

All-in-one štartovacia sada pre ESP32-P4

- Rýchle nasadenie umelej inteligencie
- 2MP kamera | 7-palcový displej
- Hardvér s otvoreným zdrojovým kódom



20+

lekcií

16

Vstavané RYX COMPONENTS



• Úvod	1
• ESP-IDF Príručka na nastavenie prostredia	2
• Lekcia 1 – Ovládanie LED GPIO	6
• Lekcia 2 – Ovládanie relé GPIO	19
• Lekcia 3 – Prepínanie dotykových tlačidiel	29
• Lekcia 4 – Ovládanie pohybu PIR	38
• Lekcia 5 – Detekcia Hallovho senzora	47
• Lekcia 6 – Ovládanie sériového LED	56
• Lekcia 7 – Blikanie LED časovača	64
• Lekcia 8 – Ovládanie serva PWM	72
• Lekcia 9 – LCD displej Hello	81
• Lekcia 10 – Ultrazvukový displej vzdialenosti	95
• Lekcia 11 – Teplota a vlhkosť DHT20	109
• Lekcia 12 – Svetelný senzor BH1750	126
• Lekcia 13 – LSM6DS3 Gyroskopický displej	136
• Lekcia 14 – Ovládanie WS2814 RGBW	155
• Lekcia 15 – Ovládanie tlačidiel ADC	164
• Lekcia 16 – Výstraha senzora dymu	174
• Lekcia 17 – Nahrávanie zvuku I2S	184
• Lekcia 18 – Prehrávanie zvuku I2S	197
• Lekcia 19 – Ovládanie dotykových LED LVGL	209

Vitajte v používateľskej príručke k štartovacej sade All-in-one pre ESP32-P4. Začneme našu cestu do sveta vývojovej dosky ESP32-P4 a jej inteligentných senzorových modulov.

Táto vývojová doska je vybavená 19 kurzami, ktoré sú starostlivo navrhnuté tak, aby boli postupne náročnejšie, zaujímavé a podnecujúce k premýšľaniu. Tieto kurzy vás krok za krokom prevedú základnými pojmami a praktickými cvičeniami. Tu sa zoznámite s rôznymi elektronickými modulmi, posilíte svoje logické myslenie, zdokonalíte svoje kreatívne dizajnérske schopnosti a implementujete funkčnosť týchto modulov prostredníctvom programovania pomocou rámca ESP-IDF.

Proces učenia sa začína nastavením vývojového prostredia ESP-IDF, po ktorom nasleduje úvod do vývojovej dosky ESP32-P4 a jej širokej škály pripojených modulov. Následne sa naučíte, ako programovať každý modul, porozumieť jeho komunikačným protokolom a uplatniť svoje vedomosti v praktických aplikáciách. Každý krok je jasne vysvetlený, čo uľahčuje začiatočníkom rýchle osvojenie si vstavaného vývoja pomocou jazyka C v prostredí IDF.

All-in-one Starter Kit pre ESP32-P4 obsahuje 16 elektronických modulov, z ktorých každý má odlišné vlastnosti a funkcie, čo z neho robí ideálnu voľbu pre začiatočníkov, ktorí chcú preskúmať vývoj hardvéru aj softvéru. Napríklad senzor teploty a vlhkosti vám umožňuje monitorovať údaje o prostredí, zatiaľ čo reléové a motorové moduly vám pomáhajú ovládať reálne zariadenia prostredníctvom kódu.

Zhrnutie: Prácou s touto vývojovou doskou získate solídne znalosti o senzoroch a aktuátoroch, naučíte sa dôležité pojmy, ako sú digitálne a analógové signály, PWM, ADC, DAC a komunikačné rozhrania, ako sú UART, I²C a SPI. Osvojíte si tiež integráciu sieťových funkcií, ako sú Wi-Fi a Bluetooth, a dokonca budete vedieť vo svojich projektoch aplikovať jednoduché funkcie umelej inteligencie. A čo je najdôležitejšie, prostredníctvom programovania ESP-IDF získate hlboké znalosti o vstavaných systémoch a zdokonalíte svoje logické a problémové riešiteľské schopnosti.

Ako programovací softvér budeme využívať vývojový rámec ESP-IDF. ESP-IDF je oficiálna a výkonná open-source platforma spoločnosti Espressif, ktorá vývojárom ponúka plnú kontrolu nad hardvérom a umožňuje profesionálne vstavané programovanie. Je to jeden z najlepších nástrojov na učenie sa vývoja reálnych aplikácií IoT a AI.

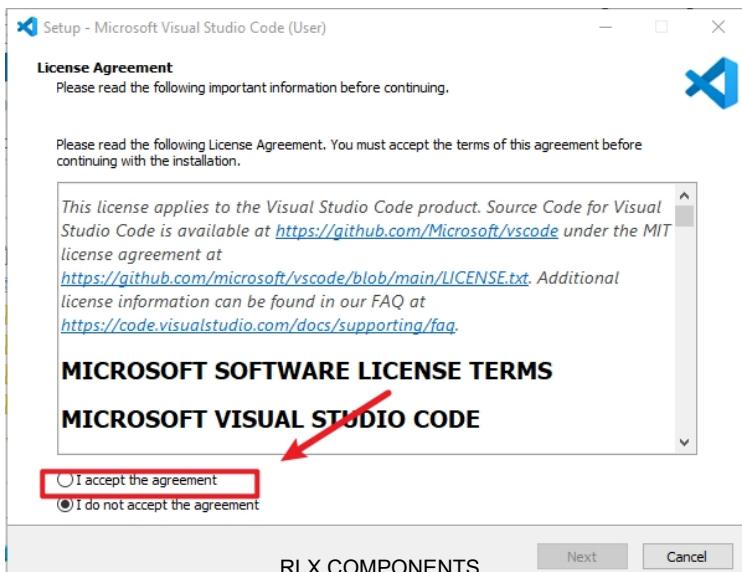
ESP-IDF Príručka na nastavenie prostredia

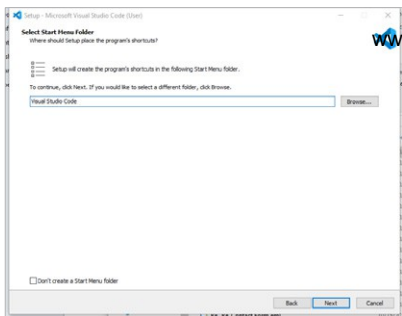
Inštalácia Visual Studio Code

- Najskôr stiahnite Visual Studio Code z <https://code.visualstudio.com/>. Vyberte verziu kompatibilnú s operačným systémom vášho počítača a stiahnite ju.

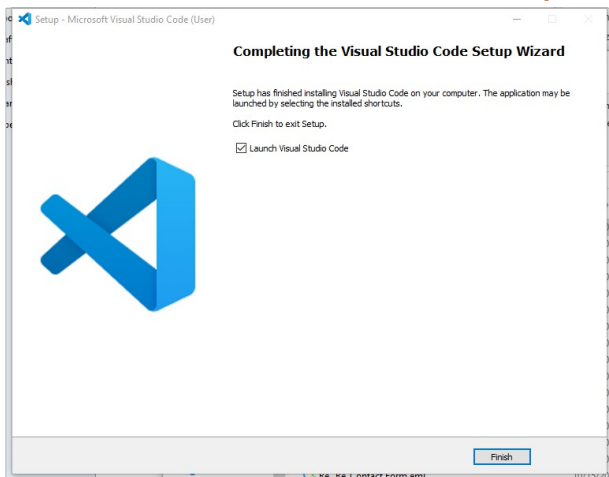
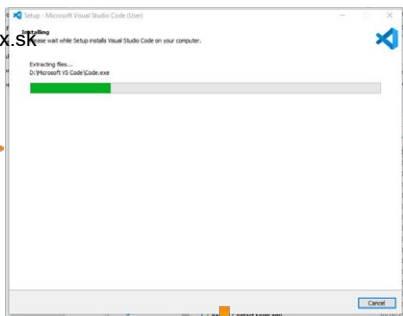


- Dvojitým kliknutím nainštalujte softvér Visual Studio Code a jednoducho pokračujte v predvolenej inštalácii.

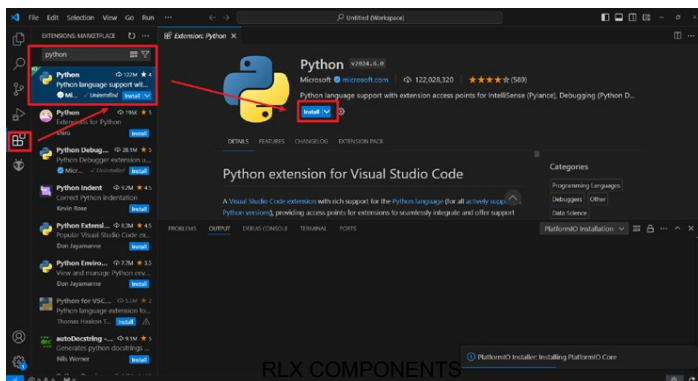




www.rlx.sk

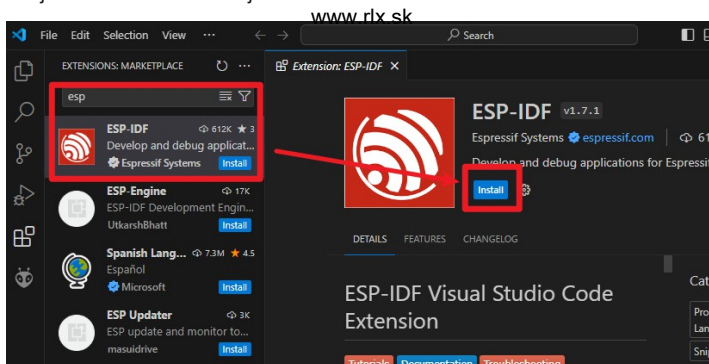


2. Otvorte Visual Studio Code, kliknite na „Rozšírenia“, vyhľadajte Python a nainštalujte ho.

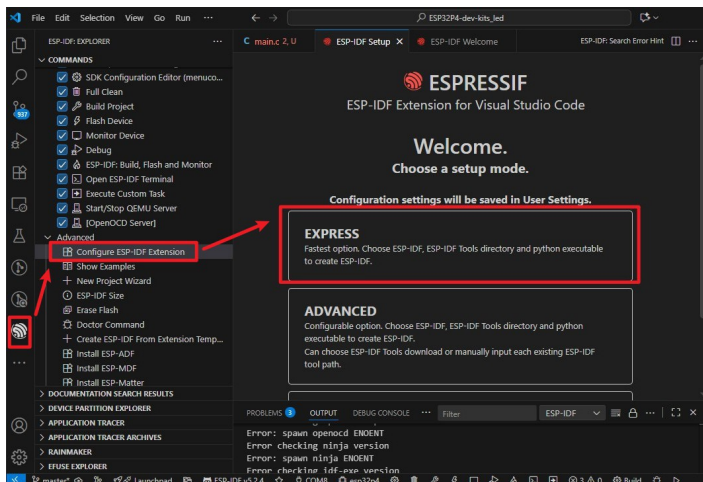


RLX COMPONENTS

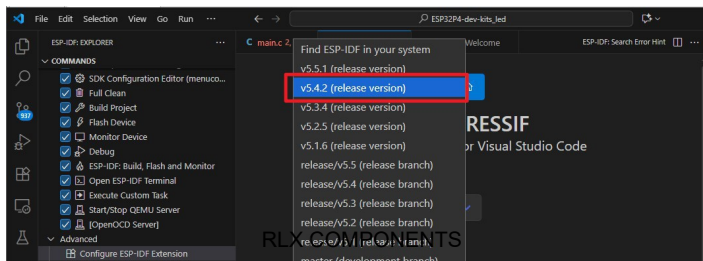
3. Vyhľadajte ESP-IDF a nainštalujte ho.



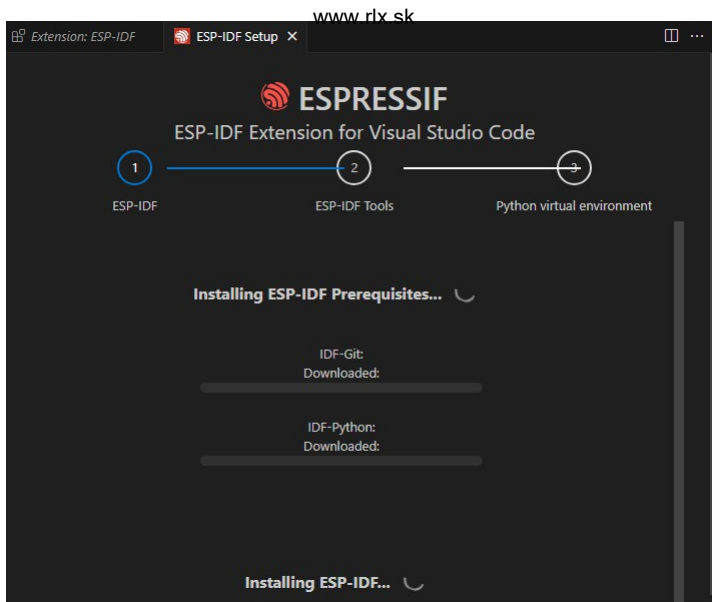
4. Nainštalujte nástroje ESP-IDF.



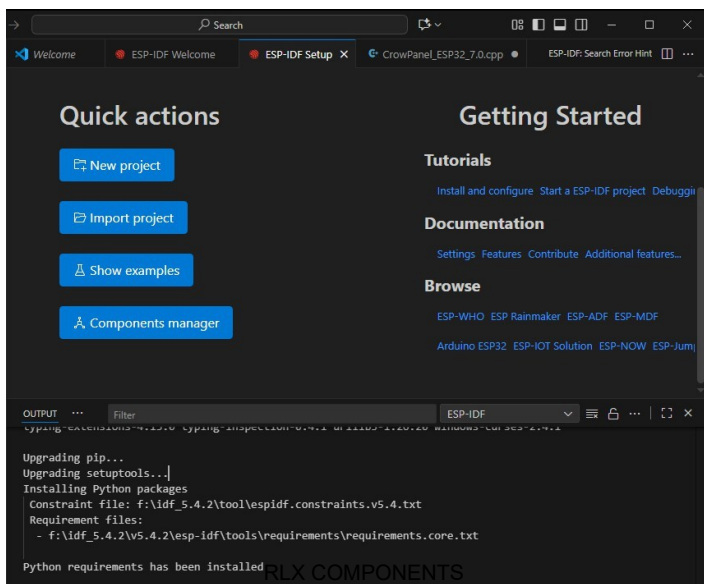
5. Vyberte verziu 5.4.2 a nakonfigurujte umiestnenie úložiska.



6. Čaká sa na inštaláciu.



7. Inštalácia bola úspešná.

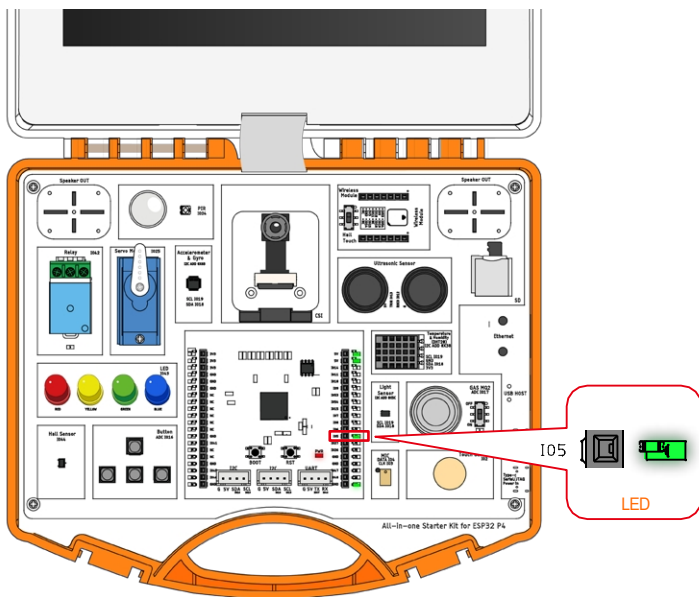


Lekcia 1 – Ovládanie LED GPIO

Úvod

Táto kapitola tutoriálu predstavuje aplikácie výstupu GPIO ESP32-P4 pomocou príkladu rozsvietenia, ktorý pomáha pochopiť jeho základné funkcie. Ako klasický testovací prípad poskytuje demonštrácia rozsvietenia čitateľom jednoduché, ale komplexné pochopenie aplikácií ESP32-P4, čím vytvára základ pre zložitejšie projekty, ktoré budú nasledovať.

Demonštrácia projektu Efekt



Táto kapitola je rozdelená do nasledujúcich podkapitol

- 1.1 Úvod do GPIO a LED
- 1.2 Návrh hardvéru
- 1.3 Návrh programu
- 1.4 Stiahnutie a overenie

1.1.1 Úvod do GPIO

Čip ESP32-P4 poskytuje 55 funkcií všeobecného vstupu/výstupu (GPIO), ktoré ponúkajú flexibilitu a prispôsobivosť v širokej škále aplikácií. Kľúčové vlastnosti týchto GPIO zahŕňajú:

- ① Univerzálnosť: Každý pin GPIO môže fungovať nielen ako vstup alebo výstup, ale môže byť tiež konfigurovaný prostredníctvom IO MUX pre rôzne úlohy (podrobnosti nájdete v kapitole 2), ako napríklad PWM, ADC, I2C, SPI a ďalšie. To umožňuje ESP32-P4 prispôsobiť sa rôznym periférnym pripojeniam.
- ② Vysoký výstupný prúd: Piny GPIO ESP32-P4 podporujú výstupný prúd až 40 mA, čo umožňuje priame riadenie nízkoprikonových záťaží, ako sú LED diódy. Tým sa znižuje zložitosť externých radiacích obvodov.
- ③ Programovateľnosť: Prostredníctvom vývojového rámca ESP-IDF (SDK) môžu používatelia flexibilne konfigurovať režim vstupu/výstupu každého GPIO, parametre pull-up/pull-down a ďalšie nastavenia tak, aby vyhovovali špecifickým požiadavkám aplikácie.
- ④ Podpora prerušenia: Piny GPIO podporujú funkciu prerušenia, ktorá je schopná spúšťať prerušenia pri zmenách signálu. To je vhodné pre aplikácie s reakciou v reálnom čase, ako je detekcia tlačidiel a spúšťanie senzorov.
- ⑤ Indikácia stavu: Piny GPIO môžu fungovať ako LED indikátory, čo umožňuje vizualizáciu stavu prostredníctvom jednoduchého prepínania medzi vysokou a nízkou úrovňou. To uľahčuje používateľom ladenie a monitorovanie prevádzky systému. Funkcie GPIO ESP32-P4 poskytujú vývojárom robustnú hardvérovú podporu. V tejto kapitole sa budeme podrobne zaoberať aplikáciami a konfiguráciou GPIO prostredníctvom príkladu s osvetlením.

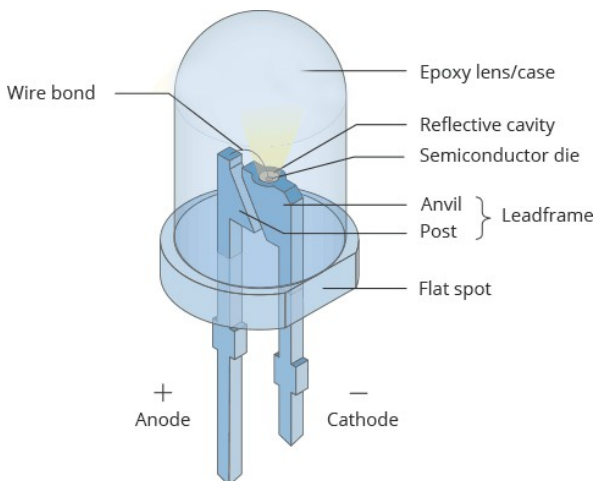
1.1.2 Úvod do LED

LED (svetelné diódy) sú vysoko účinné, dlhotrváce miniatúrne polovodičové zariadenia, ktoré vyžarujú svetlo, keď nimi prechádza elektrický prúd. Ponúkajú výhody, ako je vysoká účinnosť konverzie energie, nízka tvorba tepla a šetrnosť k životnému prostrediu. LED diódy sa bežne používajú v kontrolkách, displejoch a osvetľovacích zariadeniach, poskytujú rýchlu odozvu a širokú škálu farieb, vďaka čomu majú široké uplatnenie v elektronických výrobkoch. V demonštrácii osvetlenia ESP32-P4 zjednodušuje a intuitívne uľahčuje ovládanie GPIO prepínanie LED diód, čo pomáha používateľom pochopiť ich praktické aplikácie.

① Princíp emisii svetla LED

LED zariadenia sú svetlo emitujúce komponenty založené na polovodičovej technológii. Keď sa na polovodičový materiál s PN prechodom pripojí dopredný prúd, rekombinácia nosičov náboja v polovodiči uvoľňuje energiu vo forme fotónov, čím vzniká svetlo. LED diódy sú teda studené svetelné zdroje, ktoré negenerujú teplo ako žiarovky s vláknom, čím eliminujú

problémy, ako je napríklad prehorenie. Nižšie uvedený diagram ilustruje princíp fungovania LED zariadenia.
www.rlx.sk



Princíp fungovania LED zariadení

Na obrázku vyššie polovodičový PN prechod vykazuje charakteristiky priameho vedenia, spätného blokovania a zlyhania. Keď nie je aplikované žiadne vonkajšie predpätie a prechod je v tepelnej rovnováhe, nedochádza k rekombinácii nosičov v rámci PN prechodu, a teda ani k emisii svetla. Keď je však aplikované priame predpätie, proces vyžarovania svetla PN prechodom možno rozdeliť do troch fáz: Po prvé,

nosiče sa vstreknú pod dopredným predpätím;

Po druhé, elektróny a diery sa rekombinujú v oblasti P, čím uvoľňujú energiu;

Nakoniec sa energia uvoľnená počas rekombinácie vyžaruje von vo forme svetla. Jednoducho povedané, keď prúd preteká cez PN prechod, elektróny migrujú smerom k P-oblasť pod vplyvom elektrického poľa. Tam sa rekombinujú s dierami, uvoľňujú prebytočnú energiu a generujú fotóny, čím umožňujú luminiscenčnú funkciu PN prechodu.

Poznámka: Farba svetla vyžarovaného LED diódou je určená šírkou zakázaného pásma použitého polovodičového materiálu. Rôzne materiály produkujú svetlo s rôznymi vlnovými dĺžkami, čo umožňuje rôznorodé farebné výstupy. Tento vysoko účinný luminiscenčný mechanizmus viedol k širokému uplatneniu LED diód v osvetľovacích aj indikačných aplikáciách.

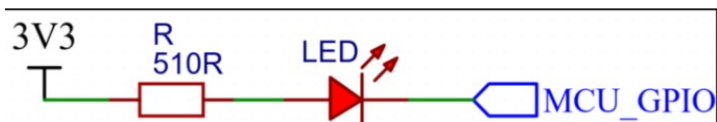
② Princípy ovládačov LED osvetlenia

www.rlx.sk

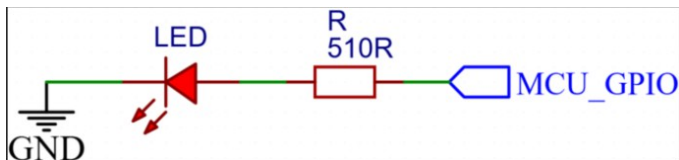
Ovládanie LED diód znamená dodávanie vhodného prúdu a napätia prostredníctvom stabilného zdroja energie, aby sa zabezpečilo správne osvetlenie. Primárnymi metódami ovládania LED diód sú konštantný prúd a konštantné napätie, pričom konštantný prúd je preferovaný pre svoju schopnosť obmedzovať prúd. Keďže LED lampy sú veľmi citlivé na kolísanie prúdu, prekročenie ich menovitého prúdu môže spôsobiť poškodenie. Konštantný prúd preto chráni prevádzku LED diód tým, že udržiava stabilný tok prúdu. Ďalej sa pozrieme na dve metódy ovládania LED diód.

1) Pripojenie s vstrekováním prúdu. Ide o externé napájanie prevádzkového prúdu LED, ktoré vstrekuje prúd do nášho MCU.

Riziko spočíva v tom, že kolísanie externého napájania môže ľahko spôsobiť spálenie pinov MCU.



2) Konfigurácia odberového prúdu. To znamená, že MCU dodáva napätie a prúd a výstupný prúd do LED. Ak sa GPIO MCU používa na priame riadenie LED, jeho riadiaca schopnosť je relatívne slabá a nemusí poskytovať dostatočný prúd na riadenie LED.



Obvod LED na vývojovej doske DNEP32P4 využíva konfiguráciu odberového prúdu. Tento prístup zabraňuje tomu, aby MCU priamo dodávalo napätie a prúd na riadenie LED, čím sa efektívne znižuje zaťaženie MCU. To umožňuje MCU viac sa sústrediť na vykonávanie iných základných úloh, čím sa zvyšuje celkový výkon a stabilita systému.

LED osvetlenie v každodennom živote:



RLX COMPONENTS

③ Pokles napätia LED a hnací prúd

www.rlx.sk

Obvod LED na vývojovej doske P4 využíva obvod uvedený vyššie na napájanie LED lampy. Aký je teda prúd pretekajúci LED v tomto obvode? Predtým, ako sa budeme zaoberať touto otázkou, musíme najprv pochopiť základný pojem: referenčná hodnota poklesu napätia LED. Nižšie sú uvedené referenčné hodnoty poklesu napätia pre LED diód s povrchovou montážou:

1) Červená: Pokles napätia 1,82–1,88 V, prúd 5–8 mA.

2) Zelená: Pokles napätia 1,75–1,82 V, prúd 3–5 mA.

3) Modrá: Pokles napätia 3,1–3,3 V, prúd 8–10 mA.

Pomocou uvedených referenčných hodnôt poklesu napätia SMD LED diód je možné vypočítať prúd LED diódy pomocou Kirchhoffovho zákona o napätí.

Postup výpočtu je nasledovný:

$$(3,3 - 1,8) / 510R = 2,9 \text{ mA}$$

Ak ignorujeme vlastný odpor diódy, prúd pretekajúci cez LED je 2,9 mA. Hoci táto hodnota prúdu presahuje štandardný referenčný rozsah prúdu pre povrchovo montované LED, 2,9 mA je stále dostatočné na rozsvietenie červenej LED.

V mnohých obvodoch sa bez ohľadu na farbu LED namontovanej na doske zvyčajne používajú rezistory s identickými hodnotami obmedzujúce prúd. Táto prax vyplýva predovšetkým z úvah o štandardizácii komponentov a zjednodušení konštrukcie. Použitie jednotných hodnôt rezistorov znižuje zložitosť výroby a údržby a zároveň uľahčuje správu zásob. Okrem toho štandardizované hodnoty rezistorov zefektívňujú proces návrhu obvodov, čo umožňuje konštruktérom pracovať efektívnejšie počas fázy návrhu aj ladenia.

1.2 Návrh hardvéru

1.2.1 Bežná funkčnosť

V rámci 500-milisekundového cyklu sa logický stav LED5 prepne.

1.2.2 Hardvérové zdroje

1) LED



- IO5

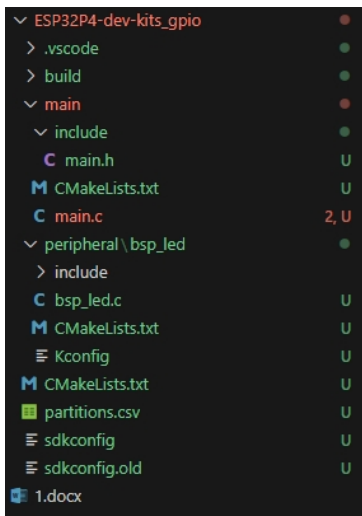
Na vyššie uvedenom schéme je LED5 ovládaná GPIO5 na ESP32-P4, ktorá určuje, či svieti alebo nie. Súčasne indikátor PWR zobrazuje stav napájania a svieti, keď je pripojený napájací zdroj.

1.3 Analýza programu

www.rlx.sk

<https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design>

Otvorte súbor projektu v VS Code podľa predchádzajúcich pokynov.



V rámci príkladu **ESP32P4-dev-kits_gpio** bola vytvorená nová zložka s názvom **bsp_led** v ceste **ESP32P4-dev-kits_gpio\peripheral**. V ceste **bsp_led** bola vytvorená nová zložka **include**, súbor **CMakeLists.txt** a súbor **Kconfig**. Zložka **bsp_led** obsahuje súbor ovládača **bsp_led.c**, zložka **include** obsahuje súbor hlavičky **bsp_led.h** a súbor **CMakeLists.txt** integruje ovládač do systému zostavovania, čím umožňuje projektu využívať funkcie ovládača LED. Súbor **Kconfig** načíta celý ovládač spolu s definíciami pinov GPIO do súboru **sdkconfig** v rámci platformy IDF (konfigurovateľnej prostredníctvom grafického rozhrania).

1.3.1 Kód ovládača LED

Tu sa zameriame výlučne na základný kód; podrobný zdrojový kód nájdete v príslušných zdrojových súboroch pre tento experiment v materiáloch kódu.

Zdrojový kód ovládača LED pozostáva z dvoch súborov: **bsp_led.c** a **bsp_led.h**.

Nižšie najprv analyzujeme program **bsp_led.h**: obsahuje relevantné definície pre piny LED a deklarácie funkcií.

/* Odkazy na súbory hlavičiek*/

```
/*-----Header file declaration-----*/
#include "esp_log.h"//References for LOG Printing Function-related API Functions
#include "esp_err.h"//References for Error Type Function-related API Functions
#include "driver/gpio.h"//References for GPIO Function-related API Functions
/*-----Header file declaration end-----*/
```

/* Definície pinov a deklarácie funkcií */

www.rlx.sk

```
/*-----Variable declaration-----*/
#define LED_TAG "LED"
#define LED_INFO(fmt, ...) ESP_LOGI(LED_TAG, fmt, ##_VA_ARGS_)
#define LED_DEBUG(fmt, ...) ESP_LOGD(LED_TAG, fmt, ##_VA_ARGS_)
#define LED_ERROR(fmt, ...) ESP_LOGE(LED_TAG, fmt, ##_VA_ARGS_)

#ifdef CONFIG_BSP_LED_ENABLED

#define LED_GPIO CONFIG_LED_GPIO //LED GPIO
esp_err_t led_init(); //Initialize the GPIO Pin of the LED
esp_err_t set_led_state(bool state); //Set the GPIO Pin Output Level State of the LED
void led_toggle(); //Toggle the GPIO Pin Output Level State of the LED

#endif
```

Ďalej budeme analyzovať kód v **bsp_led.c**: inicializačnú konfiguráciu a funkčný kód pre piny LED.

/* Inicializačná funkcia led_init */

```
esp_err_t led_init()
{
    esp_err_t err = ESP_OK;
    const gpio_config_t gpio_config = {
        .pin_bit_mask = 1ULL << LED_GPIO, /* GPIO pin: set with bit mask, each bit maps to a GPIO */
        .mode = GPIO_MODE_INPUT_OUTPUT, /* GPIO mode: set input/output mode */
        .pull_up_en = false, /* GPIO pull-up */
        .pull_down_en = false, /* GPIO pull-down */
        .intr_type = GPIO_INTR_DISABLE, /* GPIO interrupt type */
    };
    err = gpio_config(&gpio_config); /*Configure GPIO*/
    if (err != ESP_OK)
        return err;
    return err;
}
```

V rámci funkcie **led_init** sa najskôr parametrizujú rôzne členské premenné štruktúry **gpio_config_t**. Následne sa vyvolá funkcia **gpio_config**, aby sa dokončila inicializácia GPIO pomocou týchto konfiguračných parametrov.

Stojí za zmienku, že konfigurácia pinov ako vstupno-výstupného režimu (`GPIO_MODE_INPUT_OUTPUT`) je primárne kvôli tomu, že ESP-IDF neposkytuje relevantné funkcie invertovania úrovne. Preto, aby sa implementovalo invertovanie úrovne pre LED pin,

je najprv potrebné prečítať aktuálnu úroveň pinu a následne nastaviť jej opačnú úroveň. Tým sa dosiahne požadovaná inverzia. Ak je konfigurovaný ako výstupný režim (`GPIO_MODE_OUTPUT`), funkcia **gpio_get_level** nemôže byť použitá na získanie úrovne pinu, čím sa zabráni funkcii inverzie úrovne.

/* Funkcia prepínania úrovne led_toggle */

```
void led_toggle()
{
    gpio_set_level(LED_GPIO, !(gpio_get_level(LED_GPIO))); /*Set the Corresponding Output Level of GPIO*/
}
```

V rámci funkcie **led_toggle** je pin LED nakonfigurovaný tak, aby vysielal signál s invertovanou úrovňou na základe aktuálnej vstupnej úrovne.

RLX COMPONENTS

Súbor Kconfig

www.rlx.sk

Hlavnou funkciou tohto súboru je pridať požadovanú konfiguráciu do súboru sdkconfig, čím sa umožní modifikácia určitých nastavení parametrov prostredníctvom grafického rozhrania. Číslo 5 sa tu vzťahuje na GPIO_NUM_5, pin pripojený k LED.

```
menu "BSP LED Setup"
    config BSP_LED_ENABLED
        bool "Enable LED"
        default n

    if BSP_LED_ENABLED
        config LED_GPIO
            int "GPIO For LED"
            default 5
    endif
endmenu
```

Súbor CMkaLists.txt

Funkčnosť tejto ukážkovej rutiny závisí predovšetkým od ovládača **bsp_led**. Aby bolo možné úspešne vyvolať obsah priečinka bsp_led v hlavnej funkcii, je potrebné vytvoriť a nakonfigurovať súbor **CMakeLists.txt**, ktorý sa nachádza v priečinku **bsp_led**. Podrobnosti konfigurácie sú nasledovné:

```
FILE(GLOB_RECURSE component_sources "*.c")

idf_component_register(SRCS ${component_sources}
                       INCLUDE_DIRS "include"
                       REQUIRES driver)
```

V tomto súbore **CMakeLists.txt** sú najskôr definované adresáre pre zdrojové súbory a hlavičkové súbory spolu s požadovanými knižnicami ovládačov. Následne sa tieto nastavenia zaregistrujú v systéme zostavovania pomocou príkazu idf_component_register, čím sa projektu umožní využívať funkčnosť ovládača bsp_led.

Poznámka: V nasledujúcich lekciiach nebudeme vytvárať nový súbor **CMakeLists.txt** od začiatku. Namiesto toho vykonáme drobné úpravy tohto existujúceho súboru, aby sme do systému zostavovania začlenili ďalšie ovládače.

1.3.2 main

Hlavný priečinok slúži ako hlavný adresár pre spúšťanie programu a obsahuje hlavnú spustiteľnú funkciu main.c a hlavičkový súbor main.h v priečinku include. Pridajte hlavný priečinok do súboru **CMakeLists.txt** systému zostavenia.

Súbor main.h odkazuje predovšetkým na požadované hlavičkové súbory: funkcie využívajúce ovládač **bsp_led**

vyžadujú zaradenie hlavičkového súboru **bsp_led.h**

RELX COMPONENTS

Nižšie je analýza programu main.c: inicializácia systému a spustenie funkcií špecifických pre LED. www.rlx.sk

```
#ifdef CONFIG_BSP_LED_ENABLED
    err = led_init(); /*LED Driver Initialization*/
    if (err != ESP_OK)
        init_fail("led", err);
#endif
```

Tento kód sa nachádza vo funkcii init, ktorá sa používa na ukladanie inicializačných funkcií vyžadujúcich vyvolanie a na vyhodnotenie výsledku takejto inicializácie. Ak vrátený stav nie je ESP_OK, kód zobrazí chybovú správu a ukončí ďalšie vykonávanie.

```
void app_main(void)
{
    MAIN_INFO("-----Demo version-----");
    MAIN_INFO("-----Start the test-----");
    Init();
#ifdef CONFIG_BSP_LED_ENABLED
    while (1)
    {
        led_toggle();           /*Toggle the GPIO Pin Output Level State of the LED*/
        vTaskDelay(pdMS_TO_TICKS(500)); /*Delay 500ms*/
    }
#endif
}
```

V rámci funkcie **app_main** vytvorte slučku, ktorá opakuje nasledujúci príkaz: každých 500 milisekúnd prepínajte logickú úroveň pinu LED (aby ste dosiahli efekt blikania LED).

1.3.3 Súbor CMkaLists.txt

Aby bolo možné úspešne vyvolať obsah priečinka bsp_led v rámci hlavnej funkcie, je potrebné vytvoriť a nakonfigurovať súbor **CMakeLists.txt** v hlavnom priečinku. Konfigurácia by mala byť nasledovná:

```
FILE(GLOB_RECURSE main ${CMAKE_SOURCE_DIR}/main/*.c)

idf_component_register(SRCS ${main}
                       INCLUDE_DIRS "include"
                       REQUIRES bsp_led)
```

Najskôr sa definujú adresáre pre zdrojové súbory a hlavičkové súbory spolu s požadovanou knižnicou ovládačov – konkrétne knižnicou ovládačov pre prepojenie **bsp_led**. Následne sa tieto nastavenia zaregistrujú v systéme zostavovania pomocou príkazu **idf_component_register**, čím sa hlavnej funkcii umožní využívať funkcie ovládača **bsp_led**.

Poznámka: V nasledujúcich lekciiach nebudeme vytvárať nový súbor **CMakeLists.txt** od začiatku. Namiesto toho vykonáme drobné úpravy tohto existujúceho súboru, aby sme do hlavnej funkcie začlenili ďalšie ovládače.

~~www.rlx.sk~~

Pripojte kábel USB C

The screenshot shows the Visual Studio Code editor with the file `ESP124-dev-kits_led.c` open. The file contains the following code:

```

main()
{
    //header file declaration
    #include "main.h"
    //header file declaration end

    //variable declaration
    #ifdef CONFIG_ESP_LED_ENABLED
    FastLED<led_t> led;
    #endif

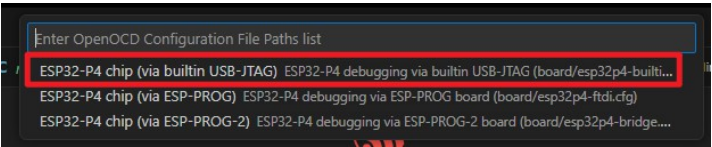
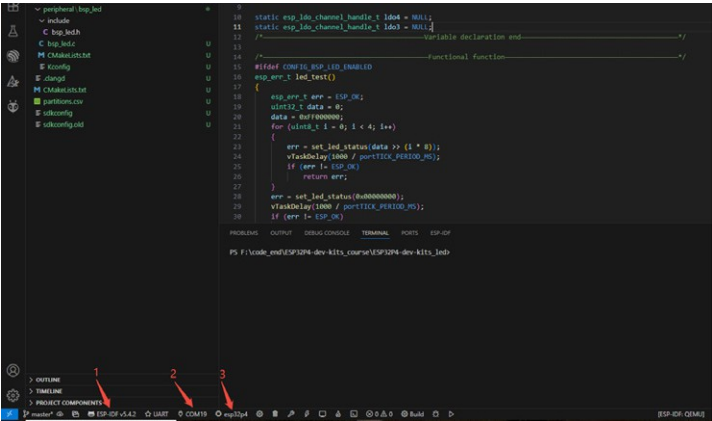
    static esp_led_channel_handle_t ledot = NULL;
    static esp_led_channel_handle_t ledot = NULL;
    //variable declaration end

    //functional function
    #ifdef CONFIG_ESP_LED_ENABLED
    esp_err_t led_test()
    {
        esp_err_t err = ESP_OK;
        uint32_t data = 0;
        data = 0xFFFFFFFF;
        for (uint32_t i = 0; i < 4; i++)
        {
            err = set_led_status(data * (1 << i));
            vTaskDelay(1000 / portTICK_PERIOD_MS);
            if (err != ESP_OK)
                return err;
        }
        err = set_led_status(0xFFFFFFFF);
        vTaskDelay(1000 / portTICK_PERIOD_MS);
        if (err != ESP_OK)
    }
}

```

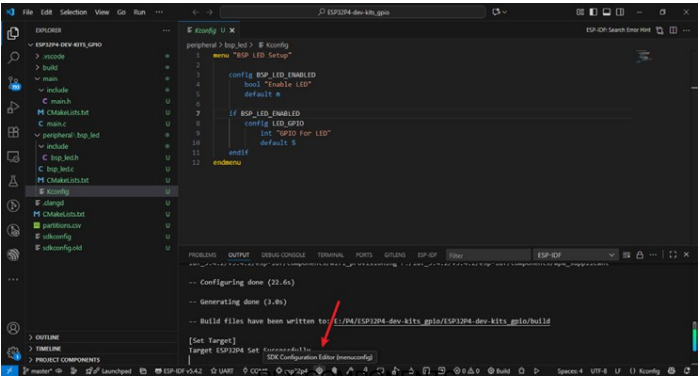
The bottom status bar shows the file path: `PS F:\code_end\ESP124-dev-kits_course\ESP124-dev-kits_led`.

1.4.2 Nakonfigurujte prostredie IDF a model čipu pre kompiláciu a nastavte číslo sériového portu pre programovanie. www.rlx.sk

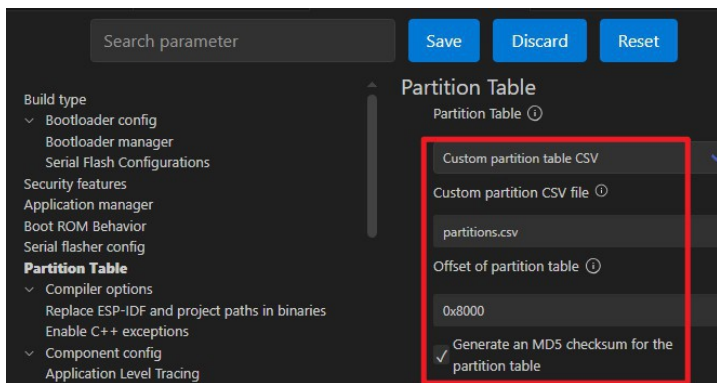


1.4.3 Nastavte požadované nastavenia prostredníctvom SDKConfig (nastavte bootloader na napájovanie súborov CVS a vyberte rýchlosť napájovania, aktivujte PSRAM a vyberte rýchlosť, aktivujte inicializáciu pre komponent BSP_LED špecifický pre kurz; ak nie je k dispozícii, vyhľadajte príslušný obsah priamo v vyhľadávacom poli). Následne vykonajte funkciu: skompilujte, napajte a otvorte monitor.

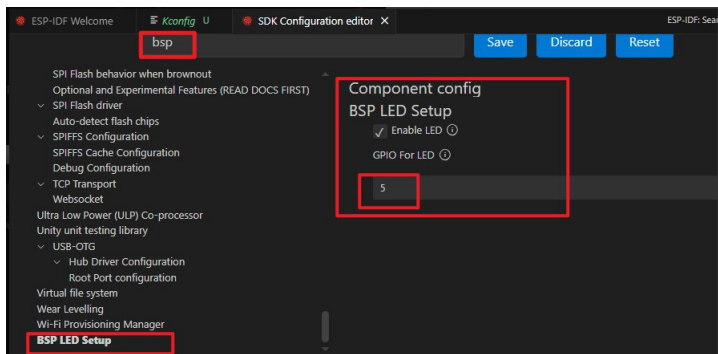
- ① Kliknite na SDK_config nižšie, aby ste získali prístup k nastaveniam.



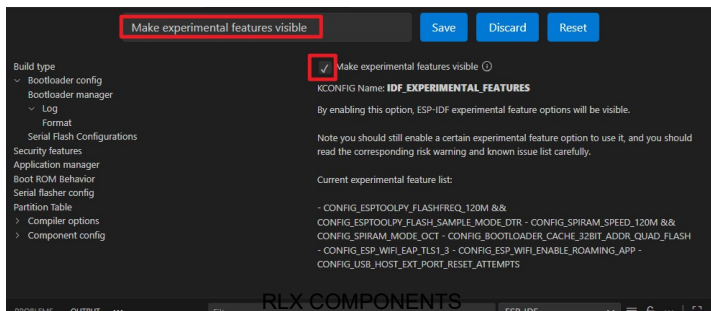
- ② Prejdite nadol do časti Partition Table (Tabuľka oddielov) a nakonfigurujte nastavenia podľa obrázku. www.rlx.sk



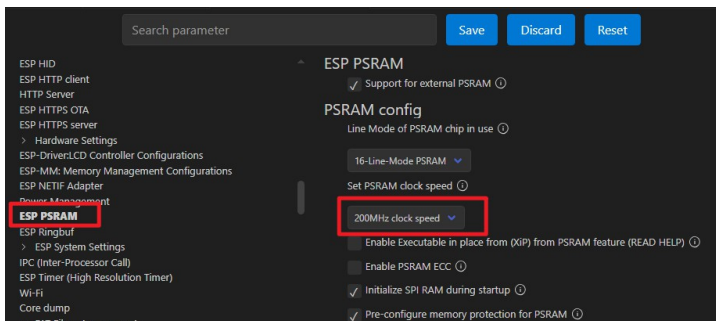
- ③ Môžete vyhľadávať priamo v vyhľadávacom poli pre BSP. Nastavte piny LED svetla.



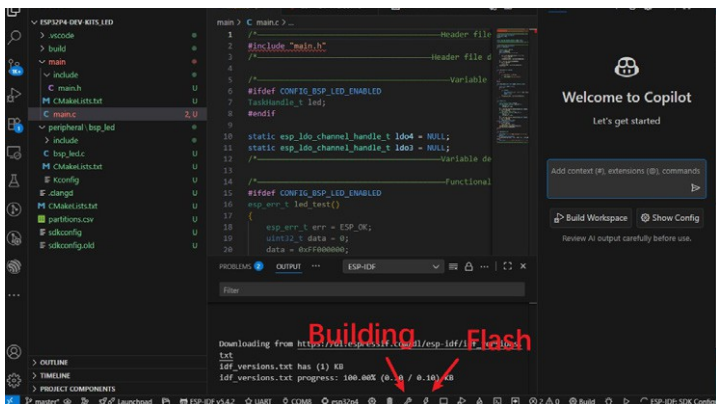
- ④ Môžete vyhľadávať priamo v poli vyhľadávania „Zobraziť experimentálne funkcie“ a zaškrtnúť možnosť zobrazenú na obrázku.



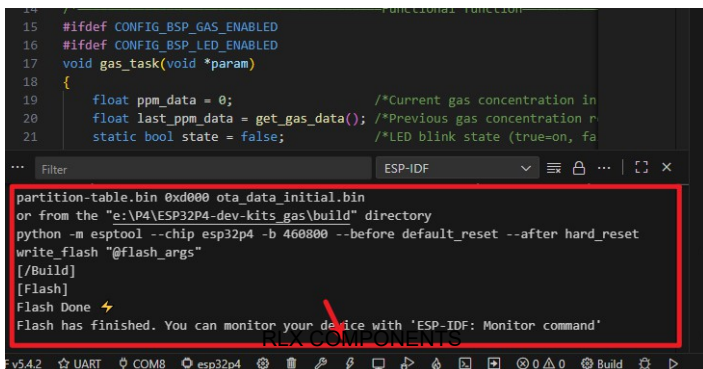
- ⑤ Môžete vyhľadávať priamo v vyhľadávacom poli alebo vyhľadať ESP PSRAM a nastaviť ho na 200 MHz. www.rlx.sk



1.4.4 Kliknite na „Kompilovať“. Po úspešnej kompilácii kliknite na „Stiahnuť“.



Oznámenie o úspešnom stiahnutí :

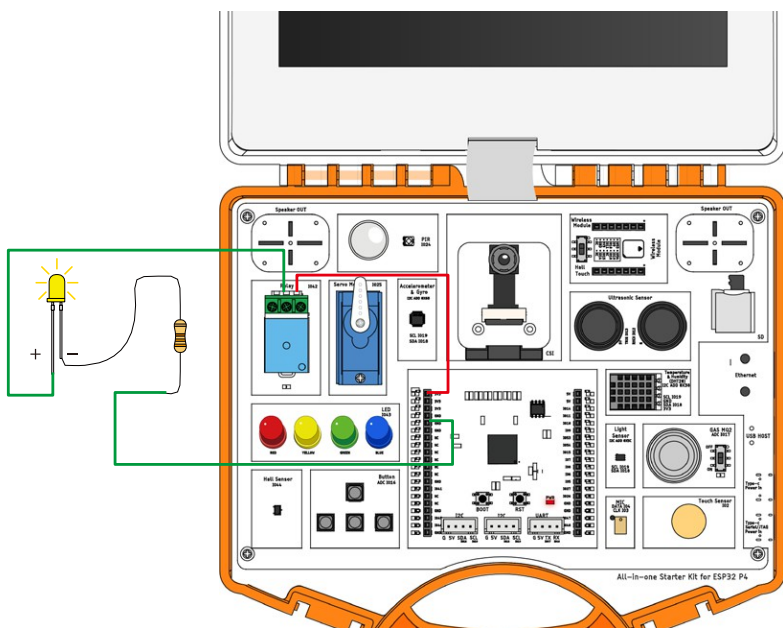


Lekcia 2 – Ovládanie relé GPIO

Úvod

Táto kapitola tutoriálu predstavuje aplikácie výstupu GPIO ESP32-P4. Prostredníctvom príkladu ovládania relé pomáha pochopiť praktické použitie GPIO pri ovládaní zaťaženia. Ako bežné spínacie zariadenie umožňujú relé izoláciu a ovládanie medzi mikrokontrolérmi a vysokonapäťovými zariadeniami s vysokým prúdom, čím tvoria základ pre štúdium projektov inteligentného hardvéru.

