry a bázovou adresu pro TRNG:

```
constexpr unsigned long TRNG_Base = Peripheral_Base + 0x00104000;

enum class TRNG_Reg
{
  Control   = 0,
  Status    = 1,
  Data      = 2,
  Threshold = 3,
  Int_Mask  = 4,
};
```

Z registrů výše vidíme, že obsahuje jeden `Control` registr. Nastavením bitu 1 TRNG spustíme. Druhým bitem bychom generování čísel zrychlili za cenu horší míry náhodnosti.

Dále obsahuje registr `Status` – ten v dolních 20 bitech obsahuje počet slov, které má generátor vygenerovat „na prázdno“, u kterých negarantuje míru náhodnosti. Horních 8 bitů pak obsahuje číslo, které udává počet slov (čísel), které jsou ve frontě připravené k přečtení. Čtením registru `Data` pak z této fronty jedno slovo vybereme a můžeme použít.

Další dva registry nás pro teď zajímat nebudou, ale do budoucna se k ním možná vrátíme. Registr `Int_Mask` slouží k maskování přerušení – TRNG totiž umí při určitém počtu dostupných slov ve frontě vygenerovat přerušení. Tento počet udává obsah registru `Threshold`. V budoucnu se nám může mechanismus přerušení hodit, jelikož v momentě, kdy si proces řekne o náhodné číslo, ale

žádné nebude k dispozici, máme v podstatě dvě možnosti. Buď aktivně čekat, dokud nějaké číslo nebude dostupné, nebo se přesunout do stavu „blokovaný“ a počkat právě na přerušení, které proces odblokuje. K tomu ale až jindy, pro teď přerušení vždy maskujme (zakažme, nastavme 1 do registru `Int_Mask`).

K systémovému řešení driveru pro TRNG – není dobré vyžadovat exkluzivní přístup. Entropie čísel je zaručená hardwarem (další číslo nepustí, dokud není entropie dostatečná), takže jediné, co může vícenásobný přístup způsobit, je celkové zpomalení generování čísel z pohledu všech přistupujících procesů. Stejně ale potřebujeme vědět, zda někdo čísla odebírá – nějaký příznak tedy chtít budeme. Místo prostého binárního čítače (exkluzivní přístup) zvolme celé číslo, které svou filozofií odpovídá čítači referencí – první, kdo TRNG otevře, ho zároveň inicializuje. Poslední, kdo ho zavře (čítač klesne na 0) ho zároveň deinicializuje.

Pozn.: někteří v tomto přístupu poznali semafor – momentálně zakazujeme přerušení, takže k paralelnímu otevření a zavření nedojde. V budoucnu, až budeme mít implementovaná synchronizační primitiva a blokování procesů se však k těmto kusům kódu vrátíme a zabezpečíme je.

Z důležitých kusů kódu uveďme kód inicializace:

```
volatile unsigned int* regs = reinterpret_cast<volatile unsigned
    int* const>(trng_reg_base);

regs[static_cast<uint32_t>(TRNG_Reg::Status)] = 0x40000;
regs[static_cast<uint32_t>(TRNG_Reg::Int_Mask)] |= 1;
regs[static_cast<uint32_t>(TRNG_Reg::Control)] |= 1;

while (!(mTrng_Regs[static_cast<uint32_t>(hal::TRNG_Reg::Status)]
     >> 24))
  ;
```

Ve výše uvedeném kódu lze vidět inicializaci `Status` registru na číslo `0x40000` – to je dokumentací navržené číslo, které označuje již zmíněný počet zahozených slov. Dále vypínáme přerušení a zapínáme TRNG na „běžný“ režim. Nakonec čekáme, až se entropie ustálí (je vygenerováno `0x40000` slov) – tento krok lze pak vynechat a ponechat ho pouze v kódu čtení. Nicméně vygenerovat tolik slov může být relativně časově náročné. Pak je otázka, zda chceme potenciálně blokovat při otevírání proudu, nebo obětovat tento čas při prvním čtení.

Dále kód pro deinicializaci:

```
regs[static_cast<uint32_t>(TRNG_Reg::Control)] = 0;
```

A nakonec úryvek kódu pro čtení čísel z fronty – zde opět čekáme, až ve frontě nějaké číslo bude (v budoucnu se budeme i blokovat) a pak ho z fronty vybereme:

```
while (!(regs[static_cast<uint32_t>(TRNG_Reg::Status)] >> 24))
  ;

return regs[static_cast<uint32_t>(TRNG_Reg::Data)];
```

Filesystem driver implementujte po vlastní ose, podobně jako v předchozím případě. Nahookujte ho na cestu `DEV:trng`

5 I2C

I2C (správně IIC nebo I^2C) je sériové komunikační rozhraní pro komunikaci typu master-slaves (1:N). Používá se typicky na velmi krátké vzdálenosti (jednotky centimetrů), což typicky znamená komunikaci s nějakou přímo připojenou perifierií (senzor, malý displej, přídavný řadič, ...).

