KIV/OS - cvičení č. 2

Martin Úbl

26. července 2021

1 Obsah cvičení

- struktura projektu OS
- CMake
- miniUART a bootloader
- inicializace jádra a C++ v „bare“ projektu
- driver pro GPIO

2 Struktura projektu

Jelikož budeme vytvářet poměrně složitý projekt, je vhodné si včas rozmyslet adresářovou a logickou strukturu. Nemusíme však nic přehánět – nebudeme psát druhý Linux. Určitě je ale vhodné rozdělit složku se zdrojovými soubory a složku s hlavičkovými soubory, a to primárně z důvodu, že vybrané moduly mohou vyžadovat hlavičky jaderných modulů pro svou práci (např. pro definice nějakých konstant a struktur), ale už nemusí vidět implementaci. Všechny jaderné soubory umístíme do adresáře `kernel`. Pro snazší distribuci v něm oddělíme hlavičkové soubory do adresáře `include` a zdrojové soubory do složky `src`. Na stejnou úroveň vložíme skript pro linker `link.ld`.

Pro potřeby předmětu budeme vyvíjet monolitické jádro systému reálného času. Respektive – nejprve půjde o systém interaktivní (pro který implementujeme time-slicing plánovač) a ten poté přetvoříme v systém reálného času v druhé polovině semestru.

Pro teď nechme zdrojové soubory specifické pro jádro přímo ve složce `src`. Tam vytvořme soubor `start.s` obsahující náš vstupní bod do jádra (viz minulé cvičení) a soubor `main.cpp` (obsahující pro teď zbytek implementace). Později dnes zde ještě přibydou další soubory.

Nyní vytvořme adresář `drivers` v obou složkách (`src` i `include`). Sem budeme vkládat zdrojové a hlavičkové soubory ovladačů periferií.

Ve složce `include` by bylo dobré oddělit definice pro různé varianty desky, kdybychom se například rozhodli podporovat kromě RPi Zero i třeba jinou

edici RPi nebo klidně i úplně jinou desku. V zájmu obecnosti proto vytvořme složku `board` a v ní podsložku pro každou podporovanou desku – pro nás je to teď jen `rpi0`. Návod pro sestavení by pak měl nastavovat *include* cesty tak, aby směřovaly vždy i do příslušného adresáře podle desky. Zde budeme tvořit jakousi HAL (Hardware Abstraction Layer), tedy vrstvu, která odstíní konkrétní způsob ovládání periferií od rozhraní. V našem případě vzhledem k poměrně unifikovanému rozhraní RPi to budou pouze bázové adresy a offsety pro různé periferie a další detaily. Vytvořme proto podsložku `hal` a v ní soubor `peripherals.h`. Rovněž můžeme vytvořit soubor `intdef.h`, kde budeme definovat datové typy s pevnou (známou) délkou pro daný typ desky.

Do kořenového adresáře celého projektu ještě vložíme návody k sestavení (`Makefile` nebo za chvíli spíše `CMakeLists.txt`) a volitelně skript, kterým budeme řešení sestavovat (v mém případě `build.sh`)

Výsledná struktura by měla vypadat zhruba takto:

```
.
+-- kernel +-- include +-- board +-- rpi0 +-- hal +-- intdef.h
|          |           |                        +-- peripherals.h
|          |           +-- drivers
|          +-- src +-- drivers
|          |       +-- main.cpp
|          |       +-- start.s
|          +-- link.ld
+-- build.sh
+-- CMakeLists.txt
```

3 CMake

Pro sestavení projektu budeme potřebovat nějaký nástroj a „recept“ pro něj, a to ideálně takový, abychom nemuseli vše ručně doplňovat při každé změně ve struktuře. Pro tyto účely se hodí nástroj CMake. Pokud nechcete používat CMake, můžete použít a vhodně modifikovat `Makefile` z minulého cvičení, nebo si napsat vlastní recept ve Vámi oblíbeném nástroji – stačí, když zvládnete specifikovat kompilátor, kterým chcete zdrojové soubory překládat a dodat jim příslušné přepínače.