Demonštrácia projektu Efekt



Táto kapitola je rozdelená do nasledujúcich podkapitol

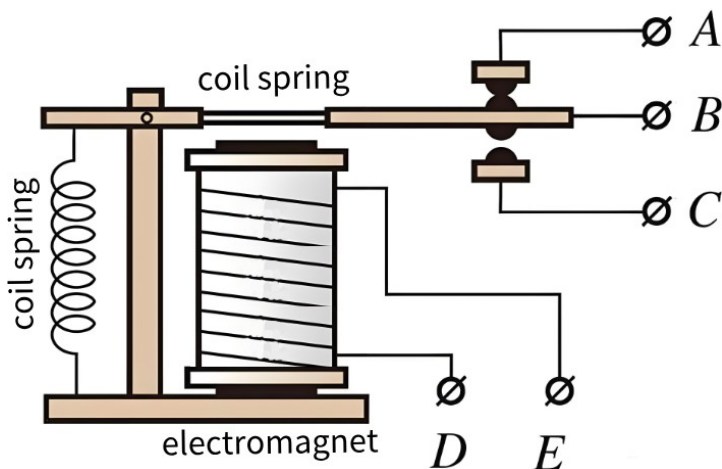
- 1.1 Úvod do relé
- 1.2 Návrh hardvéru
- 1.3 Návrh programu
- 1.4 Stiahnutie a overenie

RLX COMPONENTS

1.1 Úvod do relé

1.1.1 Úvod do relé

Relé je elektromagnetický ovládací spínač, ktorý pomocou malého prúdu ovláda väčší prúd. Široko sa používa na izoláciu obvodov a ovládanie elektrických zariadení. V základnom príklade LED osvetlenia ESP32-P4 sme použili LED diódu; v tejto časti budeme namiesto toho ovládať relé na osvetlenie externých zariadení, ako je malý ventilátor alebo žiarovka.



Popis komponentov v schéme

Cievka elektromagnetu: Vytvára magnetické pole, keď ňou preteká prúd.

Kotva (železná doska pripojená k pružine): Stiahnutá magnetickým poľom, čím sa mení stav kontaktu.

Vratná pružina: Vráti kotvu do pôvodnej polohy, keď je elektromagnet odpojený od napájania.

Kontaktná zostava:

A—B: Normálne uzavretý (NC) kontakt, uzavretý v predvolenom stave.

B—C: Normálne otvorený (NO) kontakt, otvorený v predvolenom stave. problémy, ako napríklad prehorenie. Nižšie uvedený diagram ilustruje princíp fungovania LED zariadenia.

D, E: Dva svorkové terminály elektromagnetu, ktoré slúžia na pripojenie radiaceho prúdu.

① Ako funguje relé

www.rlx.sk

Relé sa zvyčajne skladajú z **cievky**, **kotvy** a **kontaktov**. Keď GPIO vysielá prúd na poháňanie cievky relé, cievka generuje magnetické pole, ktoré priťahuje kotvu, čím mení otvorený alebo uzavretý stav kontaktu.

Princíp činnosti

1. Počiatočný stav (bez prúdu)

Cievky D a E nie sú pod napätím → Elektromagnet nemá žiadnu magnetickú silu.

Pružina ťahá kotvu späť a udržuje ju v hornej polohe. V tomto

momente:

Kontakty B—C sú otvorené (nevedú prúd).

Kontakty A—B sú uzavreté (vedú prúd).

2. Aktívny stav (napájaný)

Keď sa na svorky D a E pripojí radiaci prúd, elektromagnet vytvorí magnetické pole.

Magnetická sila stiahne kotvu nadol. Kotva posunie pohyblivý kontakt B nadol:

B—C sú spojené (uzavreté). A—B

sú oddelené (otvorené).

3. Vrátenie do pôvodného stavu (bez napájania)

Keď je radiaci prúd prerušený, elektromagnet stráca svoju magnetickú silu. Pružina stiahne kotvu späť do pôvodnej polohy.

Kontakty sa vrátia do svojho pôvodného stavu:

B—C je
uzavretý.

A—B je
otvorené.

Táto charakteristika riadenia vysokých prúdov nízkymi prúdmi umožňuje mikrokontrolérom bezpečnú prevádzku zariadení s napätím 220 V striedavého prúdu.

② Spôsob pohonu relé

Keďže cievky relé zvyčajne vyžadujú 5 V/desiatky miliampérov a výstupná kapacita GPIO ESP32-P4 nie je dostatočná na priame riadenie, je potrebný tranzistorový ovládač alebo reléový modul:

GPIO → Tranzistor → Reléová cievka → VCC (5 V)

Súčasne je paralelne pripojená dióda (v reverznom predpätí cez svorky cievky), aby absorbovala reverznú elektromotorickú silu generovanú pri odpojení cievky, čím chráni komponenty.

③ Hlavné parametre relé

www.rlx.sk

Bežné parametre relé zahŕňajú:

Menovité napätie: Zvyčajne 3,3 V, 5 V alebo 12 V (napätie na strane ovládania). (Tu používame 5 V) Prúd cievky: Rozsah od desiatok do stoviek miliampérov.

Menovitá hodnota kontaktu: Určuje regulovateľnú zaťažiteľnosť (napr. 15 A 250 V AC). Normálne otvorené (NO)/normálne uzavreté (NC) kontakty:

NO: V predvolenom stave otvorené, pri napájaní sa uzatvárajú. NC: V predvolenom stave uzavreté, pri napájaní sa otvárajú.

V rámci inteligentných domácich systémov sa relé bežne používajú na ovládanie zariadení, ako sú osvetlenie, klimatizačné jednotky a vodné čerpadlá.



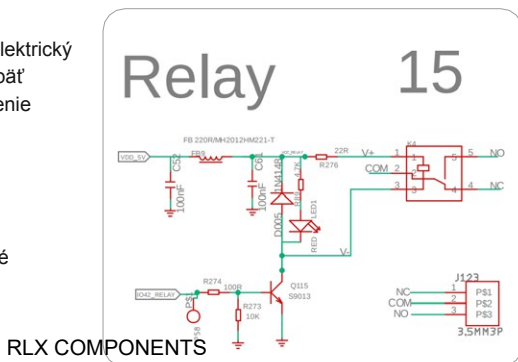
1.2 Hardvérový dizajn

1.2.1 Funkčnosť

V rámci cyklu 5000 milisekúnd sa elektrický stav relé prepne, pričom sa raz za päť sekúnd zapne, aby ovládalo osvetlenie externej LED lampy.

1.2.2 Hardvérové zdroje

Na vyššie uvedenom schéme je relé ovládané GPIO42 na ESP32-P4.



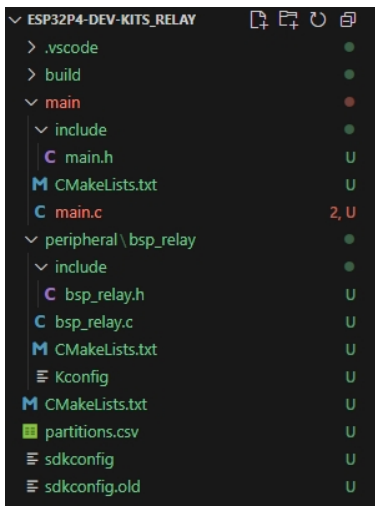
RLX COMPONENTS

1.3 Analýza programu

www.rlx.sk

<https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design>

Otvorte súbor projektu v VS Code podľa predchádzajúcich pokynov.



V rámci príkladu **ESP32P4-dev-kits_relay** bola v adresári **ESP32P4-dev-kits_relay\peripheral**

vytvorená nová zložka s názvom **bsp_relay**.

V adresári **bsp_relay** bol vytvorený nový priečinok include, súbor **CMakeLists.txt** a súbor **Kconfig**. Priečinok

bsp_relay obsahuje súbor ovládača **bsp_relay.c**, priečinok include obsahuje súbor hlavičky **bsp_relay.h** a

súbor **CMakeLists.txt** integruje ovládač do systému zostavovania, čím umožňuje projektantom využívať funkcie ovládača Relay. Súbor **Kconfig** načíta celý ovládač spolu s definíciami pinov GPIO do súboru **sdkconfig** v rámci platformy IDF (konfigurovateľné prostredníctvom grafického rozhrania).

1.3.1 Kód ovládača Relay

Tu sa zameriame výlučne na základný kód; podrobný zdrojový kód nájdete v príslušných zdrojových súboroch pre tento experiment v materiáloch kódu.

Zdrojový kód ovládača relé pozostáva z dvoch súborov: **bsp_relay.c** a **bsp_relay.h**.

Nižšie najprv analyzujeme program **bsp_relay.h**: obsahuje relevantné definície pre piny relé a deklarácie funkcií.

/* Odkazy na hlavičkové súbory */

```
/*-----Header file declaration-----*/
#include "esp_log.h"//References for LOG Printing Function-related API Functions
#include "esp_err.h"//References for Error Type Function-related API Functions
#include "driver/gpio.h"//References for GPIO Function-related API Functions
/*-----Header file declaration end-----*/
```

/* Definície pinov a deklarácie funkcií */

www.rlx.sk

```
#ifndef CONFIG_BSP_RELAY_ENABLED

#define RELAY_GPIO CONFIG_RELAY_GPIO // Relay GPIO
esp_err_t relay_init(); // Initialize the GPIO Pin of the Relay
esp_err_t set_relay_state(bool state); // Set the GPIO Pin Output Level State of the Relay
void relay_toggle(); // Toggle the GPIO Pin Output Level State of the Relay

#endif
```

Ďalej budeme analyzovať kód v **bsp_relay.c**: inicializačnú konfiguráciu a funkčný kód pre pin Relay.

/* Inicializačná funkcia relay_init */

```
esp_err_t relay_init()
{
    esp_err_t err = ESP_OK;
    const gpio_config_t gpio_cfg = {
        .pin_bit_mask = 1ULL << RELAY_GPIO, /* GPIO pin: set with bit mask, each bit maps to a GPIO */
        .mode = GPIO_MODE_INPUT_OUTPUT, /* GPIO mode: set input/output mode */
        .pull_up_en = false, /* GPIO pull-up */
        .pull_down_en = false, /* GPIO pull-down */
        .intr_type = GPIO_INTR_DISABLE, /* GPIO interrupt type */
    };
    err = gpio_config(&gpio_cfg); /*Configure GPIO*/
    if (err != ESP_OK)
        return err;
    return err;
}
```

V rámci funkcie `relay_init` boli najskôr nakonfigurované jednotlivé členské premenné štruktúry `gpio_config_t` s parametrami. Následne bola vyvolaná funkcia `gpio_config`, aby sa dokončila inicializácia GPIO pomocou týchto konfiguračných parametrov.

Stojí za zmienku, že konfigurácia pinu ako režimu vstup-výstup (`GPIO_MODE_INPUT_OUTPUT`) je primárne kvôli tomu, že ESP-IDF neposkytuje špecializovanú funkciu invertovania úrovne. Preto, aby sa dosiahla inverzia úrovne pre pin relé, je potrebné najprv prečítať aktuálnu úroveň pinu a potom nastaviť jeho opačnú úroveň, aby sa dosiahla inverzia. Ak je nakonfigurovaný ako výstupný režim (`GPIO_MODE_OUTPUT`), funkcia **`gpio_get_level`** nemôže byť použitá na získanie úrovne pinu, čím sa zabráni funkcii inverzie úrovne.

/* Funkcia inverzie úrovne relay_toggle */

```
void relay_toggle()
{
    gpio_set_level(RELAY_GPIO, !(gpio_get_level(RELAY_GPIO))); /*Set the Corresponding Output Level of GPIO*/
}
```

V rámci funkcie **`relay_toggle`** je pin relé nakonfigurovaný tak, aby prepínal svoj výstupný stav na základe aktuálnej úrovne vstupu.

1.3.2 Súbor Kconfig

www.rlx.sk

Hlavnou funkciou tohto súboru je pridať požadovanú konfiguráciu do súboru `sdkconfig`, čím sa umožní modifikácia určitých nastavení parametrov prostredníctvom grafického rozhrania. Tu 42 zodpovedá `GPIO_NUM_42`.

```
menu "BSP_RELAY Setup"

    config BSP_RELAY_ENABLED
        bool "Enable RELAY"
        default n

    if BSP_RELAY_ENABLED
        config RELAY_GPIO
            int "GPIO For RELAY"
            default 42
    endif

endmenu
```

1.3.3 Súbor CMakeLists.txt

Funkčnosť tejto ukážkovej rutiny závisí primárne od ovládača **bsp_relay**. Aby bolo možné úspešne vyvolať obsah priečinka **bsp_relay** v rámci hlavnej funkcie, je potrebné nakonfigurovať súbor **CMakeLists.txt**, ktorý sa nachádza v priečinku **bsp_relay**. Podrobnosti konfigurácie sú nasledovné:

```
FILE(GLOB_RECURSE component_sources "*.c")

idf_component_register(SRCS ${component_sources}
                       INCLUDE_DIRS "include"
                       REQUIRES driver)
```

V tomto súbore **CMakeLists.txt** sú najskôr definované adresáre pre zdrojové súbory a hlavičkové súbory spolu s požadovanými knižnicami ovládačov. Následne sa tieto nastavenia zaregistrujú v systéme zostavovania pomocou príkazu `idf_component_register`, čo umožňuje projektu využívať funkčnosť ovládača **bsp_relay**.

1.3.4 Hlavný priečinok

Hlavný priečinok slúži ako hlavný adresár pre spúšťanie programu a obsahuje hlavnú spustiteľnú funkciu **main.c** a hlavičkový súbor **main.h** v priečinku `include`. Pridajte hlavný priečinok do súboru `CMakeLists.txt` systému zostavenia.

Súbor **main.h** odkazuje predovšetkým na požadované hlavičkové súbory: funkcie využívajúce ovládač

bsp_relay vyžadujú zaradenie hlavičkového súboru **bsp_relay.h**.

Nižšie je analýza programu `main.c`: inicializácia systému a vykonávanie špecifické pre funkciu relé.

```
#ifndef CONFIG_BSP_RELAY_ENABLED
    err = relay_init(); /*Relay Driver Initialization*/
    if (err != ESP_OK)
        init_fail("relay", err);
#endif
```

RLX COMPONENTS

Tento kód sa nachádza vo funkcii `init`, ktorá sa používa na ukladanie inicializačných funkcií vyžadujúcich vyvolanie a na vyhodnotenie výsledku tejto inicializácie. Ak vrátený stav nie je **ESP_OK**, kód zobrazí chybovú správu a ukončí ďalšie vykonávanie.

```
void app_main(void)
{
    MAIN_INFO("-----Demo version-----");
    MAIN_INFO("-----Start the test-----");
    Init();
#ifdef CONFIG_BSP_RELAY_ENABLED
    while (1)
    {
        relay_toggle();           /*Toggle the GPIO Pin Output Level State of the Relay*/
        vTaskDelay(pdMS_TO_TICKS(5000)); /*Delay 5000ms*/
    }
#endif
}
```

V rámci funkcie `app_main` vytvorte slučku, ktorá opakuje nasledujúcu činnosť: každých 5000 milisekúnd prepínajte úroveň reléového pinu (aby ste dosiahli funkciu prepínania relé). Toto sa líši od prístupu v predchádzajúcej lekcii k ovládaniu intervalov blikania LED, pretože mechanická konštrukcia reléového spínača má obmedzenú životnosť a nevydrží časté aktivovanie.

1.3.5 Súbor **CMkaLists.txt**

Aby bolo možné úspešne vyvolať obsah priečinka `bsp_relay` v rámci hlavnej funkcie, je potrebné nakonfigurovať súbor **CMakeLists.txt**, ktorý sa nachádza v hlavnom priečinku. Podrobnosti konfigurácie sú nasledovné:

```
FILE(GLOB_RECURSE main ${CMAKE_SOURCE_DIR}/main/*.c)

idf_component_register(SRCS ${main}
                       INCLUDE_DIRS "include"
                       REQUIRES bsp_relay)
```

Najskôr sa definujú adresáre pre zdrojové súbory a hlavičkové súbory spolu s požadovanou knižnicou ovládačov, konkrétne knižnicou ovládačov pre prepojenie **bsp_relay**. Následne sa tieto nastavenia zaregistrujú v systéme zostavovania pomocou príkazu `idf_component_register`, čím sa hlavnej funkcii umožní využívať funkciu ovládača `bsp_relay`.

1.4 Postup programovania

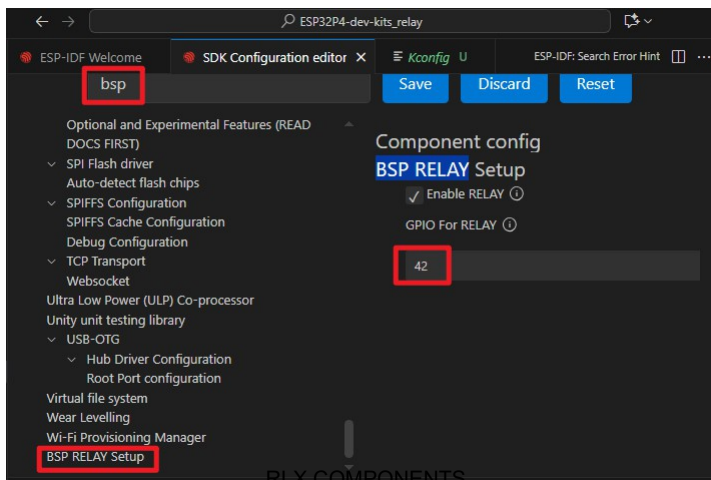
www.rlx.sk

Pripojte zariadenie P4 k počítaču cez USB



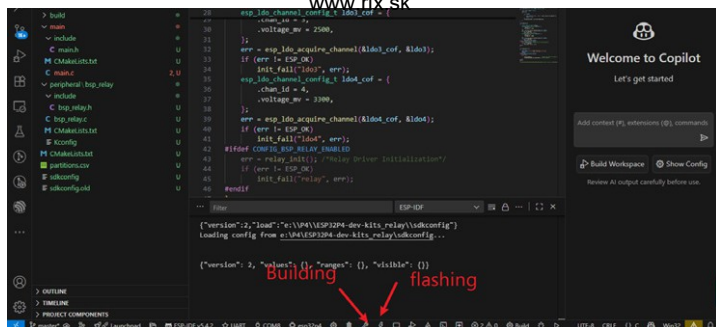
1.4.1 Po klonovaní kódu prostredníctvom Git (odkaz bude potvrdený) vymažte všetky lokálne informácie o kompilácii. Nakonfigurujte prostredie IDF a model čipu pre kompiláciu podľa lekcie 1 a nastavte číslo sériového portu pre programovanie.

1.4.2 Nasledujúca konfigurácia SDKConfig je z veľkej časti identická s lekciou 1, vyžaduje len rekonfiguráciu reléového pinu.



1.4.3 Kliknite na Kompilovať. Po úspešnej kompilácii kliknite na Stiahnuť.

www.rlx.sk

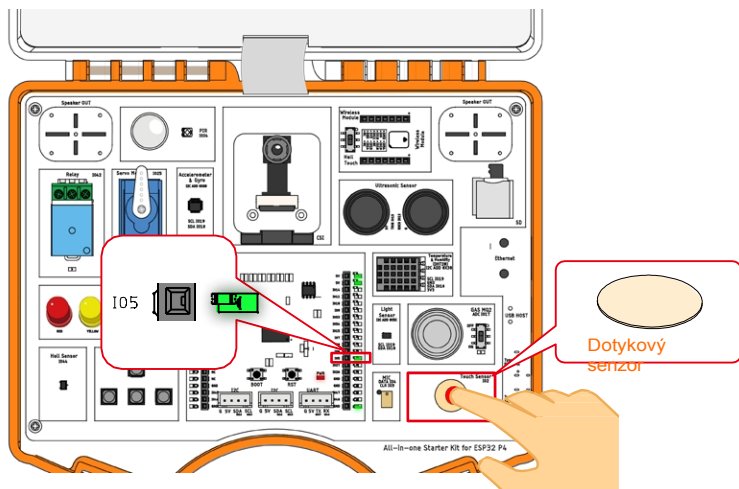


Lekcia 3 – Prepínanie dotykového tlačidla

Úvod

Táto kapitola tutoriálu predstavuje aplikáciu vstupu GPIO pre ESP32-P4. Prostredníctvom príkladu dotykového senzora pomáha pochopiť funkciu detekcie vstupu GPIO. Ako bežné rozhranie človek-stroj umožňujú dotykové senzory detekciu vstupu bez mechanických tlačidiel a slúžia ako dôležitá prípadová štúdia pre štúdiu inteligentných interaktívnych aplikácií.

Demonštrácia projektu Efekt



RLX COMPONENTS

Táto kapitola je rozdelená do nasledujúcich podkapitol

www.rlx.sk

- 1.1 Úvod do dotykových senzorov
- 1.2 Návrh hardvéru
- 1.3 Návrh programu
- 1.4 Stiahnutie a overenie

1.1 Úvod do dotykových senzorov

1.1.1 Úvod do dotykových senzorov

Dotykové senzory sú vstupné zariadenia založené na kapacitnom efekte. Keď sa ľudský prst priblíži alebo dotkne senzora, spôsobí to zmenu kapacity, ktorú detekuje čip. Sú široko používané v inteligentných domácich systémoch, spotrebnej elektronike a aplikáciách interakcie človek-stroj.

① Princíp fungovania dotykových senzorov

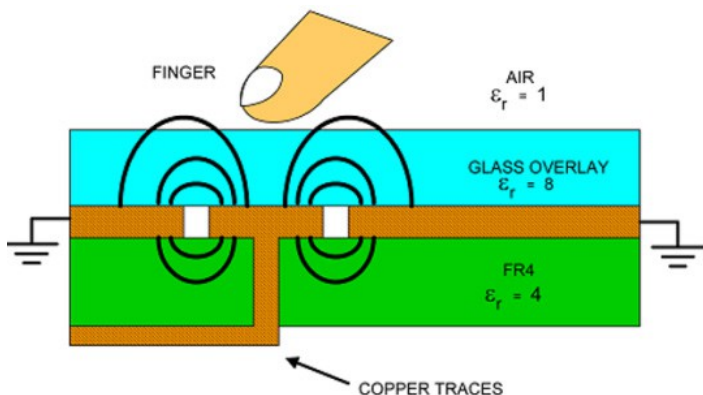
Detekcia dotyku závisí od zmeny kapacity:

Keď nie je prítomný žiadny prst, kapacita medzi elektródou a zemou zostáva stabilná.

Keď sa prst priblíži k elektróde, vodivosť tela zvýši hodnotu kapacity.

Obvod na detekciu dotyku v rámci ESP32-P4 pravidelne meria túto hodnotu kapacity a porovnáva ju s vopred nastavenou prahovou hodnotou.

Akonáhle hodnota kapacity prekročí túto prahovú hodnotu, detekuje sa „dotyková udalosť“.



RLX COMPONENTS

② Spôsob ovládania dotykového senzora

www.rlx.sk

ESP32-P4 obsahuje integrovaný modul dotykového senzora, ktorý umožňuje vývojárom využívať ho bez potreby dodatočných obvodov. Proces detekcie pozostáva z nasledujúcich krokov:

1. Inicializácia GPIO do režimu dotykového vstupu.
2. Prečítanie úrovne napätia tohto GPIO na určenie, či došlo k dotyku. V porovnaní s tradičnými mechanickými tlačidlami ponúkajú dotykové senzory nasledujúce výhody:

- Absencia mechanického opotrebenia
- Rýchla odozva
- Praktický vodotesný dizajn

③ Kľúčové parametre dotykových senzorov

Bežné parametre dotykových senzorov zahŕňajú:

Doba odozvy: desiatky milisekúnd, čo je výrazne rýchlejšie ako v prípade tradičných mechanických tlačidiel.

Citlivosť: Nastaviteľná prostredníctvom softvérových prahov.

Odolnosť: Bez mechanických komponentov, čo zaručuje dlhšiu životnosť.

Prispôsobivosť: Schopnosť detekovať dotyk cez nekovové materiály, ako je sklo a akryl.

Dotykové senzory sa tiež široko používajú v každodennom živote.

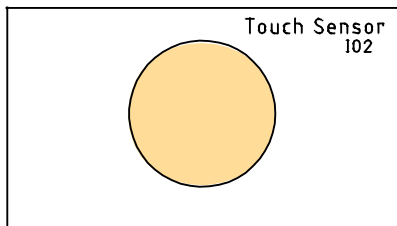


1.2 Hardvérový dizajn

1.2.1 Bežná funkčnosť

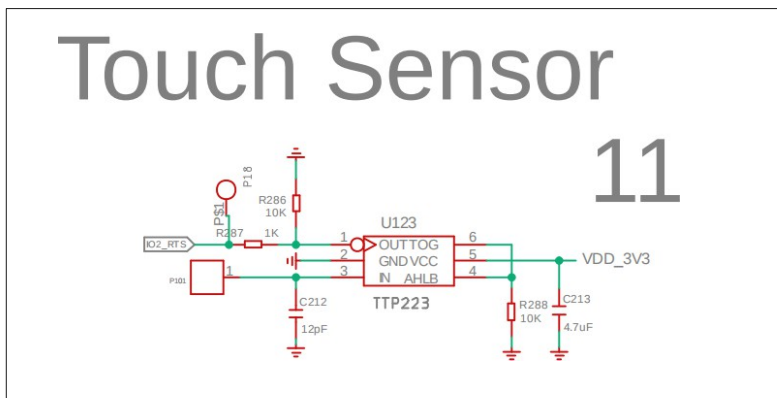
V tomto experimente ESP32-P4 využíva buď svoje integrované, alebo externé dotykové elektródy (kovové dosky/vodivé plochy), ktoré nevyžadujú žiadne ďalšie komponenty. Po detekcii dotyku sa rozsvieti LED dióda ovládaná GPIO5.

1.2.2 Hardvérové zdroje



GPIO 2 je pripojené k kovovému touchpadu (alebo oblasti medenej fólie na doske plošných spojov).

Kapacita sa mení, keď sa k nemu priblíži alebo ho dotkne ľudský prst, a ESP32-P4 detekuje tieto zmeny hodnoty kapacity.

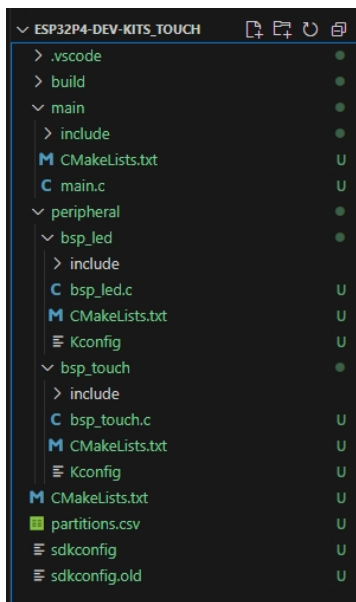


1.3 Analýza programu

www.rlx.sk

<https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design>

Otvorte súbor projektu v VS Code podľa predchádzajúcich pokynov.



V rámci príkladu **ESP32P4-dev-kits_touch** bola v adresári **ESP32P4-dev-kits_touch\peripheral** vytvorená nová zložka s názvom **bsp_touch**.

V rámci cesty **bsp_touch** bola vytvorená nová zložka **include**, súbor **CMakeLists.txt** a súbor **Kconfig**. Zložka **bsp_touch** obsahuje súbor ovládača **bsp_touch.c**, zložka **include** obsahuje súbor hlavičky **bsp_touch.h** a súbor **CMakeLists.txt** integruje ovládač do systému zostavovania, čím umožňuje projektu využívať funkcie ovládača Relay. Súbor **Kconfig** načíta celú konfiguráciu ovládača, vrátane definícií pinov GPIO, do súboru **sdkconfig** v rámci platformy IDF (konfigurovateľného prostredníctvom grafického rozhrania).

Tu sa zameriame výlučne na základný kód; podrobný zdrojový kód nájdete v príslušných zdrojových súboroch pre tento experiment v materiáloch kódu.

Zdrojový kód ovládača dotykového displeja pozostáva z dvoch súborov: **bsp_touch.c** a **bsp_touch.h**.

Nižšie najprv analyzujeme program **bsp_touch.h**: obsahuje relevantné definície pre dotykové piny a deklarácie funkcií.

/* Odkazy na súbory hlavičiek */

```
/*-----Header file declaration-----*/
#include "esp_log.h"//References for LOG Printing Function-related API Functions
#include "esp_err.h"//References for Error Type Function-related API Functions
#include "driver/gpio.h"//References for GPIO Function-related API Functions
/*-----Header file declaration end-----*/
```

/* Definície pinov a deklarácie funkcií */

```

#ifdef CONFIG_BSP_INDEPENDENT_TOUCH_ENABLED

#define INDEPENDENT_TOUCH_GPIO CONFIG_INDEPENDENT_TOUCH_GPIO
esp_err_t independent_touch_init();           // Initialize the GPIO Pin of the Touch
int get_touch_state();                       // Get the current corresponding level of the GPIO for the Touch button

#endif

```

Ďalej budeme analyzovať kód v **bsp_touch.c**: inicializačnú konfiguráciu a funkčný kód pre dotykové piny.

/* Inicializačná funkcia touch_init */

```

esp_err_t independent_touch_init()
{
    esp_err_t err = ESP_OK;
    const gpio_config_t gpio_cfg = {
        .pin_bit_mask = 1ULL << INDEPENDENT_TOUCH_GPIO, /* GPIO pin: set with bit mask, each bit maps to a GPIO */
        .mode = GPIO_MODE_INPUT,                          /* GPIO mode: set input/output mode */
        .pull_up_en = false,                               /* GPIO pull-up */
        .pull_down_en = false,                             /* GPIO pull-down */
        .intr_type = GPIO_INTR_DISABLE,                    /* GPIO interrupt type */
    };
    err = gpio_config(&gpio_cfg); /*Configure GPIO*/
    if (err != ESP_OK)
        return err;
    return err;
}

```

V rámci funkcie **touch_init** sa najskôr nakonfigurujú členské premenné štruktúry **gpio_config_t** pomocou parametrov. Následne sa vyvolá funkcia **gpio_config**, aby sa dokončila inicializácia GPIO pomocou týchto konfiguračných parametrov. Tu je režim GPIO nastavený na režim vstupu, aby sa prečítal stav úrovne portov I/O.

/* Funkcia **get_touch_state** na získanie stavu dotykového tlačidla */

```

int get_touch_state()
{
    return gpio_get_level(INDEPENDENT_TOUCH_GPIO); /*Read the current corresponding level of GPIO*/
}

```

V rámci funkcie `get_touch_state` sa číta aktuálny stav úrovne dotykového tlačidla: vysoká úroveň znamená, že dotykové tlačidlo je stlačené, zatiaľ čo nízka úroveň znamená, že dotykové tlačidlo je uvoľnené.

1.3.2 Súbor Kconfig

Hlavnou funkciou tohto súboru je pridať požadovanú konfiguráciu do súboru `sdkconfig`, čím sa umožní úprava určitých nastavení parametrov prostredníctvom grafického rozhrania. Číslo 2 sa tu vzťahuje na **GPIO_NUM_2**.

```

menu "BSP INDEPENDENT TOUCH Setup"

config BSP_INDEPENDENT_TOUCH_ENABLED
    bool "Enable INDEPENDENT TOUCH"
    default n

if BSP_INDEPENDENT_TOUCH_ENABLED
    config INDEPENDENT_TOUCH_GPIO
        int "GPIO For INDEPENDENT TOUCH"
        default 2
    endif
endif
endmenu

```

RLX COMPONENTS

1.3.3 Súbor CMkaLists.txt

Funkčnosť tejto ukážkovej rutiny závisí predovšetkým od ovládača **bsp_touch**. Aby bolo možné úspešne vyvolať obsah priečinka **bsp_touch** v hlavnej funkcii, je potrebné nakonfigurovať súbor **CMakeLists.txt**, ktorý sa nachádza v priečinku **bsp_touch**. Podrobnosti konfigurácie sú nasledovné:

```
FILE(GLOB_RECURSE component_sources "*.c")

idf_component_register(SRCS ${component_sources}
                       INCLUDE_DIRS "include"
                       REQUIRES driver)
```

V tomto súbore CMakeLists.txt sú najskôr definované adresáre pre zdrojové súbory a hlavičkové súbory spolu s požadovanými knižnicami ovládačov. Následne sa tieto nastavenia zaregistrujú v systéme zostavovania pomocou príkazu `idf_component_register`, čím sa projektu umožní využívať funkčnosť ovládača **bsp_touch**.

1.3.4 Hlavný priečinok

Hlavný priečinok slúži ako hlavný adresár pre spúšťanie programu a obsahuje hlavnú spustiteľnú funkciu **main.c** a hlavičkový súbor **main.h** v priečinku `include`. Pridajte hlavný priečinok do súboru **CMakeLists.txt** systému zostavenia.

Súbor **main.h** odkazuje predovšetkým na požadované hlavičkové súbory: funkcie využívajúce ovládač **bsp_touch** vyžadujú zaradenie hlavičkového súboru **bsp_touch**, zatiaľ čo funkcie využívajúce ovládač **bsp_led** vyžadujú hlavičkový súbor **bsp_led**.

Nižšie je analýza programu **main.c**: inicializácia systému a spustenie funkcie LED a dotykovej funkcie.

```
#ifndef CONFIG_BSP_LED_ENABLED
    err = led_init(); /*LED Driver Initialization*/
    if (err != ESP_OK)
        init_fail("led", err);
#endif
#ifdef CONFIG_BSP_INDEPENDENT_TOUCH_ENABLED
    err = independent_touch_init(); /*Touch Driver Initialization*/
    if (err != ESP_OK)
        init_fail("independent touch", err);
#endif
```

Tento kód sa nachádza vo funkcii `init`, ktorá sa používa na ukladanie inicializačných funkcií vyžadujúcich vyvolanie a na vyhodnotenie výsledku takejto inicializácie. Ak vrátený stav nie je `ESP_OK`, kód zobrazí chybovú správu a ukončí ďalšie vykonávanie.


```

void app_main(void)
{
    MAIN_INFO("-----Demo version-----");
    MAIN_INFO("-----Start the test-----");
    Init();
#ifdef CONFIG_BSP_INDEPENDENT_TOUCH_ENABLED
#ifdef CONFIG_BSP_LED_ENABLED

    bool state = false;    /*Touch button status*/
    bool last_state = false; /*Previous state of the touch button*/
    while (1)
    {
        state = get_touch_state();    /*Get the current touch button status*/
        if ((state) && (state != last_state)) /*If the previous state of the touch button was released, and the current state of the touch button is pressed*/
        {
            set_led_state(true); /*Set the Corresponding Output Level of GPIO*/
        }
        else if (!(state) && (state != last_state)) /*If the previous state of the touch button was pressed, and the current state of the touch button is released*/
        {
            set_led_state(false); /*Set the Corresponding Output Level of GPIO*/
        }
        last_state = state;    /*Save the previous touch button state*/
        vTaskDelay(10 / portTICK_PERIOD_MS); /*Delay 10ms*/
    }

#endif
#endif
}

```

www.rlx.sk

V rámci funkcie **app_main** inicializujete premennú aktuálneho stavu a premennú minulého stavu pre dotykové tlačidlo. Následne vytvorte slučku, ktorá bude opakovať nasledujúcu činnosť: každých 10 milisekúnd vyhodnotiť stav. Funkcia na získanie stavu dotykového tlačidla získa aktuálny stav a porovná ho s predchádzajúcim stavom. Ak sa stav zmenil, vykoná sa funkcia v rámci ovládača **bsp_led**, ktorá nastaví stav LED. Táto funkcia prijíma parameter na nastavenie úrovne LED (nízka úroveň rozsvieti LED, vysoká úroveň ju zhasne). Aktuálny stav dotykového tlačidla sa potom priradí, aby sa zachoval minulý stav. Konkrétna funkcia je nasledovná: stlačením dotykového tlačidla sa LED zhasne, uvoľnením sa LED rozsvieti.

1.3.5 Súbor CMkaLists.txt

Aby bolo možné úspešne vyvolať obsah priečinkov **bsp_led** a **bsp_touch** v rámci hlavnej funkcie, je potrebné nakonfigurovať súbor CMakeLists.txt, ktorý sa nachádza v hlavnom priečinku. Podrobnosti konfigurácie sú nasledovné:

```

FILE(GLOB_RECURSE main ${CMAKE_SOURCE_DIR}/main/*.c)

idf_component_register(SRCS ${main}
                       INCLUDE_DIRS "include"
                       REQUIRES bsp_led bsp_touch)

```

Najskôr sa definujú adresáre pre zdrojové súbory a hlavičkové súbory spolu s požadovanými knižnicami ovládačov – konkrétne knižnice ovládačov potrebné na prepojenie **bsp_led** a **bsp_touch**. Následne sa tieto nastavenia zaregistrujú v systéme zostavovania pomocou príkazu **idf_component_register**, čím sa hlavnej funkcii umožní využívať tieto funkcie ovládačov.

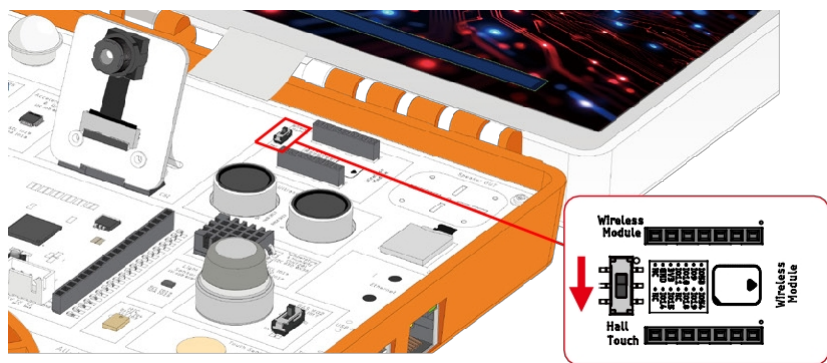
1.4 Postup programovania

www.rlx.sk

Pripojte zariadenie P4 k počítaču prostredníctvom USB



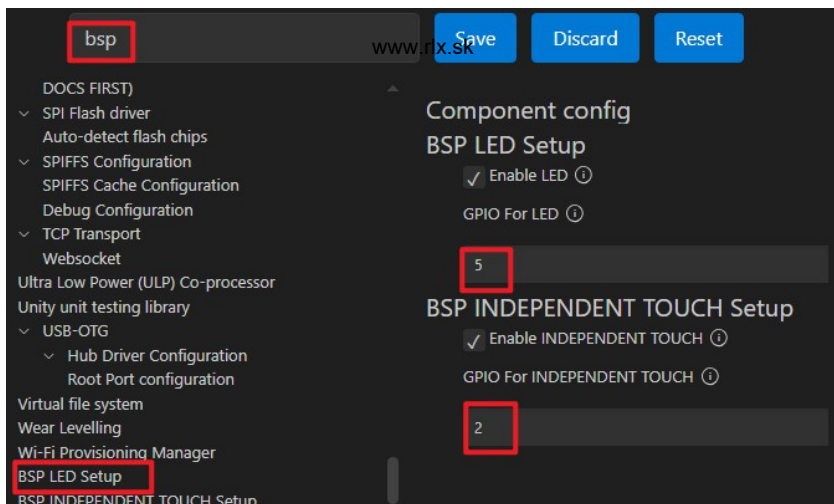
Pri použití dotkových a Hallových senzorov musí byť prepínač v blízkosti bezdrôtového modulu nastavený do polohy Hall a dotyk.



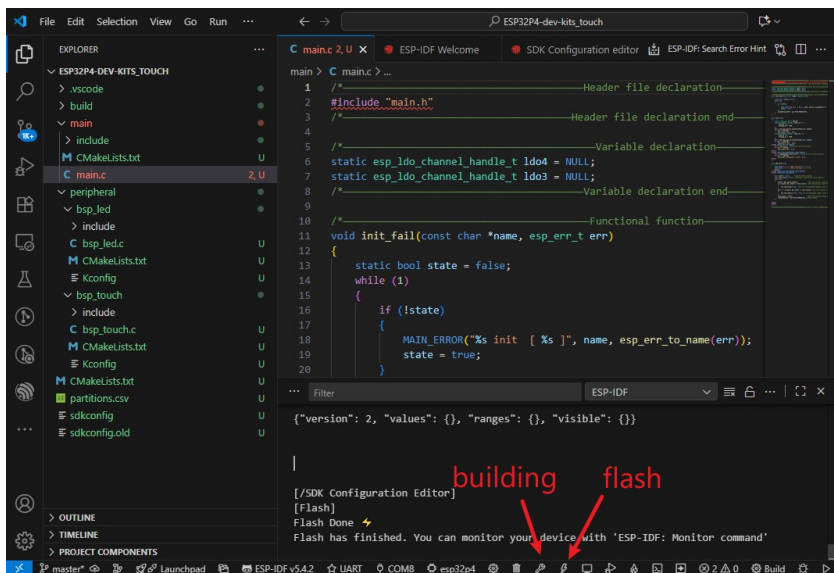
1.4.1 Po klonovaní kódu prostredníctvom Git (odkaz bude potvrdený) vymažte všetky lokálne informácie o kompilácii. Nakonfigurujte prostredie IDF a model čipu pre kompiláciu podľa lekcii 1 a nastavte číslo sériového portu pre programovanie.

1.4.2 Nasledujúca konfigurácia SDKConfig je z veľkej časti identická s lekciami 1, vyžaduje len rekonfiguráciu LED svetla a dotkových pinov.

RLX COMPONENTS



1.4.3 Kliknite na Kompilovať. Po úspešnej kompilácii kliknite na Stiahnuť.



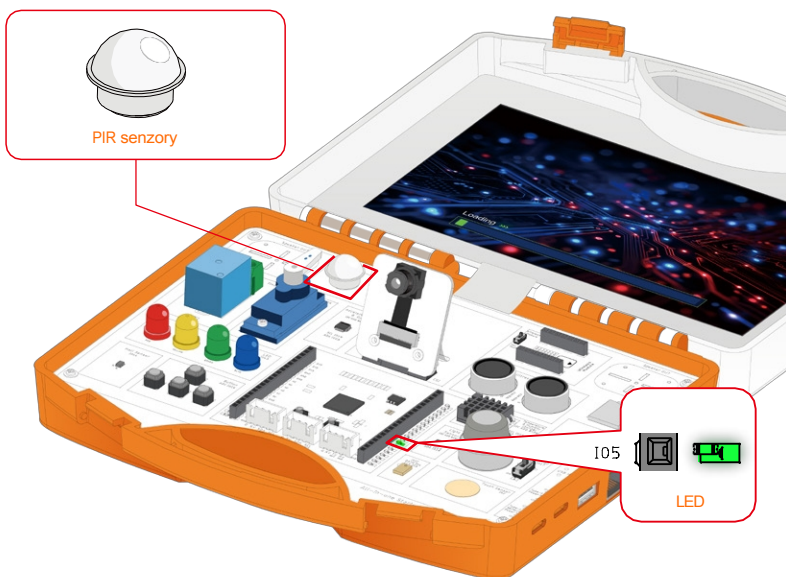
RLX COMPONENTS

Lekcia 4 – Ovládanie pohybu PIR

Úvod

Tento tutoriál demonštruje aplikáciu vstupu GPIO v ESP32-P4. Prostredníctvom príkladu detekcie senzorom PIR (pasívny infračervený senzor) pomáha používateľom pochopiť funkciu detekcie vstupu GPIO. Ako bežné zariadenie na detekciu prítomnosti ľudí, senzor PIR umožňuje automatickú detekciu ľudskej aktivity v prostredí, čo z neho robí dôležitú prípadovú štúdiu pre štúdium inteligentnej bezpečnosti a automatizovaného riadenia.

Demonštrácia projektu Efekt



Táto kapitola je rozdelená do nasledujúcich podkapitol

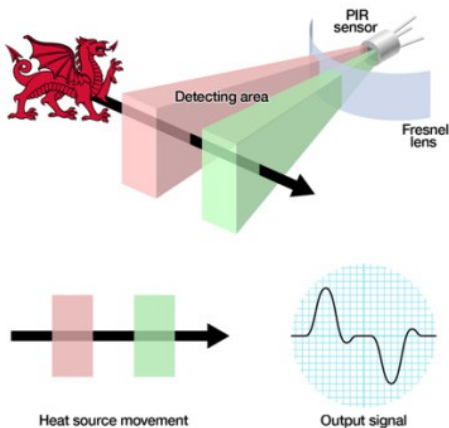
- 1.1 Úvod do senzora PIR
- 1.2 Návrh hardvéru
- 1.3 Návrh programu
- 1.4 Stiahnutie a overenie

1.1 Úvod do senzorov PIR

1.1.1 Úvod do senzorov PIR

PIR (pasívny infračervený senzor) je detekčné zariadenie založené na snímaní infračerveného žiarenia, ktoré sa používa predovšetkým na detekciu pohybu ľudí. Keďže ľudské telo vyžaruje infračervené žiarenie s vlnovou dĺžkou medzi 8 a 14 mikrometrami, senzory PIR detekujú zmeny v tomto žiarení, aby určili, či je prítomná ľudská aktivita.

① Ako fungujú senzory PIR



PIR sa vnútorne skladá z pyroelektrického infračerveného senzora a Fresnelovej šošovky:

Keď je distribúcia infračerveného žiarenia v okolí stabilná, výstup senzora zostáva na nízkej úrovni.

Keď ľudské telo vstúpi do snímacej oblasti, infračervené žiarenie vyžarované telom sa zameria na senzor prostredníctvom Fresnelovej šošovky, čo spôsobí zmenu vstupného náboja.

Táto zmena náboja je spracovaná zosilňovacím a porovnávacím obvodom, čo má za následok výstup impulzu s vysokou úrovňou.

PIR teda priamo nezisťuje teplotu, ale skôr detekuje dynamické zmeny v infračervenom žiarení.

② Princíp fungovania senzorov PIR

Moduly PIR zvyčajne obsahujú obvody na zosilňovanie a porovnávanie signálov, ktoré vysielajú digitálne signály (vysoká/nízka úroveň) na priame pripojenie k GPIO ESP32-P4:

Zistená prítomnosť človeka → Výstupy na vysokej úrovni (GPIO detekuje

1). Nezistená prítomnosť človeka → Výstupy na nízkej úrovni (GPIO detekuje 0).

Toto jednoduché rozhranie eliminuje potrebu používateľov navrhovať zložité analógové obvody.

www.rlx.sk

③ Kľúčové parametre senzorov PIR

Bežné moduly PIR (napríklad HC-SR501) majú nasledujúce kľúčové špecifikácie:

Prevádzkové napätie: 3,3 V až 5 V (priamo kompatibilné s rozhraniami ESP32-P4).

Pohotovostný prúd: približne 50 μ A (nízka spotreba energie).

Detekčný rozsah: typicky 3 až 7 metrov, nastaviteľný.

Uhol snímania: približne 100°–120°.

Výstupný formát: Vysoká/nízka úroveň, kompatibilný s TTL.

PIR senzory sa využívajú v mnohých každodenných aplikáciách, napríklad v inteligentných osvetľovacích systémoch.



1.2 Hardvérový dizajn

V tomto experimente je VCC modulu PIR pripojené k 3,3 V, GND je uzemnené a výstup OUT je pripojený k GPIO24 ESP32-P4.

Keď je detegovaná ľudská aktivita, OUT vysiela vysokú úroveň. LED kontrolovaná GPIO5 sa rozsvieti.

RLX COMPONENTS

1.3.1 Kód ovládača PIR

www.rlx.sk

Tu vysvetlíme len základný kód. Podrobný zdrojový kód nájdete v príslušnom zdrojovom kóde pre tento experiment v materiáloch kódu.

Zdrojový kód ovládača PIR pozostáva z dvoch súborov: bsp_pir.c a bsp_pir.h.

Nižšie najprv analyzujeme program bsp_pir.h: definuje pin PIR a deklaruje funkcie.

/* Odkazy na hlavičkový súbor*/

```
/*-----Header file declaration-----*/
#include "esp_attr.h" //API Function References Related to Memory Configuration
#include "esp_log.h" //References for LOG Printing Function-related API Functions
#include "esp_err.h" //References for Error Type Function-related API Functions
#include "driver/gpio.h" //References for GPIO Function-related API Functions
/*-----Header file declaration end-----*/
```

/* Definície pinov a deklarácie funkcií*/

```
#ifndef CONFIG_BSP_PIR_ENABLED

#define PIR_GPIO CONFIG_PIR_GPIO // PIR GPIO
esp_err_t pir_init(); // Initialize the GPIO Pin of the PIR
esp_err_t get_pir_state(); // Get the status for the PIR

#endif
```

Ďalej budeme analyzovať kód v bsp_pir.c: konfiguráciu inicializácie a kód funkcie pre pin PIR.