Kompletní teorie a principy fungování nejsou pro nás až tak důležité a navíc se tím zabývají jiné předměty (např. KIV/NMS). Tady se zaměříme spíše na způsob implementace driveru a softwarového obalu za minimální potřebné znalosti podlehlého hardware.

I2C vyžaduje ke své funkci 2 piny zvané `SDA` (datový) a `SCL` (časovací). Ty jsou na desce RPi Zero vyvedeny na specifické piny (resp. dvojice pinů), a lze mezi nimi volit pomocí režimu pinu (lze vybrat příslušnou alternativní funkci pinu, dle dokumentace). My budeme používat I2C spolu s deskou KIV-DPP-01, na které je na I2C připojený OLED displej.

V mikrokontroléru BCM2835 je I2C schovaný v periferii s názvem BSC (Broadcom Serial Controller). Ty jsou v pouzdře celkem 3, což znamená, že máme k dispozici celkem 3 I2C kanály. Prakticky to ale neznamená komunikaci jen se třemi periferiemi – I2C umožňuje adresaci 7 nebo 10 bity. Prakticky ale bývá v periferii adresa nastavena napevno (nebo se dá měnit jen omezeně, např. propojením vývodů), takže pokud bychom chtěli mít dvě takové periferie, musíme použít jiný kanál.

Rozšiřme proto `peripherals.h` o definice bázových adres BSC a sady registrů:

```
constexpr unsigned long BSC0_Base = Peripheral_Base + 0x00205000;
constexpr unsigned long BSC1_Base = Peripheral_Base + 0x00804000;
constexpr unsigned long BSC2_Base = Peripheral_Base + 0x00805000;

enum class BSC_Reg
{
  Control       = 0,
  Status        = 1,
  Data_Length   = 2,
  Slave_Address = 3,
  Data_FIFO     = 4,
```

```
  Clock_Div     = 5,
  Data_Delay    = 6,
  CLKT          = 7,
};
```

Dále vytvořme třídu, která bude I2C driver zaobalovat – zde opět ponechme implementační detaily na čtenáři. Dbejte ale při návrhu na možnost tuto třídu parametrizovat, a to jak bázovou adresou, tak dvojicí pinů pro data a hodinový signál. BSC jako takové nevyžaduje žádnou inicializaci – stačí, když přepnete příslušné piny do jejich alternativní funkce, která odpovídá BSC. Samotné aktivování stačí provést až před samotným přenosem. Výchozí režim I2C (BSC) na desce RPi Zero je ten, se kterým je kompatibilní každá periferie – není tedy třeba nastavovat frekvenci hodin, ani jiné parametry přenosu.

Některé konstrukce a metody, které by se mohly hodit, jsou například vyčkání na dokončení operace:

```
volatile uint32_t& s = mRegs[BSC_Reg::Status];

while( !(s & (1 << 1)) )
  ;
```

Zde `mRegs` označuje ukazatel na příslušnou registrovou sadu. Dále odesílání konkrétní slave periferii (s adresou `address`, délkou bufferu `length` a bufferem `buffer`):

```
mRegs[BSC_Reg::Slave_Address] = address;
mRegs[BSC_Reg::Data_Length] = length;

for (uint32_t i = 0; i < length; i++)
  mRegs[BSC_Reg::Data_FIFO] = buffer[i];

mRegs[BSC_Reg::Status] = (1 << 9) | (1 << 8) | (1 << 1);
mRegs[BSC_Reg::Control] = (1 << 15) | (1 << 7);

Wait_Ready();
```

Do `Status` registru ukládáme hodnotu, kterou resetujeme stav *slave clock hold*, *slave fail* a *status* bitů. Do `Control` registru ukládáme příznak pro povolení BSC a započetí přenosu. Metoda `Wait_Ready()` představuje výše uvedené čekání na dokončení operace.