Pro potřeby demonstrace budu používat nástroj CMake. Pokud byste se rozhodli pro tentýž nástroj, lze ho stahnout pro Windows na adrese ¡TODO¿, popř. ve většině balíčkovacích systémů pro Linux nebo macOS ho lze najít pod prostým názvem `cmake`. Na Debian-based (apt-based) distribucích ho můžeme nainstalovat příkazem

```
apt install cmake
```

Po instalaci stačí specifikovat soubor `CMakeLists.txt` a dodat toolchain soubor. Toolchain soubor pro náš překlad v ARM-specific gcc stahnete např. zde: ¡TODO¿.

Obsah `CMakeLists.txt` souboru může vypadat např. takto:

```
CMAKE_MINIMUM_REQUIRED(VERSION 3.0)

PROJECT(kiv_os_rpios CXX C ASM)

SET(LINKER_SCRIPT
    "${CMAKE_CURRENT_SOURCE_DIR}/kernel/link.ld")
SET(CMAKE_EXE_LINKER_FLAGS
    "${CMAKE_EXE_LINKER_FLAGS} -T ${LINKER_SCRIPT}")

INCLUDE_DIRECTORIES(
    "${CMAKE_CURRENT_SOURCE_DIR}/kernel/include/")

INCLUDE_DIRECTORIES(
    "${CMAKE_CURRENT_SOURCE_DIR}/kernel/include/board/rpi0/")
ADD_DEFINITIONS(-DRPI0)

FILE(GLOB_RECURSE kernel_src "kernel/*.cpp" "kernel/*.c"
     "kernel/*.h" "kernel/*.hpp" "kernel/*.s")

ADD_EXECUTABLE(kernel ${kernel_src})

ADD_CUSTOM_COMMAND(
  TARGET kernel POST_BUILD
  COMMAND ${CMAKE_OBJCOPY} ./kernel${CMAKE_EXECUTABLE_SUFFIX}
      -O binary ./kernel.img
  COMMAND ${CMAKE_OBJDUMP} -l -S -D
      ./kernel${CMAKE_EXECUTABLE_SUFFIX} > ./kernel.asm
  COMMAND ${CMAKE_OBJCOPY} --srec-forceS3
      ./kernel${CMAKE_EXECUTABLE_SUFFIX} -O srec kernel.srec
  WORKING_DIRECTORY
      ${CMAKE_BINARY_DIR})
```

Většina příkazů je poměrně jasná – nastavíme vyžadovanou verzi nástroje CMake, jméno projektu a jazyky v něm obsažené, skript který se má předat linkeru (paměťová mapa) a přepínače a include cesty obecné pro celý projekt. Následují dva příkazy specifické pro překlad pro RPi Zero (resp. takto jsme si je navrhli) – include cesty do specifické `boards` složky a nastavení preprocesorového makra `RPI0`.

Poté nalezneme všechny soubory v podsložce `kernel` vyhovující filtru a přidáme target s názvem `kernel`, který je zahrnuje (kompiluje).

Následuje vlastní post-build sekvence příkazů, kterými transformujeme výstup do obrazu paměti k nahrání (`kernel.img`), pak provedeme zpětný překlad do assembly (`kernel.asm`), abychom mohli výsledný překlad zkoumat, a následuje řádka, kterou budeme potřebovat až později v tomto cvičení – transformace binárního obrazu do Motorola S-record formátu (`kernel.srec`). Tato transformace přijde vhod v momentě, kdy budeme chtít nahrávat obraz jádra přes

komunikační rozhraní.