/* Inicializačná funkcia pir_init */

```
esp_err_t pir_init()
{
    esp_err_t err = ESP_OK;
    const gpio_config_t gpio_cofig = {
        .pin_bit_mask = 1ULL << PIR_GPIO, /* GPIO pin: set with bit mask, each bit maps to a GPIO */
        .mode = GPIO_MODE_INPUT, /* GPIO mode: set input/output mode */
        .pull_up_en = false, /* GPIO pull-up */
        .pull_down_en = false, /* GPIO pull-down */
        .intr_type = GPIO_INTR_POSEDGE, /* GPIO interrupt type */
    };
    err = gpio_config(&gpio_cofig); /*Configure GPIO*/
    if (err != ESP_OK)
        return err;
    err = gpio_isr_handler_add(PIR_GPIO, PIR_ISR, (void *)PIR_GPIO); /*Add ISR handler for the corresponding GPIO pin*/
    if (err != ESP_OK)
        return err;
    return err;
}
```

V rámci funkcie pir_init sa najskôr nakonfigurujú parametre pre každú členskú premennú štruktúry gpio_config_t. Ďalej sa volá funkcia gpio_config, aby sa dokončila inicializácia GPIO pomocou týchto konfiguračných parametrov. Nakoniec funkcia gpio_isr_handler_add zaregistruje funkciu spätného volania prerušenia a priradí ju k príslušnému GPIO. Tu je režim GPIO nastavený na režim vstupu, aby sa prečítal stav úrovne portu IO, a typ prerušenia je vybraný ako prerušenie stúpajúcej hrany.

/* Funkcia spätného volania prerušenia PIR pin PIR_ISR */


```
static void IRAM_ATTR PIR_ISR(void *arg) /*Rising Edge Interrupt Handler for the PIR Pin*/
{
    if ((arg == (void *)PIR_GPIO) && (!PIR_flag))
        PIR_flag = true;
}
```

www.rlx.sk

V rámci funkcie PIR_ISR sa nastaví vlajka prerušenia kontrolou, či bolo vymazané číslo pinu, ktoré spustilo prerušenie, a stav vlajky prerušenia PIR_flag. (static bool PIR_flag = false; Typ vlajky prerušenia by mal byť definovaný ako globálna premenná.)

/* Funkcia get_pir_state na získanie stavu PIR */

```
esp_err_t get_pir_state()
{
    if (PIR_flag)
    {
        if (gpio_get_level(PIR_GPIO)) /*Read the current corresponding level of GPIO*/
            return ESP_OK;
        else
        {
            PIR_flag = false;
            return ESP_FAIL;
        }
    }
    else
        return ESP_FAIL;
    return ESP_FAIL;
}
```

1.3.2 Súbor Kconfig

Hlavnou funkciou tohto súboru je pridať požadované konfigurácie do súboru sdkconfig, čo umožňuje vykonávať určité úpravy parametrov prostredníctvom grafického rozhrania. Tu sa číslo 24 vzťahuje na GPIO_NUM_24.

```
menu "BSP PIR Setup"
    config BSP_PIR_ENABLED
        bool "Enable PIR config"
        default n
    if BSP_PIR_ENABLED
        config PIR_GPIO
            int "GPIO For PIR"
            default 24
    endif
endmenu
```

1.3.3 Súbor CMkaLists.txt

Funkčnosť tohto príkladu spočíva predovšetkým na ovládači bsp_pir. Aby ste mohli úspešne vyvolať obsah priečinka bsp_pir v hlavnej funkcii, musíte nakonfigurovať súbor CMakeLists.txt, ktorý sa nachádza v priečinku bsp_pir. Podrobnosti konfigurácie sú nasledovné:

```
FILE(GLOB_RECURSE component_sources "*.c")

idf_component_register(SRCS ${component_sources}
                       INCLUDE_DIRS "include"
                       REQUIRES driver)
```

RLX COMPONENTS

V tomto súbore CMakeLists.txt sú najskôr definované adresáre pre zdrojové súbory a hlavičkové súbory spolu s požadovanými knižnicami ovládačov. Potom sa tieto nastavenia zaregistrujú v systéme zostavovania pomocou príkazu `idf_component_register`, čím sa projektu umožní využívať funkčnosť ovládača `bsp_pir`.

1.3.4 hlavný priečinok

Hlavný priečinok slúži ako hlavný adresár pre spúšťanie programu. Obsahuje hlavnú spustiteľnú funkciu `main.c` a hlavičkový súbor `main.h` umiestnený v priečinku `include`. Pridajte hlavný priečinok do súboru `CMakeLists.txt` systému zostavenia.

Súbor `main.h` odkazuje predovšetkým na požadované hlavičkové súbory: funkcie využívajúce ovládač `bsp_pir` vyžadujú hlavičkový súbor `bsp_pir`, zatiaľ čo funkcie využívajúce ovládač `bsp_led` vyžadujú hlavičkový súbor `bsp_led`.

Nižšie je analýza programu `main.c`: inicializácia systému a spustenie funkcie LED a PIR.

```
err = gpio_install_isr_service(0); /*Install the GPIO driver's ETS_GPIO_INTR_SOURCE ISR handler service, which allows per-pin GPIO interrupt handlers*/
if (err != ESP_OK)
    init_fail("gpio isr service", err);
#ifdef CONFIG_BSP_LED_ENABLED
err = led_init(); /*LED Driver Initialization*/
if (err != ESP_OK)
    init_fail("led", err);
#endif
#ifdef CONFIG_BSP_PIR_ENABLED
err = pir_init(); /*Pir Driver Initialization*/
if (err != ESP_OK)
    init_fail("pir", err);
#endif
```

Tento kód sa nachádza vo funkcii `init`, ktorá sa používa na ukladanie inicializačných funkcií, ktoré je potrebné vyvolať, a na vyhodnotenie ich návratových hodnôt. Ak návratový stav nie je `ESP_OK`, kód vytlačí chybovú správu a zastaví ďalšie vykonávanie. Stojí za zmienku, že sme pridali funkciu `gpio_install_isr_service` na registráciu skupiny prerušení pre všetky prerušenia GPIO.

```
void app_main(void)
{
    MAIN_INFO("-----Demo version-----");
    MAIN_INFO("-----Start the test-----");
    init();
#ifdef CONFIG_BSP_PIR_ENABLED
#ifdef CONFIG_BSP_LED_ENABLED
bool state = false; /*PIR status*/
bool last_state = false; /*Previous state of the PIR*/
if (get_pir_state() == ESP_OK) /*First judgment of PIR state*/
    set_led_state(false);
else
    set_led_state(true);
while (1)
{
    if (get_pir_state() == ESP_OK) /*Get the current status feedback from the PIR*/
        state = true;
    else
        state = false;
    if ((state) && (state != last_state)) /*If the previous PIR state was "object stationary" and the current PIR state is "object moving"*/
        set_led_state(false); /*Set the Corresponding Output Level of GPIO*/
    else if ((!state) && (state != last_state)) /*If the previous PIR state was "object moving" and the current PIR state is "object stationary"*/
        set_led_state(true); /*Set the Corresponding Output Level of GPIO*/
    last_state = state; /*Save the previous PIR state*/
    vTaskDelay(10 / portTICK_PERIOD_MS); /*Delay 10ms*/
}
#endif
#endif
```

V funkcii `app_main` inicializujete premennú aktuálneho stavu PIR a premennú minulého stavu. Potom použijete prvý stav PIR na určenie počiatočného stavu LED. Nakoniec vytvorte slučku, ktorá

opakuje nasledujúce: kontroluje stav každých 10 ms. Funkcia na získanie stavu PIR určuje aktuálny stav. Tento stav sa porovnáva s predchádzajúcim stavom. Ak sa stav zmení, vykoná sa funkcia v ovládači bsp_led, ktorá nastaví stav LED. Táto funkcia používa parameter na nastavenie úrovne LED (nízka úroveň zapne LED, vysoká úroveň ju vypne). Aktuálny stav PIR sa potom priradí k predošlej premennej stavu na uchovanie. Konkrétna funkcia je: LED svieti, keď sa zistí pohyb objektu, a zhasne, keď objekt zostane nehybný.

1.3.5 Súbor CMkaLists.txt

Aby ste mohli úspešne vyvolať obsah priečinkov bsp_led a bsp_pir v hlavnej funkcii, musíte nakonfigurovať súbor CMakeLists.txt, ktorý sa nachádza v hlavnom priečinku. Podrobnosti konfigurácie sú nasledovné:

```
FILE(GLOB_RECURSE main ${CMAKE_SOURCE_DIR}/main/*.c)

idf_component_register(SRCS ${main}
                        INCLUDE_DIRS "include"
                        REQUIRES bsp_led bsp_pir)
```

Najskôr sa definujú adresáre pre zdrojové súbory a hlavičkové súbory spolu s požadovanými knižnicami ovládačov – konkrétne knižnice ovládačov potrebné na prepojenie bsp_led a bsp_pir. Potom sa tieto nastavenia zaregistrujú v kompilačnom systéme pomocou príkazu idf_component_register, čím sa hlavnej funkcii umožní využívať tieto funkcie ovládača.

1.4 Postup programovania

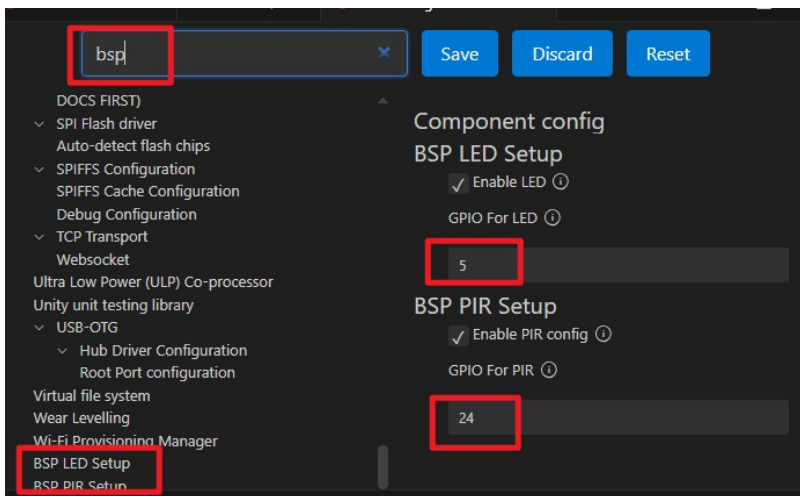
Pripojte zariadenie P4 k počítaču prostredníctvom USB



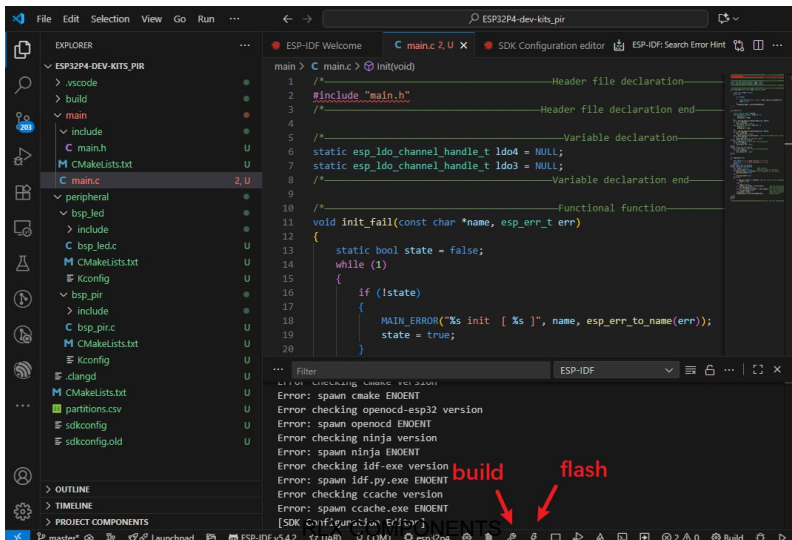
RLX COMPONENTS

1.4.1 Po klonovaní kódu prostredníctvom Git (odkaz bude potvrdený) vymažte všetky lokálne informácie o kompilácii. Nakonfigurujte prostredie IDF a model čipu pre kompiláciu podľa lekcii 1 a nastavte číslo sériového portu pre programovanie.

1.4.2 Nasledujúca konfigurácia SDKConfig je z veľkej časti identická s lekciami 1, vyžaduje len rekonfiguráciu pinov pir a led.



1.4.3 、Kliknite na Compile (Kompilovať). Po úspešnej kompilácii kliknite na Download (Stiahnuť).

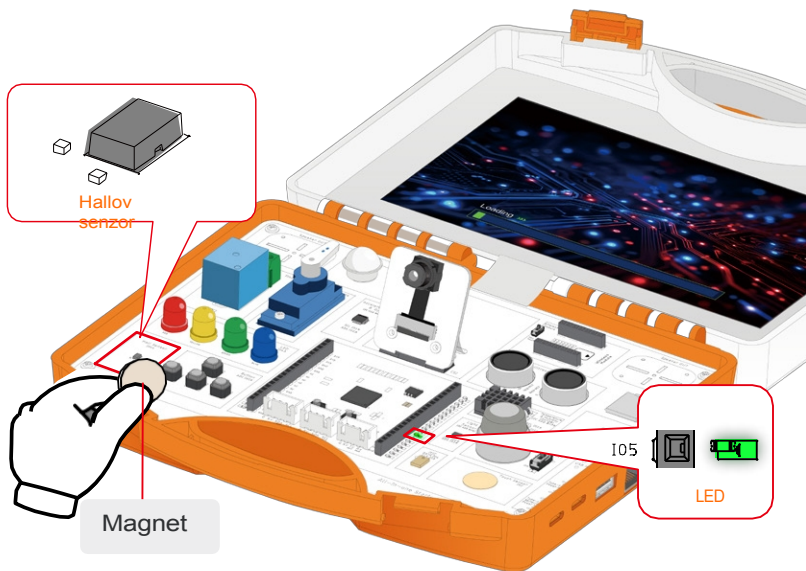


Lekcia 5 – Detekcia Hallovým senzorom

Úvod

Táto kapitola tutoriálu predstavuje aplikácie vstupov GPIO ESP32-P4 na príklade Hallovho senzora, aby vám pomohla pochopiť jeho základné funkcie. Ako bežné zariadenie na detekciu magnetického poľa môže Hallov senzor priamo odrážať zmeny vonkajších magnetických polí, vďaka čomu sa široko používa v scenároch, ako je detekcia polohy, meranie rýchlosti a snímanie prúdu. Učebné príklady v tejto kapitole poskytnú čitateľom jasné pochopenie schopností vstupu GPIO ESP32-P4, čím položia základ pre zložitejšie aplikácie senzorov v nasledujúcich častiach.

Demonštrácia projektu Efekt



Táto kapitola je rozdelená do nasledujúcich podkapitol

- 1.1 Úvod do Hallových senzorov
- 1.2 Návrh hardvéru
- 1.3 Návrh softvéru
- 1.4 Stiahnutie a overenie

RLX COMPONENTS

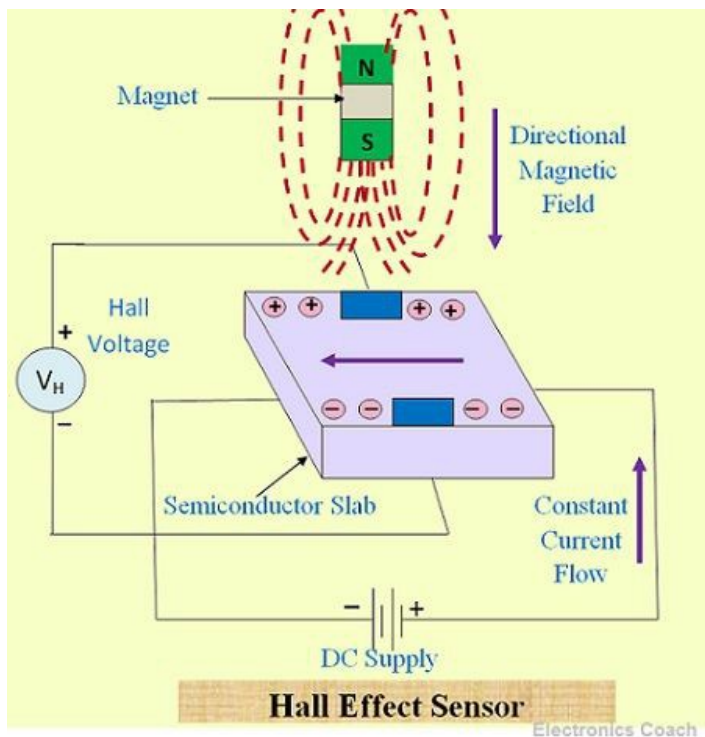
1.1 Úvod do Hallových senzorov

www.rlx.sk

1.1.1 Úvod do Hallových senzorov

Hallovo sensor je zariadenie na detekciu magnetického poľa, ktoré využíva Hallov efekt. Keď magnetické pole prechádza polovodičovým materiálom, generuje napätový signál na svorkách materiálu. ESP32-P4 integruje interný analógový Hall sensor, ktorý umožňuje priamu detekciu zmien intenzity magnetického poľa bez potreby externých komponentov.

1.1.2 Princíp fungovania PIR senzorov



① Princíp činnosti Hallových senzorov

Hallovo efekt označuje jav, pri ktorom pri prechode prúdu vodičom alebo polovodičom magnetické pole kolmé na smer prúdu indukuje potenciálny rozdiel medzi dvoma koncami kolmými na smer prúdu aj magnetického poľa. Tento jav sa nazýva Hallovo napätie.

Zjednodušené povedané:

Žiadne magnetické pole → Výstupné napätie sa blíži k nule.

Magnetické pole → Výstupné napätie sa mení v závislosti od sily magnetického poľa.

Odčítaním tejto hodnoty napätia prostredníctvom ADC alebo špecializovaného rozhrania je možné získať informácie o magnetickom poli.

② Princíp činnosti Hallovho senzora

Vstavaný Hallov senzor ESP32-P4 nevyžaduje žiadny dodatočný hardvér. Hodnoty intenzity magnetického poľa je možné získať prostredníctvom rozhrania SDK `hall_sensor_read()`.

Keď sa magnet priblíži k čipu, hodnota merania sa výrazne zmení.

Keď sa magnet vzdiali alebo nie je prítomný, hodnota sa priblíži k základnej hodnote.

Používatelia môžu nastaviť prahové hodnoty na určenie detekcie magnetického poľa, čím sa aktivujú funkcie ako detekcia polohy a meranie rýchlosti otáčania.

③ Kľúčové parametre Hallových senzorov

Kľúčové špecifikácie Hallových senzorov zahŕňajú:

Citlivosť: Určuje citlivosť senzora na magnetické polia. Prevádzkové napätie:

Vstavaný Hallov senzor ESP32 pracuje priamo pri 3,3 V.

Rýchlosť odozvy: Hallove senzory sú zariadenia s rýchlou odozvou, vhodné na detekciu vysokorýchlostných rotorov.

Teplotná stabilita: Teplotné výkyvy môžu ovplyvniť namerané hodnoty; softvérové filtrovanie alebo kalibračné metódy môžu tieto výkyvy kom

V experimentoch s vývojovou doskou ESP32-P4 môžeme overiť výkon snímania magnetického poľa bez externého Hallovho čipu priamym volaním rozhraní poskytovaných SDK.

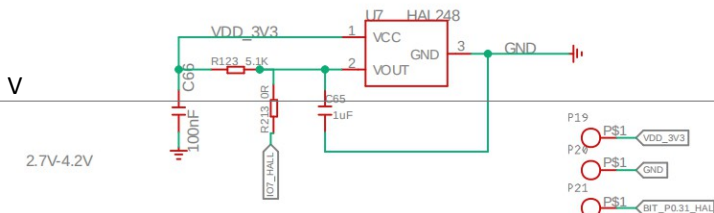
1.2 Hardvérový dizajn

Pri použití vstavaného Hallovho senzora na ESP32-P4 nie je potrebný žiadny externý obvod. Stačí umiestniť malý magnet v blízkosti čipu.

Ak chcete rozšíriť o externý Hallov senzor (napríklad A3144), musíte: Pripojiť VCC k 3,3 V,

pripojiť GND k uzemneniu,

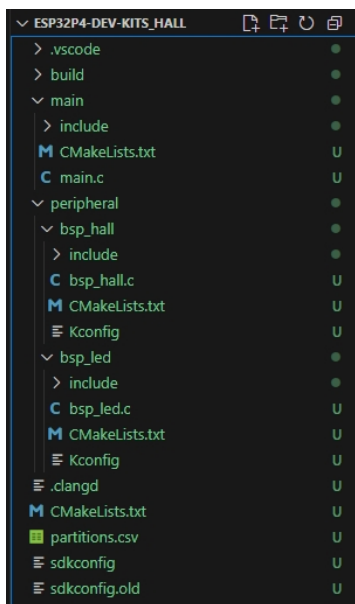
pripojiť výstupný pin k vstupnému pinu GPIO s pull-up rezistorom. Toto nastavenie umožňuje detekciu magnetického poľa.



1.3 Analýza programu

<https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design>

Otvorte súbor projektu v VS Code podľa predchádzajúcich pokynov.



V príklade **ESP32P4-dev-kits_hall** bol vytvorený nový priečinok **bsp_hall** pod adresárom

ESP32P4-dev-kits_hall\peripheral. Do adresára **bsp_hall** bol pridaný nový priečinok **include**, súbor **CMakeLists.txt** a súbor **Kconfig**.

Zložka **bsp_hall** obsahuje súbor ovládača **bsp_hall.c**, zložka **include** obsahuje súbor hlavičky **bsp_hall.h** a súbor **CMakeLists.txt** integruje ovládač do systému zostavovania, čím umožňuje projektu využívať funkcie ovládača HALL. Súbor **Kconfig** načíta celú konfiguráciu ovládača, vrátane definícií pinov GPIO, do súboru **sdkconfig** v rámci platformy IDF (konfigurovateľného prostredníctvom grafického rozhrania).

1.3.1 Kód ovládača HALL

www.rlx.sk

Tu vysvetlíme len základný kód. Podrobný zdrojový kód nájdete v príslušnom zdrojovom kóde pre tento experiment v materiáloch kódu.

Zdrojový kód ovládača HALL pozostáva z dvoch súborov: **bsp_hall.c** a **bsp_hall.h**.

Nižšie najprv analyzujeme program v **bsp_hall.h**: definuje pin HALL a deklaruje funkcie.

/* Odkazy na hlavičkový súbor */

```
/*-----Header file declaration-----*/
#include "esp_attr.h" //API Function References Related to Memory Configuration
#include "esp_log.h" //References for LOG Printing Function-related API Functions
#include "esp_err.h" //References for Error Type Function-related API Functions
#include "driver/gpio.h" //References for GPIO Function-related API Functions
/*-----Header file declaration end-----*/
```

/* Definície pinov a deklarácie funkcií */

```
#ifndef CONFIG_BSP_HALL_ENABLED

#define HALL_GPIO CONFIG_HALL_GPIO // HALL GPIO
esp_err_t hall_init(); // Initialize the GPIO Pin of the HALL
bool get_hall_status(); // Get the status for the HALL

#endif
```

Ďalej budeme analyzovať kód v **bsp_hall.c**: konfiguráciu inicializácie a kód funkcie pre pin HALL.

/* Inicializačná funkcia hall_init */

```
esp_err_t hall_init()
{
    esp_err_t err = ESP_OK;
    const gpio_config_t gpio_cofig = {
        .pin_bit_mask = 1ULL << HALL_GPIO, /* GPIO pin: set with bit mask, each bit maps to a GPIO */
        .mode = GPIO_MODE_INPUT, /* GPIO mode: set input/output mode */
        .pull_up_en = false, /* GPIO pull-up */
        .pull_down_en = false, /* GPIO pull-down */
        .intr_type = GPIO_INTR_ANYEDGE, /* GPIO interrupt type */
    };
    err = gpio_config(&gpio_cofig); /*Configure GPIO*/
    if (err != ESP_OK)
    {
        return err;
    }
    err = gpio_isr_handler_add(HALL_GPIO, HALL_ISR, (void *)HALL_GPIO); /*Add ISR handler for the corresponding GPIO pin*/
    if (err != ESP_OK)
    {
        return err;
    }
    return err;
}
```

V funkcii hall_init sú najskôr konfigurované členské premenné štruktúry **gpio_config_t** pomocou parametrov. Následne je volaná funkcia **gpio_config** na inicializáciu GPIO pomocou týchto konfiguračných parametrov. Nakoniec sa funkcia **gpio_isr_handler_add** použije na registráciu funkcie spätného volania prerušenia a jej priradenie k príslušnému GPIO. Tu je režim GPIO nastavený na režim vstupu, aby sa prečítal stav úrovne portu IO. Typ prerušenia je konfigurovaný ako plný okraj (spúšťa funkciu spätného volania prerušenia buď na stúpajúcom, alebo klesajúcom okraji).

/* Funkcia spätného volania prerušenia pinov HALL HALL_ISR */

```
static void IRAM_ATTR HALL_ISR(void *arg) /*Hall Pin's Full-Edge Interrupt Handler*/
{
    if ((arg == (void *)HALL_GPIO) && (!gpio_get_level(HALL_GPIO)))
        hall_state = true;
    else if ((arg == (void *)HALL_GPIO) && (gpio_get_level(HALL_GPIO)))
        hall_state = false;
}
```

Vo funkcii **HALL_ISR** sa nastaví vlajka prerušenia kontrolou čísla pinu, ktorý spustil prerušenie, a aktuálneho stavu úrovne pinu. (true označuje klesajúcu hranu, čo znamená, že sa magnet približuje; false označuje stúpajúcu hranu, čo znamená, že sa magnet vzdaluje.) (static bool hall_state = false; Definícia typu vlajky prerušenia by mala byť globálnou premennou.)

/* Funkcia get_hall_status na získanie stavu HALL */

```
bool get_hall_status() /*Get the state of the Hall sensor*/
{
    return hall_state;
}
```

Vo funkcii **get_hall_status** vráťte vlajku prerušenia.

1.3.2 Súbor Kconfig

Hlavnou funkciou tohto súboru je pridať požadovanú konfiguráciu do súboru **sdkconfig**, čím sa umožní úprava určitých nastavení parametrov prostredníctvom grafického rozhrania. Číslo 7 tu predstavuje GPIO_NUM_7.

```
menu "BSP HALL Setup"
    config BSP_HALL_ENABLED
        bool "Enable HALL"
        default n
    if BSP_HALL_ENABLED
        config HALL_GPIO
            int "GPIO For HALL"
            default 7
    endif
endmenu
```

1.3.3 Súbor CMkaLists.txt

Funkčnosť tohto príkladu spočíva predovšetkým na ovládači **bsp_hall**. Aby ste mohli úspešne vyvolať obsah priečinka **bsp_hall** v hlavnej funkcii, musíte nakonfigurovať súbor **CMakeLists.txt**, ktorý sa nachádza v priečinku **bsp_hall**. Podrobnosti konfigurácie sú nasledovné:

```
FILE(GLOB_RECURSE component_sources "*.c")

idf_component_register(SRCS ${component_sources}
    INCLUDE_DIRS "include"
    REQUIRES driver)
```

V tomto súbore **CMakeLists.txt** sú najskôr definované adresáre pre zdrojové súbory a hlavičkové súbory spolu s požadovanými knižnicami ovládačov. Potom sa tieto nastavenia zaregistrujú v systéme zostavovania pomocou príkazu **idf_component_register**, čím sa projektu umožní využívať funkčnosť ovládača **bsp_hall**.

RLX COMPONENTS

1.3.4 hlavný priečinok

www.rlx.sk

Hlavný priečinok slúži ako hlavný adresár pre spúšťanie programu. Obsahuje hlavnú spustiteľnú funkciu `main.c` a hlavičkový súbor `main.h`, ktorý sa nachádza v priečinku `include`. Pridajte hlavný priečinok do súboru **CMakeLists.txt** systému zostavenia.

Súbor `main.h` odkazuje predovšetkým na požadované hlavičkové súbory: funkcie využívajúce ovládač **bsp_hall** vyžadujú hlavičkový súbor **bsp_hall**, zatiaľ čo funkcie využívajúce ovládač **bsp_led** vyžadujú hlavičkový súbor **bsp_led**.

Nižšie je analýza programu `main.c`: inicializácia systému a spustenie funkcií LED a Hallovho senzora.

```
err = gpio_install_isr_service(0); /*Install the GPIO driver's ETS_GPIO_INTR_SOURCE ISR handler service, which allows per-pin GPIO interrupt handlers*/
if (err != ESP_OK)
    init_fail("gpio isr service", err);
#ifdef CONFIG_BSP_LED_ENABLED
err = led_init(); /*LED Driver Initialization*/
if (err != ESP_OK)
    init_fail("led", err);
#endif
#ifdef CONFIG_BSP_HALL_ENABLED
err = hall_init(); /*HALL Driver Initialization*/
if (err != ESP_OK)
    init_fail("hall", err);
#endif
```

Tento kód sa nachádza vo funkcii `init`, ktorá sa používa na ukladanie inicializačných funkcií, ktoré je potrebné vyvolať, a na vyhodnotenie ich návratových hodnôt. Ak návratový stav nie je `ESP_OK`, kód vytlačí chybovú správu a zastaví ďalšie vykonávanie. Stojí za zmienku, že sme pridali funkciu **gpio_install_isr_service** na registráciu skupiny prerušení pre všetky prerušenia GPIO.

```
void app_main(void)
{
    MAIN_INFO("-----Demo version-----");
    MAIN_INFO("-----Start the test-----");
    Init();
#ifdef CONFIG_BSP_HALL_ENABLED
#ifdef CONFIG_BSP_LED_ENABLED
bool state = false; /*HALL status*/
bool last_state = false; /*Previous state of the HALL*/
if (get_hall_status()) /*First judgment of HALL state*/
    set_led_state(false);
else
    set_led_state(true);
while (1)
{
    state = get_hall_status(); /*Get the current status feedback from the HALL*/
    if ((state && (state != last_state)) /*If the previous Hall sensor state was "magnet moving away" and the current Hall sensor state is "magnet approaching"*/
        set_led_state(false); /*Set the Corresponding Output Level of GPIO*/
    else if ((!state && (state != last_state)) /*If the previous Hall sensor state was "magnet approaching" and the current Hall sensor state is "magnet moving away"*/
        set_led_state(true); /*Set the Corresponding Output Level of GPIO*/
    last_state = state; /*Save the previous HALL state*/
    vTaskDelay(10 / portTICK_PERIOD_MS); /*Delay time*/
}
#endif
#endif
}
```

V funkcii **app_main** inicializujete premennú aktuálneho stavu HALL a premennú minulého stavu. Potom použijete prvý stav HALL na určenie počiatočného stavu LED. Nakoniec vytvorte slučku, ktorá opakuje nasledujúce: kontrolujte raz za 10 ms oneskorenia. Funkcia na získanie stavu HALL načítá aktuálny stav a porovná ho s predchádzajúcim stavom. Ak sa stav zmení, vykoná funkciu v ovládači **bsp_led**, ktorá nastaví stav LED. Táto funkcia prijíma parameter na nastavenie úrovně LED (nízka úroveň zapne LED, vysoká úroveň ju vypne). Aktuálny stav HALL sa potom priradí k premennej minulého stavu na uchovanie. Konkrétna funkcia je: LED svieti, keď sa magnet priblíži k Hallovmu senzoru. LED zhasne, keď sa magnet vzdiali od Hallovho senzora.

1.3.5 Súbor CMkaLists.txt

Aby ste mohli úspešne vyvolať obsah priečinkov **bsp_led** a **bsp_hall** v hlavnej funkcii, musíte nakonfigurovať súbor **CMakeLists.txt**, ktorý sa nachádza v hlavnom priečinku. Podrobnosti konfigurácie sú nasledovné:

```
FILE(GLOB_RECURSE main ${CMAKE_SOURCE_DIR}/main/*.c)

idf_component_register(SRCS ${main}
                        INCLUDE_DIRS "include"
                        REQUIRES bsp_led bsp_hall)
```

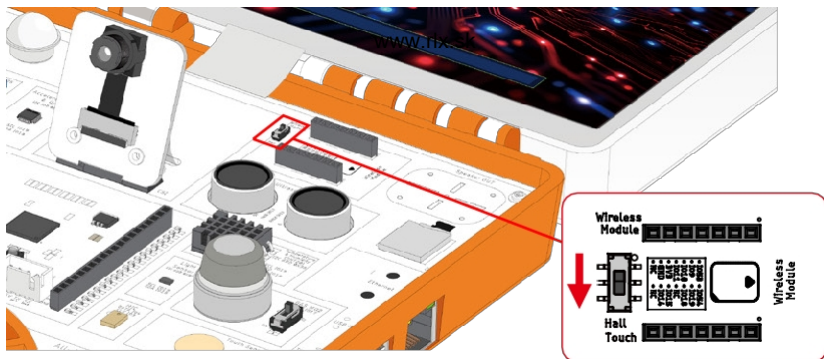
Najskôr sa definujú adresáre pre zdrojové súbory a hlavičkové súbory spolu s požadovanými knižnicami ovládačov – konkrétne knižnice ovládačov potrebné na prepojenie **bsp_led** a **bsp_hall**. Potom sa tieto nastavenia zaregistrujú v systéme zostavovania pomocou príkazu **idf_component_register**, čím sa hlavnej funkcii umožní využívať tieto funkcie ovládača.

1.4 Postup programovania

Pripojte zariadenie P4 k počítaču prostredníctvom USB

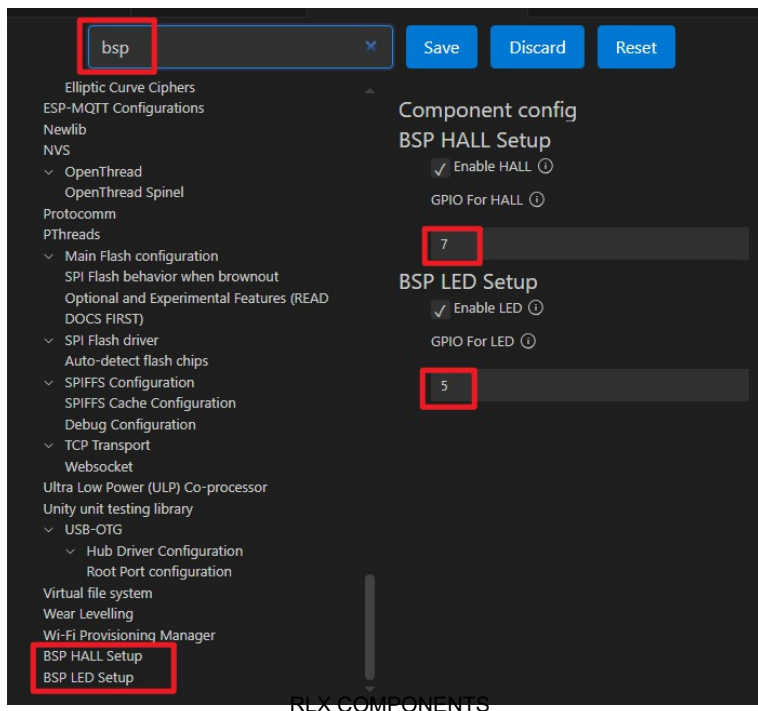


Pri použití dotykových a Halloových senzorov musí byť prepínač v blízkosti bezdrôtového modulu nastavený do polohy Hall a dotyk.



1.4.1 Po klonovaní kódu cez Git (odkaz bude potvrdený) vymažte všetky lokálne kompilačné informácie. Nakonfigurujte prostredie IDF a model čipu pre kompiláciu podľa lekcie 1 a nastavte číslo sériového portu pre programovanie.

1.4.2 Nasledujúca konfigurácia SDKConfig je z veľkej časti identická s lekciovou 1, vyžaduje však rekonfiguráciu pinov Hall a LED.



1.4.3 Kliknite na Kompilovať. Po úspešnej kompilácii kliknite na Stiahnuť.

www.rlx.sk

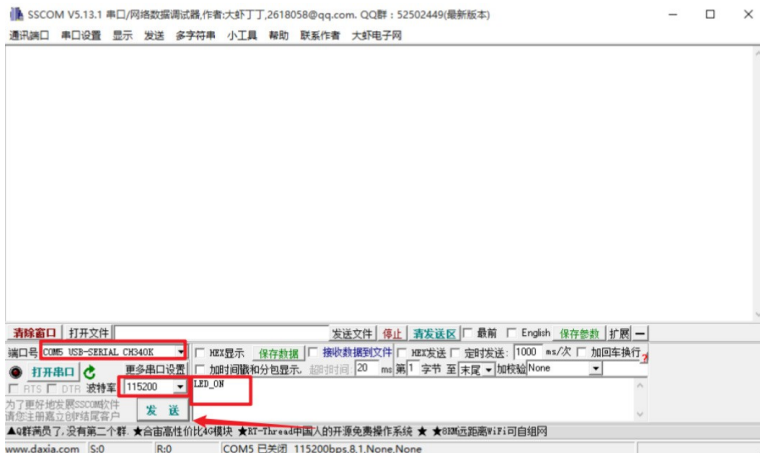


Lekcia 6 – Sériové ovládanie LED

Úvod

Táto kapitola tutorialu predstavuje aplikáciu UART v ESP32-P4, ktorá využíva sériové komunikačné rutiny na uľahčenie pochopenia jej základných funkcií. Ako jedna z najrozšírenejších komunikačných metód vo vývoji vstavaných systémov, UART umožňuje vývojárom rýchlo implementovať výmenu dát medzi vývojovými doskami, PC a periférnymi modulmi, čím vytvára základ pre zložitejšie komunikačné projekty.

Demonštrácia projektu Efekt



Nástroj sériového portu Adresa na stiahnutie :

<https://drive.google.com/drive/folders/1gWt4DU5ANLgNkN1XKIRMDNgYmBAf?usp=sharing>

RLX COMPONENTS



Táto kapitola je rozdelená do nasledujúcich podkapitol

- 1.1 Úvod do UART
- 1.2 Návrh hardvéru
- 1.3 Návrh programu
- 1.4 Stiahnutie a overenie

1.1 Úvod do UART

1.1.1 Úvod do UART

UART (Universal Asynchronous Receiver/Transmitter) je bežný sériový komunikačný protokol, ktorý využíva asynchrónny prenos dát. Nevyžaduje žiadny dodatočný hodinový signál a prenos dát realizuje pomocou iba dvoch signálnych vedení: TX (vysielanie) a RX (prijímanie).

Čip ESP32-P4 obsahuje viacero radičov UART s nasledujúcimi vlastnosťami:

- ① Podpora viacerých kanálov: ESP32-P4 poskytuje až päť rozhraní UART, čo umožňuje simultánnu komunikáciu s viacerými periférnymi zariadeniami.
- ② Flexibilná prenosová rýchlosť: UART podporuje konfigurácie prenosovej rýchlosti v rozmedzí od 300 bps do 5 Mbps, čím vyhovuje rôznym scenárom použitia.
- ③ Hardvérová FIFO: UART obsahuje internú vyrovnávaciu pamäť FIFO, čím znižuje zaťaženie CPU

pri spracovaní komunikačných dát a zvyšuje efektivitu.

- ④ Podpora prerušenia: Rozhranie UART podporuje prerušovacie udalosti, ako je dokončenie prenosu a dokončenie prijatia, vďaka čomu je vhodné pre komunikáciu v reálnom čase.
- ⑤ Vysoká kompatibilita: Protokol UART je jednoduchý a široko používaný v aplikáciách vrátane modulov GPS, modulov Bluetooth, senzorov a výtlačkov na odstraňovanie chýb.

1.1.2 Princíp fungovania UART

1.1.2 Princíp fungovania UART

Komunikácia UART prenáša dáta v bitových jednotkách, zvyčajne používa 8-bitový formát dát s 1 štartovacím bitom a 1 stop bitom. Niektoré aplikácie dodatočne obsahujú paritný bit.

Formát dátového rámca je znázornený nižšie:

Štartovací bit | Dátové bity (D0–D7) | Paritný bit (voliteľný) | Stop

bit Stručný popis prevádzky:

- 1) Keď hostiteľ prenáša dáta do podriadeného zariadenia, pin TX vysiela signálovú úroveň;
- 2) Pin RX prijíma tento signál a dekoduje každý bit v rámci dohodnutého časového intervalu prenosovej rýchlosti;
- 3) Kompletný bajt je nakoniec rekonštruovaný, čo umožňuje bod-bodovú komunikáciu. Na rozdiel od ovládania LED diód prostredníctvom GPIO, UART kladie väčší dôraz na formát údajov a načasovanie. V dôsledku toho je nevyhnutné zabezpečiť, aby bola prenosová rýchlosť na vysielaťacom aj prijímaťacom konci konzistentná; inak dôjde k poškodeniu údajov.

1.2 Hardvérový dizajn

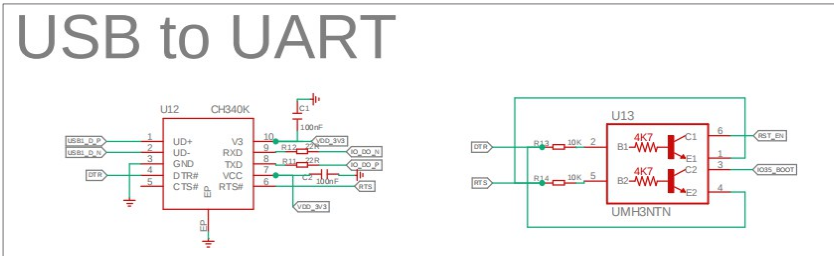
V tomto príklade využívame UART0 vývojovej dosky ESP32-P4 na komunikáciu s PC.

Hardvérové pripojenia sú nasledovné:

TXD0 → čip prevodníka USB na sériový port → softvér sériového terminálu počítača (napr. SecureCRT, XCOM)

RXD0 → prevodník USB na sériový port → softvér sériového terminálu počítača GND ↔ Spoločné uzemnenie

UART0 ESP32-P4 je zvyčajne vopred pripojený k integrovanému čipu prevodníka USB na sériový port. Používatelia nepotrebujú žiadne ďalšie zapojenie; komunikácia sa realizuje pomocou jediného dátového kábla typu C.



Schematický diagram

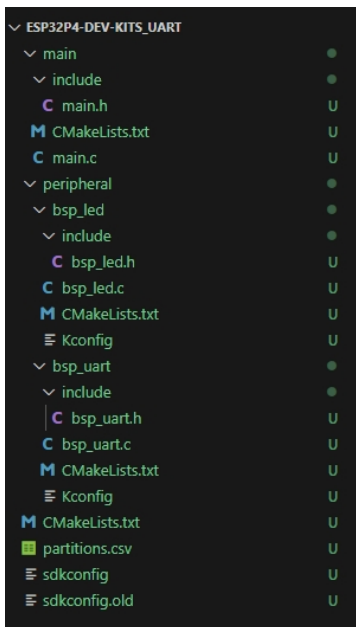
RLX COMPONENTS

1.3 Analýza programu

www.rlx.sk

<https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design>

Otvorte súbor projektu v VS Code podľa predchádzajúcich pokynov.



V rámci príkladu **ESP32P4-dev-kits_uart** bola v adresári **ESP32P4-dev-kits_uart\peripheral** vytvorená nová zložka s názvom **bsp_uart**. V rámci cesty **bsp_uart** bola vytvorená nová zložka **include**, súbor **CMakeLists.txt** a súbor **Kconfig**. Zložka **bsp_uart** obsahuje súbor ovládača **bsp_uart.c**, zložka **include** obsahuje súbor hlavičky **bsp_uart.h** a súbor **CMakeLists.txt** integruje ovládač do systému zostavovania, čím umožňuje projektu využívať funkcie ovládača **HALL**. Súbor **Kconfig** načíta celú konfiguráciu ovládača, vrátane definícií pinov **GPIO**, do súboru **sdkconfig** v rámci platformy **IDF** (konfigurovateľného prostredníctvom grafického rozhrania).

Tu sa zameriame výlučne na základný kód; podrobný zdrojový kód nájdete v príslušných zdrojových súboroch pre tento experiment v materiáloch kódu.

Zdrojový kód ovládača **UART** pozostáva z dvoch súborov: **bsp_uart.c** a **bsp_uart.h**.

Nižšie najprv analyzujeme program **bsp_uart.h**: obsahuje relevantné definície pre piny **UART** a deklarácie funkcií.

/* Odkazy na súbory hlavičiek */

```

esp_err_t uart_init()
{
    esp_err_t err = ESP_OK;
    const uart_config_t uart_config = {
        .baud_rate = 115200,           /*UART baud rate*/
        .data_bits = UART_DATA_8_BITS, /*UART byte size*/
        .parity = UART_PARITY_DISABLE, /*UART parity mode*/
        .stop_bits = UART_STOP_BITS_1, /* UART stop bits*/
        .flow_ctrl = UART_HW_FLOWCTRL_DISABLE, /*UART HW flow control mode (cts/rts)*/
        .source_clk = UART_SCLK_DEFAULT, /*UART source clock selection*/
    };
    err = uart_driver_install(UART_NUM_0, 1024 * 2, 0, 0, NULL, 0); /*Install UART driver and set the UART to the default configuration*/
    if (err != ESP_OK)
    {
        UART_ERROR("uart driver install fail");
        return err;
    }
    err = uart_param_config(UART_NUM_0, &uart_config); /*Set UART configuration parameters*/
    if (err != ESP_OK)
    {
        return err;
    }
    return err;
}

```

V rámci funkcie **uart_init** sa najskôr nakonfigurujú rôzne členské premenné **štruktúry** **uart_config** pomocou parametrov. Následne sa vyvolá funkcia **uart_driver_install**, aby sa zaregistroval príslušný radič sériového portu a konfigurácia vyrovnávacej pamäte. Nakoniec sa pomocou funkcie **uart_param_config** priradia konfiguračné parametre k príslušnému radiču UART.

Stojí za zmienku, že tu konfigurujeme **UART_NUM_0**, čo je predvolený sériový port naprogramovaný počas počiatočného napájovania. Ten využíva predvolené piny **GPIO_NUM_37** a **GPIO_NUM_38**, takže nie je potrebná žiadna dodatočná konfigurácia. Ak chcete použiť iný pin, môžete zavolať funkciu **uart_set_pin** a nastaviť príslušné číslo pinu.

1.3.2 Súbor Kconfig

Hlavnou funkciou tohto súboru je začleniť požadovanú konfiguráciu do súboru **sdkconfig**, čo umožňuje vykonávať určité úpravy parametrov prostredníctvom grafického rozhrania. Konfigurácia pinov tu nie je potrebná; slúži výlučne ako konfigurácia definície makra na aktiváciu funkcií.

```

menu "BSP UART Setup"

    config BSP_UART_ENABLED
        bool "Enable UART"
        default n

endmenu

```

1.3.3 Súbor CMakeLists.txt

Funkčnosť tejto ukážkovej rutiny závisí primárne od ovládača **bsp_uart**. Aby bolo možné úspešne vyvolať obsah priečinka **bsp_uart** v hlavnej funkcii, je potrebné nakonfigurovať súbor **CMakeLists.txt**, ktorý sa nachádza v priečinku **bsp_uart**. Podrobnosti konfigurácie sú nasledovné:

```

FILE(GLOB_RECURSE component_sources "*.c")

idf_component_register(SRCS ${component_sources}
    INCLUDE_DIRS "include"
    REQUIRES driver)

```

RLX COMPONENTS

V tomto súbore CMakeLists.txt sú najskôr definované adresáre pre zdrojové súbory a hlavičkové súbory spolu s požadovanými ~~využívanými~~ ovládačom. Následne sa tieto nastavenia zaregistrujú v systéme zostavovania pomocou príkazu `idf_component_register`, čím sa projektu umožní využívať funkčnosť ovládača `bsp_uart`.

1.3.4 hlavný priečinok

Hlavný priečinok slúži ako hlavný adresár pre spúšťanie programu a obsahuje hlavnú spustiteľnú funkciu `main.c` a hlavičkový súbor `main.h` v priečinku `include`. Pridajte hlavný priečinok do súboru `CMakeLists.txt` systému zostavenia.

Súbor `main.h` odkazuje predovšetkým na požadované hlavičkové súbory: funkcie využívajúce ovládač `bsp_uart` vyžadujú zaradenie hlavičkového súboru `bsp_uart`, zatiaľ čo funkcie využívajúce ovládač `bsp_led` vyžadujú hlavičkový súbor `bsp_led`.

Nížšie je analýza programu `main.c`: inicializácia systému a spustenie funkcie LED a UART.

```
#ifndef CONFIG_BSP_LED_ENABLED
    err = led_init(); /*LED Driver Initialization*/
    if (err != ESP_OK)
        init_fail("led", err);
#endif
#ifdef CONFIG_BSP_UART_ENABLED
    err = uart_init(); /*Uart Driver Initialization*/
    if (err != ESP_OK)
        init_fail("uart", err);
#endif
```

Tento kód sa nachádza vo funkcii `init`, ktorá sa používa na ukladanie inicializačných funkcií vyžadujúcich vyvolanie a na vyhodnotenie výsledku takejto inicializácie. Ak vrátený stav nie je `ESP_OK`, kód zobrazí chybovú správu a zastaví ďalšie vykonávanie.

```
void app_main(void)
{
    MAIN_INFO("-----Demo version-----");
    MAIN_INFO("-----Start the test-----");
    Init();
#ifdef CONFIG_BSP_UART_ENABLED
#ifdef CONFIG_BSP_LED_ENABLED
    int Rx_Bytes = 0; /*Data length read from the serial port buffer*/
    char *data = (char *)malloc(512 + 1); /*Storage space for data read from the serial port buffer*/
    while (1)
    {
        Rx_Bytes = uart_read_bytes(UART_NUM_0, data, 512, pdMS_TO_TICKS(1000)); /*UART read bytes from UART buffer*/
        if (Rx_Bytes > 0)
        {
            data[Rx_Bytes] = 0; /*Append \0 at the end of the string*/
            MAIN_INFO("read data: %s, length: %d", data, Rx_Bytes);
            if (strcmp(data, CMD_LED_ON) == 0) /*Compare whether the string matches the preset command*/
            {
                MAIN_INFO("Received the LED_ON command");
                set_led_state(false); /*Set the Corresponding Output Level of GPIO*/
            }
            else if (strcmp(data, CMD_LED_OFF) == 0) /*Compare whether the string matches the preset command*/
            {
                MAIN_INFO("Received the LED_OFF command");
                set_led_state(true); /*Set the Corresponding Output Level of GPIO*/
            }
            else
            {
                MAIN_ERROR("Unknown Command: %s", data);
            }
        }
    }
#endif
#endif
}
```

V rámci funkcie `app\_main` inicializujte premennú pre počet bajtov prijatých cez sériový port a ukazovateľ pre prijaté dáta, pričom pridajte www.idf.espressif.com veľkosť pamäťového priestoru. Následne vytvorte slučku, ktorá sa bude opakovať:

- (1) Pomocou funkcie `uart_read_bytes` prečítajte 512 bajtov dát s časovým limitom nastaveným na 1 sekundu. To znamená, že dáta v vyrovnávacej pamäti sa prečítajú do 1 sekundy; ak je časový limit prekročený, operácia čítania sa automaticky ukončí a vráti počet prečítaných bajtov.
- (2) Ak je počet prečítaných bajtov väčší ako nula (čo znamená prítomnosť dát v vyrovnávacej pamäti), pripojte na koniec dát nulový znak „0“ na základe celkového počtu prečítaných bajtov. Následne porovnajte údaje s preddefinovaným reťazcom príkazov. Ak funkcia `strcmp` vráti nulu, reťazce sú identické. Príslušný príkaz potom ovláda LED (reťazec `LED_ON` ju zapne, reťazec `LED_OFF` ju vypne). Ak prijatý reťazec nezodpovedá prednastavenému, sériový port vytlačí chybovú správu LOG.