Poté bude dobré umět data i přijmout. Kód vypadá podobně:

```
mRegs[BSC_Reg::Slave_Address] = address;
```

```
mRegs[BSC_Reg::Data_Length] = length;

mRegs[BSC_Reg::Status] = (1 << 9) | (1 << 8) | (1 << 1);
mRegs[BSC_Reg::Control] = (1 << 15) | (1 << 7) | (1 << 4) | (1 <<
     0);

Wait_Ready();

for (uint32_t i = 0; i < length; i++)
  buffer[i] = mRegs[BSC_Reg::Data_FIFO];
```

Kód výše nastavuje očekávanou délku dat, adresu odesílatele (od koho data očekáváme) a nastavuje identický obsah `Status` registru. Do `Control` registru však ještě přibydou bity pro indikaci operace čtení a vyprázdnění fronty před samotným příjmem. Následně je vyčkáno na dokončení operace, po které lze z datového registru znak po znaku přečíst vstup. Zde opatrně na optimalizace kompilátoru – tady je kritické označení `mRegs` jako `volatile`. Kompilátor by se mohl domnívat, že pravá strana výrazu v přiřazení se nemá šanci během iterace cyklu změnit, a tak by optimalizoval celý cyklus na jednorázové přiřazení. S klíčovým slovem `volatile` mu podobné optimalizace nedovolíme.

Co je dále vhodné zohlednit je to, že se I2C provoz obvykle realizuje v obalu, který vzdáleně připomíná transakce. Často totiž potřebujeme logicky oddělit nějaké operace, ale příkazy, které tyto operace generují, musí být přeneseny najednou. Toto chování bude očekávat právě například připojený OLED displej.

Definujme proto třídu transakce, která může vypadat třeba takto:

```
constexpr uint32_t I2C_Transaction_Max_Size = 8;

class CI2C_Transaction
{
  friend class CI2C;
  private:
    bool mIn_Progress = false;
    uint8_t mBuffer[I2C_Transaction_Max_Size];
    uint32_t mLength = 0;
    uint16_t mAddress = 0;

  public:
    CI2C_Transaction() = default;

    void Set_Address(uint16_t addr)
    {
      mAddress = addr;
    }
```

```
    template<typename T>
    CI2C_Transaction& operator<<(const T& chr)
    {
      if (mLength >= I2C_Transaction_Max_Size)
        return *this;

      mBuffer[mLength++] = static_cast<uint8_t>(chr);

      return *this;
    }
};
```

Není to nic vyloženě složitého a ve své podstatě jde jen o jakousi syntaktickou kličku, abychom se nezaobírali přetypováním a pozicí v bufferu a mohli data jen skládat za sebe.

Z pohledu C++ (tady spíše pro ty, kteří C++ moc nepoužívají) jsou zde dvě výrazné věci – deklarace I2C třídy jako `friend` a šablonový přetížený operátor `<<` níže. Označení `friend` pro I2C třídu je zde proto, aby si mohla I2C třída sahnout pro buffer, reálnou délku a adresu, ačkoliv jde o privátní atributy. Šablonové přetížení operátoru `<<` je zde proto, abychom mohli knihovně-obvyklým způsobem do bufferu poslat cokoliv, co lze staticky přetypovat na `uint8_t`, a vnitřní logika se sama postarala o přetypování i kontrolu toho, že tato typová konverze je vůbec možná (kompilátor by se ozval).

Integrace této transakční třídy už je opět ponecháno jako cvičení pro čtenáře. Pokuste se ale zachovat standardní jmenné konvence – `Begin_Transaction()`, `Commit_Transaction()` a `Abort_Transaction()` (případně souhrnná metoda `End_Transaction()` s parametrem příznaku, zda transakci provést či ne).

6 OLED displej

Na desce KIV-DPP-01 je připojený displej s označením Adafruit SSD1306. Ten je připojen na I2C kanál 1 (druhé BSC) na piny 2 a 3. V tomto předmětu nemá moc smysl rozebírat, jak takový displej funguje – to je obsahem jiných předmětů a nebo si můžete funkci displeje nastudovat po vlastní ose. V tento moment však není důležité, jak přesně displej funguje, ani jaké konkrétní příkazy mu musíme poslat, aby se něco ukázalo.

Zdrojové kódy, které volně integrujte do svého projektu, si můžete stahnout zde: ¡TODO¿.

Poskytnuté kódy obsahují pro teď vše, co je třeba ke zobrazení sady pixelů na výstup. Pochopitelně lze část ještě zefektivnit tím, že budeme překreslovat pouze část displeje (a ne vždy celý buffer) a tak podobně.

Co je ale důležité je to, jak takový displej zpřístupnit uživatelským procesům. Jednou z možností je (resp. bylo by, až bychom měli implementované stránkování) namapovat buffer určený pro zobrazovací driver do paměťového

prostoru procesu. To ale nemusí být úplně elegantní a univerzální – buffer u větších displejů často ani není alokovaný celý (nebo ani neexistuje) a jednotlivé vykreslovací příkazy se posílají okamžitě. Navíc často potřebujeme mít i možnost, jak ovládat parametry displeje, jako například jas, převrácení, a tak podobně.