Poté je potřeba jen nějak spustit sestavovací proces spolu s toolchain souborem. Proto jsme definovali soubor `build.sh` v minulé kapitole. Jeho obsah může vypadat například takto:

```
#!/bin/bash

mkdir -p build >/dev/null 2>&1
cd build

cmake -G "Unix Makefiles" \
 -DCMAKE_TOOLCHAIN_FILE="toolchain-arm-none-eabi-rpi0.cmake" \
  ..

make
```

Pokud jsme všechno správně nastavili a umístili, výsledkem bude mj. soubor `kernel.img` a `kernel.srec` v podsložce `build`.

4 UART bootloader

Jedním z nepříjemných aspektů vývoje pro RPi může být to, že je jádro nutné uložit na SD kartu – to vyžaduje opětovné přenášení karty do čtečky pro PC, nahrávání souboru a přenos zpět (často označováno anglickým termínem *SD card dance*). S tím je pochopitelně spojeno mechanické opotřebení SD karty, čtečky, slotu na desce RPi Zero a podobné.

Bylo by proto dobré minimálně pro prvotní fázi vývoje najít způsob, jak toto přendavání minimalizovat nebo úplně odstranit. V budoucnu, až by bylo jádro stabilní a mělo konektivitu do počítačové sítě lze uvažovat např. o modularizaci jádra a nahrazování komponent na SD kartě přes síť.

Pro teď ale zvolíme způsob, který je dosti oblíbený ve světě embedded zařízení – využijeme USB-TTL převodník pro PC spolu s jeho USB rozhraním, a na straně RPI využijeme UART rozhraní. Vytvoříme tak stále spojení vývojového PC s RPi přes sériový port, kterého můžeme dále využít. Na straně počítače je to snadné - můžeme klasicky zapisovat a číst sériový port. RPi ale vyžaduje obslužný program, který z tohoto portu něco přijme, zapíše to do paměti a následně je schopen provést skok na dané místo v paměti, které je vstupním bodem do nahraného programu. Tento program nazveme *bootlader*, fakticky se jedná o jakousi druhou část bootovacího procesu na straně systému.

Rovněž je vhodné vybrat nějaký formát pro přenos jádra. Již výše jsme zmínili formát Motorola S-record, který je textovým formátem a obsahuje základní značky pro začátek přenosu, uložení datového balíčku na danou adresu a příkaz pro skok na vstupní bod. Je tedy pro tyto účely vhodným kandidátem.

Potřebujeme tedy bootloader:

- s minimalistickým UART ovladačem

- s dekodérem S-record formátu
- transparentní – je schopen být zavedený v paměťovém prostoru a zároveň nahrát do téhož paměťového prostoru jádro

Jakmile takový bootloader budeme mít, stačí z hostitelského počítače zapsat celý soubor `kernel.srec` na sériový port a tím by mělo být jádro nahráno a zavedeno.

UART driveru se budeme věnovat později při integraci do jádra systému. Dekodér S-record formátu je opět relativně snadnou záležitostí a proto se tím více zabývat nebudeme – na zdrojové kódy a obraz bootloaderu naleznete odkaz níže v této kapitole.

Zajímavou částí je však transparentnost bootloaderu vzhledem k jádru. Jak jsme již uvedli v minulém cvičení, „stage 1“ bootloaderu RPi očekává, že bude program nahrán tak, že na adrese `0x8000` bude jeho vstupní bod. My ale nechceme přizpůsobovat jádro pro běh s naším bootloaderem nebo bez, a tedy jeho vstupní bod budeme určitě situovat na adresu `0x8000`. Bootloader se ale rovněž musí nějak spustit, a vzhledem k očekávanému startu na dané adrese musí i jeho kód být umístěn na adrese `0x8000`. Při nahrávání kódu jádra nesmí dojít k přepsání kódu, který jádro nahrává. Jinak hrozí, že se začne „najednou“ provádět úplně jiný kód, který ovšem předpokládá úplně jiný stav systému a nejspíše dojde k selhání (zaseknutí).