1.3.5 Súbor CMkaLists.txt

Aby bolo možné úspešne vyvolať obsah priečinkov `bsp_led` a `bsp_uart` v hlavnej funkcii, je potrebné nakonfigurovať súbor `CMakeLists.txt`, ktorý sa nachádza v hlavnom priečinku. Podrobnosti konfigurácie sú nasledovné:

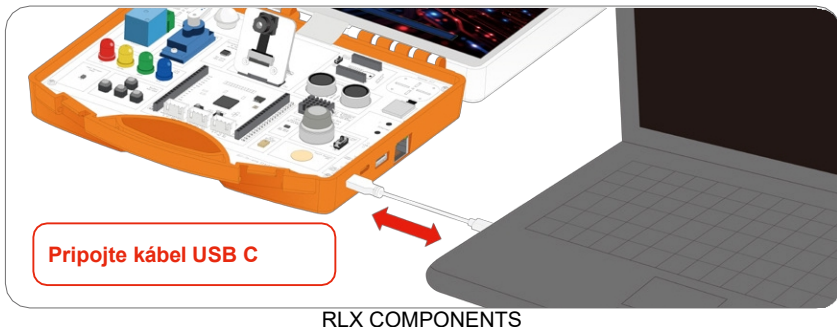
```
FILE(GLOB_RECURSE main ${CMAKE_SOURCE_DIR}/main/*.c)

idf_component_register(SRCS ${main}
                        INCLUDE_DIRS "include"
                        REQUIRES bsp_led bsp_uart)
```

Najskôr sa definujú adresáre pre zdrojové súbory a hlavičkové súbory spolu s požadovanými knižnicami ovládačov – konkrétne knižnice ovládačov potrebné na prepojenie `bsp_led` a `bsp_uart`. Následne sa tieto nastavenia zaregistrujú v kompilačnom systéme pomocou príkazu `idf_component_register`, čím sa hlavnej funkcii umožní využívať tieto schopnosti ovládača.

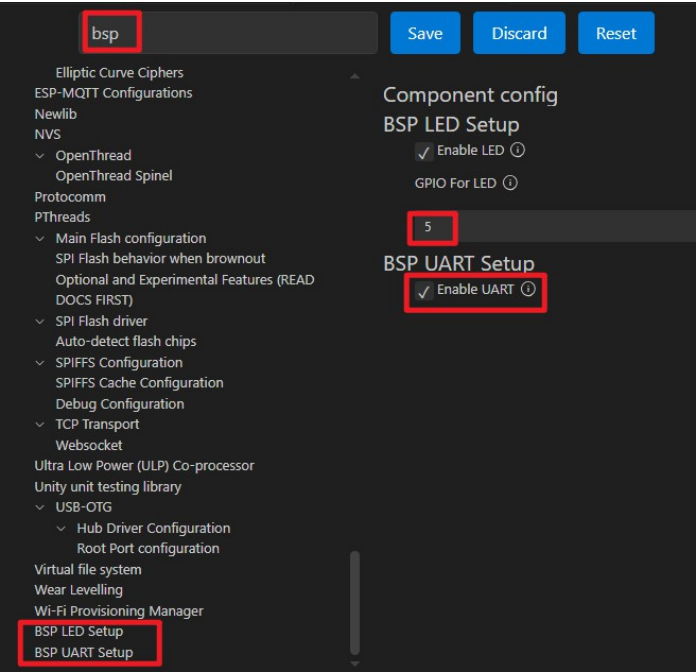
1.4 Postup programovania

Pripojte zariadenie P4 k počítaču prostredníctvom USB

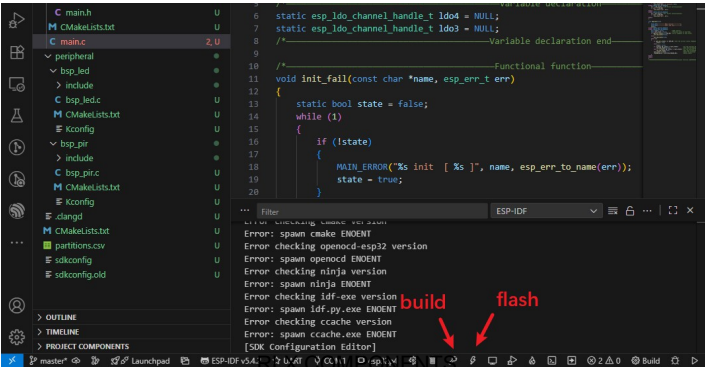


1.4.1 Po klonovaní kódu prostredníctvom Git (odkaz bude potvrdený) vymažte všetky lokálne kompilačné informácie. Nakonfigurujte IDE podľa nasledujúceho odkazu a model čipu pre kompiláciu podľa lekcii 1 a nastavte číslo sériového portu pre programovanie.

1.4.2 Nasledujúca konfigurácia SDKConfig je z veľkej časti identická s lekciami 1. Jednoducho prekonfigurujte piny LED a aktivujte rozhranie UART.



1.4.3 Kliknite na Compile (Kompilovať). Po úspešnej kompilácii kliknite na Download (Stiahnuť).

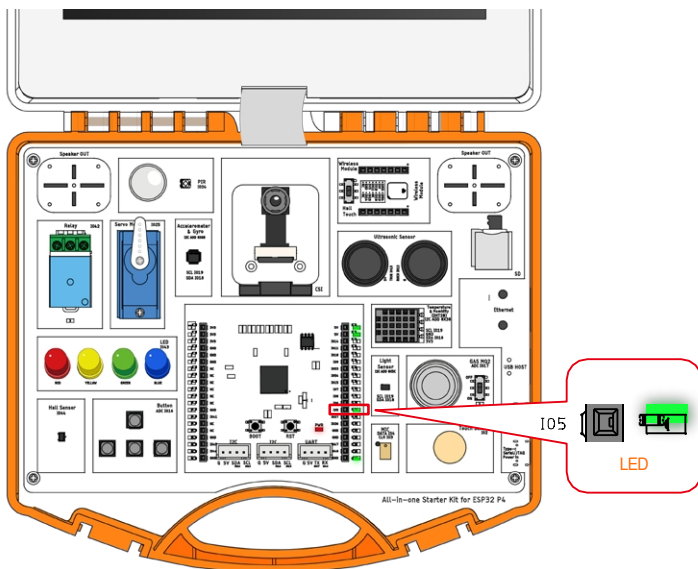


Lekcia 7 – Blikanie LED časovača

Úvod

Táto kapitola tutoriálu predstavuje aplikáciu Timer pre ESP32-P4, pričom na pomoc pri pochopení jej základných funkcií používa príklad rutiny na blikanie LED v časových intervaloch. Ako základné periférne zariadenie v integrovaných systémoch môže časovač presne generovať signály časových intervalov a je široko používaný v scenároch, ako je plánovanie úloh, spúšťanie udalostí a PWM ovládanie. V tejto kapitole budeme ovládať blikanie LED diódy pomocou časovača, aby sme čitateľom pomohli osvojiť si jeho základné použitie a položili základy pre zložitejšie projekty.

Demonštrácia projektu Efekt



Táto kapitola je rozdelená na nasledujúcich podkapitol

- 1.1 Úvod do časovača
- 1.2 Návrh hardvéru
- 1.3 Návrh programu
- 1.4 Stiahnutie a overenie

RLX COMPONENTS

1.1.1 Úvod do časovača

Čip ESP32-P4 integruje viacero časovačov všeobecného použitia s nasledujúcimi charakteristikami:

- ① Multiplexované časovače: Podporuje viacero nezávislých skupín časovačov, čo umožňuje simultánne vykonávanie viacerých časových úloh.
- ② Vysoká presnosť: Časovače pracujú na základe hardvérových hodín a dosahujú presnosť na úrovni mikrosekúnd.
- ③ Funkcia prerušenia: Časovače môžu spúšťať prerušenia po dosiahnutí vopred nastavených hodnôt, aby vykonali špecifické úlohy.
- ④ Flexibilná konfigurácia: Podporuje periodický aj jednorazový režim, aby vyhovoval rôznym požiadavkám aplikácií.
- ⑤ Rozsiahle aplikácie: Bežne sa používa na blikanie LED, plánovanie úloh, počítanie udalostí, detekciu časového limitu, PWM a podobné scenáre.

V podstate časovač funguje ako hardvérový počítadlo, ktoré sa zvyšuje alebo znižuje podľa vopred nastavenej frekvencie hodín. Po dosiahnutí nakonfigurovanej hodnoty spustí udalosť (napríklad prerušenie), čím vykoná úlohy definované používateľom.

1.1.2 Princíp fungovania časovača

Fungovanie časovača možno rozdeliť do nasledujúcich krokov:

- 1) Nastavte zdroj hodín a deliteľný faktor na určenie frekvencie počítania;
- 2) Nastavte časové obdobie (t. j. koľko impulzov spustí udalosť);
- 3) Spustíte časovač, aby začal počítat;
- 4) Keď počítadlo dosiahne prednastavenú hodnotu, spustíte prerušenie;
- 5) Vykonajte úlohu v rámci rutiny obsluhy prerušenia (napr. prepnete stav LED). Proces je znázornený nižšie:

Systémové hodiny → Deliteľ frekvencie → Akumulácia počítadla → Zhodná hodnota → Spúšťacie prerušenie → Vykonanie úlohy

1.2 Hardvérový dizajn

Tento experiment naďalej využíva integrovanú LED diódu na vývojovej doske ESP32-P4 ako výstupný testovací subjekt.

Pin GPIO: LED dióda na doske je pripojená k GPIO5.

Štruktúra obvodu zostáva identická s predchádzajúcou kapitolou, nevyžaduje žiadne dodatočné hardvérové pripojenia.

1.3 Analýza programu

www.rlx.sk

<https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design>

Otvorte súbor projektu v VS Code podľa predchádzajúcich pokynov.



V rámci príkladu **ESP32P4-dev-kits_timer** bola v adresári **ESP32P4-dev-kits_timer\peripheral** vytvorená nová zložka s názvom **bsp_timer**. V adresári **bsp_timer** bola vytvorená nová zložka **include**, súbor **CMakeLists.txt** a súbor **Kconfig**. Zložka **bsp_timer** obsahuje ovládač **bsp_timer.c**, zložka **include** obsahuje hlavičkový súbor **bsp_timer.h** a súbor **CMakeLists.txt** integruje ovládač do systému zostavovania, čím umožňuje projektu **využívať** funkcie ovládača **časovača**. Súbor **Kconfig** načíta celý ovládač spolu s definíciami pinov GPIO do súboru **sdkconfig** v rámci platformy IDF (konfigurovateľný prostredníctvom grafického rozhrania).

1.3.1 Kód ovládača časovača

Tu sa zameriame výlučne na základný kód; podrobný zdrojový kód nájdete v príslušných zdrojových súboroch pre tento experiment v materiáloch kódu.

Zdrojový kód ovládača časovača pozostáva z dvoch súborov: **bsp_timer.c** a **bsp_timer.h**.

Nižšie najprv analyzujeme program **bsp_timer.h**: deklaruje funkcie pre časovač.

/ Odkazy na súbory hlavičiek */*

```
/*-----Header file declaration-----*/
#include "esp_log.h" //References for LOG Printing Function-related API Functions
#include "esp_err.h" //References for Error Type Function-related API Functions
#include "esp_timer.h" //References for high-precision timers Function-related API Functions
/*-----Header file declaration end-----*/
```


/* Deklarácie funkcií a deklarácie definícií makriér */

www.rlx.sk

```
#ifndef CONFIG_BSP_TIMER_ENABLED
esp_err_t timer_init();           /*Initialize the high-precision timer*/
esp_err_t start_timer(uint64_t time); /*Start timer*/
esp_err_t stop_timer(esp_timer_handle_t timer); /*Stop timer*/
bool get_timer_flag();           /*Get the timer flag*/
void reset_timer_flag();         /*Reset the timer flag*/
#endif
```

Ďalej budeme analyzovať program **bsp_timer.c**: inicializácia a konfigurácia časovača spolu s rôznymi volateľnými funkciami API.

/* timer_init */

```
esp_err_t timer_init()
{
    esp_err_t err = ESP_OK;

    /*Timer configuration passed to esp_timer_create()*/
    const esp_timer_create_args_t timer_args = {
        .callback = &periodic_timer_callback, /* Callback function to execute when timer expires*/
        .arg = NULL,                          /*Argument to pass to callback*/
        .name = "Mytimer"                     /*Timer name*/
    };

    err = esp_timer_create(&timer_args, &periodic_timer); /*Create an esp_timer instance*/
    if (err != ESP_OK)
        return err;

    return err;
}
```

V rámci funkcie **timer_init** sa najskôr nakonfigurujú parametre pre každú členskú premennú štruktúry **esp_timer_create_args_t**. (Najdôležitejším prvkom je tu funkcia vykonávania spätného volania, ktorá bude podrobne opísaná neskôr.) Následne sa vyvolá funkcia **esp_timer_create** na vytvorenie ovládača časovača, ktorý prijíma identifikátor prostredníctvom štruktúry **esp_timer_handle_t**. Tento identifikátor uľahčuje následné operácie s ovládačom.

Stojí za zmienku, že na inicializáciu ovládača časovača nie je potrebné volať funkciu **esp_timer_init()**. Táto funkcia sa spustí automaticky po zapnutí čipu. Reinicializácia ovládača je potrebná len v prípade, ak boli použité iné časovače.

/* periodic_timer_callback */

```
void periodic_timer_callback(void *arg) /*Timer Interrupt Handler*/
{
    if (timer_flag == false)
        timer_flag = true;
}
```

V rámci funkcie spätného volania časovača nastavíme vlajku časovača **timer_flag** na hodnotu **true**, čo znamená, že nastavená dĺžka trvania časovača uplynula. Iné funkcie môžu túto vlajku prečítať, aby určili, či bolo dosiahnuté požadované časové obdobie. Následne sa vlajka vymaže, aby sa zabezpečilo, že časovač môže byť opäť spustený.

/* Načítanie vlajky časovača get_timer_flag a vynulovanie vlajky časovača reset_timer_flag */

```

bool get_timer_flag() /*Get the timer flag*/
{
    return timer_flag;
}

void reset_timer_flag() /*Reset the timer flag*/
{
    timer_flag = false;
}

```

/* Spustite funkciu časovača start_timer */

```

esp_err_t start_timer(uint64_t time)
{
    esp_err_t err = ESP_OK;
    err = esp_timer_start_periodic(periodic_timer, time); /*Start a periodic timer(period: us)*/
    if (err != ESP_OK)
    {
        TIMER_ERROR("Start a periodic timer error");
        return err;
    }
    return err;
}

```

V rámci tejto funkcie sa vyvolá funkcia **esp_timer_start_periodic**, aby sa spustil periodický časovač (t. j. pre cyklické vykonávanie). Vstupný parameter určuje časový interval v mikrosekundách. Úpravou hodnoty tejto funkcie je možné nastaviť časovanie na rôzne trvania.

1.3.2 Súbor Kconfig

Hlavnou funkciou tohto súboru je začleniť požadovanú konfiguráciu do súboru **sdkconfig**, čím sa umožní vykonávať určité úpravy parametrov prostredníctvom grafického rozhrania. Konfigurácia pinov nie je v tomto prípade potrebná; slúži výlučne ako konfigurácia definície makra na povolenie funkčnosti.

```

menu "BSP Timer Setup"

    config BSP_TIMER_ENABLED
        bool "Enable Timer"
        default n

endmenu

```

1.3.3 Súbor CMkaLists.txt

Funkčnosť tejto ukážkovej rutiny závisí primárne od ovládača **bsp_timer**. Na úspešné vyvolanie obsahu priečinka **bsp_timer** v hlavnej funkcii je potrebné nakonfigurovať súbor CMakeLists.txt, ktorý sa nachádza v priečinku **bsp_timer**.

RLX COMPONENTS

Podrobnosti konfigurácie sú nasledovné:

www.rlx.sk

```
FILE(GLOB_RECURSE component_sources "*.c")

idf_component_register(SRCS ${component_sources}
                       INCLUDE_DIRS "include"
                       REQUIRES esp_timer)
```

V tomto súbore CMakeLists.txt sú najskôr definované adresáre pre zdrojové súbory a hlavičkové súbory spolu s požadovanými knižnicami ovládačov. Následne sa tieto nastavenia zaregistrujú v systéme zostavovania pomocou príkazu idf_component_register, čím sa projektu umožní využívať funkčnosť ovládača bsp_timer.

1.3.4 Hlavný priečnik

Hlavný priečnik slúži ako hlavný adresár pre spúšťanie programu a obsahuje hlavnú spustiteľnú funkciu main.c a hlavičkový súbor main.h v priečniku include. Pridajte hlavný priečnik do súboru **CMakeLists.txt** systému zostavenia.

Súbor main.h odkazuje predovšetkým na požadované hlavičkové súbory: funkcie využívajúce ovládač **bsp_timer** vyžadujú zaradenie hlavičkového súboru **bsp_timer**, zatiaľ čo funkcie využívajúce ovládač **bsp_led** vyžadujú hlavičkový súbor **bsp_led**.

Nižšie je analýza programu main.c: inicializácia systému a vykonávanie funkcií LED a časovača.

```
#ifdef CONFIG_BSP_LED_ENABLED
    err = led_init(); /*LED Driver Initialization*/
    if (err != ESP_OK)
        init_fail("led", err);
#endif
#ifdef CONFIG_BSP_TIMER_ENABLED
    err = timer_init(); /*Timer Driver Initialization*/
    if (err != ESP_OK)
        init_fail("timer", err);
#endif
```

Tento kód sa nachádza vo funkcii init, ktorá sa používa na ukladanie inicializačných funkcií vyžadujúcich vyvolanie a na vyhodnotenie výsledku takejto inicializácie. Ak vrátený stav nie je **ESP_OK**, kód zobrazí chybovú správu a ukončí ďalšie vykonávanie.

```

void app_main(void)
{
    MAIN_INFO("-----Demo version-----");
    MAIN_INFO("-----Start the test-----");
    Init();
#ifdef CONFIG_BSP_TIMER_ENABLED
#ifdef CONFIG_BSP_LED_ENABLED
    start_timer(3000000); /*Start a periodic timer -3s*/
    while (1)
    {
        if (get_timer_flag()) /*Get the timer flag*/
        {
            led_toggle();
            reset_timer_flag(); /*Reset the timer flag*/
        }
        vTaskDelay(pdTICKS_TO_MS(10));
    }
#endif
#endif
}

```

V rámci funkcie ``app_main`` sa funkcia ``start_timer`` používa na spustenie periodického časovača s cyklom 3 sekundy.

Následne sa vytvorí slučka. V rámci tejto slučky sa každých 10 milisekúnd kontroluje vlajka časovača. Ak je návratová hodnota pravdivá, vykoná sa funkcia prepínania LED (pozri kód z lekcie 1). Po vykonaní sa vlajka časovača vymaže, aby sa pripravila na ďalšiu iteráciu kontroly.

1.3.5 Súbor CMkaLists.txt

Aby bolo možné úspešne vyvolať obsah priečinkov **bsp_led** a **bsp_timer** v hlavnej funkcii, je potrebné nakonfigurovať súbor **CMakeLists.txt**, ktorý sa nachádza v hlavnom priečinku. Podrobnosti konfigurácie sú nasledovné:

```

FILE(GLOB_RECURSE main ${CMAKE_SOURCE_DIR}/main/*.c)

idf_component_register(SRCS ${main}
                       INCLUDE_DIRS "include"
                       REQUIRES bsp_led bsp_timer)

```

Najsôr sa definujú adresáre pre zdrojové súbory a hlavičkové súbory spolu s požadovanými knižnicami ovládačov – konkrétne knižnice ovládačov potrebné na prepojenie **bsp_led** a **bsp_timer**. Následne sa tieto nastavenia zaregistrujú v systéme zostavenia pomocou príkazu **idf_compo-nent_register**, čím sa hlavnej funkcii umožní využívať tieto funkcie ovládačov.

1.4 Postup programovania

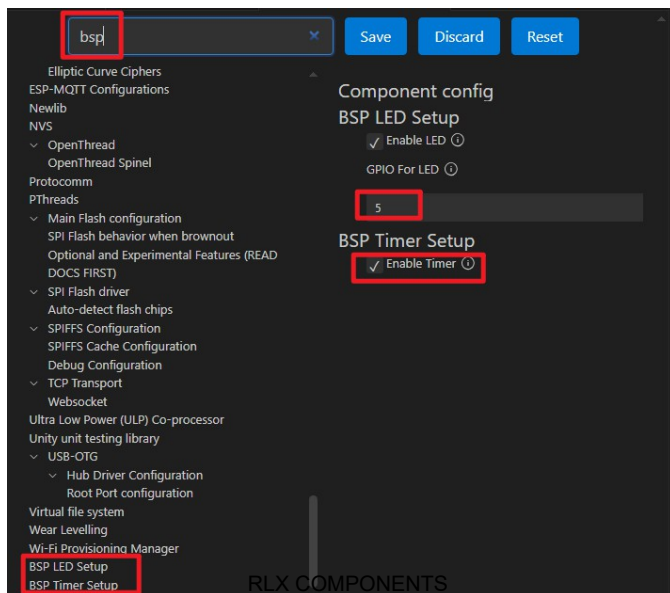
www.rlx.sk

Pripojte zariadenie P4 k počítaču cez USB

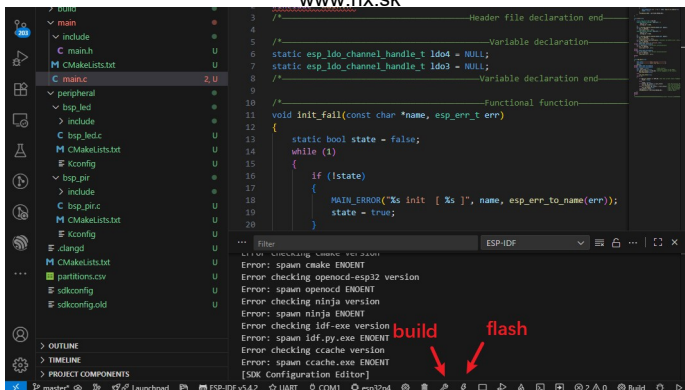


1.4.1 Po klonovaní kódu prostredníctvom Git (odkaz bude potvrdený) vymažte všetky lokálne kompilačné informácie. Nakonfigurujte prostredie IDF a model čipu pre kompiláciu podľa lekcii 1 a nastavte číslo sériového portu pre programovanie.

1.4.2 Nasledujúca konfigurácia SDKConfig je z veľkej časti identická s lekciami 1. Jednoducho prekonfigurujte piny LED a aktivujte rozhranie Timer.



www.rlx.sk



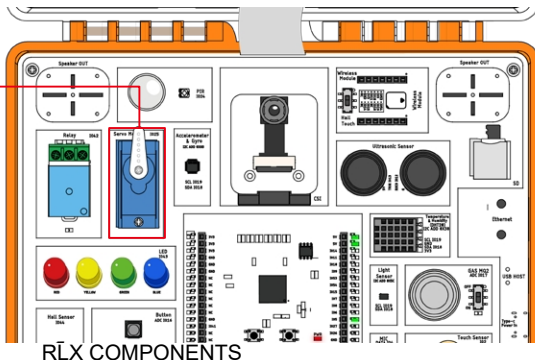
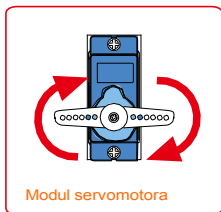
Lekcia 8 – PWM servo ovládanie

Úvod

Táto kapitola tutoriálu predstavuje aplikáciu výstupu PWM ESP32-P4 pomocou príkladu servo riadenia, ktorý vám pomôže pochopiť základné funkcie PWM.

Servá, ako bežné pohony, sú nevyhnutnými komponentmi v robotoch, diaľkovo ovládaných modeloch a automatizovaných zariadeniach. V tejto kapitole sa čitatelia naučia, ako generovať PWM vlny pomocou pinov GPIO ESP32-P4 na ovládanie 360-stupňového serva, čím si vytvoria základ pre zložitejšie projekty riadenia pohybu.

Demonštrácia projektu Efekt



- 1.1 Servomotory a PWM Úvod
- 1.2 Návrh hardvéru
- 1.3 Návrh programu
- 1.4 Stiahnutie a overenie

1.1 Úvod do servomotorov a PWM

1.1.1 Úvod do PWM

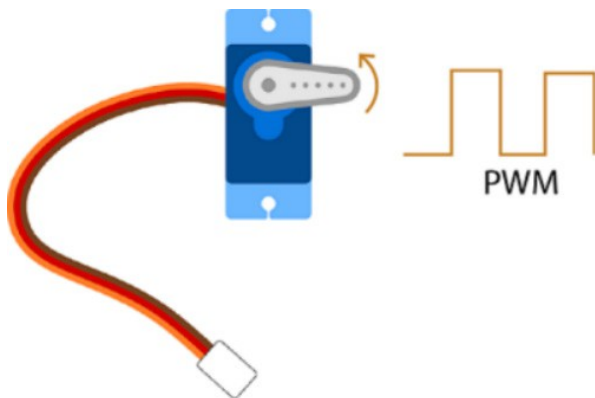
Pulzná šírková modulácia (PWM) je bežná metóda digitálneho riadenia signálu, ktorá reguluje priemerné napätie výstupného signálu ovládaním pomeru trvania vysokej úrovne k perióde cyklu (pracovný cyklus). ESP32-P4 poskytuje bohatú škálu PWM kanálov vhodných na ovládanie zariadení, ako sú motory, servá a podsvietenie.

Kľúčové vlastnosti:

- ① Flexibilita: Programovateľný pracovný cyklus a frekvencia, vhodný pre rôzne aplikácie od stmievania LED až po reguláciu otáčok motora.
- ② Vysoká presnosť: PWM ESP32-P4 obsahuje časovače s vysokým rozlíšením, ktoré umožňujú plynulé ovládanie.
- ③ Viackanálová schopnosť: Súčasne poháňa viacero servomotorov alebo motorov, aby splnil komplexné požiadavky na pohyb.

1.1.2 Úvod do servomotorov

Servo je motor s riadením uhla, ktorý pozostáva z jednosmerného motora, prevodoviek a obvodov spätnej väzby polohy.



RLX COMPONENTS

Štandardný 180° servomotor: Ovládaný uhlom, zvyčajne 0° až 180°.

360° servo s nepretržitým otáčaním: Nema pevný uhol, ale riadi smer a rýchlosť otáčania. V tomto tutoriáli sa používa 360-stupňové servo, ktoré sa ovláda takto:

PWM cyklus: Zvyčajne 20 ms (50 Hz).

Pracovný cyklus: Určuje smer a rýchlosť otáčania.

Približne 1,0 ms → Otáčanie v smere hodinových ručičiek

Približne 1,5 ms → Zastavenie

Približne 2,0 ms → Otáčanie proti smeru hodinových ručičiek

1.1.3 Princípy servomotorového pohonu

Servo riadenie sa v zásade dosahuje prostredníctvom modulácie šírky impulzov

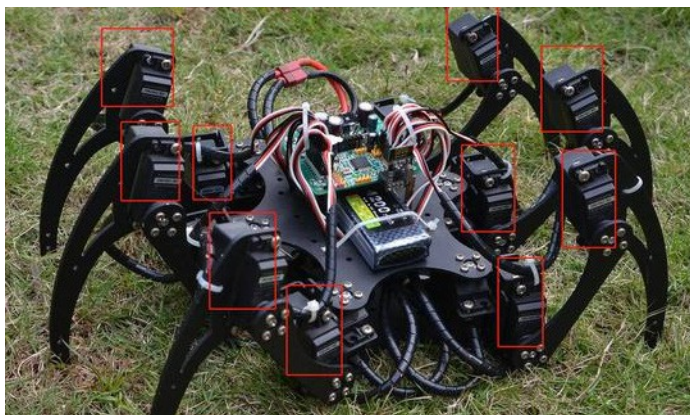
PWM: ESP32-P4 vysiela PWM vlnu s periódou 20 ms;

Rôzne šírky impulzov (1–2 ms) predstavujú odlišné stavy otáčania;

Vnútorne obvody serva upravujú prevádzku motora na základe tohto signálu, čím riadia rýchlosť a smer otáčania.

Na rozdiel od riadenia LED, ktoré zahŕňa jednoduché stavy „zapnuté/vypnuté“, servá vyžadujú „nepretržitý PWM signál“, aby si udržali svoju polohu.

Servá nachádzajú široké uplatnenie v rôznych scenároch, ktoré vyžadujú uhlové ovládanie.



1.2 Hardvérový dizajn

V tomto príklade používame vývojovú dosku ESP32-P4 spárovanú s 360-stupňovým servomotorom.

Konfigurácia pripojenia je nasledovná:

RLX COMPONENTS

Servo VCC → 5V napájanie (alebo vývojová doska 5V) Servo GND

→ Vývojová doska GND

www.rlx.sk

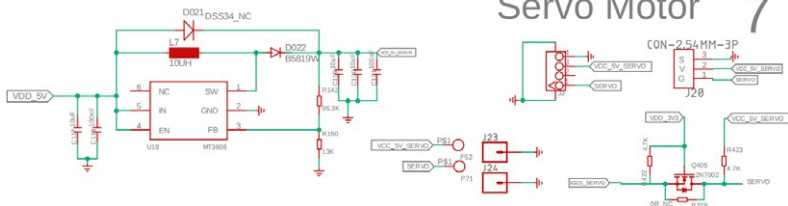
Signálny pin serva → Voľný PWM pin na ESP32-P4 (napr. GPIO25)

Dôležité poznámky:

Servá pracujú pri 5 V, zatiaľ čo ESP32-P4 vysiela signály 3,3 V. Väčšina serv je však kompatibilná.

Ak má pracovať viacero serv súčasne, je potrebný dodatočný napájací zdroj, aby sa predišlo nedostatočnému prúdu z USB portu vývojovej dosky.

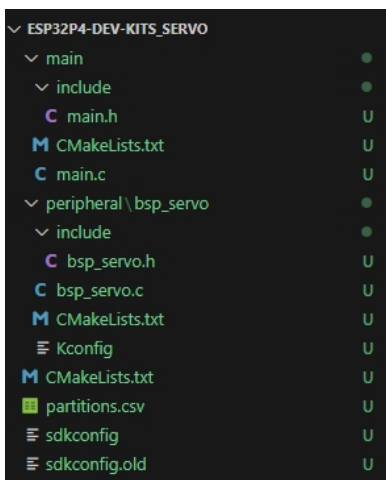
Servo Motor 7



1.3 Analýza programu

<https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design>

Otvorte súbor projektu v VS Code podľa predchádzajúcich pokynov.



V rámci príkladu **ESP32P4-dev-kits_servo** bol v adresári **ESP32P4-dev-kits_servo\peripheral** vytvorený nový priečinok s názvom **bsp_servo**. V priečinku **bsp_servo** bol vytvorený nový priečinok include, súbor **CMakeLists.txt** a súbor **Kconfig**. Zložka **bsp_servo** obsahuje súbor ovládača **bsp_servo.c**, zložka include obsahuje súbor hlavičky **bsp_servo.h** a súbor **CMakeLists.txt** integruje ovládač do systému zostavovania, čím umožňuje projektu využívať funkciu riadenia servomotora. Súbor **Kconfig** načíta celú konfiguráciu ovládača, vrátane definícií pinov GPIO, do súboru **sdkconfig** v rámci platformy IDF (konfigurovateľnej prostredníctvom grafického rozhrania).

1.3.1 Kód ovládača SERVO

www.rlx.sk

Tu sa zameriame výlučne na základný kód, podrobnejší zdrojový kód nájdete v príslušných zdrojových súboroch pre tento experiment v materiáloch kódu.

Zdrojový kód ovládača SERVO pozostáva z dvoch súborov: **bsp_servo.c** a **bsp_servo.h**.

Nižšie najprv analyzujeme program **bsp_servo.h**: obsahuje relevantné definície pre servo piny a deklarácie funkcií.

/* Odkazy na súbory hlavičiek */

```
/*-----Header file declaration-----*/
#include "esp_log.h" //References for LOG Printing Function-related API Functions
#include "esp_err.h" //References for Error Type Function-related API Functions
#include "driver/gpio.h" //References for GPIO Function-related API Functions
#include "driver/ledc.h" //References for LEDC PWM Function-related API Functions
/*-----Header file declaration end-----*/
```

/* Deklarácie funkcií a deklarácie definícií makriér */

```
#ifndef CONFIG_BSP_SERVO_ENABLED
#define SERVO_GPIO CONFIG_SERVO_GPIO

typedef enum
{
    forward_dir = 1,
    reverse_dir,
    stop,
} servo_dir; /*Servo Motor Direction Type*/

esp_err_t servo_init(); // Initialize the GPIO Pin and PWM channel of the LED*/
esp_err_t set_servo_status(servo_dir servo_dir_status, int speed); /*Set the servo motor direction and speed*/

#endif
```

Ďalej budeme analyzovať program **bsp_servo.c**: inicializácia a konfigurácia servo pinov a volanie funkcie setup.

/* Inicializačná funkcia servo_init */

```
esp_err_t servo_init()
{
    esp_err_t err = ESP_OK;
    const gpio_config_t gpio_config = {
        .pin_bit_mask = 1ULL << SERVO_GPIO, /* GPIO pin: set with bit mask, each bit maps to a GPIO */
        .mode = GPIO_MODE_OUTPUT, /* GPIO mode: set input/output mode */
        .pull_up_en = false, /* GPIO pull-up */
        .pull_down_en = false, /* GPIO pull-down */
        .intr_type = GPIO_INTR_DISABLE, /* GPIO Interrupt type */
    };
    err = gpio_config(&gpio_config); /*Configure GPIO*/
    if (err != ESP_OK)
        return err;

    const ledc_timer_config_t servo_timer_config = {
        .clk_cfg = LEDC_USE_PPL_DIV_CLK, /* Configure LEDC source clock from ledc_clk_cfg_t*/
        .duty_resolution = LEDC_TIMER_11_BIT, /*LEDC channel duty resolution*/
        .freq_hz = 50, /*LEDC timer frequency (Hz)*/
        .speed_mode = LEDC_LOW_SPEED_MODE, /*LEDC speed speed_mode, high-speed mode (only exists on esp32) or low-speed mode*/
        .timer_num = LEDC_TIMER_1, /* The timer source of channel (0 - LEDC_TIMER_MAX-1)*/
    };
    const ledc_channel_config_t channel_config = {
        .gpio_num = SERVO_GPIO, /*The LEDC output gpio_num*/
        .speed_mode = LEDC_LOW_SPEED_MODE, /*LEDC speed speed_mode, high-speed mode (only exists on esp32) or low-speed mode*/
        .channel = LEDC_CHANNEL_1, /*LEDC channel (0 - LEDC_CHANNEL_MAX-1)*/
        .intr_type = LEDC_INTR_DISABLE, /*Configure interrupt, fade interrupt enable or fade interrupt disable*/
        .timer_sel = LEDC_TIMER_1, /*Select the timer source of channel (0 - LEDC_TIMER_MAX-1)*/
        .duty = 154, /*LEDC channel duty, the range of duty setting is [0, (2^duty_resolution)] */
        .hpoint = 0, /*LEDC channel hpoint value, the range is [0, (2^duty_resolution)-1]*/
    };
    err = ledc_timer_config(&servo_timer_config); /*LEDC timer configuration Configure LEDC timer with the given source timer/frequency(Hz)/duty_resolution*/
    if (err != ESP_OK)
        return err;
    err = ledc_channel_config(&channel_config); /*LEDC channel configuration Configure LEDC channel with the given channel/output gpio_num/interrupt/source timer/frequency(Hz)/LEDC dut*/
    if (err != ESP_OK)
        return err;
    return err;
}
```

V rámci funkcie `servo_init` boli najskôr nakonfigurované členské premenné štruktúry `gpio_config_t` s parametrami. Následne sa vyvolá funkcia `gpio_config`, aby sa dokončila inicializácia GPIO pomocou týchto konfiguračných parametrov. Potom sa nakonfiguruje štruktúra `ledc_timer_config_t`, ktorá nastavuje parametre časovača LEDC (frekvencia, rozlíšenie atď.). Potom sa konfiguruje štruktúra `ledc_channel_config_t`, ktorá slúži na prepojenie používaného portu GPIO s príslušným časovačom LEDC a kanálom LEDC. Nakoniec sa vyvolajú funkcie `ledc_timer_config` a `ledc_channel_config`, aby sa dokončila inicializácia radiča LEDC a jeho kanálov.

`/* Nastavenie funkcie pohybu serva set_servo_status */`

```
/*speed - (0 - 4(fastest)) */
esp_err_t set_servo_status(servo_dir_t servo_dir_status, int speed)
{
    esp_err_t err = ESP_OK;
    if ((speed < 0) || (speed > 4)) /*The speed is not within the specified range*/
        return ESP_ERR_INVALID_ARG;
    switch (servo_dir_status) /**/
    {
        case stop:
            err = ledc_set_duty(LEDC_LOW_SPEED_MODE, LEDC_CHANNEL_1, 154); /*LEDC set duty*/
            if (err != ESP_OK)
                return err;
            err = ledc_update_duty(LEDC_LOW_SPEED_MODE, LEDC_CHANNEL_1); /*LEDC update channel parameters*/
            if (err != ESP_OK)
                return err;
            break;
        case forward_dir:
            err = ledc_set_duty(LEDC_LOW_SPEED_MODE, LEDC_CHANNEL_1, (132 - (speed * 20)));
            if (err != ESP_OK)
                return err;
            err = ledc_update_duty(LEDC_LOW_SPEED_MODE, LEDC_CHANNEL_1);
            if (err != ESP_OK)
                return err;
            break;
        case reverse_dir:
            err = ledc_set_duty(LEDC_LOW_SPEED_MODE, LEDC_CHANNEL_1, (speed * 20) + 176);
            if (err != ESP_OK)
                return err;
            err = ledc_update_duty(LEDC_LOW_SPEED_MODE, LEDC_CHANNEL_1);
            if (err != ESP_OK)
                return err;
            break;
    }
    return err;
}
```

Táto funkcia má dva parametre: smer serva a rýchlosť serva. Typ smeru serva je definovaný v súbore `bsp_servo.h` a zahŕňa dopredu, dozadu a zastavenie. Premenná rýchlosti serva má rozsah od 0 do 4, čím sa rýchlosť konfiguruje do piatich rôznych nastavení. Táto funkcia určuje, ktoré z funkcií `ledc_set_duty` a

`ledc_update_duty` sa má vykonať na základe smeru serva. Tieto funkcie nastavujú pracovný cyklus pre aktuálny kanál PWM a aktualizujú aktuálne nastavenia na vykonanie.

Stojí za zmienku, že používame 360-stupňový servo, nie 180-stupňový servo. Preto PWM vlnová forma, ktorú vysielame, neovláda uhol otáčania serva, ale skôr jeho rýchlosť otáčania.

1.3.2 Súbor Kconfig

www.rlx.sk

Hlavnou funkciou tohto súboru je pridať požadovanú konfiguráciu do súboru sdkconfig, čím sa umožní modifikácia určitých nastavení parametrov prostredníctvom grafického rozhrania. Tu sa 25 vzťahuje na GPIO_NUM_25.

```
menu "BSP_SERVO Setup"
    config BSP_SERVO_ENABLED
        bool "Enable SERVO config"
        default n

    if BSP_SERVO_ENABLED
        config SERVO_GPIO
            int "GPIO For SERVO"
            default 25
    endif
endmenu
```

1.3.3 Súbor CMakeLists.txt

Funkčnosť tejto ukážkovej rutiny závisí predovšetkým od ovládača **bsp_servo**. Aby bolo možné úspešne vyvolať obsah priečinka **bsp_servo** v hlavnej funkcii, je potrebné nakonfigurovať súbor **CMakeLists.txt**, ktorý sa nachádza v priečinku **bsp_servo**. Podrobnosti konfigurácie sú nasledovné:

```
FILE(GLOB_RECURSE component_sources "*.c")

idf_component_register(SRCS ${component_sources}
                       INCLUDE_DIRS "include"
                       REQUIRES driver)
```

V tomto súbore **CMakeLists.txt** sú najskôr definované adresáre pre zdrojové súbory a hlavičkové súbory spolu s požadovanými knižnicami ovládačov. Následne sa tieto nastavenia zaregistrujú v systéme zostavovania pomocou príkazu **idf_component_register**, čo umožňuje projektu využívať funkčnosť ovládača **bsp_servo**.

1.3.4 Hlavný priečinok

Hlavný priečinok slúži ako hlavný adresár pre spúšťanie programu a obsahuje hlavnú spustiteľnú funkciu **main.c** a hlavičkový súbor **main.h** v priečinku **include**. Pridajte hlavný priečinok do súboru **CMakeLists.txt** systému zostavenia.

Súbor **main.h** odkazuje predovšetkým na požadované hlavičkové súbory: funkcie využívajúce ovládač **bsp_servo** vyžadujú zaradenie hlavičkového súboru **bsp_servo**.

Nižšie je analýza programu **main.c**: inicializácia systému a spustenie funkcie **servo**.

```
int main(void)
{
    #ifdef CONFIG_BSP_SERVO_ENABLED
        err = servo_init(); /*Servo Driver Initialization*/
        if (err != ESP_OK)
            init_fail("servo", err);
    #endif
}
```

RLX COMPONENTS

Tento kód sa nachádza vo funkcii `init`, ktorá sa používa na ukladanie inicializačných funkcií vyžadujúcich vyvolanie a na vyhodnotenie výsledku takejto inicializácie. Ak vrátený stav nie je `ESP_OK`, kód zobrazí chybovú správu a ukončí ďalšie vykonávanie.

```
void app_main(void)
{
    MAIN_INFO("-----Demo version-----");
    MAIN_INFO("-----Start the test-----");
    Init();
#ifdef CONFIG_BSP_SERVO_ENABLED
    while (1)
    {
        for (uint8_t i = 0; i < 5; i++) /*Execute control with different speeds in a loop*/
        {
            set_servo_status(forward_dir, i); /*Set the servo motor direction and speed*/
            vTaskDelay(2000 / portTICK_PERIOD_MS);
            set_servo_status(reverse_dir, i); /*Set the servo motor direction and speed*/
            vTaskDelay(2000 / portTICK_PERIOD_MS);
        }
        set_servo_status(stop, 1); /*Set the servo motor stop*/
        vTaskDelay(2000 / portTICK_PERIOD_MS);
    }
#endif
}
```

V rámci funkcie `app_main` vytvorte slučku. V rámci tejto slučky vytvorte vnořenú slučku, ktorá je navrhnutá tak, aby vykonala svoj účel päťkrát. Táto vnořená slučka bude prechádzať nasledujúcou sekvenciou: najskôr otočte servo v smere hodinových ručičiek; po dvojsekundovom oneskorení otočte smer otáčania; potom postupne vykonajte všetky rýchlosti otáčania, pričom postupne zvyšujte rýchlosť otáčania. Po dokončení tejto sekvencie a opustení vnořenej slučky vykonajte príkaz na zastavenie serva. Po dvojsekundovom oneskorení pokračujte vo vykonávaní pôvodnej slučky.

1.3.5 Súbor `CMkaLists.txt`

Aby bolo možné úspešne vyvolať obsah priečinka `bsp_servo` v hlavnej funkcii, je potrebné nakonfigurovať súbor `CMakeLists.txt`, ktorý sa nachádza v hlavnom priečinku. Podrobnosti konfigurácie sú nasledovné:

```
FILE(GLOB_RECURSE main ${CMAKE_SOURCE_DIR}/main/*.c)

idf_component_register(SRCS ${main}
                       INCLUDE_DIRS "include"
                       REQUIRES bsp_servo)
```

Najskôr sa definujú adresáre pre zdrojové súbory a hlavičkové súbory spolu s požadovanou knižnicou ovládačov, konkrétne knižnicou ovládačov pre prepojenie `bsp_servo`. Následne sa tieto nastavenia zaregistrujú v systéme zostavovania pomocou príkazu `idf_component_register`, čím sa hlavnej funkcii umožní využívať tieto funkcie ovládačov.

1.4 Postup programovania

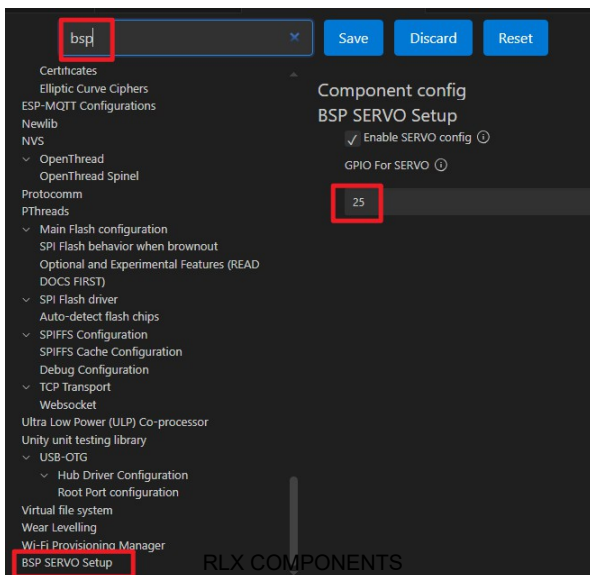
www.rlx.sk

Pripojte zariadenie P4 k počítaču prostredníctvom USB

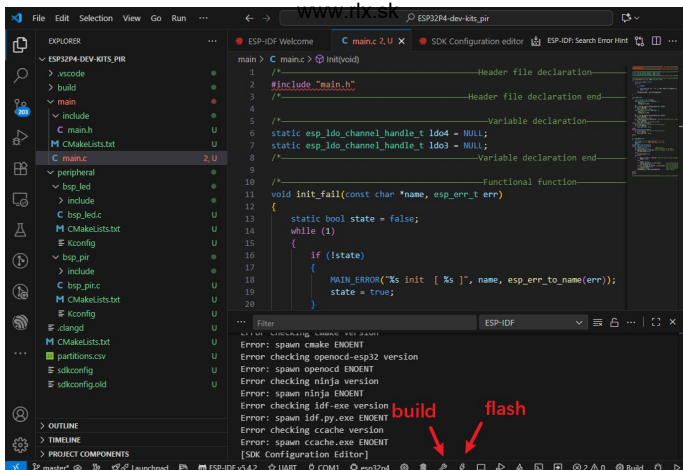


1.4.1 Po klonovaní kódu cez Git (odkaz bude potvrdený) vymažte všetky lokálne kompilačné informácie. Nakonfigurujte prostredie IDF a model čipu pre kompiláciu podľa lekcie 1 a nastavte číslo sériového portu pre programovanie.

1.4.2 Nasledujúca konfigurácia SDKConfig je z veľkej časti identická s lekciami 1, vyžaduje len rekonfiguráciu servo pinu.



1.4.3 Kliknite na Kompilovať. Po úspešnej kompilácii kliknite na Stiahnuť.

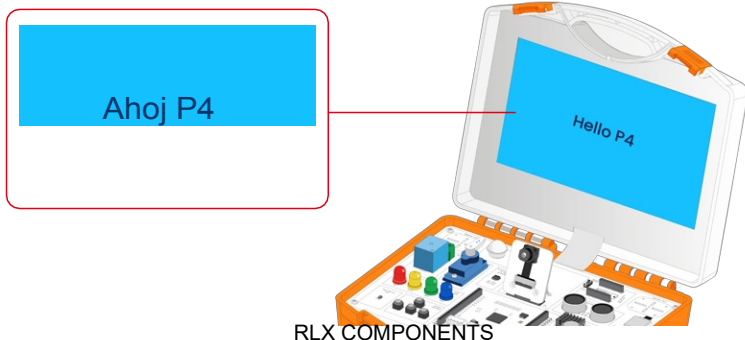


Lekcia 9 – LCD displej Hello

Úvod

Táto kapitola tutoriálu predstavuje grafické zobrazovacie aplikácie pre ESP32-P4. Prostredníctvom príkladov rutín využívajúcich LVGL (Light and Versatile Graphics Library) v kombinácii s rozhraním MIPI DSI pomáha pochopiť jeho základné schopnosti. Rozsvietenie obrazovky a vykresľovanie základnej grafiky slúžia ako klasické testovacie prípady, ktoré čitateľom umožňujú získať jednoduché, ale komplexné pochopenie aplikácií pre zobrazenie na ESP32-P4. To vytvára základ pre zložitejšie projekty GUI v nasledujúcich fázach.