Sice trochu pracnějším, ale zato univerzálním řešením je definice protokolu, který bude realizován nad souborovým systémem. Jinak řečeno uživatelský proces si otevře soubor, který představuje jednotku displeje. Do něj bude zapisovat zprávy protokolu s daným významem (zapiš pixel, vymaž displej, ...) a filesystem driver tyto zprávy bude dekódovat a předávat ovladači displeje. Pokud se rozhodneme, že protokol bude pro všechny druhy displejů stejný, lze samotné dekódování odsunout až do ovladače displeje (a tedy interpretaci příkazů), nebo naopak vytvořit společnou dekódovací mezivrstvu a pro ovladače displejů vytvořit rozhraní (to je rozhodně systematičtější řešení).

Z důvodu jednoduchosti a omezeného času na cvičení proveďme první možnost – dekódování protokolových zpráv v ovladači displeje, do kterého je bude předávat jeho filesystem driver. Návrhově korektní by však bylo spíše druhé řešení. Tím lze mnohem lépe generalizovat celý zobrazovací subsystém operačního systému, a do budoucna tak třeba zjednodušit implementaci „velkých“ displejů připojených na rozhraní HDMI, kterým RPi Zero taktéž disponuje.

6.1 Protokol pro uživatelské procesy

Co od displeje vlastně potřebujeme? Sada operací je vesměs docela malá:

- vymazat displej
- „propsat“ interní buffer na výstup displeje
- nakreslit pixel
- vykreslit bitmapu do obdélníku – abychom nemuseli pro každý pixel realizovat relativně drahé systémové volání

Pro teď počítejme, že máme displej pouze monochromatický (a pro každý pixel tedy teoreticky stačí 1 bit) s relativně malými rozměry, aby se souřadnice pixelu vešla do 16-bitového čísla.

Definujme si teď hlavičkový soubor `display_protocol.h` a v něm vytvořme výčet příkazů:

```
enum class NDisplay_Command : uint8_t
{
  Nop                     = 0,
  Flip                    = 1,
  Clear                   = 2,
  Draw_Pixel_Array        = 3,
  Draw_Pixel_Array_To_Rect = 4,
};
```

Všimněte si přidané operace `Nop` – je zvyklostí operaci s číslem nula vyhradit pro prázdnou operaci.

Dále definujme hlavičku zprávy protokolu:

```
struct TDisplay_Packet_Header
{
  NDisplay_Command cmd;
};
```

Ta je, jak vidíte, velmi jednoduchá – obsahuje pro teď jen číslo operace.

Můžeme také definovat strukturu pro nastavovaný pixel:

```
struct TDisplay_Pixel_Spec
{
  uint16_t x;
  uint16_t y;
  uint8_t set;
};
```

A pak stačí definovat struktury zpráv všech ostatních operací:

```
struct TDisplay_NonParametric_Packet
{
  TDisplay_Packet_Header header;
};

struct TDisplay_Clear_Packet
{
  TDisplay_Packet_Header header;
  uint8_t clearSet;
};

struct TDisplay_Draw_Pixel_Array_Packet
{
  TDisplay_Packet_Header header;
  uint16_t count;

  // tady bude nasledovat pole TDisplay_Pixel_Spec
};

struct TDisplay_Pixels_To_Rect
{
```

```
  TDisplay_Packet_Header header;
  uint16_t x1, y1;
  uint16_t w, h;

  // tady bude nasledovat pole bitu (za sebou serializovane v
    bajtech zleva), tzn. na 8 pixelu staci 1 bajt, prvni pixel je
     nejvyssi bit v bajtu
};
```

Význam struktur a jejich členských atributů si lze snadno domyslet. U „clear“ zprávy je navíc atribut `clearSet`, který říká, zda displej vymazat černou nebo „bílou“ barvou.

Nyní stačí protokol integrovat jak do driveru displeje, tak do uživatelských procesů. První část integrace proveďte dle uvážení sami. Na příštím cvičení bude kód k dispozici. Druhé části integrace se budeme věnovat v příštím cvičení – budeme psát sdílenou standardní knihovnu, která obsahuje mimo jiné i tento protokol pro displej.

7 Úkol za body

Implementujte ovladač pulzně-šířkové modulace a ověřte ho na rozšiřující desce KIV-DPP-01 (resp. jejím pasivním bzučáku). Pokud nemáte k dispozici potřebný HW vzhledem k omezenému počtu kusů, stačí, když implementujete základ bez ověření – hodnotit se bude práce s manuálem a dodržení návrhové filozofie systému.