Řešení je poměrně snadné - kód bootloaderu můžeme umístit vlastně kam chceme. Na adrese `0x8000` pak stačí, když bude obsažena právě jedna instrukce skoku na námi zvolenou adresu. Pak jen stačí vybrat pro kód bootloaderu takovou adresu, která nebude přepsána kódem jádra. Zvolíme proto například `0x200000`. V kódu pak lze vidět tento odskok jako vložení prázdného místa mezi vstupní bod a vlastní výkonný kód:

```
.global _start
_start:
  b skip

.space 0x200000-0x8004,0

skip:
  mov sp,#0x08000000
  bl loader_main
```

Problém by pochopitelně nastal, kdyby kód našeho jádra byl větší, než cca 2 MiB (`0x200000 - 0x8000`). Tento problém lze řešit posunutím adresy o kus dál. Naše jádro však stěží překročí tuhle velikost a tak můžeme pro teď být v klidu.

Bootloader (zdrojové soubory i obraz) si můžete stahnout zde: ¡TODO¿

5 Inicializace jádra a C++

V běžných programech jsme zvyklí, že za nás spousty práce odvede buď operační systém sám nebo tak učiní tzv. CRT0 (C-Runtime 0). Nyní se budeme orientovat na část druhou – v té musíme nastavit běhové prostředí tak, aby došlo ke správné inicializaci prostředí tak, že budeme moci pracovat se staticky inicializovanými entitami „jak jsme zvyklí“. Jádro navíc budeme psát v C++, a tak je třeba učinit ještě dodatečné kroky.

Budeme potřebovat:

1. zajistit, že vstupní bod jádra bude vždy na adrese `0x8000`
2. vynulovat obsah sekce `bss`
3. zavolat konstruktory globálních instancí tříd při inicializaci
4. zavolat destruktory globálních instancí tříd při deinicializaci (k tomu teď sice nedojde, ale připravme si to)
5. definovat minimalistické C++ ABI
   - pro zabránění rekurzivní inicializace
   - pro pokus o volání čistě virtuální metody

Bod 1 lze vyřešit snadno – buď můžeme použít nějakou z direktiv překladače (hint, atribut), aby umístil symbol na danou adresu, a nebo (dle mého jistější cesta) pro daný symbol vytvořit samostatnou sekci (např. `text.start`) a tu v linker skriptu umístit na pevně dané místo. Kód `start.s` pak může vypadat třeba takto:

```
.global _start

;@ tato sekce se vlozi na adresu 0x8000
.section .text.start

_start:
  mov sp,#0x8000
  bl _kernel_main
hang:
  b hang

.section .text
;@ ...
```

V linker skriptu pak sekci umístíme před samotnou `text` sekci – jelikož je `text` první, a blok `ram` začíná na adrese `0x8000`, k umístění dojde dle očekávání:

```
.text :
{
  *(.text.start*)
```

```
  *(.text*)
} > ram
```

Do skriptu ještě přidáme zarážky a direktivy, které pomohou vyřešit bod 2 - start a konec `bss` sekce a její umístění za `text` sekci:

```
_bss_start = .;

.bss :
{
        *(.bss*)
} > ram

_bss_end = .;
```

Zarážky `_bss_start` a `_bss_end` jsou pak součástí procesu linkování – lze je tedy v C++ kódu definovat jako externí symboly a budou nám normálně zpřístupněny.

Stejně tak rovnou můžeme vložit speciální sekci pro konstruktory a destruktory globálních instancí tříd. Jejich volání generuje sám kompilátor do sekce `text` a adresy těchto generovaných volání vkládá ve formě seznamu do virtuálních sekcí `ctors` (`init_array`) a `dtors` (`fini_array`). Linkerem je vložíme do sekce `data` společně se zarážkami, abychom viděli, kde seznamy začínají a končí:

```
.data :
{
  __CTOR_LIST__ = .; *(.ctors) *(.init_array) __CTOR_END__ = .;
  __DTOR_LIST__ = .; *(.dtors) *(.fini_array) __DTOR_END__ = .;
  data = .;
  _data = .;
  __data = .;
  *(.data)
} > ram
```

Zarážky si opět budeme moci vyzvednout v kódu jako externí symboly.