Demonštrácia projektu Efekt



- 1.1 Úvod do LVGL a MIPI DSI
- 1.2 Návrh hardvéru
- 1.3 Návrh softvéru
- 1.4 Stiahnutie a overenie

1.1 Úvod do LVGL a MIPI DSI

1.1.1 Úvod do LVGL

LVGL je open-source vývojový rámec pre vstavané grafické používateľské rozhranie, ktorý sa vyznačuje nízkou hmotnosťou, multiplatformovosťou a vysokým výkonom a je široko využívaný na rôznych platformách MCU a SoC.

Medzi jeho hlavné charakteristiky patria:

- ① Rozsiahla knižnica ovládacích prvkov: Poskytuje komponenty grafického používateľského rozhrania, ako sú tlačidlá, indikátory priebehu, grafy a obrázky, pre rýchlu konštrukciu rozhraní človek-stroj.
- ② Podpora hardvérovej akcelerácie: LVGL je integrovaný s 2D akcelerátorom a DMA ESP32-P4, čím výrazne zvyšuje rýchlosť vykresľovania;
- ③ Podpora multitaskingu: Kompatibilný s RTOS pre plynulé prepínanie rozhraní a komplexné spracovanie logiky;
- ④ Prispôsobiteľnosť: Témy, štýly a písma sú konfigurovateľné, aby sa prispôbili rôznym požiadavkám produktov;
- ⑤ Kompatibilita medzi platformami: Podporuje nasadenie od low-end MCU po vysokovýkonné čipy, čo uľahčuje prenosnosť a škálovateľnosť.

Prostredníctvom LVGL môžu vývojári rýchlo implementovať sofistikované návrhy rozhraní bez potreby priamej implementácie základných grafických algoritmov.

1.1.2 Úvod do MIPI DSI

MIPI DSI (Display Serial Interface) je vysokorýchlostný sériový protokol rozhrania displeja, ktorý sa široko používa v displejoch smartfónov a tabletov. ESP32-P4 obsahuje integrovaný radič MIPI DSI, ktorý umožňuje priame ovládanie displejov s vysokým rozlíšením.

Medzi jeho kľúčové vlastnosti patria:

- ① Vysokorýchlostná sériová komunikácia: Podporuje prenosové rýchlosti presahujúce 500 Mbps až 1 Gbps, vhodné pre displeje s vysokým rozlíšením;
- ② Podpora viacerých kanálov: voliteľné režimy prenosu 1–4 kanálov, aby vyhovovali rôznym požiadavkám na rozlíšenie a obnovovaciu frekvenciu;
- ③ Nízka spotreba energie: Využívanie diferenciálneho prenosu signálu pre zníženú spotrebu energie a zvýšenú odolnosť proti rušeniu;

RLX COMPONENTS

- V príklade displeja ESP32-P4 slúži LVGL ako grafický renderovací engine, zatiaľ čo MIPI DSI funguje ako hardvérové prenosové rozhranie. Ich integrácia umožňuje rýchle osvetlenie obrazovky a zobrazenie rozhrania.

Z hľadiska hardvéru sa vývojová doska ESP32-P4 pripája k LCD obrazovke prostredníctvom rozhrania MIPI DSI. Obrazovka zvyčajne obsahuje vlastný integrovaný obvod na riadenie napájania (PMIC) a obvody na riadenie podsvietenia.

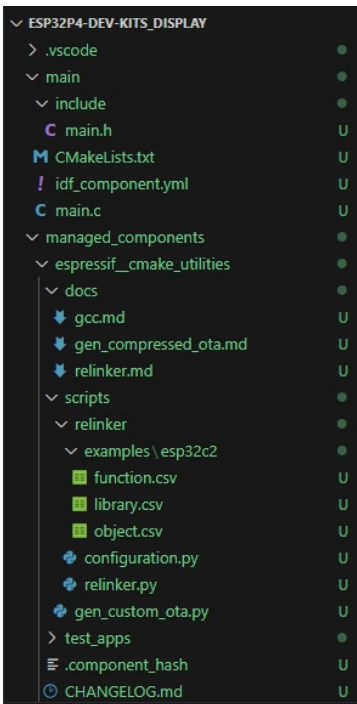
ESP32-P4 vyžaduje na ovládanie displeja iba správne napájanie hodín a inicializačné príkazy.

1.3 Analýza programu

www.rlx.sk

<https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design>

Otvorte súbor projektu v VS Code podľa predchádzajúcich pokynov.



V rámci príkladu **ESP32P4-dev-kits_display** bola v adresári **ESP32P4-dev-kits_display\peripheral** vytvorená nová zložka s názvom **bsp_display**.

V rámci cesty **bsp_display** bol vytvorený nový priečinok **include**, súbor **CMakeLists.txt** a súbor **Kconfig**. Zložka **bsp_display** obsahuje súbor ovládača **bsp_display.c**, zložka **include** obsahuje súbor hlavičky **bsp_display.h** a súbor **CMakeLists.txt** integruje ovládač do systému zostavovania, čím umožňuje projektu využívať funkcie ovládača displeja. Súbor **Kconfig** načíta celú konfiguráciu ovládača, vrátane definícií pinov GPIO, do súboru **sdkconfig** v rámci platformy IDF (konfigurovateľného prostredníctvom grafického rozhrania).

1.3.1 Kód ovládača displeja

Tu sa zameriame výlučne na základný kód; podrobný zdrojový kód nájdete v príslušných zdrojových súboroch pre tento experiment v materiáloch kódu.

Zdrojový kód ovládača displeja pozostáva z dvoch súborov: **bsp_display.c** a **bsp_display.h**.

Nižšie najprv analyzujeme program **bsp_display.h**: obsahuje relevantné definície pre piny displeja a deklarácie funkcií.

```
/* Odkazy na hlavičkové súbory */
```

```

/*                                     Header file declaration                                     */
#include "esp_log.h" //References for LOG Printing Function-related API Functions
#include "esp_err.h" //References for Error Type/Value-Related API Functions
#include "esp_ldo_regulator.h" //References for LDO Function-related API Functions
#include "esp_lcd_ek79807.h" //References for lcd ek79807 Function-related API Functions
#include "esp_lcd_mipi_dsi.h" //References for lcd mipi dsi Function-related API Functions
#include "esp_lcd_panel_ops.h" //References for lcd panel ops Function-related API Functions
#include "esp_lcd_panel_io.h" //References for lcd panel io Function-related API Functions
#include "esp_lvgl_port.h" //References for LVGL port Function-related API Functions
#include "driver/gpio.h" //References for GPIO Function-related API Functions
#include "driver/ledc.h" //References for LEDC PWM Function-related API Functions
#include "lvgl.h" //References for LVGL Function-related API Functions
/*                                     Header file declaration end                                     */

```

/* Deklarácie funkcií a definície makier */

```

#ifdef CONFIG_BSP_DISPLAY_ENABLED

#define V_size CONFIG_V_SIZE // Vertical resolution
#define H_size CONFIG_H_SIZE // Horizontal resolution
#define BITS_PER_PIXEL CONFIG_BITS_PER_PIXEL // Number of image display bits of the display screen

#define LCD_GPIO_RESET CONFIG_LCD_GPIO_RESET // LCD Reset GPIO
#define LCD_GPIO_BLIGHT CONFIG_LCD_GPIO_BLIGHT // LCD Blight GPIO
#define LCD_GPIO_UPDN CONFIG_LCD_GPIO_UPDN // LCD UPDN GPIO
#define LCD_GPIO_SHLR CONFIG_LCD_GPIO_SHLR // LCD SHLR GPIO
#define BLIGHT_PWM_Hz CONFIG_BLIGHT_PWM_Hz // LCD Blight PWM GPIO

#define LV_COLOR_RED lv_color_make(0xFF, 0x00, 0x00) // LVGL Red
#define LV_COLOR_GREEN lv_color_make(0x00, 0xFF, 0x00) // LVGL Green
#define LV_COLOR_BLUE lv_color_make(0x00, 0x00, 0xFF) // LVGL Blue
#define LV_COLOR_WHITE lv_color_make(0xFF, 0xFF, 0xFF) // LVGL White
#define LV_COLOR_BLACK lv_color_make(0x00, 0x00, 0x00) // LVGL Black
#define LV_COLOR_GRAY lv_color_make(0x80, 0x80, 0x80) // LVGL gray
#define LV_COLOR_YELLOW lv_color_make(0xFF, 0xFF, 0x00) // LVGL yellow

esp_err_t display_init(); // Display Screen Initialization Function
esp_err_t set_lcd_blight(uint32_t brightness); // Set the screen backlight
#endif

```

Ďalej budeme analyzovať program **bsp_display.c**: inicializácia a konfigurácia pinov displeja a volanie funkcie setup.

/* Funkcia inicializácie podsvietenia blight_init */

```

#ifdef CONFIG_BSP_DISPLAY_ENABLED
static esp_err_t blight_init(void)
{
    esp_err_t err = ESP_OK;
    const gpio_config_t gpio_config = {
        .pin_bit_mask = (1ULL << LCD_GPIO_RESET) | (1ULL << LCD_GPIO_BLIGHT) | (1ULL << LCD_GPIO_UPDN) | (1ULL << LCD_GPIO_SHLR), /* GPIO pin: set with bit mask, each bit maps to a GPIO */
        .mode = GPIO_MODE_OUTPUT, /* GPIO mode: set input/output mode */
        .pull_up_en = false, /* GPIO pull-up */
        .pull_down_en = false, /* GPIO pull-down */
        .intr_type = GPIO_INTR_DISABLE, /* GPIO Interrupt type */
    };
    err = gpio_config(&gpio_config); /*Configure GPIO*/
    if (err != ESP_OK)
        return err;

    const ledc_timer_config_t timer_config = {
        .clk_cfg = LEDC_USE_FRC_DIV_CLK, /* Configure LEDC source clock from ledc_clk_cfg */
        .duty_resolution = LEDC_TIMER_11_BIT, /*LEDC channel duty resolution*/
        .freq_hz = BLIGHT_PWM_Hz, /*LEDC timer frequency (Hz)*/
        .speed_mode = LEDC_LOW_SPEED_MODE, /*LEDC speed speed_mode, high-speed mode (only exists on esp32) or low-speed mode*/
        .timer_num = LEDC_TIMER_0, /* The timer source of channel (0 - LEDC_TIMER_MAX-1)*/
    };
    const ledc_channel_config_t channel_config = {
        .gpio_num = LCD_GPIO_BLIGHT, /*the LEDC output gpio_num*/
        .speed_mode = LEDC_LOW_SPEED_MODE, /*LEDC speed speed_mode, high-speed mode (only exists on esp32) or low-speed mode*/
        .channel = LEDC_CHANNEL_0, /*LEDC channel (0 - LEDC_CHANNEL_MAX-1)*/
        .intr_type = LEDC_INTR_DISABLE, /*configure interrupt, fade interrupt enable or fade interrupt disable*/
        .timer_sel = LEDC_TIMER_0, /*Select the timer source of channel (0 - LEDC_TIMER_MAX-1)*/
        .duty = 0, /*LEDC channel duty, the range of duty setting is [0, (2**duty_resolution)-1] */
        .hpoint = 0, /*LEDC channel hpoint value, the range is [0, (2**duty_resolution)-1]*/
    };
    err = ledc_timer_config(&timer_config); /*LEDC timer configuration Configure LEDC timer with the given source timer/frequency(Hz)/duty_resolution*/
    if (err != ESP_OK)
        return err;
    err = ledc_channel_config(&channel_config); /*LEDC channel configuration Configure LEDC channel with the given channel/output gpio_num/interrupt/source timer/frequency(Hz)/LEDC duty*/
    if (err != ESP_OK)
        return err;
    return err;
}

```

RLX COMPONENTS

V rámci funkcie **blight_init** sa najskôr parametrizujú členské premenné štruktúry **gpio_config_t**. Následne sa vyvolá funkcia **gpio_pin_config**, aby sa dokončila inicializácia GPIO pomocou týchto konfiguračných parametrov. (Tu sa inicializujú všetky piny potrebné pre následné operácie). Následne sa konfiguruje štruktúra **ledc_timer_config_t**, čím sa nastaví parametre časovača LEDC (frekvencia, rozlíšenie atď.). Potom sa konfiguruje štruktúra **ledc_channel_config_t**, aby sa pripojil používaný port GPIO s príslušným **časovačom LEDC** a **kanálom LEDC**. Nakoniec sa vyvolajú funkcie **`ledc\_timer\_config`** a **`ledc\_channel\_config`**, aby sa dokončila inicializácia radiča LEDC a jeho kanálov. (Tým sa zaregistruje rozhranie PWM pre ovládanie podsvietenia.)

/* Funkcia na nastavenie jasu podsvietenia: `set\_lcd\_brightness` */

```
/*brightness = (0 ~ 100) */
esp_err_t set_lcd_blight(uint32_t brightness)
{
    esp_err_t err = ESP_OK;
    if (brightness != 0)
    {
        err = ledc_set_duty(LEDC_LOW_SPEED_MODE, LEDC_CHANNEL_0, ((brightness * 18) + 200)); /*LEDC set duty*/
        if (err != ESP_OK)
            return err;
        err = ledc_update_duty(LEDC_LOW_SPEED_MODE, LEDC_CHANNEL_0); /*LEDC update channel parameters*/
        if (err != ESP_OK)
            return err;
    }
    else
    {
        err = ledc_set_duty(LEDC_LOW_SPEED_MODE, LEDC_CHANNEL_0, 0);
        if (err != ESP_OK)
            return err;
        err = ledc_update_duty(LEDC_LOW_SPEED_MODE, LEDC_CHANNEL_0);
        if (err != ESP_OK)
            return err;
    }
    return err;
}
```

Táto funkcia má jeden parameter: percentuálnu hodnotu jasu podsvietenia (0–100). Vykonáva funkcie **`ledc\_set\_duty`** a **`ledc\_update\_duty`** na základe zadaného percentuálneho jasu. Tieto funkcie nastavujú pracovný cyklus pre aktuálny kanál PWM a aktualizujú aktuálne nastavenia na vykonanie.

/* Funkcia ovládača portu displeja: `display\_port\_init` */

```
static esp_err_t display_port_init(void)
{
    esp_err_t err = ESP_OK;
    lcd_color_rgb_pixel_format_t dpi_pixel_format; /*LCD color pixel format in RGB color space*/
    /*MIPI DSI bus configuration structure*/
    esp_lcd_dsi_bus_config_t bus_config = {
        .bus_id = 0, /*Select which DSI controller, index from 0*/
        .num_data_lanes = 2, /*Number of data lanes, if set to 0, the driver will fallback to use maximum number of lanes*/
        .phy_clk_src = MIPI_DSI_PHY_CLK_SRC_DEFAULT, /*MIPI DSI PHY clock source*/
        .lane_bit_rate_mbps = 900, /*Lane bit rate in Mbps*/
    };
    err = esp_lcd_new_dsi_bus(&bus_config, &mipi_dsi_bus); /*Create MIPI DSI bus handle*/
    if (err != ESP_OK)
        return err;
    /*Panel ID configuration structure, for MIPI DSI command interface*/
    esp_lcd_dbi_io_config_t dbi_config = {
        .virtual_channel = 0, /*Virtual channel ID, index from 0*/
        .lcd_cmd_bits = 8, /*Bit-width of LCD command*/
        .lcd_param_bits = 8, /*Bit-width of LCD parameter*/
    };
    err = esp_lcd_new_panel_io_dbi(mipi_dsi_bus, &dbi_config, &mipi_dbi_io); /*Create LCD panel IO, for MIPI DSI DSI interface*/
    if (err != ESP_OK)
        return err;
    /*Select according to the number of data display digits*/
    if (BITS_PER_PIXEL == 24)
        dpi_pixel_format = LCD_COLOR_PIXEL_FORMAT_RGB888;
    else if (BITS_PER_PIXEL == 18)
        dpi_pixel_format = LCD_COLOR_PIXEL_FORMAT_RGB666;
    else if (BITS_PER_PIXEL == 16)
        dpi_pixel_format = LCD_COLOR_PIXEL_FORMAT_RGB565;
}
```

Najskôr nakonfigurujete štruktúru `esp_lcd_dsi_bus_config_t`, aby ste nastavili rozhranie MIPI DSI. Keďže používame 2-pásmový displej, parametre `num_data_lanes` je nastavený na 2. Použite funkciu `esp_lcd_new_dsi_bus` na vytvorenie nového ovládača zbernice DSI. Ďalej nakonfigurujete štruktúru `esp_lcd_dbi_io_config_t` a vytvorte rukoväť rozhrania IO pre rozhranie MIPI DSI DBI pomocou funkcie `esp_lcd_new_panel_io_dbi`. Nakoniec vyberte typ vykresľovania farieb obrazovky prostredníctvom rôznych nastavení bitovej hĺbky (predvolené nastavenie je RGB565).

```
/*MIPI DSI DPI panel configuration structure*/
const esp_lcd_dpi_panel_config_t dpi_config = {
    .dpi_clk_src = MIPI_DSI_DPI_CLK_SRC_DEFAULT, /*MIPI DSI DPI clock source*/
    .dpi_clock_freq_mhz = 51, /* DPI clock frequency in MHz*/
    .virtual_channel = 0, /*Virtual channel ID, index from 0*/
    .pixel_format = dpi_pixel_format, /*Pixel format that used by the MIPI LCD device*/
    .num_fb = 1, /*Number of screen-sized frame buffers that allocated by the driverd */
    .video_timing = {
        .h_size = H_size, /*Horizontal resolution*/
        .v_size = V_size, /*Vertical resolution*/
        .hsync_back_porch = 160, /*Horizontal back porch, number of pixel clock between hsync and start of line active data*/
        .hsync_pulse_width = 70, /*Horizontal sync width, in pixel clock*/
        .hsync_front_porch = 160, /*Horizontal front porch, number of pixel clock between the end of active data and the next hsync*/
        .vsync_back_porch = 22, /*Vertical back porch, number of invalid lines between vsync and start of frame*/
        .vsync_pulse_width = 10, /*Vertical sync width, in number of lines*/
        .vsync_front_porch = 12, /*Vertical front porch, number of invalid lines between the end of frame and the next vsync*/
    },
    .flags.use_dma2d = true, /*Use DMA2D to copy user buffer to the frame buffer when necessary*/
};

/*LCD panel vendor configuration*/
ek79007_vendor_config_t vendor_config = {
    .mipi_config = {
        .dsi_bus = mipi_dsi_bus, /*MIPI-DSI bus configuration*/
        .dpi_config = &dpi_config, /*MIPI-DPI panel configuration */
    },
};

/*configuration structure for panel device*/
const esp_lcd_panel_dev_config_t panel_config = {
    .reset_gpio_num = LCD_GPIO_RESET, /* GPIO used to reset the LCD panel, set to -1 if it's not used*/
    .rgb_ele_order = LCD_RGB_ELEMENT_ORDER_RGB, /*Set RGB element order, RGB or BGR*/
    .bits_per_pixel = BITS_PER_PIXEL, /*Color depth, in bpp*/
    .vendor_config = &vendor_config, /*vendor specific configuration, optional, left as NULL if not used*/
};

err = esp_lcd_new_panel_ek79007(mipi_dbi_io, &panel_config, &panel_handle); /*Create LCD panel for model EK79007*/
if (err != ESP_OK)
    return err;
err = esp_lcd_panel_reset(panel_handle); /*Reset LCD panel*/
if (err != ESP_OK)
    return err;
err = esp_lcd_panel_init(panel_handle); /*Initialize LCD panel*/
if (err != ESP_OK)
    return err;
return err;
```

Konfigurujeme štruktúru `esp_lcd_dpi_panel_config_t`, aby sme nastavili parametre špecifické pre náš displej. Potom konfigurujeme štruktúru `ek79007_vendor_config_t`, do ktorej začleňujeme našu registrovanú konfiguráciu displeja a nastavenia rozhrania zbernice MIPI DSI. Následne konfigurujeme štruktúru `esp_lcd_panel_dev_config_t`, ktorá nastavuje našu farebnú bitovú hĺbku, režim zobrazenia a pin na resetovanie displeja.

Funkcia `esp_lcd_new_panel_ek79007` vytvorí nové ovládacie rozhranie pre náš ovládač displeja. V prípade výmeny obrazovky je potrebné túto funkciu prekonfigurovať s príslušným ovládačom displeja. Zaregistruje identifikátor pre tento konkrétny ovládač displeja. Nasledujúce volania funkcie `esp_lcd_panel_reset` resetujú túto rukoväť a nakoniec funkcia `esp_lcd_panel_init` inicializuje rukoväť a konfiguruje inicializáciu rozhrania displeja. Stojí za zmienku, že konfigurácia štruktúry `video_timing` zahŕňa špecifické parametre pre displej. Tieto parametre musia byť nastavené podľa dátového manuálu displeja, pričom rozlíšenie a ostatné parametre musia presne zodpovedať špecifikáciám uvedeným v tomto manuáli.

/* Funkcia ovládača portu lvgl lvgl_init */

RLX COMPONENTS

```
static esp_err_t lvgl_init()
{
    esp_err_t err = ESP_OK;
    const lvgl_port_cfg_t lvgl_cfg = {
        .task_priority = configMAX_PRIORITIES - 4, /* LVGL task priority */
        .task_stack = 8192, /* LVGL task stack size */
        .task_affinity = -1, /* LVGL task pinned to core (-1 is no affinity) */
        .task_max_sleep_ms = 10, /* Maximum sleep in LVGL task */
        .timer_period_ms = 5, /* LVGL timer tick period in ms */
    };
    err = lvgl_port_init(&lvgl_cfg); /*Initialize LVGL portation*/
    if (err != ESP_OK)
    {
        DISPLAY_ERROR("LVGL port initialization failed");
    }
}
```

www.rlx.sk

Najskôr nakonfigurujete štruktúru **lvgl_port_cfg_t**, ktorá nastavuje parametre, ako je kontrola zásobníka a priorita pre vlákno lvgl. Potom použijete funkciu **lvgl_port_init** na inicializáciu a vytvorenie vlákna **lvgl**.

```
const lvgl_port_display_cfg_t disp_cfg = {
    .io_handle = esp_lcd_io, /*LCD IO handle*/
    .panel_handle = panel_handle, /*LCD panel handle*/
    .control_handle = panel_handle, /*LCD panel control handle*/
    .buffer_size = (H_size * V_size * ((BITS_PER_PIXEL * 7) / 8)), /*Size of the buffer for the screen in pixels*/
    .double_buffer = true, /*true, if should be allocated two buffers*/
    .hres = H_size, /*LCD display horizontal resolution*/
    .vres = V_size, /*LCD display vertical resolution*/
    .monochrome = false, /*true, if display is monochrome and using bit for ipa*/
};
#if LVGL_VERSION_MAJOR >= 9
    .color_format = LV_COLOR_FORMAT_RGB565,
#endif
.rotation = {
    .swapped_xy = false, /*LCD Screen swapped X and Y (in esp_lcd driver)*/
    .mirror_x = false, /*LCD Screen mirrored X (in esp_lcd driver)*/
    .mirror_y = false, /*LCD Screen mirrored Y (in esp_lcd driver)*/
},
.flags = {
    .buff_dma = false, /*Allocated LVGL buffer will be DMA capable*/
    .buff_spiram = true, /*Allocated LVGL buffer will be in PSRAM*/
    .sw_rotate = true, /* Use software rotation (slower) or PPA if available*/
};
#if LVGL_VERSION_MAJOR >= 9
    .swap_bytes = true, /*Swap bytes in RAM064 (16-bit) before send to LCD driver*/
#endif
#if CONFIG_DISPLAY_LVGL_FULL_REFRESH
    .full_refresh = true,
#else
    .full_refresh = false, /*Always make the whole screen redrawn*/
#endif
#if CONFIG_DISPLAY_LVGL_DIRECT_MODE
    .direct_mode = true,
#else
    .direct_mode = false, /*Use screen-sized buffers and draw to absolute coordinates*/
#endif
};
const lvgl_port_display_dsi_cfg_t lvgl_dsi_cfg = {
    .flags = {
        .avoid_tearing = true,
        .avoid_tearing = false, /*Use internal MIPI-DSI buffers as a LVGL draw buffers to avoid tearing effect, enabling this option requires over two LCD buffers and may reduce the frame rate*/
    }
};
```

Najskôr nakonfigurujete štruktúru **lvgl_port_display_cfg_t** nastavením rukovätí získaných z funkcie ovládača portu displeja v rámci štruktúry **lvgl**. Parameter **buffer_size** definuje vyrovnávaciu pamäť displeja, ktorú je možné nakonfigurovať na základe požiadaviek na obnovovaciu frekvenciu, napríklad vyrovnávacia pamäť na celú obrazovku alebo polovicu obrazovky. Parameter **double_buffer** určuje, či je povolené dvojité vyrovnávanie, čo výrazne zlepšuje obnovovaciu frekvenciu displeja.

Kľúčový bod: Pole **buff_dma** v štruktúre **flags** určuje, či je vyrovnávacia pamäť alokovaná v oblasti DMA. Nastavenie tejto hodnoty na **true** ponúka vysokú efektivitu zobrazenia a rýchle obnovovacie frekvencie. Prostriedky čipového priestoru DMA sú však zvyčajne obmedzené, čo túto možnosť vo všeobecnosti robí nepraktickou. Navyše, obmedzenia priestoru často bránia

konfigurácii na celú obrazovku, čo je problém pri použití displeja s nízkym rozlíšením.

RLX COMPONENTS

buff_spi ram konfiguruje vyrovnávaciu pamäť v oblasti PSRAM; táto možnosť sa vzájomne vylučuje s **buff_dma**. Nastavenie tejto možnosti na hodnotu **true** umožňuje **vyrovnávanie na celej obrazovke**, poskytuje dostatočný priestor a spĺňa požiadavky na rýchlosť, čím je vhodná pre displeje **s vysokým rozlíšením**.

Po konfigurácii inicializujte a zaregistrujte konfiguráciu LVGL displeja pomocou funkcie **lvgl_port_add_disp_dsi**, ktorá vráti kontrolný identifikátor.

/* Funkcia inicializácie obrazovky display_init */

```
esp_err_t display_init()
{
    esp_err_t err = ESP_OK;
    err = blight_init(); /*Backlight initialization function*/
    if (err != ESP_OK)
    {
        return err;
    }
    err = display_port_init(); /*Display screen interface initialization function*/
    if (err != ESP_OK)
    {
        return err;
    }
    err = lvgl_init(); /*Screen interface registration LVGL function*/
    if (err != ESP_OK)
    {
        {
            DISPLAY_ERROR("Display init fail");
            return err;
        }
    }
    gpio_set_level(LCD_GPIO_UPDN, 0); /*Set the display screen vertical mirroring*/
    gpio_set_level(LCD_GPIO_SHLR, 1); /*Set the display screen horizontal mirroring*/
    set_lcd_blight(0); /*Set the backlight to turn off*/
    return err;
}
```

Táto funkcia volá **inicializáciu podsvietenia**, **inicializáciu ovládača** displeja a **inicializáciu lvgl**, pričom kombinuje tieto inicializačné funkcie do jedného inicializačného procesu. Následne volá funkciu **gpio_set_level**, aby nastavila vertikálny zrkadlový pin na nízku úroveň a horizontálny zrkadlový pin na vysokú úroveň. Tieto dva zrkadlovacie efekty sú konfigurované podľa požiadaviek používaného displeja. Nakoniec nastaví jas podsvietenia obrazovky na 0 % (t. j. vypne podsvietenie).

1.3.2 Súbor Kconfig

Hlavnou funkciou tohto súboru je začleniť potrebné konfigurácie do **súboru sdkconfig**, čo umožňuje vykonávať určité úpravy parametrov prostredníctvom grafického rozhrania. Tieto parametre zahŕňajú: farebnú hĺbku obrazovky: **16 bitov pre RGB565 a 24 bitov pre RGB888**; pin na resetovanie **obrazovky**, pin na ovládanie **podsvietenia**, **vertikálny zrkadlový pin**, **horizontálny zrkadlový pin** a frekvenciu **PWM podsvietenia**. Číselné označenia pinov zodpovedajú sekvencii **GPIO_NUM**.

```

menu "BSP_DISPLAY Setup"
config BSP_DISPLAY_ENABLED
bool "Enable display functions"

if BSP_DISPLAY_ENABLED
config H_SIZE
int "Horizontal Resolution"
default 1024

config V_SIZE
int "Vertical Resolution"
default 600

config BITS_PER_PIXEL
int "Color depth"
default 16

config LCD_GPIO_RESET
int "GPIO For LCD RESET"
default 5

config LCD_GPIO_BLIGHT
int "GPIO For LCD BACKLIGHT"
default 20

config LCD_GPIO_UPDN
int "GPIO For LCD UPDN"
default 32

config LCD_GPIO_SHLR
int "GPIO For LCD SHLR"
default 33

config BLIGHT_PWM_Hz
int "the PWM output HZ"
default 30000

config DISPLAY_LVGL_FULL_REFRESH
bool "Enable fullscreen refresh"

config DISPLAY_LVGL_DIRECT_MODE
bool "Use screen-sized buffers"

config DISPLAY_LVGL_AVOID_TEAR
bool "Enable to avoid fragmentation"
endif
endmenu

```

www.rlx.sk

1.3.3 Súbor CMkaLists.txt

Funkčnosť tejto ukážkovej rutiny závisí predovšetkým od ovládača **bsp_display**. Aby bolo možné úspešne vyvolať obsah priečinka **bsp_display** v hlavnej funkcii, je potrebné nakonfigurovať súbor **CMakeLists.txt**, ktorý sa nachádza v priečinku **bsp_display**. Podrobnosti konfigurácie sú nasledovné:

```

FILE(GLOB_RECURSE component_sources "*.c")

idf_component_register(SRCS ${component_sources}
                      INCLUDE_DIRS "include"
                      REQUIRES driver esp_lcd_ek79007 lvgl esp_lvgl_port)

```

V tomto súbore **CMakeLists.txt** sú najskôr definované adresáre pre zdrojové súbory a hlavičkové súbory spolu s požadovanými knižnicami ovládačov (knižnica ovládača pre čip ovládača displeja **ek79007** a knižnica ovládača **lvgl**). Následne sa tieto nastavenia zaregistrujú v systéme zostavovania pomocou príkazu **idf_component_register**, čím sa projektu umožní **využívať** funkcie ovládača **bsp_display**.

1.3.4 Hlavný priečinok

Hlavný priečinok slúži ako hlavný adresár pre vykonávanie programu a obsahuje

RLX COMPONENTS

hlavnú funkciu `main.c` a hlavičkový súbor `main.h` v priečinku `include`. Pridajte hlavný priečinok do súboru `CMakeLists.txt` systému zostavenia `cmake`.

Súbor `main.h` odkazuje predovšetkým na požadované hlavičkové súbory: funkcie využívajúce ovládač `bsp_display` vyžadujú zaradenie hlavičkového súboru `bsp_display`.

Nižšie je uvedená analýza programu `main.c`: inicializácia systému a vykonávanie funkcií špecifických pre displej.

```
#ifndef CONFIG_BSP_DISPLAY_ENABLED
    err = display_init(); /*Display Initialization*/
    if (err != ESP_OK)
        init_fail("display", err);
#endif
```

Tento kód sa nachádza vo funkcii `init`, ktorá slúži na ukladanie inicializačných funkcií vyžadujúcich vyvolanie a hodnotenie ich výsledkov. Ak sa stav návratu odchyľuje od `ESP_OK`, kód vypíše chybovú správu a ukončí ďalšie vykonávanie.

/* Inicializácia obrazovky a funkcia zobrazenia `display_test` */

```
#ifndef CONFIG_BSP_DISPLAY_ENABLED
void display_test()
{
    if (lvgl_port_lock(0))
    {
        lv_obj_t *label = lv_label_create(lv_scr_act()); /*Create a label object*/
        lv_label_set_text(label, "Hello P4!"); /*Set a new text for a label. Memory will be allocated to store the text by the label.*/

        static lv_style_t label_style;
        lv_style_init(&label_style); /*Initialize a style*/
        lv_style_set_bg_opa(&label_style, LV_OPA_TRANSP); /*Set the style LVGL background color*/
        lv_obj_add_style(label, &label_style, LV_PART_MAIN); /*Add a style to an object*/
        lv_obj_center(label); /*Align an object to the center on its parent*/
        lv_obj_set_style_text_color(label, LV_COLOR_BLACK, LV_PART_MAIN); /*Set the style LVGL text color*/
        lv_obj_set_style_text_font(label, &lv_font_montserrat_42, LV_PART_MAIN); /*Set the style LVGL text font*/

        lv_obj_set_style_bg_color(lv_scr_act(), LV_COLOR_BLACK, LV_PART_MAIN); /*Set the screen's LVGL background color*/
        lv_obj_set_style_bg_opa(lv_scr_act(), LV_OPA_COVER, LV_PART_MAIN); /*Set the screen's LVGL background transparency*/
        lvgl_port_unlock();
    }
}
#endif
```

Táto funkcia primárne konfiguruje počiatočný obsah zobrazenia obrazovky: nastavuje farbu pozadia a zobrazenie textu prostredníctvom ovládacieho prvku `lvgl`.

Funkcia `lv_label_set_text` nastavuje text zobrazený na ovládacom prvku.

Funkcia `lv_obj_set_style_text_color` nastavuje farbu zobrazenia textu.

Funkcia `lv_obj_set_style_text_font` nastavuje veľkosť písma textu.

Funkcia `lv_obj_set_style_bg_color` nastavuje farbu pozadia.

Funkcia `lv_obj_set_style_bg_opa` nastavuje priehľadnosť pozadia.

Stojí za zmienku, že pri volaní funkcií `Lvgl` mimo vlákien `Lvgl` je potrebné získať vzájomný vylučný zámok. Funkcia `lvgl_port_lock` získava vzájomný vylučný zámok, zatiaľ čo funkcia `lvgl_port_unlock` ho uvoľňuje.

```

void app_main(void)
{
    MAIN_INFO("-----Demo version-----");
    Init();
#ifdef CONFIG_BSP_DISPLAY_ENABLED
    set_lcd_blight(100); /*Set the screen backlight to maximum brightness*/
    if (lvgl_port_lock(0)) /*Take LVGL mutex*/
    {
        display_test(); /*Set the screen's LVGL default display page*/
        lvgl_port_unlock(); /*Give LVGL mutex*/
    }
    while (1)
    {
        if (lvgl_port_lock(0))
        {
            lv_obj_set_style_bg_color(lv_scr_act(), LV_COLOR_RED, LV_PART_MAIN); /*Set the screen's LVGL background color*/
            lv_obj_set_style_bg_opa(lv_scr_act(), LV_OPA_COVER, LV_PART_MAIN); /*Set the screen's LVGL background transparency*/
            lvgl_port_unlock();
        }
        vTaskDelay(pdMS_TO_TICKS(2000));
        if (lvgl_port_lock(0))
        {
            lv_obj_set_style_bg_color(lv_scr_act(), LV_COLOR_YELLOW, LV_PART_MAIN);
            lv_obj_set_style_bg_opa(lv_scr_act(), LV_OPA_COVER, LV_PART_MAIN);
            lvgl_port_unlock();
        }
        vTaskDelay(pdMS_TO_TICKS(2000));
        if (lvgl_port_lock(0))
        {
            lv_obj_set_style_bg_color(lv_scr_act(), LV_COLOR_BLUE, LV_PART_MAIN);
            lv_obj_set_style_bg_opa(lv_scr_act(), LV_OPA_COVER, LV_PART_MAIN);
            lvgl_port_unlock();
        }
        vTaskDelay(pdMS_TO_TICKS(2000)); /*Delay 2000ms*/
    }
#endif
    MAIN_INFO("-----Start the test-----");
}

```

V rámci funkcie **app_main** sa najskôr nastaví **jas podsvietenia** na 100 % a potom **sa inicializuje** obsah zobrazenia na obrazovke. Následne sa vytvorí slučka, v rámci ktorej sa každé dve sekundy vykonáva funkcia **lvgl** na nastavenie farby pozadia. Farba pozadia sa cyklicky mení (v poradí červená-žltá-modrá). Tieto farby je možné podľa potreby modifikovať, pričom definície farieb sú špecifikované v hlavičkovom súbore **bsp_display.h**.

1.3.5 Súbor CMkaLists.txt

Aby bolo možné úspešne vyvolať obsah priečinka **bsp_display** v rámci hlavnej funkcie, je potrebné nakonfigurovať súbor **CMakeLists.txt**, ktorý sa nachádza v hlavnom priečinku. Podrobnosti konfigurácie sú nasledovné:

```

FILE(GLOB_RECURSE main ${CMAKE_SOURCE_DIR}/main/*.c)

idf_component_register(SRCS ${main}
                       INCLUDE_DIRS "include"
                       REQUIRES bsp_display)

```

Najskôr sa definujú adresáre pre zdrojové súbory a hlavičkové súbory spolu s požadovanou knižnicou ovládačov – konkrétne knižnicou ovládačov pre prepojenie **bsp_display**. Následne sa tieto nastavenia zaregistrujú v systéme zostavovania pomocou príkazu **idf_component_register**, čím sa hlavnej funkcii umožní využívať tieto schopnosti ovládačov.

1.4 Postup programovania

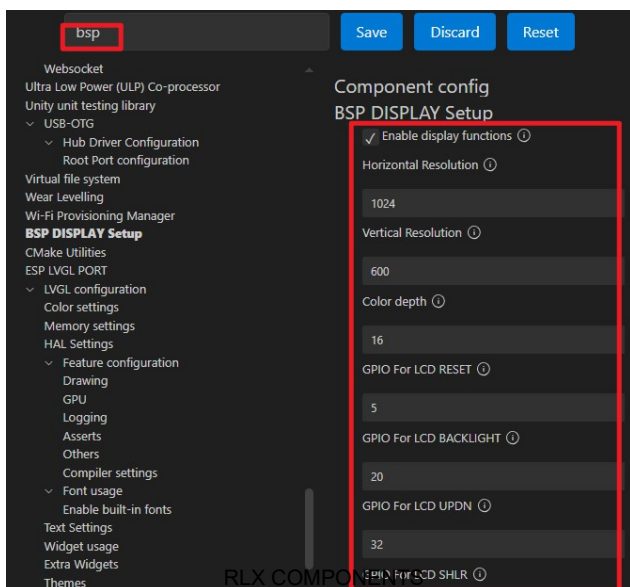
www.rlx.sk

Pripojte zariadenie P4 k počítaču prostredníctvom USB



1.4.1 Po klonovaní kódu prostredníctvom Git (odkaz bude potvrdený) vymažte všetky lokálne informácie o kompilácii. Nakonfigurujte prostredie IDF a model čipu pre kompiláciu podľa lekcie 1 a nastavte číslo sériového portu pre programovanie.

1.4.2 Nasledujúca konfigurácia SDKConfig je z veľkej časti identická s lekciou 1, stačí len prekonfigurovať piny DSI.

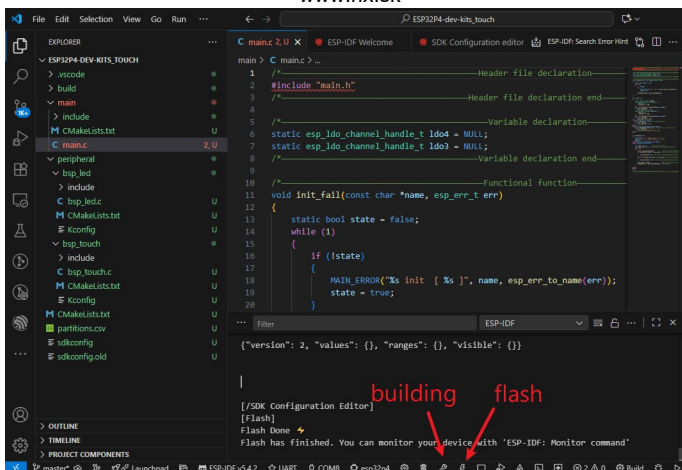


Save

Discard

Reset

1.4.3 Kliknite na Kompilovať. Po úspešnej kompilácii kliknite na Stiahnuť.
www.rlx.sk

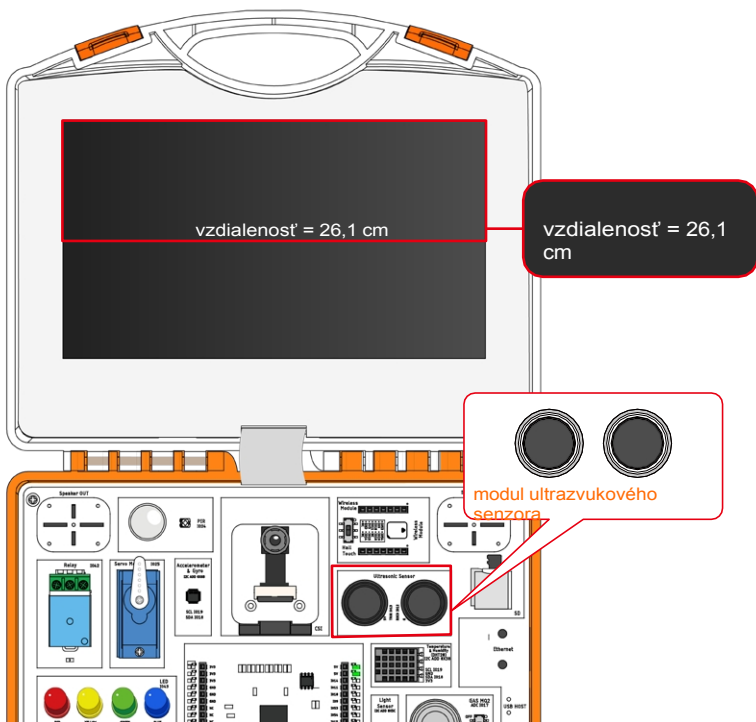


Lekcia 10 – Ultrazvukový displej vzdialenosti

Úvod

Táto kapitola tutoriálu predstavuje rozhranie medzi ESP32-P4 a ultrazvukovým senzorom, ktoré využíva rutinu merania vzdialenosti na lepšie pochopenie jeho základných funkcií. Ako bežný prípad použitia senzora, ultrazvukové meranie vzdialenosti poskytuje čitateľom intuitívne pochopenie toho, ako ESP32-P4 interaguje s periférnymi zariadeniami, čím vytvára základ pre zložitejšie projekty inteligentnej detekcie a riadenia.

RLX COMPONENTS



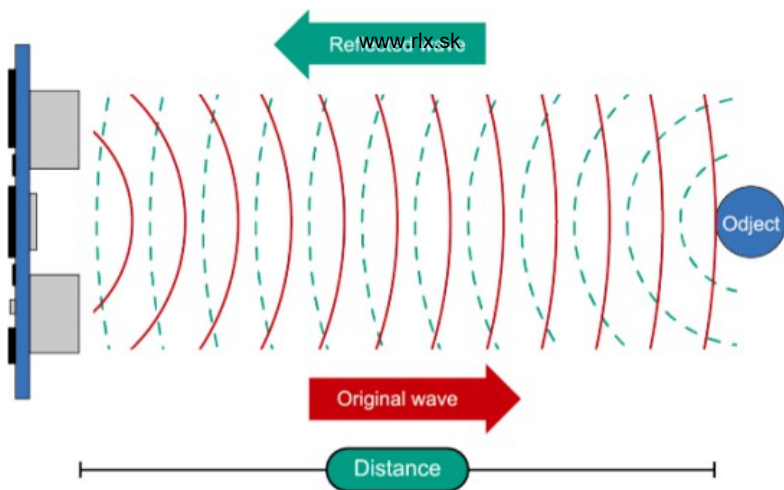
Táto kapitola je rozdelená do nasledujúcich podkapitol

- 1.1 Úvod do ultrazvukových senzorov
- 1.2 Návrh hardvéru
- 1.3 Návrh programu
- 1.4 Stiahnutie a overenie

1.1 Úvod do ultrazvukových senzorov

1.1.1 Princíp fungovania ultrazvukových senzorov

Ultrazvukové senzory sú zariadenia, ktoré využívajú princíp odrazu zvukových vln na bezkontaktné meranie vzdialenosti, ktoré sa bežne používajú v situáciách, ako je detekcia prekážok, meranie hladiny kvapalín a vyhýbanie sa prekážkam robotmi. Na príklade široko používaného modulu HC-SR04 sa meranie vykonáva prostredníctvom nasledujúcej sekvencie: vysielanie ultrazvukovej vlny → výpočet vzdialenosti.



Vysielanie: Keď ESP32-P4 aplikuje na spúšťací pin vysokú úroveň presahujúcu 10 μ s, senzor vysiela ultrazvukový impulz s frekvenciou 40 kHz.

Šírenie a odraz: Ultrazvuková vlna sa odráža späť pri stretnutí s prekážkou.

Prijímanie: Pin Echo senzora vysiela impulz s vysokou úrovňou, ktorého trvanie je úmerné času, za ktorý sa ultrazvuková vlna vráti späť.

Výpočet vzdialenosti: Vzdialenosť od cieľa sa vypočíta pomocou vzorca:

$$\text{Vzdialenosť (cm)} = \frac{\text{Čas } (\mu\text{s}) \times \text{Rýchlosť zvuku (340 m/s)}}{2 \times 10\,000} \quad \text{Vzdialenosť (cm)} = \frac{\text{Čas } (\mu\text{s}) \times \text{Rýchlosť zvuku (340 m/s)}}{2 \times 10\,000}$$

Napríklad, ak signál Echo na vysokej úrovni pretrváva 2 ms, vzdialenosť cieľa je približne 34 cm.

1.1.2 Popis pinov modulu

HC-SR04 má napríklad štyri piny: VCC: Napájacie napätie 5 V

(niektoré modely podporujú 3,3 V) GND: Uzemnenie

Spúšť (Trig): Vstupný pin spúšťa, vyžaduje vysokú úroveň 10 μ s

Echo: Výstupný terminál Echo, kde šírka vysokej úrovne je úmerná vzdialenosti

Upozornenie: GPIO modulu ESP32-14 pracuje na logických úrovniach 3,3 V, zatiaľ čo niektoré ultrazvukové moduly vysielajú Echo na 5 V. Preto je potrebná ochrana proti zníženiu napätia prostredníctvom deliča napätia alebo modulu na zmenu úrovne.

1.1.3 Faktory ovplyvňujúce meranie vzdialenosti

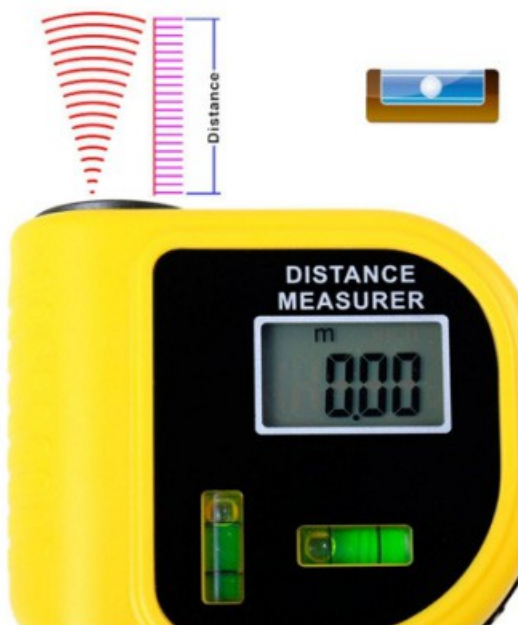
Okolité teplota: Rýchlosť zvuku sa mení s teplotou (približne 340 m/s pri 20 °C, zvyšuje sa približne o 0,6 m/s na každý 1 °C nárast).

Uhol merania: Uhol vyžarovania senzora je zvyčajne okolo 15°, čo vyžaduje, aby sa cieľ nachádzal v tomto rozsahu pre presné meranie vzdialenosti.

Materiál objektu: Mäkké alebo zvuk pohlcujúce materiály (ako napríklad tkanina alebo špongia) môžu spôsobiť útlm ozveny alebo zlyhanie merania.

Rozsah merania: Typický efektívny rozsah merania pre HC-SR04 je 2 cm až 400 cm.

Ultrazvuk nachádza široké uplatnenie v rôznych situáciách každodenného života.



Ultrazvukový diaľkomer

1.2 Hardvérový dizajn

www.rlx.sk

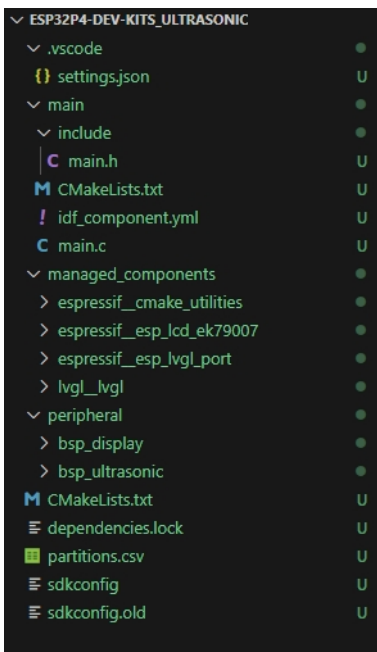
Nasledujúca tabuľka zobrazuje spôsob pripojenia pre ESP32-P4 a HC-SR04:

Ultrazvukový senzor	ESP32-P4	Špecifikácie
VCC	5	Napájanie
GND	GND	Spoločné uzemnenie
Trig	GPIO10	Výstup spúšťacieho signálu
Echo	GPIO11	Vstup signálu Echo (vyžaduje delenie napätia)

1.3 Analýza programu

<https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design>

Otvorte súbor projektu v VS Code podľa predchádzajúcich pokynov.



V rámci príkladového projektu **ESP32P4-dev-kits_ultrasonic** bola vytvorená nová zložka s názvom `bsp_ultrasonic` pod **ESP32P4-dev-kits_ultrasonic/peripheral**. V rámci cesty `bsp_ultrasonic` bol vytvorený nový priečinok `include`, súbor **CMakeLists.txt** a súbor **Kconfig**. Zložka `bsp_ultrasonic` obsahuje ovládač **bsp_ultrasonic.c**, zložka `include` obsahuje hlavičkový súbor **bsp_ultrasonic.h** a súbor **CMakeLists.txt** integruje ovládač do systému zostavovania, čím umožňuje projektu využívať funkcie ultrazvukového ovládača. Súbor **Kconfig** načíta celý ovládač a definície pinov GPIO do súboru **sdksconfig** v rámci platformy IDF (konfigurovateľné prostredníctvom grafického rozhrania).

1.3.1 Kód ultrazvukového ovládača

www.rlx.sk

Tu sa zameriame výlučne na základný kód; podrobný zdrojový kód nájdete v príslušných zdrojových súboroch pre tento experiment v materiáloch kódu.

Zdrojový kód ultrazvukového ovládača pozostáva z dvoch súborov: **bsp_ultrasonic.c** a **bsp_ultra-sonic.h**.

Nižšie najprv analyzujeme program **bsp_ultrasonic.h**: obsahuje relevantné definície pre ultrazvukové piny a deklarácie funkcií.

/* Odkazy na hlavičkové súbory */

```
/*-----Header file declaration-----*/
#include "freertos/FreeRTOS.h" //References for Freertos Function-related API Functions
#include "freertos/task.h" //References for Freertos Task Function-related API Functions
#include "esp_log.h" //References for LOG Printing Function-related API Functions
#include "esp_err.h" //References for Error Type Function-related API Functions
#include "esp_timer.h" //References for high-precision timers Function-related API Functions
#include "esp32p4/rom/ets_sys.h" //References for system timers Function-related API Functions
#include "driver/gpio.h" //References for GPIO Function-related API Functions
#include "driver/mcpwm_cap.h" //References for MCPWM capture timer Function-related API Functions
/*-----Header file declaration end-----*/
```

/* Deklarácie funkcií a deklarácie definícií makier */

```
#ifndef CONFIG_BSP_ULTRASONIC_ENABLED

#define ULTRASONIC_GPIO_ECHO CONFIG_ULTRASONIC_GPIO_ECHO // ULTRASONIC ECHO GPIO
#define ULTRASONIC_GPIO_TRIG CONFIG_ULTRASONIC_GPIO_TRIG // ULTRASONIC TRIG GPIO

typedef struct
{
    uint32_t ultrasonic_time; // Total Ultrasonic Transmission Time
    uint32_t ultrasonic_start_time; // Ultrasonic transmission start time
    uint32_t ultrasonic_stop_time; // Ultrasonic transmission end time
    float ultrasonic_distance; // Distance measured by ultrasonic
} ultrasonic_struct;

esp_err_t ultrasonic_init(); // Ultrasonic Initialization function
esp_err_t ultrasonic_callback_register(TaskHandle_t handle); // Ultrasonic task callback registration function
esp_err_t send_ultrasonic_start(); // Send the ultrasonic start signal function
float get_ultrasonic_distance(uint32_t time); // Convert the collected time data into distance function

#endif
```

Ďalej budeme analyzovať program **bsp_ultrasonic.c**: inicializácia a konfigurácia ultrazvukových pinov, volanie nastavení a vykonávanie funkcie spätného volania.

/* Funkcia inicializácie ultrazvuku ultrasonic_init */

```

esp_err_t ultrasonic_init()
{
    esp_err_t err = ESP_OK;
    mcpwm_capture_timer_handle_t cap_timer = NULL; /*Type of MCPWM capture timer handle*/
    mcpwm_capture_timer_config_t cap_conf = {
        .clk_src = MCPWM_CAPTURE_CLK_SRC_DEFAULT, /*MCPWM capture timer clock source*/
        .group_id = 0, /*Specify from which group to allocate the capture timer*/
    }; /*MCPWM capture timer configuration structure*/
    err = mcpwm_new_capture_timer(&cap_conf, &cap_timer); /*Create MCPWM capture timer*/
    if (err != ESP_OK)
        return err;
    mcpwm_capture_channel_config_t cap_ch_conf = {
        .gpio_num = ULTRASONIC_GPIO_ECHO, /*GPIO used capturing input signal*/
        .prescale = 1, /*Prescale of input signal, effective frequency = cap_input_clk/prescale*/
        .flags.neg_edge = true, /*Whether to capture on negative edge*/
        .flags.pos_edge = true, /*Whether to capture on positive edge*/
        .flags.pull_up = true, /*Whether to pull up internally*/
    }; /*MCPWM capture channel configuration structure*/
    err = mcpwm_new_capture_channel(cap_timer, &cap_ch_conf, &cap_chan); /*Create MCPWM capture channel*/
    if (err != ESP_OK)
        return err;
    const gpio_config_t gpio_cofig = {
        .pin_bit_mask = 1ULL << ULTRASONIC_GPIO_TRIG, /* GPIO pin: set with bit mask, each bit maps to a GPIO */
        .mode = GPIO_MODE_OUTPUT, /* GPIO mode: set input/output mode */
        .pull_up_en = false, /* GPIO pull-up */
        .pull_down_en = false, /* GPIO pull-down */
        .intr_type = GPIO_INTR_DISABLE, /* GPIO interrupt type */
    };
    err = gpio_config(&gpio_cofig); /*Configure GPIO*/
    if (err != ESP_OK)
        return err;
    err = mcpwm_capture_timer_enable(cap_timer); /*Enable MCPWM capture timer*/
    if (err != ESP_OK)
        return err;
    err = mcpwm_capture_timer_start(cap_timer); /*Start MCPWM capture timer*/
    if (err != ESP_OK)
        return err;
    ultrasonic_data.ultrasonic_start_time = 0;
    ultrasonic_data.ultrasonic_stop_time = 0;
    ultrasonic_data.ultrasonic_time = 0;
    return err;
}

```

V rámci funkcie **ultrasonic_init** sa najskôr nakonfigurujú členské premenné **štruktúry** **mcpwm_capture_timer_config_t** pomocou parametrov. Následne sa vyvolá funkcia **mcpwm_new_capture_timer**, aby sa vytvoril nový časovač mcpwm. Potom sa nakonfiguruje štruktúra **mcpwm_capture_channel_config_t**. Parameter **gpio_num** zodpovedá pinom GPIO pre vstupný signál časovača, parameter **prescale** zodpovedá koeficientu preddeľenia, **flags.neg_edge** označuje, či sa má zachytiť klesajúca hrana signálu, **flags.pos_edge** označuje, či sa má zachytiť stúpajúca hrana, a **flags.pull_up** označuje, či sa má povoliť interný pull-up. Po konfigurácii funkcia **mcpwm_new_capture_channel** zaregistruje nový kanál zachytávania časovača. Následne sa nakonfiguruje štruktúra **gpio_config_t** a funkcia **gpio_config** nastaví výstupný pin signálu. Nakoniec sa zachytávací časovač aktivuje pomocou **mcpwm_capture_timer_enable** a časovač zachytávania sa spustí pomocou funkcie **mcpwm_capture_timer_start**.

/* Funkcia registrácie spätného volania ultrazvuku **ultrasonic_callback_register** */

```

esp_err_t ultrasonic_callback_register(TaskHandle_t handle)
{
    esp_err_t err = ESP_OK;
    /*Group of supported MCPWM capture event callbacks*/
    mcpwm_capture_event_callbacks_t cbs = {
        .on_cap = hc_sr04_echo_callback, /*callback function that would be invoked when capture event occurred*/
    };
    err = mcpwm_capture_channel_register_event_callbacks(cap_chan, &cbs, handle); /*Set event callbacks for MCPWM capture channel*/
    if (err != ESP_OK)
        return err;
    err = mcpwm_capture_channel_enable(cap_chan); /*Enable MCPWM capture channel*/
    if (err != ESP_OK)
        return err;
    return err;
}

```

Táto funkcia prijíma jeden argument, ktorým je metóda na odovzdávanie výsledkov prostredníctvom oznámení úloh po získaní údajov v rámci funkcie spätného volania. Odovzdaný parameter je identifikátor vlákna **FreeRTOS**. Táto funkcia konfiguruje štruktúru `mcpwm_capture_event_callbacks_t`, zaregistruje funkciu spätného volania prostredníctvom `mcpwm_capture_channel_register_event_callbacks` a nakoniec aktivuje kanál časovača zachytávania volaním `mcpwm_capture_channel_enable`.

/* Funkcia vykonania spätného volania ultrazvukového echa `hc_sr04_echo_callback` */

```

static bool hc_sr04_echo_callback(mcpwm_cap_channel_handle_t cap_chan, const mcpwm_capture_event_data_t *edata, void *user_data)
{
    TaskHandle_t task_to_notify = (TaskHandle_t)user_data;
    BaseType_t high_task_wakeup = pdFALSE;
    if (edata->cap_edge == MCPWM_CAP_EDGE_POS)
    {
        ultrasonic_data.ultrasonic_start_time = edata->cap_value;
        ultrasonic_data.ultrasonic_stop_time = ultrasonic_data.ultrasonic_start_time;
    }
    else
    {
        ultrasonic_data.ultrasonic_stop_time = edata->cap_value;
        ultrasonic_data.ultrasonic_time = ultrasonic_data.ultrasonic_stop_time - ultrasonic_data.ultrasonic_start_time;
        xTaskNotifyFromISR(task_to_notify, ultrasonic_data.ultrasonic_time, eSetValueWithOverwrite, &high_task_wakeup);
    }
    return high_task_wakeup == pdTRUE;
}

```

Najskôr funkcia spätného volania priradí zachytený čas nábežnej hrany signálu ako čas začiatku ultrazvuku, potom priradí zachytený čas zostupnej hrany signálu ako čas ukončenia ultrazvuku. Odpočítaním týchto hodnôt sa získa celkový čas prechodu ultrazvuku, ktorý sa potom prenesie prostredníctvom oznámenia úlohy.

/* Funkcia `send_ultrasonic_start` na odoslanie signálu štartu pred prenosom ultrazvuku */

```

esp_err_t send_ultrasonic_start()
{
    esp_err_t err = ESP_OK;
    err = gpio_set_level(ULTRASONIC_GPIO_TRIG, 0); /*Set the Corresponding Output Level of GPIO*/
    if (err != ESP_OK)
        return err;
    ets_delay_us(20);
    err = gpio_set_level(ULTRASONIC_GPIO_TRIG, 1);
    if (err != ESP_OK)
        return err;
    ets_delay_us(10);
    err = gpio_set_level(ULTRASONIC_GPIO_TRIG, 0);
    if (err != ESP_OK)
        return err;
    return err;
}

```

Vyvolajte funkciu `gpio_set_level`, aby ste nastavili pin TRIG na výstup vlny **s nízkou úrovňou --- vysokou úrovňou --- nízkou úrovňou** s intervalom 10 μ s. Toto je časová požiadavka pre štartovací signál ultrazvukového senzora.

/* Funkcia `get_ultrasonic_distance` na prevod času na vzdialenosť v ultrazvuku */

```
float get_ultrasonic_distance(uint32_t time)
{
    static float last_ultrasonic_distance = 0;
    ultrasonic_data.ultrasonic_distance = time * 0.01715; /*Ultrasonic Conversion Formula*/
    if (ultrasonic_data.ultrasonic_distance != last_ultrasonic_distance) /*If the currently converted data is inconsistent with the previous data*/
    {
        last_ultrasonic_distance = ultrasonic_data.ultrasonic_distance;
        return ultrasonic_data.ultrasonic_distance;
    }
    last_ultrasonic_distance = ultrasonic_data.ultrasonic_distance;
    return -1; /*If the currently converted data is consistent with the previous data*/
}
#endif
```

Táto funkcia prevádza vstupnú hodnotu „čas“ meranú v mikrosekundách na vzdialenosť pomocou nasledujúceho vzorca: `ultrasonic_data.ultrasonic_distance = time * 0.01715`; Vzorec pre prevod je čas vynásobený hodnotou 0,01715. Konkrétny vzorec je nasledujúci:

Vzdialenosť = trvanie na vysokej úrovni $\times$ rýchlosť zvuku (340 m/s) $\div$ 2. Jednotku rýchlosti zvuku možno previesť takto: 340 m/s = 0,0343 cm/ μ s. Vydelením 0,0343 cm/ μ s číslom 2 dostaneme 0,01715, z čoho vyplýva prevodný vzorec na čas $\times$ 0,01715.

Stojí za zmienku, že po prevode vzdialenosti vykonáme kontrolu: ak sa novo vypočítaná vzdialenosť zhoduje s predchádzajúcou, vrátime -1.

1.3.2 Súbor Kconfig

Hlavnou funkciou tohto súboru je začleniť požadované konfigurácie do súboru `sdkconfig`, čo umožňuje vykonávať určité úpravy parametrov prostredníctvom grafického rozhrania. Tu 12 zodpovedá `GPIO_NUM_12` a 13 zodpovedá `GPIO_NUM_13`.

```
menu "BSP_ULTRASONIC Setup"
    config BSP_ULTRASONIC_ENABLED
        bool "Enable ULTRASONIC config"
        default n

    if BSP_ULTRASONIC_ENABLED
        config ULTRASONIC_GPIO_ECHO
            int "GPIO For ULTRASONIC ECHO"
            default 12

        config ULTRASONIC_GPIO_TRIG
            int "GPIO For ULTRASONIC TRIG"
            default 13
    endif
endmenu
```

1.3.3 Súbor `CMkaLists.txt`

Funkčnosť tejto ukážkovej rutiny závisí predovšetkým od ovládača **bsp_ultrasonic**. Aby bolo možné úspešne vyvolať obsah priečinka **bsp_ultrasonic** v hlavnej funkcii, je potrebné nakonfigurovať súbor **CMakeLists.txt**, ktorý sa nachádza v priečinku **bsp_ultrasonic**. Podrobnosti konfigurácie sú nasledovné:

```
FILE(GLOB_RECURSE component_sources "*.c")
                                www.rlx.sk
idf_component_register(SRCS ${component_sources}
                        INCLUDE_DIRS "include"
                        REQUIRES driver esp_timer esp_rom)
```

V tomto súbore **CMakeLists.txt** sú najskôr definované adresáre pre zdrojové súbory a hlavičkové súbory spolu s požadovanými knižnicami ovládačov (**esp_timer** pre časovač zachytávania a **esp_rom** pre časovanie na úrovni mikrosekúnd). Následne sa tieto nastavenia zaregistrujú v systéme zostavovania pomocou príkazu **idf_component_register**, čím sa projektu umožní **využívať** funkčnosť ovládača **bsp_ultrasonic**.

1.3.4 Hlavný priečinok

Hlavný priečinok slúži ako hlavný adresár pre vykonávanie programu a obsahuje hlavnú funkciu **main.c** a hlavičkový súbor **main.h** v priečinku **include**. Pridajte hlavný priečinok do súboru **CMakeLists.txt** systému zostavenia.

Súbor **main.h** odkazuje predovšetkým na požadované hlavičkové súbory: funkcie využívajúce ovládač **bsp_display** vyžadujú zaradenie hlavičkového súboru **bsp_display**, zatiaľ čo funkcie využívajúce ovládač **bsp_ultrasonic** vyžadujú hlavičkový súbor **bsp_ultrasonic**.

Nižšie je analýza programu **main.c**: inicializácia systému spolu s vykonávaním funkcie zobrazenia a ultrazvukových schopností.

```
#ifdef CONFIG_BSP_DISPLAY_ENABLED
    err = display_init(); /*Display Initialization*/
    if (err != ESP_OK)
        init_fail("display", err);
#endif
#ifdef CONFIG_BSP_ULTRASONIC_ENABLED
    err = ultrasonic_init(); /*ultrasonic Initialization*/
    if (err != ESP_OK)
        init_fail("ultrasonic", err);
#endif
```

Tento kód sa nachádza vo funkcii **init**, ktorá slúži na ukladanie inicializačných funkcií vyžadujúcich vyvolanie a hodnotenie výsledku takejto inicializácie. Ak stav návratu nie je **ESP_OK**, kód vypíše chybovú správu a ukončí ďalšie vykonávanie.

/* Inicializácia obrazovky a funkcia zobrazenia **ultrasonic_display** */

www.rlx.sk

```
void ultrasonic_display()
{
    if (lvgl_port_lock(0))
    {
        distance_label = lv_label_create(lv_scr_act()); /*Create a label object*/
        static lv_style_t label_style;
        lv_style_init(&label_style);
        lv_style_set_bg_opa(&label_style, LV_OPA_TRANSP);
        lv_obj_add_style(distance_label, &label_style, LV_PART_MAIN);
        lv_obj_set_style_text_color(distance_label, LV_COLOR_WHITE, LV_PART_MAIN);
        lv_obj_set_style_text_font(distance_label, &lv_font_montserrat_30, LV_PART_MAIN);
        lv_obj_center(distance_label);
        lv_obj_set_style_bg_color(lv_scr_act(), LV_COLOR_BLACK, LV_PART_MAIN);
        lv_obj_set_style_bg_opa(lv_scr_act(), LV_OPA_COVER, LV_PART_MAIN);
        lv_label_set_text(distance_label, "distance = 0.0 cm");
        lvgl_port_unlock();
    }
}
```

Táto funkcia primárne konfiguruje počiatočný obsah zobrazenia obrazovky: nastavenie farieb pozadia a zobrazenia textu prostredníctvom ovládacích prvkov lvgl.

Funkcia **lv_label_set_text** nastavuje text zobrazený na ovládacom prvku.

Funkcia **lv_style_set_bg_opa** nastavuje farbu pozadia štýlu.

Funkcia **lv_obj_set_style_text_color** nastavuje farbu zobrazenia textu. Funkcia **lv_obj_set_style_text_font** nastavuje veľkosť písma textu.

Funkcia **lv_obj_set_style_bg_color** nastavuje farbu pozadia.

Funkcia **lv_obj_set_style_bg_opa** nastavuje priehľadnosť pozadia.

Poznámka: Pri volaní funkcií lvgl mimo funkcií vlákna lvgl je potrebné získať mutex zámok. Funkcia **lvgl_port_lock** získava mutex zámok, funkcia **lvgl_port_unlock** ho uvoľňuje.

/* Funkcia aktualizácie údajov na obrazovke **update_distance_value** */

```
void update_distance_value(float new_distance_cm)
{
    if (distance_label)
    {
        char buffer[32];
        snprintf(buffer, sizeof(buffer), "distance = %.1f cm", new_distance_cm); /*Format the data into a string*/
        lv_label_set_text(distance_label, buffer); /*Set a new text for a label*/
    }
}
```

Táto funkcia používa funkciu **snprintf** na formátovanie získaných údajov typu float do reťazca a následne obnovuje zobrazený obsah prostredníctvom funkcie **lv_label_set_text**.

Stojí za zmienku, že funkcia **snprintf** pripája na koniec formátovaného reťazca ukončovaci znak. Okrem toho funkcia **lv_label_set_text** rozpoznáva reťazce identifikovaním tohto ukončovacieho znaku. V dôsledku toho je použitie funkcie **snprintf** na formátovanie reťazcov preferovaným prístupom.

```
void app_main(void)
{
    MAIN_INFO("-----Demo version-----");
    MAIN_INFO("-----Start the test-----");
    Init();
    #ifdef CONFIG_BSP_ULTRASONIC_ENABLED
    #ifdef CONFIG_BSP_DISPLAY_ENABLED
        set_lcd_blight(100); /*Set the screen backlight to maximum brightness*/
        ultrasonic_display(); /*Set the screen's LVGL default display page*/
        xtaskcreate(ultrasonic_task, 4096, NULL, configMAX_PRIORITIES - 5, &ultrasonic); /*Create an ultrasonic data display refresh thread*/
    #endif
    #endif
}
```

RLX COMPONENTS

V rámci funkcie **app_main** sa najskôr nastaví jas podsvietenia na 100 % a následne sa inicializuje obsah zobrazenia na obrazovke. Následne sa vytvorí vlákno **FreeRTOS** na spracovanie údajov a obnovovanie obrazovky.

/* Vlákno na spracovanie ultrazvukových údajov a obnovovanie obrazovky: **ultrasonic_task** */

```
void ultrasonic_task(void *param)
{
    static esp_err_t err = ESP_OK;
    uint32_t time = 0;
    float distance = 0;
    err = ultrasonic_callback_register(ultrasonic); /*Register the ultrasonic timing capture callback function (Parameter: Task Handle)*/
    if (err != ESP_OK)
    {
        MAIN_INFO("ultrasonic callback register fail");
    }
    while (1)
    {
        err = send_ultrasonic_start(); /*Send the ultrasonic start signal*/
        if (err != ESP_OK)
        {
            continue;
            MAIN_INFO("send ultrasonic start fail");
        }
        if (xTaskNotifyWait(0x000, ULONG_MAX, &time, pdMS_TO_TICKS(1000)) == pdTRUE) /*Wait for the task notification and obtain the transmitted data value*/
        {
            float pulse_width_us = time * (1000000.0 / esp_clk_apb_freq()); /*'tis = 1000000us*/
            if (pulse_width_us > 35000) /*If the measured time data exceeds 35 seconds, it is out-of-range data*/
            {
                MAIN_INFO("the distance exceeds the limit");
                MAIN_INFO("the pulse_width_us -%f", pulse_width_us);
                if (lvgl_port_lock(0))
                {
                    lv_label_set_text(distance_label, "the distance exceeds the limit"); /*Set a new text for a label*/
                    lvgl_port_unlock();
                }
            }
            else
            {
                distance = get_ultrasonic_distance(pulse_width_us); /*Convert the collected time data into distance*/
                if (distance == -1) /*If the value is -1, the distance data is consistent with the previous one*/
                {
                    MAIN_INFO("not new distance");
                }
                else
                {
                    MAIN_INFO("distance-%.2f cm", distance);
                    if (lvgl_port_lock(0))
                    {
                        update_distance_value(distance); /*Update the distance data displayed on the screen*/
                        lvgl_port_unlock();
                    }
                }
            }
        }
        vTaskDelay(1000 / portTICK_PERIOD_MS);
    }
}
```

V rámci vlákna na spracovanie ultrazvukových údajov a obnovovanie zobrazenia obrazovky sa najskôr inicializujú premenné a vyvolá sa **ultrasonic_callback_register** na registráciu funkcie spätného volania odovzdaním aktuálnej rukoväte vlákna. Následne sa vytvorí cyklus while. V rámci tejto slučky sa vyvolá funkcia na odoslanie ultrazvukového štartovacieho signálu. Ak sa to nepodarí, slučka sa opäť spustí prostredníctvom continue. Po úspešnom vykonaní vlákno čaká na oznámenie o prijatí úlohy. V rámci tohto oznámenia sa prijímú prenesené časové údaje a prevedú sa na jednotky mikrosekúnd. Systém potom posúdi, či uplynutý čas presahuje maximálnu hodnotu ultrazvukového merania 35 sekúnd. Ak je prekročená, vyvolá sa funkcia lvgl na ovládanie obnovenia zobrazenia, ktoré zobrazí reťazec „vzdialenosť presahuje limit“. Ak limit nie je prekročený, vyvolá sa funkcia **get_ultrasonic_distance** na prevod údajov na hodnoty vzdialenosti. Skontroluje sa návratová hodnota funkcie: ak je -1, údaje zodpovedajú predchádzajúcemu odčítaniu a zobrazenie sa neaktualizuje. Ak nie je -1, vyvolá sa funkcia **update_distance_value(distance)**, aby sa obnovilo zobrazenie s najnovšími údajmi. Konečné 1-sekundové oneskorenie predstavuje interval, v ktorom sa vykonáva ultrazvukové meranie vzdialenosti.

RLX COMPONENTS

1.3.4 Súbor CMkaLists.txt

Aby bolo možné úspešne vyvolať obsah priečinkov **bsp_display** a **bsp_ultrasonic** v hlavnej funkcii, je potrebné nakonfigurovať súbor **CMakeLists.txt**, ktorý sa nachádza v hlavnom priečinku. Podrobnosti konfigurácie sú nasledovné:

```
FILE(GLOB_RECURSE main ${CMAKE_SOURCE_DIR}/main/*.c)

idf_component_register(SRCS ${main}
                        INCLUDE_DIRS "include"
                        REQUIRES bsp_display bsp_ultrasonic)
```

Najskôr sa definujú adresáre pre zdrojové súbory a hlavičkové súbory spolu s požadovanými knižnicami ovládačov, **konkrétne** knižnicami ovládačov pre prepojenie **bsp_display** a **bsp_ultrasonic**. Následne sa tieto nastavenia zaregistrujú v systéme zostavovania pomocou príkazu **idf_component_register**, čím sa hlavnej funkcii umožní využívať tieto funkcie ovládačov.

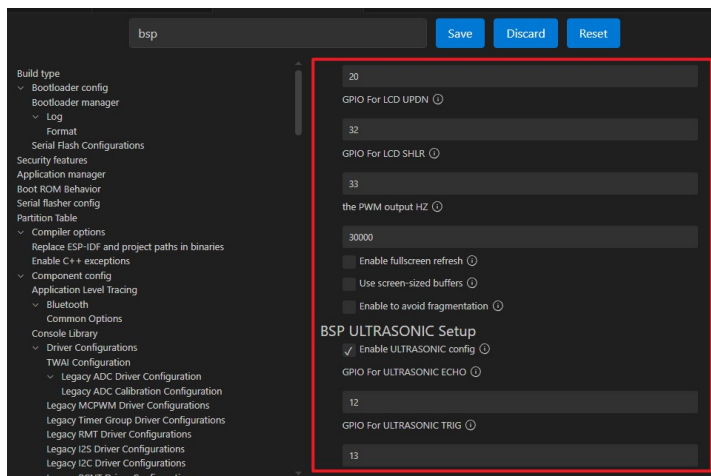
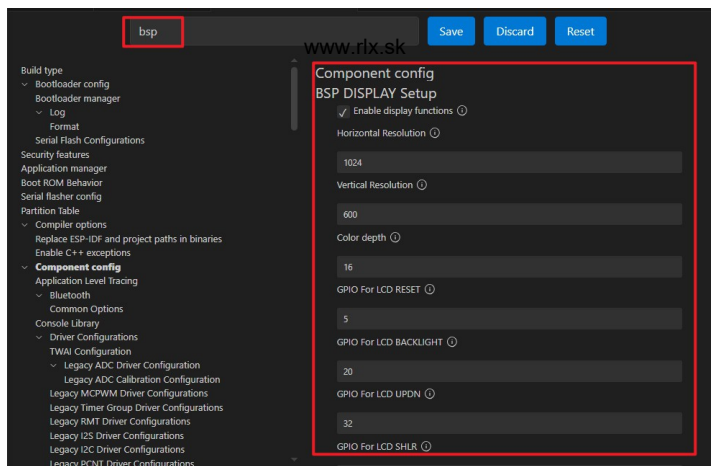
1.4 Postup programovania

Pripojte zariadenie P4 k počítaču prostredníctvom USB

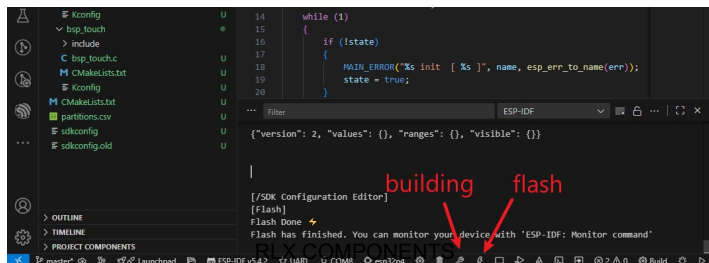


1.4.1 Po klonovaní kódu prostredníctvom Git (odkaz bude potvrdený) vymažte všetky lokálne informácie o kompilácii. Nakonfigurujte prostredie IDF a model čipu pre kompiláciu podľa lekcii 1 a nastavte číslo sériového portu pre programovanie.

1.4.2 Nasledujúca konfigurácia SDKConfig je z veľkej časti identická s lekciami 1, stačí prekonfigurovať obrazovku a ultrazvukové piny.



1.4.3 Kliknite na Compile (Kompilovat). Po úspešnej kompilácii kliknite na Download (Stiahnuť).

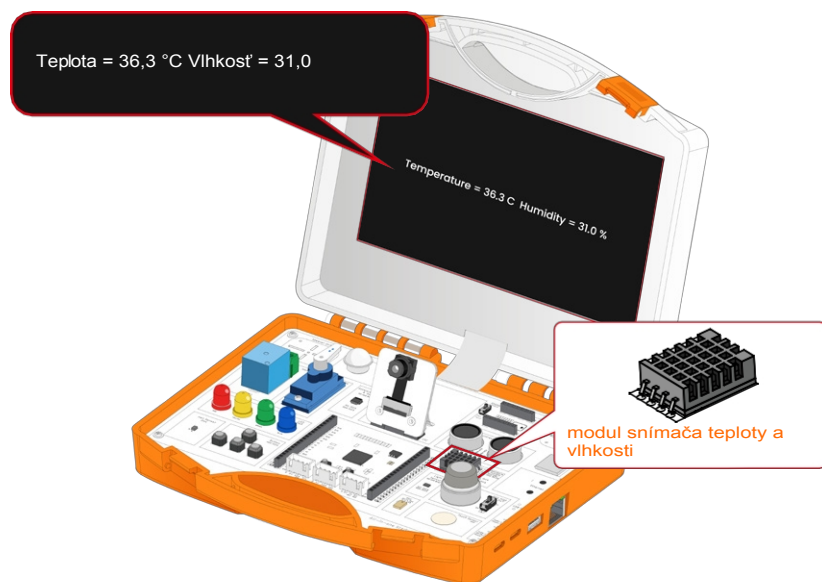


Lekcia 11 – DHT20 Teplota a vlhkosť

Úvod

Táto kapitola tutoriálu predstavuje rozhranie medzi ESP32-P4 a digitálnym senzorom teploty a vlhkosti DHT20. Prostredníctvom príkladov rutín na čítanie údajov o teplote a vlhkosti pomáha pochopiť, ako využívať digitálne senzory. Ako bežný prípad monitorovania prostredia poskytuje získavanie údajov o teplote a vlhkosti čitateľom intuitívne pochopenie interakcie medzi ESP32-P4 a periférnymi zariadeniami, čím sa vytvárajú základy pre následné komplexné projekty IoT a inteligentných domácností.

Demonštrácia projektu Efekt



Táto kapitola je rozdelená do nasledujúcich podkapitol

- 1.1 Úvod do teplotného a vlhkosťného senzora DHT20
- 1.2 Konštrukcia hardvéru
- 1.3 Návrh programu
- 1.4 Stiahnutie a overenie

1.1 Úvod do teplotného a vlhkosťného senzora DHT20

www.rlx.sk

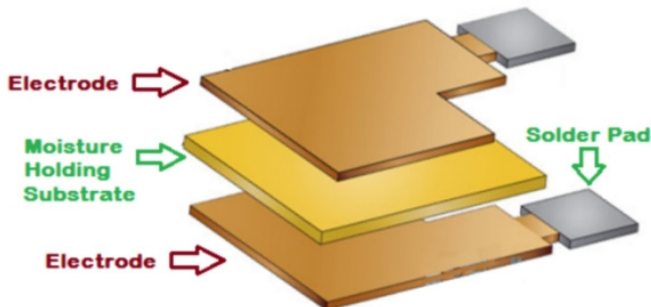
1.1.1 Úvod k DHT20

DHT20 je digitálny senzor teploty a vlhkosti, ktorý obsahuje integrovaný kapacitný senzor vlhkosti a prvok na meranie teploty, ako aj vstavaný 12-bitový ADC a obvod na spracovanie digitálneho signálu. Na rozdiel od bežných analógových senzorov komunikuje DHT20 s hostiteľským mikrokontrolérom prostredníctvom zbernice I²C, čo umožňuje priamy výstup kalibrovaných a kompenzovaných údajov o teplote a vlhkosti.

Medzi jeho kľúčové vlastnosti patria:

- ① Digitálny výstup: Komunikácia I²C eliminuje problémy s šumom a driftom spojené s analógovým snímaním;
- ② Vysoká presnosť: Typická presnosť vlhkosti $\pm 3\%$ RH, presnosť teploty $\pm 0,5\text{ }^{\circ}\text{C}$;
- ③ Nízka spotreba energie: Typický prevádzkový prúd $< 1\text{ mA}$, vhodný pre zariadenia napájané batériami;
- ④ Kalibračná kompenzácia: Kalibrované vo výrobe pre okamžité meranie po zapnutí, nevyžaduje žiadnu dodatočnú kalibráciu;
- ⑤ Rýchla odozva: Typická doba odozvy na vlhkosť menej ako 10 sekúnd.

1.1.2 Princíp činnosti



Vnútročné komponenty DHT20:

Kapacitný senzor vlhkosti: Obsahuje polymérovú fóliu citlivú na vlhkosť a elektródy. Keď sa mení vlhkosť vzduchu, mení sa dielektrická konštanta fólie, čo spôsobuje zodpovedajúcu zmenu kapacity.

Snímač teploty: Využíva vysoko presný prvok citlivý na teplotu (napr. kremíkový snímač s kompenzáciou teplotného posunu).

Obvod na spracovanie signálu: Prenáša signály teploty a vlhkosti do analógovo-digitálneho prevodníka (ADC), ktorý prostredníctvom interných kompenzačných algoritmov vydáva štandardizované digitálne hodnoty.

RLX COMPONENTS

① Princíp merania vlhkosti

Kapacitný senzor vysiela kapacitný signál, ktorý sa mení v závislosti od vlhkosti. Ten je prevádzaný na digitálnu hodnotu pomocou ADC a následne pomocou kalibračnej krivky prepočítaný na relatívnu vlhkosť (RH%).

② Princíp merania teploty

Odpor alebo napätie prvku citlivého na teplotu sa mení v závislosti od teploty. Po prevode ADC sa získa digitálna hodnota teploty (v stupňoch Celzia).

③ Vzorec na výpočet údajov

Podľa dátového listu DHT20 sú čítané surové údaje 20-bitovým binárnym číslom, ktoré sa musí previesť na skutočnú fyzikálnu veličinu:

$$RH(\%) = \frac{S_{RH}}{2^{20}} \times 100$$

$$T(^{\circ}\text{C}) = \frac{S_T}{2^{20}} \times 200 - 50$$

1.1.3 Popis pinov

Modul DHT20 má zvyčajne 4-pinové rozhranie:

VCC: Napájacie napätie 2,0 V – 5,5 V (napájanie 3,3 V vývojovej dosky ESP32-P4 môže poskytovať priame napájanie)

GND: Zem SDA:

Dátová linka I²C

SCL: Hodinová

linka I²C

Predvolená adresa I²C je 0x38, podporujúca štandardný režim (100 kHz) aj rýchly režim (400 kHz).

1.1.4 Aplikácie a ovplyvňujúce faktory

Aplikačné scenáre: Široko používané v inteligentných domácnostiach, monitorovaní životného prostredia, meteorologických staniách, sledovaní skladov a podobných kontextoch.

Vplyvné faktory:

Rýchle kolísanie teploty môže spôsobiť oneskorenie merania vlhkosti.

Pri dlhodobej prevádzke v prostredí s vysokou vlhkosťou je potrebné venovať pozornosť problémom s nasýtením senzora.

Rýchlosť prúdenia vzduchu ovplyvňuje reakčný čas senzora.

RLX COMPONENTS



1.2 Hardvérový dizajn

ESP32-P4 je pripojený k DHT20 prostredníctvom zbernice I²C. Konfigurácia zapojenia je nasledovná:

ESP32-P4 GPIOxx (SDA) → DHT20 SDA

ESP32-P4 GPIOyy (SCL) → DHT20 SCL

3,3 V → VCC

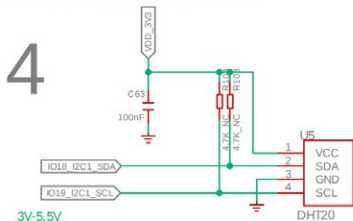
GND → GND

(Konkrétne piny pre SDA a SCL je možné vybrať podľa skutočných definícií pinov vývojovej dosky a je potrebné pridať pull-up rezistory s hodnotou 4,7 kΩ až 10 kΩ.)

Napájacie napätie je 3,3 V, s typickým prevádzkovým prúdom 0,5 mA, a môže byť priamo napájané vývojovou doskou ESP32-P4.

Temperature & Humidity(DHT20)

I2C ADDRESS:0X38



Schéma

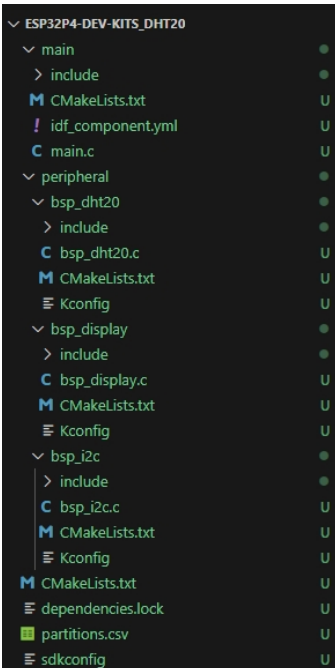
RLX COMPONENTS

1.3 Analýza programu

www.rlx.sk

<https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design>

Otvorte súbor projektu v VS Code podľa predchádzajúcich pokynov.



V rámci príkladu **ESP32P4-dev-kits_dht20** boli v adresári **ESP32P4-dev-kits_dht20\peripheral** vytvorené nové priečinky s názvami **bsp_i2c** a **bsp_dht20**. V rámci ciest **bsp_dht20** a **bsp_i2c** boli vytvorené nové priečinky **include**, súbory **CMakeLists.txt** a súbory **Kconfig**. Priečinok **bsp_i2c** obsahuje súbor ovládača **bsp_i2c.c**, zatiaľ čo priečinok **bsp_dht20** obsahuje súbor ovládača **bsp_dht20.c**. Príslušné priečinky obsahujú súbory hlavičiek **.h**, zatiaľ čo súbor **CMakeLists.txt** integruje ovládače do systému zostavovania, čím umožňuje projektové využitie ich funkcionality. Súbor **Kconfig** medzitým načíta celý ovládač spolu s definíciami pinov GPIO do súboru **SDKConfig** v rámci platformy IDF (konfigurovateľný prostredníctvom grafického rozhrania).

1.3.1 Kód ovládača I2C

Tu sa zameriame výlučne na základný kód; podrobný zdrojový kód nájdete v príslušných zdrojových súboroch pre tento experiment v materiáloch kódu.

Zdrojový kód ovládača I2C pozostáva z dvoch súborov: **bsp_i2c.c** a **bsp_i2c.h**.

Nižšie najprv analyzujeme program **bsp_i2c.h**: obsahuje relevantné definície pre piny I2C a deklarácie funkcií.

/* Odkazy na súbory hlavičiek */

```
/*-----Header file declaration-----*/
#include "esp_log.h"           //References for LOG Printing Function-related API Functions
#include "esp_err.h"          //References for Error Type Function-related API Functions
#include "driver/i2c_master.h" //References for I2C Master Function-related API Functions
/*-----Header file declaration end-----*/
```

/* Deklarácie funkcií a deklarácie definícií makriér */

www.rlx.sk

```
#ifndef CONFIG_I2C_ENABLED
#define I2C_MASTER_PORT CONFIG_I2C_PORT_NUM // I2C port number
#define I2C_SCL_SCL CONFIG_I2C_SCL // I2C SCL GPIO
#define I2C_GPIO_SDA CONFIG_I2C_GPIO_SDA // I2C SDA GPIO

esp_err_t i2c_init(void); // Initialize I2C
i2c_master_dev_handle_t i2c_dev_register(uint16_t dev_device_address); // The I2C slave device registration function
esp_err_t i2c_read(i2c_master_dev_handle_t i2c_dev, uint8_t *read_buffer, size_t read_size); // I2C Read Function
esp_err_t i2c_write(i2c_master_dev_handle_t i2c_dev, uint8_t *write_buffer, size_t write_size); // I2C Write Function
esp_err_t i2c_write_read(i2c_master_dev_handle_t i2c_dev, uint8_t read_reg, uint8_t *read_buffer, size_t read_size, uint16_t delays); // I2C Write and Read Function
esp_err_t i2c_read_reg(i2c_master_dev_handle_t i2c_dev, uint8_t reg_addr, uint8_t *read_buffer, size_t read_size); // I2C Read Register Function
esp_err_t i2c_write_reg(i2c_master_dev_handle_t i2c_dev, uint8_t reg_addr, uint8_t data); // I2C Write Register Function
#endif
```

Ďalej budeme analyzovať program **bsp_i2c.c**: inicializácia a konfigurácia pinov I2C a sprístupnenie funkcií rozhrania API.

/* Inicializačná funkcia I2C i2c_init */

```
#ifndef CONFIG_I2C_ENABLED
esp_err_t i2c_init(void)
{
    static esp_err_t err = ESP_OK;
    /*I2C master bus specific configurations*/
    i2c_master_bus_config_t conf = {
        .i2c_port = I2C_MASTER_PORT, // I2C port number*/
        .sda_io_num = I2C_GPIO_SDA, // GPIO number of I2C SDA signal*/
        .scl_io_num = I2C_GPIO_SCL, //GPIO number of I2C SCL signal*/
        .clk_source = I2C_CLK_SRC_DEFAULT, //Clock source of I2C master-bus*/
        .glitch_ignore_cnt = 7, // If the glitch period on the line is less than this value, it can be filtered out, typically value is 7*/
    };
    #ifdef CONFIG_I2C_GPIO_PULLUP
        .flags.enable_internal_pullup = true, /*Enable internal pullups*/
    #else
        .flags.enable_internal_pullup = false,
    #endif
    };
    err = i2c_new_master_bus(&conf, &i2c_bus_handle); /*Allocate an I2C master bus*/
    if (err != ESP_OK)
        return err;
    return err;
}
```

V rámci funkcie **i2c_init** sa najskôr nakonfigurujú členské premenné štruktúry **i2c_master_bus_config_t** pomocou parametrov. Následne sa vyvolá funkcia **i2c_new_master_bus**, aby sa vytvoril nový radič zbernice I2C. Parametre pre členské premenné štruktúry **i2c_master_bus_config_t** sú nasledovné:

i2c_port: Výber portu radiča zbernice I2C

sda_io_num: Dátová linka SDA zbernice I2C

scl_io_num: Hodinová linka SCL zbernice I2C

clk_source: Výber zdroja hodín zbernice I2C

glitch_ignore_cnt: Poruchy kratšie ako táto dĺžka sa ignorujú; zvyčajne nastavené na 7

flags.enable_internal_pull: Povoliť interné pull-up rezistory

/* Funkcia registrácie zariadenia I2C slave i2c_dev_register */

```
i2c_master_dev_handle_t i2c_dev_register(uint16_t dev_device_address)
{
    esp_err_t err = ESP_OK;
    i2c_master_dev_handle_t dev_handle = NULL; /*Type of I2C master bus device handle*/
    i2c_device_config_t cfg = {
        .dev_addr_length = I2C_ADDR_BIT_LEN_7, /*Select the address length of the slave device*/
        .device_address = dev_device_address, /*I2C device raw address. (The 7/10 bit address without read/write bit)*/
        .scl_speed_hz = 400000, //I2C SCL line frequency*/
    };
    err = i2c_new_master_bus_add_device(i2c_bus_handle, &cfg, &dev_handle); /*Add I2C master BUS device*/
    if (err == ESP_OK)
        return dev_handle;
    return 0;
}
```

Táto funkcia má jeden parameter: 7-bitovú adresu podriadeného zariadenia I2C, ktoré má byť

RLX COMPONENTS

zaregistrovaná. Pomocou konfiguračnej štruktúry adresy zariadenia je adresa viazaná na podriadené zariadenie prostredníctvom funkcie `i2c_master_bus_add_device`, ktorá vráti identifikátor zariadenia (použiteľný pre následné operácie čítania, zápisu atď.).

/* Funkcia čítania I2C `i2c_read` */`

```
esp_err_t i2c_read(i2c_master_dev_handle_t i2c_dev, uint8_t *read_buffer, size_t read_size)
{
    return i2c_master_receive(i2c_dev, read_buffer, read_size, 1000); /*Perform a read transaction on the I2C bus*/
}
```

Operácia čítania I2C: zadajte identifikátor zariadenia I2C, prečítajte prijímací buffer a určite množstvo čítania

/* Funkcia zápisu I2C `i2c_write` */`

```
esp_err_t i2c_write(i2c_master_dev_handle_t i2c_dev, uint8_t *write_buffer, size_t write_size)
{
    return i2c_master_transmit(i2c_dev, write_buffer, write_size, 1000); /*Perform a write transaction on the I2C bus*/
}
```

Operácia zápisu I2C: zadajte identifikátor zariadenia I2C, zapíšete pole a počet bajtov, ktoré sa majú zapísať

/* Funkcia čítania registra I2C `i2c_read_reg` */`

```
esp_err_t i2c_read_reg(i2c_master_dev_handle_t i2c_dev, uint8_t reg_addr, uint8_t *read_buffer, size_t read_size)
{
    return i2c_master_transmit_receive(i2c_dev, &reg_addr, 1, read_buffer, read_size, 1000); /*Perform a write-read transaction on the I2C bus*/
}
```

Operácia čítania registra I2C: zadajte identifikátor zariadenia I2C, adresu registra, vyrovnávaciu pamäť čítania a počet čítaní.

/* Funkcia zápisu do registra I2C `i2c_write_reg` */`

```
esp_err_t i2c_write_reg(i2c_master_dev_handle_t i2c_dev, uint8_t reg_addr, uint8_t data)
{
    uint8_t write_buf[2] = {reg_addr, data}; /*Register address and data*/
    return i2c_master_transmit(i2c_dev, write_buf, sizeof(write_buf), 1000); /*Perform a write transaction on the I2C bus*/
}
```

Operácia zápisu do registra I2C: zadajte identifikátor zariadenia I2C, určite adresu registra a zapíšete údaje (jednu hodnotu).

1.3.2 Súbor Kconfig

Hlavnou funkciou tohto súboru je začleniť požadovanú konfiguráciu do súboru `sdkconfig`, čím sa umožní modifikácia určitých nastavení parametrov prostredníctvom grafického rozhrania. Tu 18 zodpovedá

GPIO_NUM_18

a 19 zodpovedá **GPIO_NUM_19**.

```
menu "BSP I2C Setup"
config BSP_I2C_ENABLED
    bool "Enable I2C functions"

if BSP_I2C_ENABLED
    config I2C_PORT_NUM
        int "I2C Master Port"
        default 0

    config I2C_GPIO_SCL
        int "GPIO For I2C SCL"
        default 19

    config I2C_GPIO_SDA
        int "GPIO For I2C SDA"
        default 18

    config I2C_GPIO_PULLUP
        bool "Enable I2C Pullup resistor"

endif
endmenu
```

RLX COMPONENTS

1.3.3 Súbor CMkaLists.txt

www.rlx.sk

Funkčnosť tejto ukážkovej rutiny závisí predovšetkým od ovládača **bsp_i2c**. Aby bolo možné úspešne volať funkcie zo zložky **bsp_i2c** v rámci iných funkcií, je potrebné nakonfigurovať súbor **CMakeLists.txt** nachádzajúci sa v zložke **bsp_i2c**. Podrobnosti konfigurácie sú nasledovné:

```
FILE(GLOB_RECURSE component_sources "*.c")

idf_component_register(SRCS ${component_sources}
                       INCLUDE_DIRS "include"
                       REQUIRES driver)
```

V tomto súbore **CMakeLists.txt** sú najskôr definované adresáre pre zdrojové súbory a hlavičkové súbory spolu s požadovanými knižnicami ovládačov. Následne sa tieto nastavenia zaregistrujú v systéme zostavovania pomocou príkazu `idf_component_register`, čím sa projektu umožní využívať funkčnosť ovládača **bsp_i2c**.

1.3.4 Kód ovládača DHT20

Tu sa zameriame výlučne na základný kód; podrobný zdrojový kód nájdete v príslušných zdrojových súboroch pre tento experiment v materiáloch kódu.

Zdrojový kód ovládača DHT20 pozostáva z dvoch súborov: **bsp_dht20.c** a **bsp_dht20.h**.

Nižšie najprv analyzujeme program **bsp_dht20.h**: definuje príslušné piny pre senzor teploty a vlhkosti a deklaruje funkcie.