Nyní přejdeme k samotné inicializaci. Vytvořme proto nový soubor `startup.cpp`, kde ji implementujeme. Začneme sekcí `bss` a jejím vyplněním nulami. K tomu budeme potřebovat symboly `_bss_start` a `_bss_end`. Jde o číselné hodnoty adresy, které vymezují začátek a konec sekce `bss`. Teď již stačí pouze vynulovat vše mezi (např. v nějaké námi definované funkci `_c_startup`):

```
extern "C" int _bss_start;
extern "C" int _bss_end;

extern "C" int _c_startup(void)
{
  int* i;
```

```
  for (i = (int*)_bss_start; i < (int*)_bss_end; i++)
    *i = 0;

  return 0;
}
```

Následně je nutné nějak zavolat konstruktory globálních instancí tříd. Jak bylo zmíněno, C++ kompilátor generuje tato volání jako bezparametrické funkce bez návratové hodnoty, jejichž adresy vkládá do sekce k tomu určené. Každá instance globální třídy má tak vygenerovanou svou funkci, která volá konstruktor s danými parametry. V linker skriptu jsme si adresy nechali vložit mezi zarážky `__CTOR_LIST__` a `__CTOR_END__`. Zde se nachází seznam ukazatelů na tyto funkce. Nezbývá tedy než ho projít a všechny funkce zavolat.

```
extern "C" ctor_ptr __CTOR_LIST__[0];
extern "C" ctor_ptr __CTOR_END__[0];

extern "C" int _cpp_startup(void)
{
  ctor_ptr* fnptr;

  for (fnptr = __CTOR_LIST__; fnptr < __CTOR_END__; fnptr++)
    (*fnptr)();

  return 0;
}
```

Analogicky opakujme postup s destruktory:

```
extern "C" dtor_ptr __DTOR_LIST__[0];
extern "C" dtor_ptr __DTOR_END__[0];

extern "C" int _cpp_shutdown(void)
{
  dtor_ptr* fnptr;

  for (fnptr = __DTOR_LIST__; fnptr < __DTOR_END__; fnptr++)
    (*fnptr)();

  return 0;
}
```

Nakonec musíme definovat určité prvky C++ ABI tak, abychom mohli využívat všechny prvky jazyka samotného. Pro větší rozbor těchto funkcí se podívejte do příručky kompilátoru – ve zkratce jde o zabránění rekurzivní inicializace, odchycení volání čistě virtuálních metod a registraci destruktorů objektů ve sdílených knihovnách. My si navíc těla jednotlivých funkcí zjednodušíme, protože reálně je v jádře využívat nebudeme - symboly jen musí být definované, aby byl

kompilátor (linker) spokojený. Více o propojení s C++ můžete nalézt zde: `http://wiki.osdev.org/C++`.

Zde je obsah minimalistického C++ ABI pro kompilátor gcc (g++):

```
namespace __cxxabiv1
{
  __extension__ typedef int __guard __attribute__((mode(__DI__)));

  extern "C" int __cxa_guard_acquire (__guard *);
  extern "C" void __cxa_guard_release (__guard *);
  extern "C" void __cxa_guard_abort (__guard *);

  extern "C" int __cxa_guard_acquire (__guard *g)
  {
    return !*(char *)(g);
  }

  extern "C" void __cxa_guard_release (__guard *g)
  {
    *(char *)g = 1;
  }

  extern "C" void __cxa_guard_abort (__guard *)
  {
  }
}

extern "C" void __dso_handle()
{
}

extern "C" void __cxa_atexit()
{
}

extern "C" void __cxa_pure_virtual()
{
}

extern "C" void __aeabi_unwind_cpp_pr1()
{
  while (true)
    ;
}
```

Pro uživatelské programy budeme definovat implementaci C++ ABI odlišnou.