/* Odkazy na hlavičkový súbor */

```
/*-----Header file declaration-----*/
#include <string.h> //References for string Function-related API Functions
#include "freertos/FreeRTOS.h" //References for FreeRTOS Function-related API Functions
#include "freertos/task.h" //References for FreeRTOS Task Function-related API Functions
#include "esp_log.h" //References for LOG Printing Function-related API Functions
#include "esp_err.h" //References for Error Type Function-related API Functions
#include "esp_timer.h" //References for high-precision timers Function-related API Functions
#include "bsp_i2c.h"
/*-----Header file declaration end-----*/
```

/* Deklarácie funkcií a deklarácie definícií makier */

```
#ifndef CONFIG_BSP_DHT20_SENSOR_ENABLED

#define DHT20_I2C_ADDRESS CONFIG_DHT20_I2C_ADDRESS // The 7-bit I2C address of DHT20

#define DHT20_MEASURE_TIMEOUT 1000 // Measurement timeout time of DHT20
typedef struct dht20_data
{
    float temperature; // The measured temperature data
    float humidity; // the measured humidity data
    uint32_t raw_humid; // Intermediate quantity for humidity data conversion
    uint32_t raw_temp; // Intermediate quantity for temperature data conversion
} dht20_data_t;

esp_err_t dht20_begin(void); // Initialization of DHT20 sensor
esp_err_t dht20_is_calibrated(void); // The function for determining whether the DHT20 sensor is ready or not
esp_err_t dht20_read_data(dht20_data_t *data); // DHT20 Sensor Temperature and Humidity Data Reading Function
#endif
```

Ďalej budeme analyzovať program **bsp_dht20.c**: inicializácia konfigurácie senzora DHT20 a sprístupnenie funkcií rozhrania API. www.rlx.sk

/* Inicializačná funkcia DHT20 **dht20_begin** */

```
esp_err_t dht20_begin(void)
{
    esp_err_t err = ESP_OK;
    dht20_handle = i2c_dev_register(DHT20_I2C_ADDRESS); /*DHT20 sensor address registration*/
    if (dht20_handle != NULL)
    {
        if (dht20_reset_sensor() >= 255) /* Check status register, if status not eq 0x18 reset the necessary registers*/
            err = ESP_FAIL;
        else
        {
            err = ESP_FAIL;
            return err;
        }
    }
    return err;
}
```

V rámci funkcie ``dht20_begin`` sa senzor ``DHT20`` najskôr zaregistruje na zbernici I²C pomocou funkcie ``i2c_dev_register``, ktorá vráti identifikátor operácie. Následne, ak vrátený identifikátor zariadenia nie je nulový, vyvolá sa funkcia ``dht20_reset_sensor``, aby sa senzor rešetoval a zabezpečilo sa, že je v stave, v ktorom je možné čítať údaje o teplote a vlhkosti.

/* Funkcia resetovania senzora DHT20 **dht20_reset_sensor** */

```
static uint8_t dht20_reset_sensor(void)
{
    static uint8_t rst_count = 0;
    uint8_t status = dht20_status(); /*Obtain the status of the DHT20 sensor*/
    DHT20_DEBUG("Sensor status: %s - 0x%02X", print_byte(status), status);
    while ((status & 0x18) != 0x18) /*If the DHT20 sensor status code is not 0x18*/
    {
        DHT20_DEBUG("Sensor status: %s - 0x%02X", print_byte(status), status);
        rst_count++;
        if (dht20_reset_register(0x1B) != ESP_OK) /*DHT20 sensor register reset function*/
        {
            rst_count++;
        }
        if (dht20_reset_register(0x1C) != ESP_OK)
        {
            rst_count++;
        }
        if (dht20_reset_register(0x1E) != ESP_OK)
        {
            rst_count++;
        }
        if (rst_count >= 255) /*Exceed the retry count*/
            return rst_count;
        DHT20_DEBUG("Registers resetted [%d] times!", rst_count);
        status = dht20_status(); /*Obtain the status of the DHT20 sensor*/
        vTaskDelay(10 / portTICK_PERIOD_MS);
    }
    return rst_count;
}
```

Táto funkcia najskôr volá funkciu ``dht20_status``, aby získala aktuálny stav senzora DHT20. Ak aktuálny stavový kód nie je `0x18`, volá funkciu ``dht20_reset_register``, aby rešetovala registre 0x1B, 0x1C a 0x1E a posúdila, či bol reset úspešný. Po resete opäť volá funkciu ``dht20_status``, aby skontrolovala aktuálny stavový kód senzora. Ak počet opakovaných pokusov presiahne 254, funkcia sa náhle ukončí a vráti nahromadený počet opakovaných pokusov.

/* Funkcia `dht20\_status` na čítanie stavových kódov zariadenia DHT20 */

```
uint8_t dht20_status(void)
{
    esp_err_t err = ESP_OK;
    static uint8_t txbuf = 0x71;
    static uint8_t rxdata;
    err = i2c_write(dht20_handle, &txbuf, 1); /*I2C write command*/
    if (err != ESP_OK)
        return err;
    vTaskDelay(100 / portTICK_PERIOD_MS);
    err = i2c_read(dht20_handle, &rxdata, 1); /*I2C read status*/
    if (err != ESP_OK)
        return err;
    return (rxdata);
}
```

www.rlx.sk

Táto funkcia používa funkciu `i2c_write` na odoslanie hodnoty 0x71 do senzora DHT20 a potom prečíta senzor, aby získala jeho stavový kód.

/* Funkcia resetovania registra senzora DHT20 `dht20_reset_register` */

```
static esp_err_t dht20_reset_register(uint8_t reg)
{
    esp_err_t err = ESP_OK;
    static uint8_t values[3] = {0};
    uint8_t txbuffer[3] = {reg, 0x00, 0x00}; /*Register address*/
    err = i2c_write(dht20_handle, txbuffer, 3); /*I2C write command*/
    if (err != ESP_OK)
        return err;
    vTaskDelay(5 / portTICK_PERIOD_MS);
    err = i2c_read(dht20_handle, values, 3); /*I2C read status*/
    if (err != ESP_OK)
        return err;
    vTaskDelay(10 / portTICK_PERIOD_MS);

    memset(txbuffer, 0, sizeof(txbuffer));
    txbuffer[0] = (0xB0 | reg);
    txbuffer[1] = values[1];
    txbuffer[2] = values[2];

    err = i2c_write(dht20_handle, txbuffer, 3);
    if (err != ESP_OK)
        return err;
    vTaskDelay(5 / portTICK_PERIOD_MS);
    return err;
}
```

Najskôr nakonfigurujete dátové pole na zápis, potom zavolajte funkciu `i2c_write` na zápis dát. Po 5 ms oneskorení zavolajte `i2c_read` na čítanie senzora a získanie jeho hodnoty. Priradíte získané dáta k prvému a druhému prvku dátového poľa. Prvý prvok dátového poľa používa bitový operátor OR na nastavenie najvyššieho bitu adresy vstupného registra na 1, potom prepíše senzor.

/* Funkcia na určenie, či je stavový kód DHT20 normálny: `dht20_is_calibrated` */

```
esp_err_t dht20_is_calibrated(void)
{
    esp_err_t err = ESP_OK;
    uint8_t status_byte = dht20_status();
    if ((status_byte & 0x10) != 0x10) /*If the DHT20 sensor status code is not 0x10*/
    {
        err = ESP_FAIL;
    }
    return err;
}
```

RLX COMPONENTS

prečítajte prijímací buffer a zavolajte funkciu `dht20_status` s počtom prečítaní, aby ste získali stavový kód senzora. Ak stavový kód nie je rovný `0x18`, vráťte `ESP_FAIL`.

/* Funkcia na čítanie údajov o teplote a vlhkosti DHT20 `dht20_read_data` */

```
esp_err_t dht20_read_data(dht20_data_t *data)
{
    esp_err_t err = ESP_OK;
    data->humidity = 0.0f;
    data->raw_humid = 0;
    data->temperature = 0.0f;
    data->raw_temp = 0;
    static uint8_t txbuf[3] = {0xAC, 0x33, 0x00}; /*Start measuring command + 2 bytes as parameters*/
    static uint8_t status_byte[1] = {0}; /*retrieve measuring status*/
    static uint8_t rxdata[7] = {0}; /*retrieve measurement data (temp + humid)*/
    err = i2c_write(dht20_handle, txbuf, 3);
    if (err != ESP_OK)
        return err;
    vTaskDelay(80 / portTICK_PERIOD_MS); /*minimum delay as per dataheet*/
    DHT20_DEBUG("Reading registers.....");
    err = i2c_read(dht20_handle, status_byte, 1); /*read status byte to check sensor is done measuring*/
    if (err != ESP_OK)
        return err;
    unsigned long start_time = esp_timer_get_time() / 1000;
    while ((status_byte[0] >> 7) != 0) /*while measurement is ongoing*/
    {
        if ((esp_timer_get_time() / 1000) - start_time >= DHT20_MEASURE_TIMEOUT)
        {
            return ESP_ERR_TIMEOUT; /*Timeout*/
        }
        portYIELD(); /*give scheduler some time for other tasks while we're still measuring*/
    }

    err = i2c_read(dht20_handle, rxdata, 7); /*sensor idle let's read all data*/
    if (err != ESP_OK)
        return err;
    DHT20_DEBUG("Byte1: %s", print_byte(rxdata[0]));
    DHT20_DEBUG("Byte2: %s", print_byte(rxdata[1]));
    DHT20_DEBUG("Byte3: %s", print_byte(rxdata[2]));
    DHT20_DEBUG("Byte4: %s", print_byte(rxdata[3]));
    DHT20_DEBUG("Byte5: %s", print_byte(rxdata[4]));
    DHT20_DEBUG("Byte6: %s", print_byte(rxdata[5]));
    DHT20_DEBUG("CRC Byte: %s", print_byte(rxdata[6]));

    uint8_t get_crc = dht20_crc8(rxdata, 6); /*CRC calculation*/
    DHT20_DEBUG("Data byte 7: 0x%02X, calculated crc8: 0x%02X", rxdata[6], get_crc);

    if (rxdata[6] == get_crc) /*Compare CRC values and continue if they match*/
    {
        uint32_t raw_humid = rxdata[1];
        raw_humid <<= 8;
        raw_humid += rxdata[2];
        raw_humid <<= 4;
        raw_humid += rxdata[3] >> 4;
        data->raw_humid = raw_humid;
        data->humidity = (float)(raw_humid / 1048576.0f) * 100.0f; /*convert RAW to Humidity in %*/
        DHT20_DEBUG("Humidity raw: %lu - Converted: %.1f %%", data->raw_humid, data->humidity);

        uint32_t raw_temp = (rxdata[3] & 0x0F);
        raw_temp <<= 8;
        raw_temp += rxdata[4];
        raw_temp <<= 8;
        raw_temp += rxdata[5];
        data->raw_temp = raw_temp;
        data->temperature = (float)(raw_temp / 1048576.0f) * 200.0f - 50.0f; /*convert RAW to Celsius C*/
        DHT20_DEBUG("Temperature raw: %lu - Converted: %.2fC.", data->raw_temp, data->temperature);
    }
    else
    {
        DHT20_ERROR("CRC Checksum failed !!!");
        return ESP_ERR_INVALID_CRC;
    }
    return 0.0f;
}
```

Najskôr inicializujte premenné. Vyvolajte funkciu `i2c_write`, aby ste do senzora zapísali príkaz na spustenie merania. Po 80 milisekundovom oneskorení vyvolajte funkciu `i2c_read`, aby ste prečítali stavový bajt a zistili, či senzor dokončil meranie. Ak čas čítania prekročí nastavenú maximálnu dĺžku merania, ukončíte

a vráťte chybu časového limitu. Ak je meranie dokončené, znovu vyvolajte funkciu ``i2c_read``, aby ste prečítali sedem bajtov `0x00000000` (skúste meranie senzora zahŕňa kontrolný súčet CRC). Prevedte kontrolný súčet pre prvých šesť bajtov prečítaných údajov pomocou funkcie ``dht20_crc8`` (polynóm kontrolného súčtu CRC8: $\text{CRC}[7:0] = 1 + x^4 + x^5 + x^8$). Vypočítaný kontrolný súčet sa porovná s načítaným kontrolným súčtom. Ak sa zhodujú, údaje sú platné. Nakoniec sa načítané údaje prevedú pomocou nasledujúceho vzorca:

Prvý bajt prečítaných údajov je stavový kód. Horné 4 bity druhého, tretieho a štvrtého bajtu tvoria údaje o vlhkosti. Prekonvertovanie týchto údajov o vlhkosti: - Posuňte údaje druhého bajtu o 8 bitov doľava. - Pridajte údaje tretieho bajtu. - Posuňte výsledok o 4 bity doľava. - Pridajte údaje tretieho bajtu posunutú o 4 bity doprava. Premenené údaje o vlhkosti vydáme $2^{20} \times 100 \%$ (napr. ak sú údaje druhého bajtu `0x18`, údaje tretieho bajtu `0x22` a údaje štvrtého bajtu `0x11`, premena: $(((((0x18 \ll 8) | 0x22) \ll 4) | (0x11 \gg 4)) = 0x18221$
 $(0x18221 / (2^{20})) * 100 \%$ = 9,43 % (zaokrúhlené na najbližšie celé číslo)

Podobne, údaje o teplote tvoria štyri nižšie bity štvrtého bajtu, piaty bajt a šiesty bajt. Prekonvertovanie údajov o teplote:

- Pridajte `0x0F` k štvrtému bajtu a extrahujte štvrtý bit.
- Posuňte tento extrahovaný bit o 8 pozícií doľava a pridajte ho k piatemu bajtu. Po pridaní posuňte výsledok o 8 bitov doľava a pridajte údaje šiesteho bajtu. Premenené údaje o teplote vydáme $2^{20} \times 200$ a od výsledku odpočítame 50. (Napríklad, ak je štvrtý bajt `0x16`, piaty bajt `0xF5` a šiesty bajt `0xF2`, výsledok premeny je:

$(((((0x16 \& 0x0F) \ll 8) | 0xF5) \ll 8) | 0xF2) = 0x6F5F2$ $((0x6F5F2 / 2^{20}) * 200) - 50$
 = 37 stupňov (zaokrúhlené na najbližší celý stupeň))

1.3.5 Súbor Kconfig

Hlavnou funkciou tohto súboru je pridať požadovanú konfiguráciu do súboru **sdkconfig**, čím sa umožní modifikácia určitých nastavení parametrov prostredníctvom grafického rozhrania. Tu **0x38** predstavuje 7-bitovú adresu pre senzor DHT20.

```
menu "BSP DHT20 Setup"

    config BSP_DHT20_SENSOR_ENABLED
        bool "Enable DHT20 Temperature and Humidity sensor"
        depends on BSP_I2C_ENABLED

    if BSP_DHT20_SENSOR_ENABLED
        config DHT20_I2C_ADDRESS
            hex "DHT20 I2C Address"
            default 0x38
    endif

endmenu
```

1.3.6 Súbor CMakeLists.txt

Funkčnosť tejto ukážkovej rutiny závisí predovšetkým od ovládača **bsp_dht20**. Aby bolo možné úspešne volať funkcie zo zložky **bsp_dht20** v rámci iných funkcií, je potrebné nakonfigurovať súbor **CMakeLists.txt**, ktorý sa nachádza v zložke **bsp_dht20**. Podrobnosti konfigurácie sú nasledovné:

```
FILE(GLOB_RECURSE component_sources "*.c")

idf_component_register(SRCS ${component_sources}
                      INCLUDE_DIRS "include"
                      REQUIRES bsp_i2c esp_timer)
```

V tomto súbore **CMakeLists.txt** sú najskôr definované adresáre pre zdrojové súbory a hlavičkové súbory spolu s požadovanou knižnicou ovládača (**bsp_i2c**). Následne sa tieto nastavenia zaregistrujú v systéme zostavovania pomocou príkazu **idf_component_register**, čím sa projektu umožní **využívať** funkčnosť ovládača **bsp_dht20**.

1.3.7 Hlavný priečinok

Hlavný priečinok slúži ako základný adresár pre vykonávanie programu a obsahuje hlavnú funkciu **main.c** a hlavičkový súbor **main.h** v priečinku **include**. Pridajte hlavný priečinok do súboru **CMakeLists.txt** systému zostavenia.

Súbor **main.h** odkazuje predovšetkým na požadované hlavičkové súbory: funkcie využívajúce ovládač **bsp_display** vyžadujú zaradenie hlavičkového súboru **bsp_display**, zatiaľ čo funkcie využívajúce ovládač **bsp_dht20** vyžadujú hlavičkový súbor **bsp_dht20**.

Nižšie je analýza programu **main.c**: inicializácia systému a vykonávanie funkcií pre **I2C**, senzor **DHT20** a funkcie displeja.

```
#ifdef CONFIG_BSP_I2C_ENABLED
    err = i2c_init(); /*I2C Initialization*/
    if (err != ESP_OK)
        init_fail("i2c", err);
    vTaskDelay(200 / portTICK_PERIOD_MS);
#endif
#ifdef CONFIG_BSP_DHT20_SENSOR_ENABLED
    err = dht20_begin(); /*DHT20 Initialization*/
    if (err != ESP_OK)
        init_fail("dht20", err);
#endif
#ifdef CONFIG_BSP_DISPLAY_ENABLED
    err = display_init(); /*Display Initialization*/
    if (err != ESP_OK)
        init_fail("display", err);
#endif
```

Tento kód sa nachádza vo **funkcii init**, ktorá sa používa na ukladanie inicializačných funkcií vyžadujúcich vyvolanie a na vyhodnotenie ich návratového stavu. Ak návratový stav nie je **ESP_OK**, kód vytlačí chybovú správu a zastaví ďalšie vykonávanie.

RLX COMPONENTS

/* Inicializácia obrazovky a funkcia zobrazenia dht20_display */

```
void dht20_display()
{
    if (lvgl_port_lock(0))
    {
        dht20_data = lv_label_create(lv_scr_act()); /*Create a label object*/
        static lv_style_t label_style;
        lv_style_init(&label_style);
        lv_style_set_bg_opa(&label_style, LV_OPA_TRANSP); /*Initialize a style*/
        lv_obj_set_style_bg_color(dht20_data, &label_style, LV_PART_MAIN); /*Set the style LVGL background color*/
        lv_obj_set_style_text_color(dht20_data, LV_COLOR_WHITE, LV_PART_MAIN); /*Add a style to an object*/
        lv_obj_set_style_text_font(dht20_data, &lv_font_montserrat_30, LV_PART_MAIN); /*Set the style LVGL text color*/
        lv_obj_center(dht20_data); /*Set the style LVGL text font*/
        lv_obj_set_style_bg_opa(lv_scr_act(), LV_COLOR_BLACK, LV_PART_MAIN); /*Align an object to the center on its parent*/
        lv_obj_set_style_bg_color(lv_scr_act(), LV_OPA_COVER, LV_PART_MAIN); /*Set the screen's LVGL background color*/
        lv_label_set_text(dht20_data, "Temperature = 0.0 C Humidity = 0.0 %"); /*Set the screen's LVGL background transparency*/
        lvgl_port_unlock(); /*Set a new text for a label*/
    }
}
```

Táto funkcia primárne konfiguruje obsah počiatočnej obrazovky: nastavenie farieb pozadia a zobrazenia textu prostredníctvom ovládacích prvkov lvgl.

Funkcia **lv_label_set_text** nastavuje text zobrazený na ovládacom prvku.

Funkcia **lv_style_set_bg_opa** nastavuje farbu pozadia štýlu. Funkcia

lv_obj_set_style_text_color nastavuje farbu zobrazenia textu.

Funkcia **lv_obj_set_style_text_font** nastavuje veľkosť písma textu.

Funkcia **lv_obj_set_style_bg_color** nastavuje farbu pozadia.

Funkcia **lv_obj_set_style_bg_opa** nastavuje priehľadnosť pozadia.

Poznámka: Pri volaní funkcií lvgl mimo funkcií vlákna lvgl je potrebné získať mutex zámok.

Funkcia lvgl_port_lock získava mutex zámok, funkcia lvgl_port_unlock ho uvoľňuje.

/* Funkcia aktualizácie údajov na obrazovke update_dht20_value */

```
void update_dht20_value(float temperature, float humidity)
{
    if (dht20_data)
    {
        char buffer[60];
        sprintf(buffer, sizeof(buffer), "Temperature = %.1f C Humidity = %.1f %", temperature, humidity); /*Format the data into a string*/
        lv_label_set_text(dht20_data, buffer); /*Set a new text for a label*/
    }
}
```

Táto funkcia používa funkciu sprintf na formátovanie dvoch získaných údajov typu float do reťazca a následne obnovuje obsah zobrazenia pomocou funkcie **lv_label_set_text**.

Stojí za zmienku, že funkcia sprintf pripája na koniec formátovaného reťazca ukončovaci znak. Keďže **lv_label_set_text** rozpoznáva reťazce identifikáciou tohto ukončovacieho znaku, použitie sprintf na formátovanie reťazcov je preferovaným prístupom.

V rámci funkcie app_main sa najskôr nastaví jas podsvietenia na 100 % a následne sa

```
void app_main(void)
{
    MAIN_INFO("----- Demo version -----");
    Init();
    #ifdef CONFIG_BSP_DHT20_SENSOR_ENABLED
    #endif CONFIG_BSP_DISPLAY_ENABLED
    set_lcd_brightness(100); /*Set the screen backlight to maximum brightness*/
    dht20_display(); /*Set the screen's LVGL default display page*/
    xTaskCreate(dht20_read_task, "read_dht20", 4096, NULL, configMAX_PRIORITIES - 5, &read_dht20); /*Create a DHT20 data display refresh thread*/
    #endif
    #endif
    MAIN_INFO("----- Start the test -----");
}
```

inicializuje obsah zobrazenia na obrazovke. Nakoniec sa vytvorí vlákno FreeRTOS na spracovanie údajov a obnovenie obrazovky.


```
void dht20_read_task(void *param)
{
    static dht20_data_t measurements;
    while (1)
    {
        if (dht20_is_calibrated() == ESP_OK) /*The function for determining whether the DHT20 sensor is ready or not*/
        {
            MAIN_INFO("Is calibrated....");
        }
        else
        {
            MAIN_INFO("Is NOT calibrated....");
            if (dht20_begin() != ESP_OK) /*Reinitialize the DHT20 sensor*/
            {
                MAIN_ERROR("dht20 init again false");
                vTaskDelay(100 / portTICK_PERIOD_MS);
                continue;
            }
        }
        if (dht20_read_data(&measurements) != ESP_OK) /*Read the temperature and humidity data from the DHT20 sensor*/
        {
            if (lvgl_port_lock())
            {
                lv_label_set_text(dht20_data, "dht20 read data error"); /*Read failure message displayed*/
                lvgl_port_unlock();
            }
            MAIN_ERROR("dht20 read data error");
        }
        else /*Read successfully*/
        {
            if (lvgl_port_lock())
            {
                update_dht20_value(measurements.temperature, measurements.humidity); /*Update the DHT20 data displayed on the screen*/
                lvgl_port_unlock();
            }
            MAIN_INFO("Temperature: %f", measurements.temperature);
            MAIN_INFO("Humidity: %f", measurements.humidity);
        }
        vTaskDelay(1000 / portTICK_PERIOD_MS);
    }
}
```

V rámci vlákna spracovania údajov o teplote a vlhkosti DHT20 a obnovovania zobrazenia na obrazovke sa najskôr inicializujú premenné. Následne sa vytvorí cyklus while. V rámci tohto cyklu sa vyvolá funkcia `dht20_is_calibrated`, aby sa určil stavový kód senzora. Ak výsledok nie je `ESP_OK`, senzor DHT20 sa reinitializuje. Ak inicializácia zlyhá, vykoná sa príkaz `continue`, ktorý vráti cyklus. Ak je vrátená hodnota `ESP_OK`, vyvolá sa funkcia `dht20_read_data` na načítanie údajov o teplote a vlhkosti. Po úspešnom načítaní funkcia `update_dht20_value` obnoví zobrazenie na obrazovke. Ak načítanie zlyhá, na obrazovke sa zobrazí hlásenie „dht20 read data error“ (chyba načítania údajov dht20). Záverečné 1-sekundové oneskorenie zabezpečuje, že údaje sa obnovujú raz za sekundu.

1.3.8 Súbor CMKaLists.txt

Aby bolo možné úspešne vyvolať obsah priečinkov `bsp_display` a `bsp_dht20` v hlavnej funkcii, je potrebné nakonfigurovať súbor `CMakeLists.txt`, ktorý sa nachádza v hlavnom priečinku. Podrobnosti konfigurácie sú nasledovné:

```
FILE(GLOB_RECURSE main ${CMAKE_SOURCE_DIR}/main/*.c)

idf_component_register(SRCS ${main}
                        INCLUDE_DIRS "include"
                        REQUIRES bsp_i2c bsp_display bsp_dht20)
```

Najskôr sa definujú adresáre pre zdrojové súbory a hlavičkové súbory spolu s požadovanými knižnicami ovládačov – konkrétne knižnice ovládačov pre prepojenie `bsp_display`, `bsp_dht20` a `bsp_i2c`. Následne sa tieto nastavenia zaregistrujú v systéme zostavovania pomocou príkazu `idf_component_register`, čím sa hlavnej funkcii umožní využívať tieto funkcie ovládačov.

1.4 Postup programovania

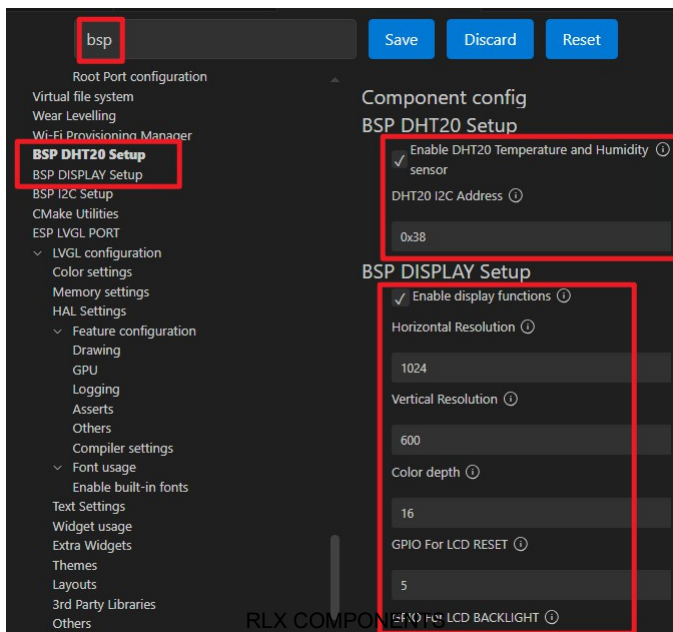
www.rlx.sk

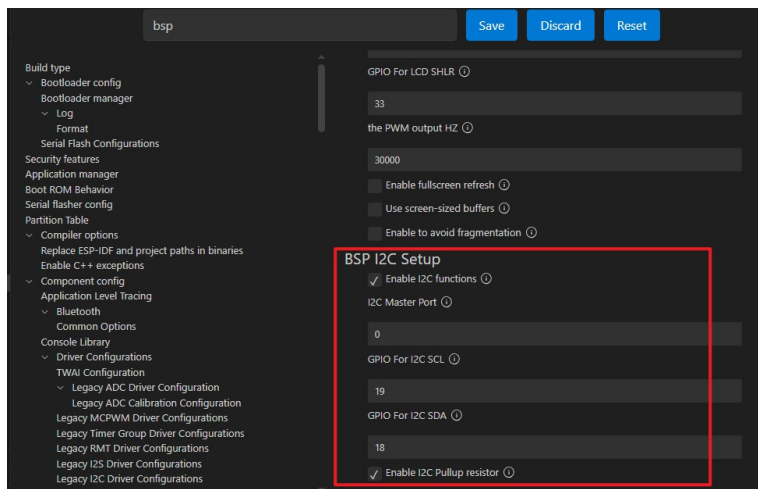
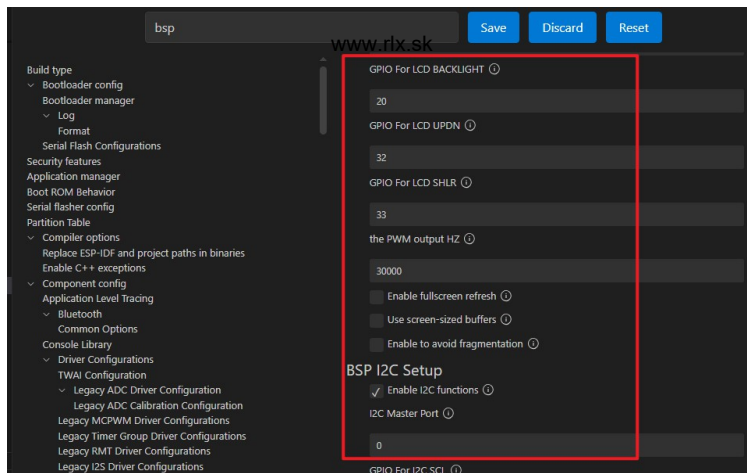
Pripojte zariadenie P4 k počítaču prostredníctvom USB



1.4.1 Po klonovaní kódu prostredníctvom Git (odkaz bude potvrdený) vymažte všetky lokálne informácie o kompilácii. Nakonfigurujte prostredie IDF a model čipu pre kompiláciu podľa lekcie 1 a nastavte číslo sériového portu pre programovanie.

1.4.2 Nasledujúca konfigurácia SDKConfig je z veľkej časti identická s lekciami 1, stačí len prekonfigurovať piny DSI a dht20.





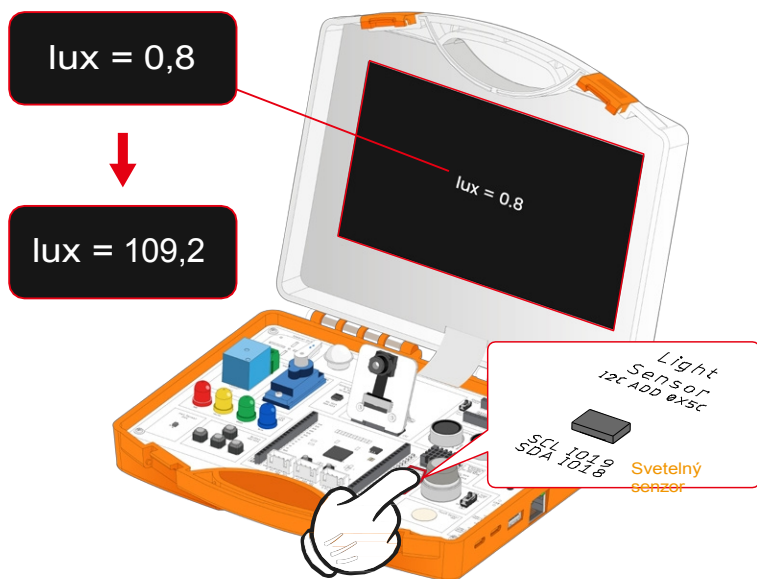
1.4.3 Kliknite na Compile (Kompilovať). Po úspešnej kompilácii kliknite na Download (Stiahnuť).

Lekcia 12 – Svetelný senzor BH1750

Úvod

Táto kapitola tutoriálu predstavuje aplikáciu periférie I2C ESP32-P4. Prostredníctvom experimentov so senzorom intenzity svetla BH1750 pomáha pochopiť základné princípy komunikácie I2C a získavania údajov o okolitej svetelnosti. BH1750 je bežne používaný digitálny luxmeter, ktorý je schopný priamo vydať hodnoty intenzity osvetlenia v jednotkách lux, vďaka čomu je veľmi vhodný pre projekty ako inteligentné osvetlenie, automatické stmievanie a monitorovanie prostredia. Táto kapitola postupuje krok za krokom a vytvára základ pre následné aplikácie zahŕňajúce ďalšie zariadenia I²C.

Demonštrácia projektu Efekt



Táto kapitola je rozdelená do nasledujúcich podkapitol

- 1.1 Úvod do BH1750 a zbernice I²C
- 1.2 Návrh hardvéru
- 1.3 Návrh softvéru
- 1.4 Stiahnutie a overenie

RLX COMPONENTS

1.1 Úvod do BH1750

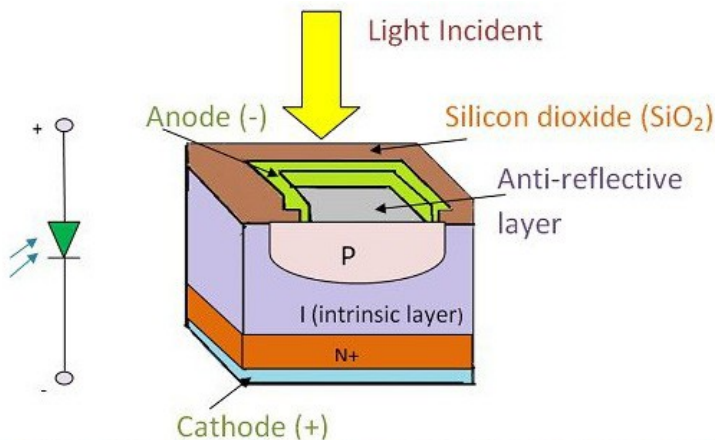
1.1.1 Úvod do BH1750

BH1750 je digitálny senzor intenzity osvetlenia vyrobený spoločnosťou ROHM, ktorý je schopný prevádzať intenzitu osvetlenia na digitálne signály a vyslať ich prostredníctvom zbernice I²C. V porovnaní s tradičnými analógovými fotorezistormi ponúka BH1750 výhody, ako je vysoká presnosť, silné protikontrastné schopnosti a rýchla odozva, vďaka čomu je veľmi vhodný pre vstavané systémy.

Kľúčové vlastnosti zahŕňajú:

- ① Digitálny výstup I²C: Eliminuje potrebu analógovo-digitálneho prevodu (ADC), čo umožňuje priame načítanie hodnôt intenzity svetla (v luxoch) prostredníctvom I²C.
- ② Široký rozsah citlivosti: 1 až 65 535 luxov, čo vyhovuje scenárom od nočného osvetlenia až po intenzívne slnečné svetlo.
- ③ Nízka spotreba energie: Prevádzkový prúd približne 0,12 mA, úspora energie.
- ④ Automatické prepínanie rozsahu: Prepína medzi režimom s vysokým rozlíšením (1 lx) a režimom s nízkym rozlíšením (4 lx).
- ⑤ Kompaktná konštrukcia: Malá veľkosť balenia uľahčuje integráciu do rôznych ovládacích dosiek.

1.1.2 Princípy merania svetla



Photodiode Symbol & Construction of Photodiode

BH1750 detekuje intenzitu okolitého osvetlenia prostredníctvom integrovanej fotodiódy, pričom využíva analógovo-digitálny prevodník (ADC) na prevod svetelných signálov na digitálne hodnoty pre výstup. Jeho vnútorná konštrukcia pozostáva predovšetkým z nasledujúcich komponentov:

- Fotodióda: Premení svetelnú energiu na elektrické signály;
 - Integračný obvod: Vykonáva časovú integráciu aktuálneho signálu s cieľom získať priemernú intenzitu svetla;
 - Analógovo-digitálny prevodník (ADC): Prevádza elektrické signály na digitálne signály;
 - Register a rozhranie I²C: Ukladá výsledky merania a prenáša ich do hostiteľského radiča prostredníctvom I²C.
- BH1750 vysiela údaje v luxoch (Lx), ktoré predstavujú svetelný tok prijatý na jednotku plochy. Inými slovami, vyššie hodnoty znamenajú jasnejšie prostredie.

1.1.3 Prevádzkové režimy BH1750

BH1750 ponúka viacero režimov merania, čo umožňuje flexibilný výber podľa scenára použitia:

režimy	Popis funkcie	Typická doba merania	Rozlíšenie
Režim H-rozlíšenia	Režim s vysokým rozlíšením	120 ms	1 lx
Režim H-rozlíšenia 2	Režim vysokej presnosti	120 ms	0,5 lx
Režim L-rozlíšenia	Režim s nízkym rozlíšením	16 ms	4 lx

Výber režimu sa dosahuje zapísaním rôznych príkazových slov (napr. 0x10, 0x11, 0x13) do BH1750, čo umožňuje vysoko efektívnu komunikáciu.

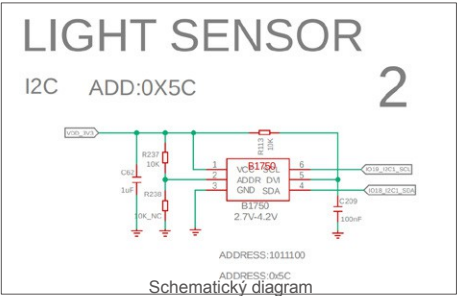
1.1.4 Adresa I²C a pokyny pre zapojenie

Predvolená adresa I²C pre modul BH1750 je 0x5C.

2.2 Hardvérový dizajn

V experimente je modul BH1750 pripojený k vývojovej doske ESP32-P4 nasledovne:

ESP32-P4	BH1750
3V3	VCC
GND	GND
IO18 (SDA)	SDA
IO19 (SCL)	SCL



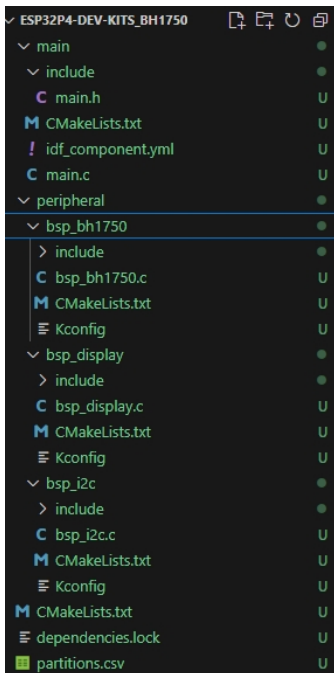
RLX COMPONENTS

1.3 Analýza programu

www.rlx.sk

<https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design>

Otvorte súbor projektu v VS Code podľa predchádzajúcich pokynov.



V rámci príkladu **ESP32P4-dev-kits_bh1750** bola v adresári **ESP32P4-dev-kits_bh1750\peripheral** vytvorená nová zložka s názvom **bsp_bh1750**. V rámci cesty **bsp_bh1750** bola vytvorená nová zložka include, súbor **CMakeLists.txt** a súbor **Kconfig**.

Zložka **bsp_bh1750** obsahuje súbor ovládača **bsp_bh1750.c**. Zložka include obsahuje súbor hlavičky **bsp_bh1750.h**, zatiaľ čo súbor **CMakeLists.txt** integruje ovládač do systému zostavovania, čím umožňuje projektové využitie jeho funkcionality. Súbor **Kconfig** načíta celý ovládač spolu s definíciami pinov GPIO do súboru **sdkconfig** v rámci platformy IDF (konfigurovateľný prostredníctvom grafického rozhrania).

1.3.1 Kód ovládača BH1750

Tu sa zameriame výlučne na základný kód; podrobný zdrojový kód nájdete v príslušných zdrojových súboroch pre tento experiment v materiáloch kódu.

Zdrojový kód ovládača BH1750 pozostáva z dvoch súborov: **bsp_bh1750.c** a **bsp_bh1750.h**.

Nižšie najprv analyzujeme program **bsp_bh1750.h**: obsahuje relevantné definície a deklarácie funkcií zodpovedajúce pinom svetelného senzora.

/* Odkazy na hlavičkové súbory */

```
/*-----Header file declaration-----*/
#include "esp_log.h" //References for LOG Printing Function-related API Functions
#include "esp_err.h" //References for Error Type Function-related API Functions
#include "bsp_i2c.h"
/*-----Header file declaration end-----*/
```

/* Deklarácie funkcií a definície makier */

```
#ifndef CONFIG_BSP_BH1750_SENSOR_ENABLED
#define CONFIG_BSP_BH1750_SENSOR_ENABLED

#define BH1750_I2C_ADDRESS CONFIG_BH1750_I2C_ADDRESS // The 7-bit I2C address of BH1750

#define BH1750_PWR_DOWN 0x00 // Power down
#define BH1750_PWR_ON 0x01 // Power on
#define BH1750_RST 0x07 // reset register
#define BH1750_CON_H 0x10 // 1lx, 120ms
#define BH1750_CON_H2 0x11 // 0.5lx, 120ms
#define BH1750_CON_L 0x13 // 4lx, 16ms
#define BH1750_ONE_H 0x20 // one test : 1lx, 120ms,
#define BH1750_ONE_H2 0x21 // one test : 0.5lx, 120ms,
#define BH1750_ONE_L 0x23 // one test : 4lx, 16ms,

esp_err_t bh1750_begin(void); // Initialization of BH1750 sensor
float bh1750_read_data(); // BH1750 Sensor lux Data Reading Function

#endif
```

Ďalej budeme analyzovať program **bsp_bh1750.c**: inicializácia a konfigurácia senzora BH1750 pri súčasnom sprístupnení funkcií rozhrania API.

/* Inicializačná funkcia BH1750 **bh1750_begin** */

```
esp_err_t bh1750_begin(void)
{
    esp_err_t err = ESP_OK;
    uint8_t tx_buf;
    tx_buf = BH1750_PWR_ON;
    bh1750_handle = i2c_dev_register(BH1750_I2C_ADDRESS); //BH1750 sensor address registration/
    if (bh1750_handle == NULL)
        return ESP_FAIL;
    err = i2c_write(bh1750_handle, &tx_buf, 1); /*Set the BH1750 sensor to start working*/
    if (err != ESP_OK)
        return err;
    tx_buf = BH1750_CON_H;
    err = i2c_write(bh1750_handle, &tx_buf, 1); /*Start measurement at 1lx resolution, Measurement Time is typically 120ms*/
    if (err != ESP_OK)
        return err;
    return err;
}
```

V rámci funkcie ``bh1750_begin`` sa senzor BH1750 najskôr zaregistruje na zbernici I²C pomocou funkcie ``i2c_dev_register``, ktorá vráti identifikátor operácie. Následne, ak vrátený identifikátor zariadenia nie je nulový, vyvolá sa funkcia ``i2c_write``, aby sa senzor aktivoval a zabezpečilo sa, že je v stave, v ktorom je možné čítať údaje o osvetlení. Potom sa zapíše testovací register, aby sa umožnili merania v intervaloch 120 ms.

/* Funkcia čítania údajov o osvetlenosti BH1750 **bh1750_read_data** */

```
float bh1750_read_data()
{
    esp_err_t err = ESP_OK;
    float lux;
    uint8_t sensorData[2] = {0};
    err = i2c_read(bh1750_handle, sensorData, 2); /*Read sensor data*/
    if (err != ESP_OK)
        return -1;
    lux = (sensorData[0] << 8 | sensorData[1]) / 1.2; /*Convert illuminance data*/
    return lux;
}
```

Najskôr inicializujte premenné. Vyvolajte funkciu `i2c_read`, aby ste prečítali dva bajty údajov zo senzora. Prečítajte dva bajty pomocou nasledujúceho vzorca:

Posuňte prvý bajt o 8 bitov doľava, vykonajte operáciu OR s druhým bajtom a výsledok vydelite 1,2.

$((X1 \ll 8) | X2) / 1,2$

(Príklad: Ak je prvý bajt 0x02 a druhý bajt 0x34, potom: $((0x02 \ll 8) |$

$0x34) / 1,2 = 470$)

RLX COMPONENTS

1.3.2 Súbor Kconfig

www.rlx.sk

Hlavnou funkciou tohto súboru je začleniť požadovanú konfiguráciu do súboru sdkconfig, čím sa umožní vykonanie určitých úprav parametrov prostredníctvom grafického rozhrania. Tu 0x5C označuje 7-bitovú adresu senzora BH1750.

```
menu "BSP BH1750 Setup"

config BSP_BH1750_SENSOR_ENABLED
    bool "Enable BH1750 illumination sensor"
    depends on BSP_I2C_ENABLED

if BSP_BH1750_SENSOR_ENABLED
    config BH1750_I2C_ADDRESS
        hex "BH1750 I2C Address"
        default 0x5C
    endif
endif

endmenu
```

1.3.3 Súbor CMkaLists.txt

Funkčnosť tejto ukážkovej rutiny závisí predovšetkým od ovládača bsp_bh1750. Aby bolo možné úspešne vyvolať obsah priečinka bsp_bh1750 z iných funkcií, je potrebné nakonfigurovať súbor CMakeLists.txt v priečinku bsp_bh1750. Podrobnosti konfigurácie sú nasledovné:

```
FILE(GLOB_RECURSE component_sources "*.c")

idf_component_register(SRCS ${component_sources}
    INCLUDE_DIRS "include"
    REQUIRES bsp_i2c)
```

V tomto súbore CMakeLists.txt sú najskôr definované adresáre pre zdrojové súbory a hlavičkové súbory spolu s požadovanou knižnicou ovládača (bsp_i2c). Následne sa tieto nastavenia zaregistrujú v systéme zostavovania pomocou príkazu idf_component_register, čím sa projektu umožní využívať funkčnosť ovládača bsp_bh1750.

1.3.4 Hlavný priečinok

Hlavný priečinok slúži ako hlavný adresár pre vykonávanie programu a obsahuje hlavnú funkciu main.c a hlavičkový súbor main.h v priečinku include. Pridajte hlavný priečinok do súboru CMakeLists.txt v rámci systému zostavovania.

Súbor main.h odkazuje predovšetkým na požadované súbory hlavičiek: funkcie využívajúce ovládač bsp_display vyžadujú zaradenie súboru hlavičky bsp_display, zatiaľ čo funkcie využívajúce ovládač bsp_bh1750 vyžadujú súbor hlavičky bsp_bh1750.

Nižšie je analýza programu main.c: inicializácia systému a vykonávanie funkcií pre I2C, senzor bh1750 a funkcie displeja.

```
#ifdef CONFIG_BSP_I2C_ENABLED
    err = i2c_init(); /*I2C Initialization*/
    if (err != ESP_OK)
        init_fail("i2c", err);
    vTaskDelay(200 / portTICK_PERIOD_MS);
#endif
#ifdef CONFIG_BSP_BH1750_SENSOR_ENABLED
    err = bh1750_begin(); /*BH1750 Initialization*/
    if (err != ESP_OK)
        init_fail("bh1750", err);
#endif
#ifdef CONFIG_BSP_DISPLAY_ENABLED
    err = display_init(); /*Display Initialization*/
    if (err != ESP_OK)
        init_fail("display", err);
#endif
```

RLX COMPONENTS

Tento kód sa nachádza vo funkcii `init`, ktorá sa používa na ukladanie inicializačných funkcií vyžadujúcich vyvolanie a na vyhodnotenie ich výsledkov. Ak sa stav návratu odchyľuje od `ESP_OK`, kód vypíše chybovú správu a zastaví ďalšie vykonávanie.

/* Inicializácia obrazovky a funkcia zobrazenia **bh1750_display** */

```
void bh1750_display()
{
    if (lvgl_port_lock(0))
    {
        bh1750_data = lv_label_create(lv_scr_act()); /*create a label object*/
        static lv_style_t label_style;
        lv_style_init(&label_style);
        lv_style_set_bg_opa(&label_style, LV_OPA_TRANSP); /*Initialize a style*/
        lv_obj_add_style(bh1750_data, &label_style, LV_PART_MAIN); /*Set the style LVGL background color*/
        lv_obj_set_style_text_color(bh1750_data, LV_COLOR_WHITE, LV_PART_MAIN); /*Add a style to an object*/
        lv_obj_set_style_text_font(bh1750_data, &lv_font_montserrat_36, LV_PART_MAIN); /*Set the style LVGL text color*/
        lv_obj_center(bh1750_data); /*Set the style LVGL text font*/
        lv_obj_set_style_bg_color(lv_scr_act(), LV_COLOR_BLACK, LV_PART_MAIN); /*Align an object to the center on its parent*/
        lv_obj_set_style_bg_opa(lv_scr_act(), LV_OPA_COVER, LV_PART_MAIN); /*Set the screen's LVGL background color*/
        lv_label_set_text(bh1750_data, "lux = 0.0"); /*Set a new text for a label*/
        lvgl_port_unlock();
    }
}
```

Táto funkcia primárne konfiguruje počiatočný obsah zobrazenia obrazovky: nastavenie farieb pozadia a zobrazenia textu prostredníctvom ovládacích prvkov `lvgl`.

Funkcia `lv_label_set_text` nastavuje text zobrazený na ovládacom prvku.

Funkcia `lv_style_set_bg_opa` nastavuje farbu pozadia štýlu. Funkcia

`lv_obj_set_style_text_color` nastavuje farbu zobrazenia textu.

Funkcia `lv_obj_set_style_text_font` nastavuje veľkosť písma textu.

Funkcia `lv_obj_set_style_bg_color` nastavuje farbu pozadia.

Funkcia `lvgl_obj_set_style_bg_opa` nastavuje priehľadnosť pozadia.

Stojí za zmienku: Pri volaní funkcií `lvgl` mimo funkcií vlákna `lvgl` je potrebné získať mutex zámok.

Funkcia `lvgl_port_lock` získava mutex zámok, zatiaľ čo funkcia `lvgl_port_unlock` ju uvoľňuje.

/* Funkcia aktualizácie údajov na obrazovke `update_bh1750_value` */

```
void update_bh1750_value(float lux)
{
    if (bh1750_data)
    {
        char buffer[40];
        sprintf(buffer, sizeof(buffer), "lux = %.1f", lux); /*Format the data into a string*/
        lv_label_set_text(bh1750_data, buffer); /*Set a new text for a label*/
    }
}
```

Táto funkcia používa funkciu `sprintf` na formátovanie získaných údajov **typu float** do reťazca a následne obnovuje zobrazený obsah prostredníctvom funkcie `lv_label_set_text`.