Nakonec je pochopitelně třeba upravit vstupní bod jádra tak, aby volal ini-

cializaci ve správném místě a pořadí:

```
_start:
  mov sp,#0x8000
  bl _c_startup
  bl _cpp_startup
  bl _kernel_main
  bl _cpp_shutdown
hang:
  b hang
```

Pozn.: z funkce `_kernel_main` nečekáme, že bychom se kdy vrátili, takže volání `_cpp_shutdown` a cyklení v `hang` je vlastně nadbytečné.

6 GPIO driver

Driver, resp. česky *ovladač* je modul operačního systému, který co možná nejvíce obecně ovládá danou periferii a zprostředkovává rozhraní zbytku systému, skrze které ji lze řídit. Ovladač by měl odstínit silně specifické úkony (jako např. přístup na konkrétní paměť a zápis konkrétních hodnot) a zamaskovat je za vyšší abstrakci – aby např. v případě GPIO existovalo rozhraní pro „zapni výstup 12“ a nemuseli jsme znát bázovou adresu pro registry, díky kterým se vůbec může výstup zapnout, a tak podobně.

GPIO je zkratka pro *General Purpose Input Output*, tedy vstupy a výstupy (piny) pro obecné použití. Některé z těchto pinů mohou být vyhrazené pro speciální funkci, když ji programově vybereme. Například komunikace skrze I2C s hardwarovou podporou je schopno jen několik předvybraných kombinací pinů. V tomto případě hovoříme o tzv. alternativní funkci pinů.

RPi Zero má header dobře zdokumentovaný (viz např. `https://pinout.xyz/`). Na něj nasadíme rozšiřující desku KIV-DPP-01, takže konkrétní piny vlastně ani neuvidíme, ale budeme vždy pracovat s něčím, co je na ně připojeno.

Pin může mít nastavený některý z uvedených režimů:

- vstupní (výchozí)
- výstupní
- alternativní funkce 0 - 5

V manuálu se dočteme, že tento režim se nastavuje zápisem do příslušného registru `GPFSEL` (0-5) na danou bitovou pozici. Vždy jde o zápis 3-bitové hodnoty na konkrétní místo odpovídající konkrétnímu pinu.

Pro ovládání pinu pak potřebujeme pro výstup registry `GPSET` a `GPCLR` (0 nebo 1) a pro čtení vstupu pak registr `GPLEV` 0 nebo 1.

Jelikož cílíme na obecnost alespoň v této rovině, bylo by dobré si v příslušném `hal` souboru `peripherals.h` definovat několik konstant, které jsou specifické pro RPi Zero.

Určitě si tam definujme bázi všech memory-mapped IO, jelikož od té budeme vždy počítat umístění všech subsystémů:

```
constexpr unsigned long Peripheral_Base = 0x20000000UL;
```

Dále definujme bázi pro GPIO řadič (viz dokumentace BCM2835):

```
constexpr unsigned long GPIO_Base = Peripheral_Base + 0x00200000UL;
```

A rovnou můžeme definovat sadu základních registrů pro GPIO (existuje jich samozřejmě víc, ale k tomu až jindy):

```
enum class GPIO_Reg
{
  // vyber funkce GPIO pinu
  GPFSEL0    = 0,
  GPFSEL1    = 1,
  GPFSEL2    = 2,
  GPFSEL3    = 3,
  GPFSEL4    = 4,
  GPFSEL5    = 5,
  // registry pro zapis "nastavovaciho priznaku"
  GPSET0     = 7,
  GPSET1     = 8,
  // registry pro zapis "odnastavovaciho priznaku"
  GPCLR0     = 10,
  GPCLR1     = 11,
  // registry pro cteni aktualniho stavu pinu
  GPLEV0     = 13,
  GPLEV1     = 14,
};
```

Pak si definujme nové soubory `gpio.h` a `gpio.cpp` v příslušných adresářích. **Nutno dodat, že konkrétní podoba rozhraní a implementace je ponechána na čtenáři** – zde pouze představíme princip fungování. Kódy ze cvičení budou k dispozici v ucelené podobě na předem domluvené adrese.

Určitě budeme potřebovat funkci pro nastavení režimu pinu. Definujme si proto výčtový typ v hlavičce:

```
enum class NGPIO_Function : uint8_t
{
  Input = 0,  // 000 - vstupni pin
  Output = 1, // 001 - vystupni pin
  Alt_5 = 2,  // 010 - alternativni funkce 5
  Alt_4 = 3,  // 011 - alternativni funkce 4
  Alt_0 = 4,  // 100 - alternativni funkce 0
  Alt_1 = 5,  // 101 - alternativni funkce 1
  Alt_2 = 6,  // 110 - alternativni funkce 2
  Alt_3 = 7,  // 111 - alternativni funkce 3
};
```

A odpovídající funkci pro nastavení režimu – z dokumentace se dočteme, že pro každý pin jsou vyhrazeny 3 konkrétní bity v konkrétním registru:

```
void Set_GPIO_Function(int pin, NGPIO_Function func)
{
  GPIO_Reg reg;
  int bit;

  volatile unsigned long* GPIO =
     reinterpret_cast<unsigned long*>(GPIO_Base);

  switch (pin / 10)
  {
    case 0: reg = GPIO_Reg::GPFSEL0; break;
    case 1: reg = GPIO_Reg::GPFSEL1; break;
    case 2: reg = GPIO_Reg::GPFSEL2; break;
    case 3: reg = GPIO_Reg::GPFSEL3; break;
    case 4: reg = GPIO_Reg::GPFSEL4; break;
    case 5: reg = GPIO_Reg::GPFSEL5; break;
  }

  bit = (pin % 10) * 3;

  const int reg_idx = static_cast<const int>(reg);

  const unsigned long old =
    GPIO[reg_idx] & (~static_cast<unsigned int>(7 << bit));

  GPIO[reg_idx] =
    old | (static_cast<unsigned int>(func) << bit);
}
```

Dále vytvořme funkce pro nastavení výstupu na vysokou nebo nízkou úroveň (zapnuto nebo vypnuto):

```
void Set_Output(int pin, bool set)
{
  int reg, bit;

  volatile unsigned long* GPIO =
    reinterpret_cast<unsigned long*>(GPIO_Base);

  if (set)
       reg = static_cast<uint32_t>((pin < 32) ?
                GPIO_Reg::GPSET0 :
                GPIO_Reg::GPSET1);
  else
```

```
        reg = static_cast<uint32_t>((pin < 32) ?
                GPIO_Reg::GPCLR0 :
                GPIO_Reg::GPCLR1);

  bit_idx = pin % 32;

  GPIO[reg] = (1 << bit);
}
```

V poslední řadě chceme funkci pro zjištění, zda je vstup zapnutý nebo vypnutý:

```
bool Get_Output(int pin)
{
  int reg, bit;

  volatile unsigned long* GPIO =
    reinterpret_cast<unsigned long*>(GPIO_Base);

  reg = static_cast<uint32_t>((pin < 32) ?
    GPIO_Reg::GPLEV0 :
    GPIO_Reg::GPLEV1);
  bit_idx = pin % 32;

  return (GPIO[reg] >> bit) & 0x1;
}
```

7 Úkol za body

Z kódu výše lze vidět, že obsahuje spousty redundantních částí a kódu, který není úplně vzhledný. Přepište kód GPIO driveru tak, aby byl co nejobecnější, čitelný, a využíval objektového paradigmatu jazyka C++ (uzavřete kód do třídy). Taktéž využijte toho, že se volají konstruktory globálních objektů. Myslete rovněž na to, že obvykle může zařízení obsahovat více zařízení stejného typu – bylo by tedy dobré nějak ovladač a jeho inicializaci parametrizovat.

Rovněž lze vidět, že je v kódu spousty datových typů bez konkrétní délky – do souboru `intdef.h` definujte typy pro 8, 16 a 32 bitové číselné hodnoty se znaménkem i bez znaménka.