Stojí za zmienku, že funkcia `sprintf` pripája na koniec formátovaného reťazca ukončovaci znak. Navyše, `lv_label_set_text` rozpoznáva reťazce identifikovaním tohto ukončovacieho znaku. V dôsledku toho je využitie funkcie `sprintf` na formátovanie reťazcov preferovaným prístupom.

```

void app_main(void)
{
    MAIN_INFO("-----Demo version-----");
    MAIN_INFO("-----Start the test-----");
    Init();
    #ifdef CONFIG_BSP_BH1750_SENSOR_ENABLED
    #ifdef CONFIG_BSP_DISPLAY_ENABLED
        set_lcd_blight(100);
        bh1750_display();
        taskCreate(bh1750_read_task, "read_bh1750", 4096, NULL, configMAX_PRIORITIES - 5, &read_bh1750); /*Set the screen backlight to maximum brightness*/
        /*Set the screen's I/O. default display page*/
        /*Create a bh1750 data display refresh thread*/
    #endif
    #endif
}

```

www.rlx.sk

V rámci funkcie `app_main` sa najskôr nastaví jas podsvietenia na 100 % a potom sa inicializuje obsah zobrazovania na obrazovke. Nakoniec sa vytvorí vlákno FreeRTOS na spracovanie údajov a obnovovanie obrazovky.

/* BH1750 spracovanie údajov o osvetlení a vlákno na obnovovanie obrazovky `bh1750_read_task` */

```

void bh1750_read_task(void *param)
{
    float lux = 0;
    while (1)
    {
        lux = bh1750_read_data(); /*Read the illuminance data of the BH1750 sensor*/
        if (lux == (-1))
        {
            if (lvgl_port_lock(0))
            {
                lv_label_set_text(bh1750_data, "bh1750 read data error"); /*Read failure message displayed*/
                lvgl_port_unlock();
            }
        }
        else
        {
            MAIN_INFO("lux:%.2f", lux);
            if (lvgl_port_lock(0))
            {
                update_bh1750_value(lux); /*Update the bh1750 data displayed on the screen*/
                lvgl_port_unlock();
            }
        }
        vTaskDelay(1000 / portTICK_PERIOD_MS);
    }
}

```

V rámci vlákna na spracovanie údajov o osvetlení BH1750 a obnovovanie zobrazovania na obrazovke sa najskôr inicializujú premenné. Následne sa vytvorí cyklus `while`. V rámci tejto slučky sa vyvolá funkcia `bh1750_read_data` na získanie údajov o osvetlení. Po úspešnom načítaní sa na obnovenie zobrazovania na obrazovke použije funkcia `update_bh1750_value`. Ak načítanie zlyhá, na obrazovke sa zobrazí hlásenie „bh1750 read data error“ (chyba načítania údajov bh1750). Konečné 1-sekundové oneskorenie zabezpečuje, že údaje sa obnovujú a načítavajú raz za sekundu.

1.3.5 Súbor `CMkaLists.txt`

Aby bolo možné úspešne vyvolať obsah priečinkov `bsp_display` a `bsp_bh1750` v hlavnej funkcii, je potrebné nakonfigurovať súbor `CMakeLists.txt`, ktorý sa nachádza v hlavnom priečinku. Podrobnosti konfigurácie sú nasledovné:

```

FILE(GLOB_RECURSE main ${CMAKE_SOURCE_DIR}/main/*.c)

idf_component_register(SRCS ${main}
                      INCLUDE_DIRS "include"
                      REQUIRES bsp_i2c bsp_display bsp_bh1750)

```

Najskôr sa definujú adresáre pre zdrojové súbory a hlavičkové súbory spolu s požadovanými knižnicami ovládačov – konkrétne knižnice ovládačov pre prepojenie `bsp_display`, `bsp_bh1750` a `bsp_i2c`. Následne sa tieto nastavenia zaregistrujú v systéme zostavovania pomocou príkazu `idf_component_register`, čím sa hlavná funkcia môže využívať tieto funkcie ovládačov.

1.4 Postup programovania

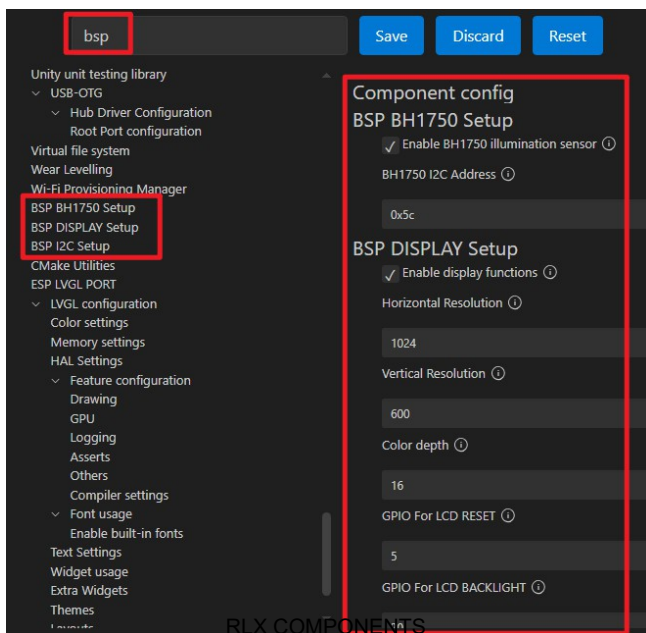
www.rlx.sk

Pripojte zariadenie P4 k počítaču cez USB

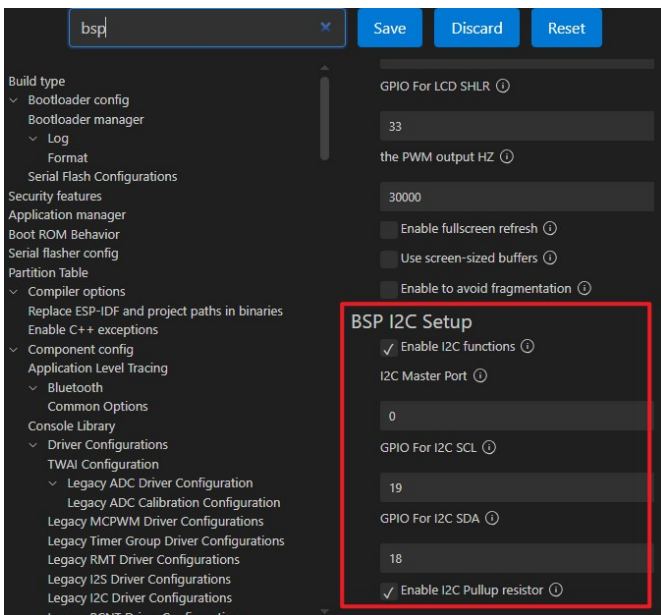
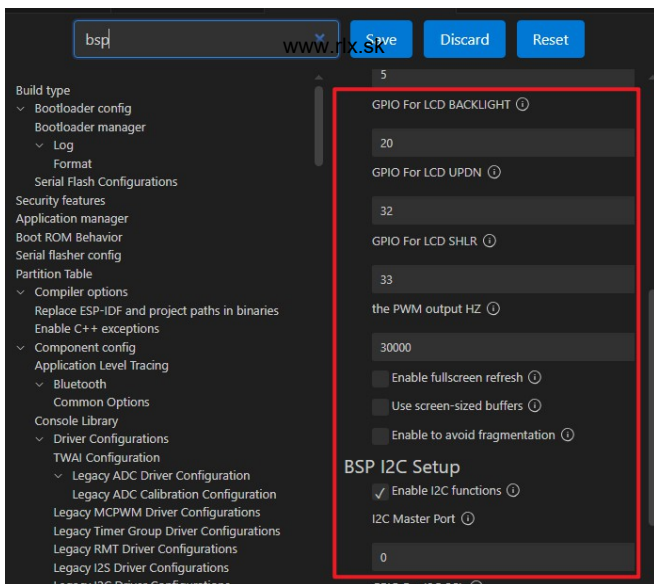


1.4.1 Po klonovaní kódu prostredníctvom Git (odkaz bude potvrdený) vymažte všetky lokálne informácie o kompilácii. Nakonfigurujte prostredie IDF a model čipu pre kompiláciu podľa lekcie 1 a nastavte číslo sériového portu pre programovanie.

1.4.2 Nasledujúca konfigurácia SDKConfig je z veľkej časti identická s lekciovou 1, stačí len prekonfigurovať piny.



RLX COMPONENTS



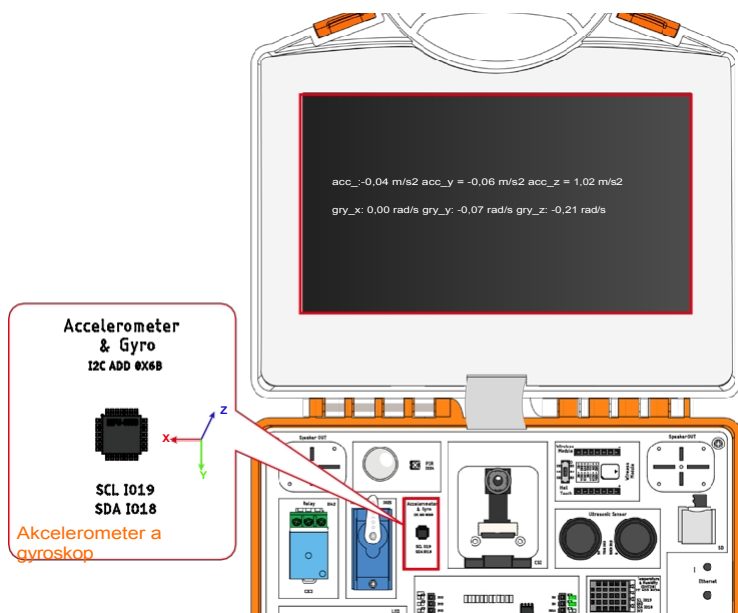
1.4.3 Kliknite na Kompilovať. Po úspešnej kompilácii kliknite na Stiahnuť.

Lekcia 13 – Zobrazenie gyroskopu LSM6DS3

Úvod

Táto kapitola tutoriálu demonštruje rozhranie medzi ESP32-P4 a gyroskopickým senzorom LSM6DS3TR. Prostredníctvom príkladu merania uhlovej rýchlosti pomáha pochopiť základné schopnosti šesťosových inerciálnych senzorov. Ako bežná súčasť detekcie pohybu umožňuje gyroskop čitateľom intuitívne pochopiť aplikácie ESP32-P4 v oblasti riadenia pohybu, odhadu polohy a nositeľných zariadení, čím vytvára základ pre zložitejšie projekty inteligentnej interakcie.

Demonštrácia projektu Efekt



Táto kapitola je rozdelená do nasledujúcich podkapitol

- 1.1 Úvod do senzora LSM6DS3TR
- 1.2 Návrh hardvéru
- 1.3 Návrh softvéru
- 1.4 Stiahnutie a overenie

1.1 Úvod do senzora LSM6DS3TR

www.rlx.sk

1.1.1 Úvod do LSM6DS3TR

LSM6DS3TR je šesťosová inerciálna meracie jednotka (IMU) vyvinutá spoločnosťou STMicroelectronics, ktorá obsahuje integrované:

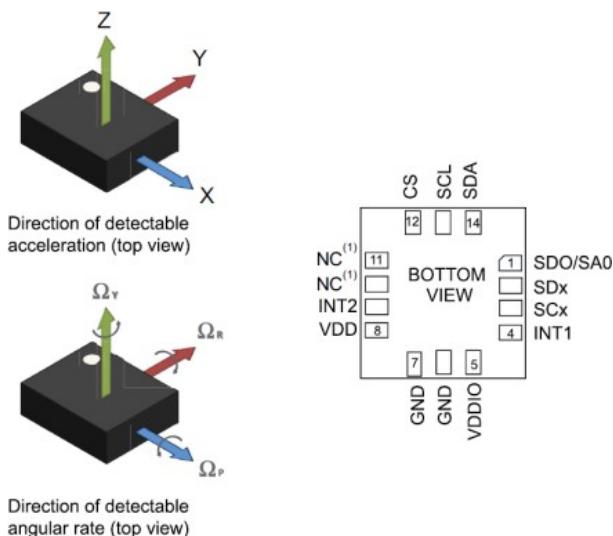
Trojosej akcelerometer ($\pm 2g$ / $\pm 4g$ / $\pm 8g$ / $\pm 16g$ voliteľné)

Trojosej gyroskop (± 125 dps / ± 250 dps / ± 500 dps / ± 1000 dps / ± 2000 dps, voliteľné)

Obsahuje jednotku digitálneho spracovania signálu (DSP) a komunikuje priamo s hostiteľským zariadením prostredníctvom rozhraní I²C alebo SPI.

Kľúčové vlastnosti:

- ① Vysoký výkon: Hustota šumu akcelerometra je len $90 \mu g/\sqrt{Hz}$; hustota šumu gyroskopu je zvyčajne $4 \text{ mdps}/\sqrt{Hz}$;
- ② Nízka spotreba energie: Typický prevádzkový prúd len $0,9 \text{ mA}$ (akcelerometer + gyroskop);
- ③ Vstavaná funkcionlita: Podporuje vyrovnávaciu pamäť FIFO, detekciu chôdze, rozpoznávanie aktivity a detekciu voľného pádu;
- ④ Flexibilná komunikácia: Podporuje I²C ($100 \text{ kHz}/400 \text{ kHz}$) a SPI (až 10 MHz);
- ⑤ Široká použiteľnosť: Vhodný pre drony, mobilné telefóny, fitness trackery, VR/AR headsety, roboty a ďalšie zariadenia.



1.1.2 Ako fungujú gyroskopy

Gyroskop je zariadenie na meranie uhlovej rýchlosti, implementované interne v LSM6DS3TR pomocou mikroeлектromechanických systémov MEMS. Jeho základný princíp funguje takto:

Keď vnútorná mikromechanická štruktúra vibruje, pôsobí na ňu Coriolisova sila;

Pri zmenách uhlovej rýchlosti v zariadení dochádza k nepatrnému posunu vibrujúcej konštrukcie.

Toto posunutie sa prostredníctvom kapacitného snímania prevádza na elektrický signál. Interný ADC a DSP prevádzajú signál na digitálny výstup.

$$F_n = 2mv\omega$$

Where:

- F_n : Coriolis force
- m : Mass
- v : Vibration velocity
- ω : Angular velocity

Gyroskop tak môže v reálnom čase merať hodnoty uhlovej rýchlosti okolo osí X, Y a Z (jednotka: dps, t. j. stupňov za sekundu).

1.1.3 Princíp fungovania akcelerometra

LSM6DS3TR obsahuje aj trojosový akcelerometer na meranie lineárneho zrýchlenia:

Keď zariadenie zaznamená zrýchlenie, jeho vnútorný hmotnostný blok sa posunie; Toto posunutie mení kapacitnú štruktúru;

Výsledný signál je prevádzaný na digitálnu hodnotu pomocou ADC a odosielaný.

Pomocou trojosového akcelerometra je možné vypočítať pohybový stav a orientáciu objektu (napríklad uhol horizontálneho naklonenia).

1.1.4 Výstup a konverzia údajov

Registre LSM6DS3TR ukladajú surové namerané hodnoty (16-bitové binárne doplnenie do dvojky).

Gyroscope Data Calculation Formula

www.rlx.sk

$$\omega \text{ (dps)} = (\text{Raw} \div 32768) \times \text{FS}_{\text{gyro}}$$

Where:

- Raw: Raw register value
- FS_{gyro} : Full-scale range, e.g., ± 250 dps

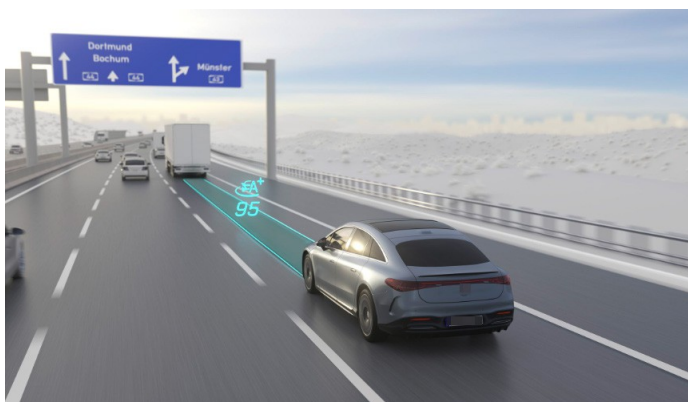
Accelerometer Data Calculation Formula

$$a \text{ (g)} = (\text{Raw} \div 32768) \times \text{FS}_{a\text{cc}}$$

Where:

- $\text{FS}_{a\text{cc}}$: Full-scale range, e.g., ± 2 g

Gyroskopy nachádzajú široké uplatnenie v každodenných predmetoch, ako sú drony a automobily.



1.2 Hardvérový dizajn

Typická konfigurácia zapojenia pre ESP32-P4 a LSM6DS3TR je nasledovná (režim I²C):

ESP32-P4 SDA → LSM6DS3TR SDA

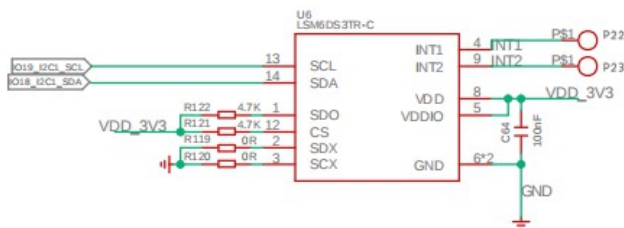
ESP32-P4 SCL → LSM6DS3TR SCL

3,3 V → VCC

GND → GND

Predvolená adresa I²C je 0x6B

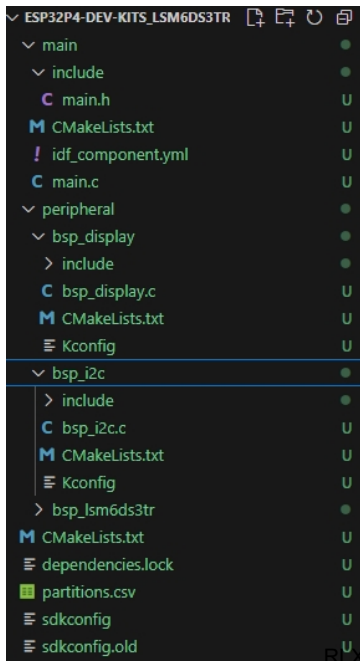
Odporúča sa pripojiť 4,7 kΩ pull-up rezistor k každej z liniek SDA a SCL.



1.3 Analýza programu

<https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design>

Otvorte súbor projektu v VS Code podľa predchádzajúcich pokynov.



V rámci príkladu **ESP32P4-dev-kits_ism6ds3tr** bol v adresári **ESP32P4-dev-kits_ism6ds3tr/peripheral** vytvorený nový priečinok s názvom **bsp_ism6ds3tr**. V rámci cesty **bsp_ism6ds3tr** bol vytvorený nový priečinok **include**, súbor **CMakeLists.txt** a súbor **Kconfig**. Zložka **bsp_ism6ds3tr** obsahuje súbor ovládača **bsp_ism6ds3tr.c**. Zložka **include** obsahuje súbor hlavičky **bsp_ism6ds3tr.h**, zatiaľ čo súbor **CMakeLists.txt** integruje ovládač do systému zostavovania, čím umožňuje projektové využitie jeho funkcionality. Súbor **Kconfig** načíta celý ovládač spolu s definíciami pinov GPIO do súboru **sdkconfig** v rámci platformy IDF (konfigurovateľné prostredníctvom grafického rozhrania).

1.3.1 Kód ovládača LSM6DS3TR-C

www.rlx.sk

Tu sa zameriame výlučne na základný kód, podrobný zdrojový kód nájdete v príslušných zdrojových súboroch pre tento experiment v materiáloch kódu.

Zdrojový kód ovládača LSM6DS3TR pozostáva z dvoch súborov: bsp_lsm6ds3tr.c a bsp_lsm6ds3tr.h.

Najskôr preskúmame súbor bsp_lsm6ds3tr.h: obsahuje relevantné definície pre piny akcelerometra a gyroskopu spolu s deklaráciami funkcií.

/ Odkazy na súbory hlavičiek */*

```
/*-----Header file declaration-----*/
#include "freertos/FreeRTOS.h" //References for FreeRTOS Function-related API Functions
#include "freertos/task.h" //References for FreeRTOS Task Function-related API Functions
#include "esp_log.h" //References for LOG Printing Function-related API Functions
#include "esp_err.h" //References for Error Type Function-related API Functions
#include "bsp_i2c.h"
/*-----Header file declaration end-----*/
```

/ Deklarácie funkcií a deklarácie definícií makrií */*

```
#ifndef CONFIG_BSP_LSM6DS3_SENSOR_ENABLED
#define CONFIG_BSP_LSM6DS3_SENSOR_ENABLED
#endif

#define LSM6DS3_I2C_ADDRESS CONFIG_LSM6DS3_I2C_ADDRESS // The 7-bit I2C address of LSM6DS3TR
#define WHO_AM_I_ADDRESS 0x0f // This register is a read-only register. Its value is fixed at 60h
#define CTRL1_XL_ADDRESS 0x10 // Linear acceleration sensor control register 1
#define CTRL2_G_ADDRESS 0x11 // Angular rate sensor control register 2
#define CTRL3_C_ADDRESS 0x12 // Control register 3
#define CTRL4_C_ADDRESS 0x13 // Control register 4
#define CTRL5_G_ADDRESS 0x14 // Angular rate sensor control register 5
#define CTRL6_G_ADDRESS 0x15 // Angular rate sensor control register 6
#define CTRL7_G_ADDRESS 0x16 // Angular rate sensor control register 7
#define CTRL8_XL_ADDRESS 0x17 // Linear acceleration sensor control register 8
#define STATUS_REG_ADDRESS 0x1E // Status register
#define OUTX_L_G_ADDRESS 0x22 // Angular rate sensor pitch axis (X) angular rate output register(L_Bit)
#define OUTX_H_G_ADDRESS 0x23 // Angular rate sensor pitch axis (X) angular rate output register(H_Bit)
#define OUTY_L_G_ADDRESS 0x24 // Angular rate sensor roll axis (Y) angular rate output register(L_Bit)
#define OUTY_H_G_ADDRESS 0x25 // Angular rate sensor roll axis (Y) angular rate output register(H_Bit)
#define OUTZ_L_G_ADDRESS 0x26 // Angular rate sensor yaw axis (Z) angular rate output register(L_Bit)
#define OUTZ_H_G_ADDRESS 0x27 // Angular rate sensor yaw axis (Z) angular rate output register(H_Bit)
#define OUTX_L_XL_ADDRESS 0x28 // Linear acceleration sensor X-axis output register(L_Bit)
#define OUTX_H_XL_ADDRESS 0x29 // Linear acceleration sensor X-axis output register(H_Bit)
#define OUTY_L_XL_ADDRESS 0x2A // Linear acceleration sensor Y-axis output register(L_Bit)
#define OUTY_H_XL_ADDRESS 0x2B // Linear acceleration sensor Y-axis output register(H_Bit)
#define OUTZ_L_XL_ADDRESS 0x2C // Linear acceleration sensor Z-axis output register(L_Bit)
#define OUTZ_H_XL_ADDRESS 0x2D // Linear acceleration sensor Z-axis output register(H_Bit)
```

```
/*Output data rate and power mode selection*/
#define LSM6DS3TRC_RATE_0 0x00
#define LSM6DS3TRC_RATE_320 0x00
#define LSM6DS3TRC_RATE_320S 0x10
#define LSM6DS3TRC_RATE_204Z 0x20
#define LSM6DS3TRC_RATE_524Z 0x30
#define LSM6DS3TRC_RATE_1040Z 0x40
#define LSM6DS3TRC_RATE_2040Z 0x50
#define LSM6DS3TRC_RATE_4096Z 0x60
#define LSM6DS3TRC_RATE_8194Z 0x70
#define LSM6DS3TRC_RATE_16384Z 0x80
#define LSM6DS3TRC_RATE_32768Z 0x90
#define LSM6DS3TRC_RATE_65536Z 0xA0
/*Accelerometer full-scale selection*/
#define LSM6DS3TRC_ACC_FSXL_2G 0x00
#define LSM6DS3TRC_ACC_FSXL_16G 0x04
#define LSM6DS3TRC_ACC_FSXL_4G 0x08
#define LSM6DS3TRC_ACC_FSXL_8G 0x0C
/*Gyroscope full-scale selection*/
#define LSM6DS3TRC_GVR_FSG_245 0x00
#define LSM6DS3TRC_GVR_FSG_500 0x04
#define LSM6DS3TRC_GVR_FSG_1000 0x08
#define LSM6DS3TRC_GVR_FSG_2000 0x0C
/*Anti-aliasing filter bandwidth selection*/
#define LSM6DS3TRC_ACC_BW0XL_1500HZ 0x00
#define LSM6DS3TRC_ACC_BW0XL_400HZ 0x01
/*Accelerometer bandwidth selection*/
#define LSM6DS3TRC_CTRL4_DRDY_XL_EN_DISABLE 0x00
#define LSM6DS3TRC_CTRL4_DRDY_XL_EN_ENABLE 0x00
/*Gyroscope sleep mode enable*/
#define LSM6DS3TRC_CTRL4_SLEEP_ENABLE 0x00
#define LSM6DS3TRC_CTRL4_SLEEP_DISABLE 0x00
/*All interrupt signals available on INT1 pin enable*/
#define LSM6DS3TRC_CTRL4_DRDY_INT1_DISABLE 0x00
#define LSM6DS3TRC_CTRL4_DRDY_INT1_ENABLE 0x20
/*Data-ready mask enable*/
#define LSM6DS3TRC_CTRL4_DRDY_MASK_DISABLE 0x00
#define LSM6DS3TRC_CTRL4_DRDY_MASK_ENABLE 0x08
```

RLX COMPONENTS

```

/*Disable I2C interface*/
#define LSM6DS3TRC_CTRL4_I2C_DISABLE 0x04
#define LSM6DS3TRC_CTRL4_I2C_ENABLE 0x00
/* Enable gyroscope digital LPF1. The bandwidth can be selected through */
#define LSM6DS3TRC_CTRL4_LPF1_SELG_ENABLE 0x02
#define LSM6DS3TRC_CTRL4_LPF1_SELG_DISABLE 0x00

#define LSM6DS3TRC_CTRL6_C_EDGE_TRIGGER 0x00 // Gyroscope data edge-sensitive trigger enable
#define LSM6DS3TRC_CTRL6_C_LEVEL_TRIGGER 0x40 // Gyroscope data level-sensitive trigger enable
#define LSM6DS3TRC_CTRL6_C_LEVEL_LATCHED 0x00 // Gyroscope level-sensitive latched enable.
#define LSM6DS3TRC_CTRL6_C_LEVEL_FIFO 0xc0 // LSM6DS3TRC_CTRL6_C_EDGE_TRIGGER|LSM6DS3TRC_CTRL6_C_LEVEL_TRIGGER
/* High-performance operating mode disable for accelerometer*/
#define LSM6DS3TRC_CTRL6_C_XL_HP_MODE_ENABLE 0x00
#define LSM6DS3TRC_CTRL6_C_XL_HP_MODE_DISABLE 0x10
/* Gyroscope's low-pass filter (LPF1) bandwidth selection */
#define LSM6DS3TRC_CTRL6_C_FTYPE_1 0x00
#define LSM6DS3TRC_CTRL6_C_FTYPE_2 0x01
#define LSM6DS3TRC_CTRL6_C_FTYPE_3 0x02
#define LSM6DS3TRC_CTRL6_C_FTYPE_4 0x03
/*High-performance operating mode*/
#define LSM6DS3TRC_CTRL7_G_HP_MODE_ENABLE 0x00
#define LSM6DS3TRC_CTRL7_G_HP_MODE_DISABLE 0x00
/*Gyroscope digital high-pass filter enable*/
#define LSM6DS3TRC_CTRL7_G_HP_EN_DISABLE 0x00
#define LSM6DS3TRC_CTRL7_G_HP_EN_ENABLE 0x40
/*Gyroscope digital HP filter cutoff selection*/
#define LSM6DS3TRC_CTRL7_G_HPM_16MHZ 0x00
#define LSM6DS3TRC_CTRL7_G_HPM_65MHZ 0x10
#define LSM6DS3TRC_CTRL7_G_HPM_260Hz 0x20
#define LSM6DS3TRC_CTRL7_G_HPM_1Hz04 0x30
/*Source register rounding function enable*/
#define LSM6DS3TRC_CTRL7_G_ROUNDING_STATUS_DISABLE 0x04
#define LSM6DS3TRC_CTRL7_G_ROUNDING_STATUS_ENABLE 0x00

```

```

/*Accelerometer ODR register setting*/
#define LSM6DS3TRC_ACC_LOW_PASS_ODR_50 0x08
#define LSM6DS3TRC_ACC_LOW_PASS_ODR_100 0x0A
#define LSM6DS3TRC_ACC_LOW_PASS_ODR_9 0xc0
#define LSM6DS3TRC_ACC_LOW_PASS_ODR_400 0xE8

#define LSM6DS3TRC_ACC_HIGH_PASS_ODR_50 0x04
#define LSM6DS3TRC_ACC_HIGH_PASS_ODR_100 0x24
#define LSM6DS3TRC_ACC_HIGH_PASS_ODR_9 0x44
#define LSM6DS3TRC_ACC_HIGH_PASS_ODR_400 0x64

#define LSM6DS3TRC_STATUS_GYROSCOPE 0x02 // Gyroscope new data available
#define LSM6DS3TRC_STATUS_ACCELEROMETER 0x01 // Accelerometer new data available

typedef struct
{
    float acc_x;
    float acc_y;
    float acc_z;
    float gry_x;
    float gry_y;
    float gry_z;
} lsm6ds3tr; // lsm6ds3tr data structure

extern lsm6ds3tr my_lsm6ds3;
esp_err_t lsm6ds3_begin(void); // Initialization of lsm6ds3tr sensor
esp_err_t lsm6ds3_get_acc(uint8_t fval); // lsm6ds3 acquires acceleration data
esp_err_t lsm6ds3_get_gry(uint8_t fag); // lsm6ds3 acquires gyroscope data
esp_err_t lsm6ds3_scan(); // The lsm6ds3 scans to determine whether the currently read data is from the gyroscope or the accelerometer
#endif

```

Tu viacero makier definuje rôzne adresy registrov a operačné príkazy pre senzor.

Ďalej budeme analyzovať program v bsp_Lsm6ds3tr.c: inicializácia konfigurácie senzora a sprístupnenie funkcií rozhrania API.

/* Inicializačná funkcia LSM6DS3TR lsm6ds3_begin */

```

esp_err_t lsm6ds3_begin(void)
{
    esp_err_t err = ESP_OK;
    lsm6ds3_handle = i2c_dev_register(LSM6DS3_I2C_ADDRESS); /*lsm6ds3 initialization*/
    if (lsm6ds3_handle != NULL)
    {
        err = lsm6ds3_getchipID(); /*lsm6ds3 obtains ID information*/
        if (err != ESP_OK)
            return err;
        err = lsm6ds3_reset(); /*Reset the sensor of lsm6ds3*/
        if (err != ESP_OK)
            return err;
        err = lsm6ds3_set_BDU(true); /*Set BDU for lsm6ds3*/
        if (err != ESP_OK)
            return err;
        err = lsm6ds3_set_acc_rate(LSM6DS3TRC_RATE_120HZ); /*Set the rate of the accelerometer for lsm6ds3*/
        if (err != ESP_OK)
            return err;
        err = lsm6ds3_set_gyr_rate(LSM6DS3TRC_RATE_120HZ); /*Set the rate of the gyroscope for lsm6ds3*/
        if (err != ESP_OK)
            return err;
        err = lsm6ds3_set_acc_fullscale(LSM6DS3TRC_ACC_FSR_16G); /*Set the full-scale of the accelerometer for lsm6ds3*/
        if (err != ESP_OK)
            return err;
        err = lsm6ds3_set_gyr_fullscale(LSM6DS3TRC_GVR_FSR_2000); /*Set the full-scale of the gyroscope for lsm6ds3*/
        if (err != ESP_OK)
            return err;
        err = lsm6ds3_set_acc_bandwidth(LSM6DS3TRC_ACC_BW_400HZ, LSM6DS3TRC_ACC_LOWPASS_ODR_100); /*Set the bandwidth of the accelerometer for lsm6ds3*/
        if (err != ESP_OK)
            return err;
        err = lsm6ds3_set_gyr_register4(LSM6DS3TRC_CTRL4_LPFS_SEL_ENABLE); /*Set the gyroscope register 4 of lsm6ds3*/
        if (err != ESP_OK)
            return err;

        return err;
    }
    else
    {
        err = ESP_FAIL;
        return err;
    }
    return err;
}

```

```

        return err;
    }
    else
    {
        err = ESP_FAIL;
        return err;
    }
    return err;
}

```

V rámci funkcie **lsm6ds3_begin** sa senzor **lsm6ds3tr** najskôr zaregistruje na zbernici I2C pomocou funkcie **i2c_dev_register**, ktorá vráti identifikátor operácie. Následne, ak je vrátený identifikátor zariadenia nenulový, vyvolajú sa rôzne konfigurácie registrov na inicializáciu senzora **lsm6ds3tr**. Špecifická funkčnosť každej funkcie bude podrobne analyzovaná nižšie.

/ Funkcia na čítanie ID zariadenia LSM6DS3TR: **lsm6ds3_getchipID** */*

```

static esp_err_t lsm6ds3_getchipID(void)
{
    esp_err_t err = ESP_OK;
    uint8_t buf = 0;
    err = i2c_read_reg(lsm6ds3_handle, WHO_AM_I_ADDRESS, &buf, 1); /*I2C Read read-only register*/
    if (err != ESP_OK)
        return err;
    if (buf != 0x6A) /*Check if the id is 0x6A*/
    {
        err = ESP_FAIL;
        return err;
    }
    return err;
}

```

Najskôr použite funkciu **i2c_read_reg** na načítanie adresy registra ID a získanie údajov ID. Určite, či je to 0x6A (predvolená hodnota pre tento register musí byť 0x6A). Ak áno, vráťte **ESP_OK**.

/ Funkcia **lsm6ds3_reset** na resetovanie zariadenia LSM6DS3TR */*

```

esp_err_t lsm6ds3_reset(void)
{
    esp_err_t err = ESP_OK;
    uint8_t buf[1] = {0};
    buf[0] = 0x80;
    err = i2c_write_reg(lsm6ds3_handle, CTRL3_C_ADDRESS, buf[0]); /*Reboot memory content*/
    if (err != ESP_OK)
        return err;
    vTaskDelay(15 / portTICK_PERIOD_MS);
    err = i2c_read_reg(lsm6ds3_handle, CTRL3_C_ADDRESS, buf, 1); /*I2C Read register status*/
    if (err != ESP_OK)
        return err;
    buf[0] |= 0x01;
    err = i2c_write_reg(lsm6ds3_handle, CTRL3_C_ADDRESS, buf[0]); /*Software reset*/
    if (err != ESP_OK)
        return err;
    while (buf[0] & 0x01) /*(0: normal mode; 1: reset device)*/
    {
        i2c_read_reg(lsm6ds3_handle, CTRL3_C_ADDRESS, buf, 1); /*I2C Read register status*/
        vTaskDelay(15 / portTICK_PERIOD_MS);
    }
    return err;
}

```

Najskôr použite funkciu `i2c_write_reg` na zapísanie hodnoty 0x80 do adresy registra 3 radiča. Tento krok vynuluje obsah pamäte. Po 15 ms oneskorení prečítajte údaje z adresy registra 3; výsledok by mal byť 0x01, čo znamená nízky bit 1. Následne použite funkciu `i2c_write_reg` na zapísanie tejto hodnoty, čím aktivujete funkciu softvérového vynulovania. Nakoniec vytvorte slučku, aby ste zistili, či bola funkcia softvérového resetu dokončená (nízky bit je resetovaný na 0).

/* Funkcia `lsm6ds3_set_BDU` na konfiguráciu BDU na `lsm6ds3tr` */

```

esp_err_t lsm6ds3_set_BDU(bool state)
{
    esp_err_t err = ESP_OK;
    uint8_t buf[1] = {0};
    err = i2c_read_reg(lsm6ds3_handle, CTRL3_C_ADDRESS, buf, 1); /*I2C Read register status*/
    if (err != ESP_OK)
        return err;
    if (state == true)
    {
        buf[0] |= 0x40;
        err = i2c_write_reg(lsm6ds3_handle, CTRL3_C_ADDRESS, buf[0]); /*1: output registers not updated until MSB and LSB have been read*/
        if (err != ESP_OK)
            return err;
    }
    else
    {
        buf[0] &= 0xBF;
        err = i2c_write_reg(lsm6ds3_handle, CTRL3_C_ADDRESS, buf[0]); /*0: continuous update*/
        if (err != ESP_OK)
            return err;
    }

    err = i2c_read_reg(lsm6ds3_handle, CTRL3_C_ADDRESS, buf, 1); /*I2C Read register status*/
    if (err != ESP_OK)
        return err;
    return err;
}

```

Najskôr použite funkciu `i2c_read_reg` na načítanie adresy registra 3 radiča. Tento krok zachová pôvodné nastavenia a zmení iba konfiguráciu parametrov BDU. Na základe vstupného boolovského parametra, ak je pravdivý, použite funkciu `i2c_write_reg` na nastavenie kontrolného bitu 1 BDU. Tým sa zabráni aktualizácii výstupného registra pred načítaním najvýznamnejšieho bitu a najmenej významného bitu. Ak je parameter nepravdivý, funkcia `i2c_write_reg` sa použije na nastavenie kontrolného bitu BDU na 0, čím sa zabezpečí, že výstupný register bude pokračovať v aktualizácii.

/* Funkcia `lsm6ds3_set_acc_rate` na konfiguráciu výstupnej rýchlosti akcelerometra na `LSM6DS3TR` */

RLX COMPONENTS

```

esp_err_t lsm6ds3_set_acc_rate(uint8_t rate)
{
    esp_err_t err = ESP_OK;
    uint8_t buf[1] = {0};
    err = i2c_read_reg(lsm6ds3_handle, CTRL1_XL_ADDRESS, buf, 1); /*I2C Read register status*/
    if (err != ESP_OK)
        return err;
    buf[0] |= rate;
    err = i2c_write_reg(lsm6ds3_handle, CTRL1_XL_ADDRESS, buf[0]); /*Accelerometer Output data rate*/
    if (err != ESP_OK)
        return err;
    return err;
}

```

www.rlx.sk

Najskôr použite funkciu `i2c\_read\_reg` na prečítanie adresy registra radiča 1. Tento krok zachová pôvodné nastavenia a zmení iba konfiguráciu parametrov pre výstupnú rýchlosť akcelerometra. Na základe vstupných parametrov

sa nakonfigurované údaje zapíšu do adresy registra radiča 1 prostredníctvom funkcie `i2c\_write\_reg`.

/* Funkcia lsm6ds3_set_acc_fullscale na konfiguráciu maximálneho rozsahu akcelerometra na LSM6DS3TR */

```

esp_err_t lsm6ds3_set_acc_fullscale(uint8_t value)
{
    esp_err_t err = ESP_OK;
    uint8_t buf[1] = {0};
    err = i2c_read_reg(lsm6ds3_handle, CTRL1_XL_ADDRESS, buf, 1);
    if (err != ESP_OK)
        return err;
    buf[0] |= value;
    err = i2c_write_reg(lsm6ds3_handle, CTRL1_XL_ADDRESS, buf[0]); /*Accelerometer full-scale selection*/
    if (err != ESP_OK)
        return err;
    return err;
}

```

Najskôr použite funkciu `i2c\_read\_reg` na načítanie adresy registra radiča 1. Tento krok zachová pôvodné nastavenia a zmení iba konfiguračný parameter pre maximálny rozsah akcelerometra. Na základe vstupných parametrov

sa špecifikované údaje zapíšu do adresy registra radiča 1 prostredníctvom funkcie `i2c\_write\_reg`.

/* Funkcia lsm6ds3_set_gyr_rate na konfiguráciu výstupnej rýchlosti gyroskopu na LSM6DS3TR */

```

esp_err_t lsm6ds3_set_gyr_rate(uint8_t rate)
{
    esp_err_t err = ESP_OK;
    uint8_t buf[1] = {0};
    err = i2c_read_reg(lsm6ds3_handle, CTRL2_G_ADDRESS, buf, 1);
    if (err != ESP_OK)
        return err;
    buf[0] |= rate;
    err = i2c_write_reg(lsm6ds3_handle, CTRL2_G_ADDRESS, buf[0]); /*Gyroscope Output data rate*/
    if (err != ESP_OK)
        return err;
    return err;
}

```

Najskôr použite funkciu `i2c\_read\_reg` na prečítanie adresy registra radiča 2. Tento krok zachová pôvodné nastavenia a zmení iba konfiguráciu výstupnej rýchlosti gyroskopu.

Na základe vstupných parametrov

sa špecifikované údaje zapisujú do adresy registra radiča 2 prostredníctvom funkcie `i2c\_write\_reg`.

/* Funkcia lsm6ds3_set_gyr_fullscale na nastavenie plného rozsahu gyroskopu na LSM6DS3TR */

www.rlx.sk

```
esp_err_t lsm6ds3_set_gyr_fullscale(uint8_t value)
{
    esp_err_t err = ESP_OK;
    uint8_t buf[1] = {0};
    err = i2c_read_reg(lsm6ds3_handle, CTRL2_G_ADDRESS, buf, 1);
    if (err != ESP_OK)
        return err;
    buf[0] |= value;
    err = i2c_write_reg(lsm6ds3_handle, CTRL2_G_ADDRESS, buf[0]); /*Gyroscope full-scale selection*/
    if (err != ESP_OK)
        return err;
    return err;
}
```

Najskôr použite funkciu `i2c\_read\_reg` na načítanie adresy registra radiča 2. Tento krok zachová pôvodné nastavenia a zmení iba konfiguráciu parametra maximálneho rozsahu gyroskopu. Na základe vstupných parametrov

nakonfigurované údaje sa zapisujú do adresy registra radiča 2 prostredníctvom funkcie `i2c\_write\_reg`.

/* Funkcia lsm6ds3_set_acc_bandwidth na konfiguráciu šírky pásma akcelerometra na LSM6DS3TR */

```
esp_err_t lsm6ds3_set_acc_bandwidth(uint8_t BWBWL, uint8_t ODR)
{
    esp_err_t err = ESP_OK;
    uint8_t buf[1] = {0};
    err = i2c_read_reg(lsm6ds3_handle, CTRL1_XL_ADDRESS, buf, 1);
    if (err != ESP_OK)
        return err;
    buf[0] |= BWBWL;
    err = i2c_write_reg(lsm6ds3_handle, CTRL1_XL_ADDRESS, buf[0]); /*Anti-aliasing filter bandwidth selection*/
    if (err != ESP_OK)
        return err;

    err = i2c_read_reg(lsm6ds3_handle, CTRL8_XL_ADDRESS, buf, 1);
    if (err != ESP_OK)
        return err;
    buf[0] |= ODR;
    err = i2c_write_reg(lsm6ds3_handle, CTRL8_XL_ADDRESS, buf[0]); /*Accelerometer ODR register setting*/
    if (err != ESP_OK)
        return err;
    return err;
}
```

Táto funkcia má dva parametre, ktoré spoločne určujú nastavenia šírky pásma akcelerometra. Nachádzajú sa v

registri radiča 1 a registri radiča 8.

/* Funkcia lsm6ds3_set_gry_register4 na nastavenie registra gyroskopu 4 na lsm6ds3tr */

```
esp_err_t lsm6ds3_set_gry_register4(uint8_t cmd)
{
    esp_err_t err = ESP_OK;
    uint8_t buf[1] = {0};

    err = i2c_read_reg(lsm6ds3_handle, CTRL4_C_ADDRESS, buf, 1);
    if (err != ESP_OK)
        return err;
    buf[0] |= cmd;
    err = i2c_write_reg(lsm6ds3_handle, CTRL4_C_ADDRESS, buf[0]); /*Enable gyroscope digital LPF1 (0: disabled; 1: enabled)*/
    if (err != ESP_OK)
        return err;
    return err;
}
```


Táto funkcia aktivuje alebo deaktivuje filter nastavením parametra filtra LPF1 v registri regulátora 4 (0: deaktivovaný, 1: aktivovaný) www.rlx.sk

/* lsm6ds3_set_gry_register6: Funkcia na konfiguráciu registra 6 gyroskopického ovládača pre LSM6DS3TR */

```
esp_err_t lsm6ds3_set_gry_register6(uint8_t cmd)
{
    esp_err_t err = ESP_OK;
    uint8_t buf[1] = {0};
    err = i2c_read_reg(lsm6ds3_handle, CTRL6_C_ADDRESS, buf, 1);
    if (err != ESP_OK)
        return err;
    buf[0] |= cmd;
    err = i2c_write_reg(lsm6ds3_handle, CTRL6_C_ADDRESS, buf[0]); /*this bit must be set to '0' for the correct operation of the device.*/
    if (err != ESP_OK)
        return err;
    return err;
}
```

Táto funkcia nastavuje register regulátora 6, ktorý musí byť nakonfigurovaný na hodnotu 0 podľa požiadaviek uvedených v technickom liste.

/* lsm6ds3_set_gry_register7: Funkcia na konfiguráciu registru 7 gyroskopického regulátora pre LSM6DS3TR */

```
esp_err_t lsm6ds3_set_gry_register7(uint8_t cmd)
{
    esp_err_t err = ESP_OK;
    uint8_t buf[1] = {0};
    err = i2c_read_reg(lsm6ds3_handle, CTRL7_C_ADDRESS, buf, 1);
    if (err != ESP_OK)
        return err;
    buf[0] |= cmd;
    err = i2c_write_reg(lsm6ds3_handle, CTRL7_C_ADDRESS, buf[0]); /*Gyroscope digital high-pass filter enable and gyroscope digital HP filter cutoff selection 260Hz*/
    if (err != ESP_OK)
        return err;
    return err;
}
```

Táto funkcia aktivuje alebo deaktivuje vysokopásmový filter gyroskopu a vyberá jeho medznú frekvenciu konfiguráciou parametrov v registri radiča 7.

/* Funkcia lsm6ds3_get_status načíta stavový register z lsm6ds3tr */

```
static uint8_t lsm6ds3_get_status(void)
{
    uint8_t buf[1] = {0};
    i2c_read_reg(lsm6ds3_handle, STATUS_REG_ADDRESS, buf, 1); /*I2C Read register status*/
    return buf[0];
}
```

Vyvolaním funkcie i2c_read_reg sa prečíta hodnota registra stavu senzora a vráti sa (určí sa, či je aktuálny stav určený na čítanie aktualizácií údajov akcelerometra alebo aktualizácií údajov gyroskopu).

/* Funkcia lsm6ds3_data_read na čítanie výstupného registra LSM6DS3TR */

```
/*I2C reads the registers, performing a cyclic reading process*/
static esp_err_t lsm6ds3_data_read(uint8_t reg_addr, uint8_t *rev_data, uint8_t length)
{
    esp_err_t err = ESP_OK;
    while (length) /*Read the length*/
    {
        err = i2c_read_reg(lsm6ds3_handle, reg_addr++, rev_data++, 1); /*I2C Read register data*/
        if (err != ESP_OK)
            return err;
        length--;
        vTaskDelay(10 / portTICK_PERIOD_MS);
    }
    return err;
}
```

Zadaním parametrov ako podmienok pre vykonanie slučky. V rámci slučky sa vyvolá funkcia `i2c_read_reg` na čítanie hodnôt z výstupného registra senzora (po každom čítaní sa zvýši adresa registra a adresa prijímacej vyrovnávacej pamäte, aby sa dosiahlo cyklické čítanie; následne sa zníži dĺžka čítaných údajov, aby sa určilo, kedy sa má slučka ukončiť).

/* Funkcia `lsm6ds3_get_acc` na čítanie parametrov akcelerometra z `lsm6ds3tr` */

```
esp_err_t lsm6ds3_get_acc(uint8_t fsk1)
{
    esp_err_t err = ESP_OK;
    uint8_t buf[6];
    int16_t acc[3];
    err = lsm6ds3_data_read(OUTX1_XL_ADDRESS, buf, 6); /*Read the value stored in the accelerometer register*/
    if (err != ESP_OK)
        return err;
    else
    {
        acc[0] = buf[1] << 8 | buf[0];
        acc[1] = buf[3] << 8 | buf[2];
        acc[2] = buf[5] << 8 | buf[4];
        switch (fsk1) /*Based on setting different full-scale parameters, convert the parameters returned by the accelerometer*/
        {
            case LSM6DS3TRC_ACC_FSK1_2G:
                my_lsm6ds3_acc_x = ((float)acc[0] * 0.001f) / 1000.0f;
                my_lsm6ds3_acc_y = ((float)acc[1] * 0.001f) / 1000.0f;
                my_lsm6ds3_acc_z = ((float)acc[2] * 0.001f) / 1000.0f;
                break;
            case LSM6DS3TRC_ACC_FSK1_4G:
                my_lsm6ds3_acc_x = ((float)acc[0] * 0.122f) / 1000.0f;
                my_lsm6ds3_acc_y = ((float)acc[1] * 0.122f) / 1000.0f;
                my_lsm6ds3_acc_z = ((float)acc[2] * 0.122f) / 1000.0f;
                break;
            case LSM6DS3TRC_ACC_FSK1_8G:
                my_lsm6ds3_acc_x = ((float)acc[0] * 0.244f) / 1000.0f;
                my_lsm6ds3_acc_y = ((float)acc[1] * 0.244f) / 1000.0f;
                my_lsm6ds3_acc_z = ((float)acc[2] * 0.244f) / 1000.0f;
                break;
            case LSM6DS3TRC_ACC_FSK1_16G:
                my_lsm6ds3_acc_x = ((float)acc[0] * 0.488f) / 1000.0f;
                my_lsm6ds3_acc_y = ((float)acc[1] * 0.488f) / 1000.0f;
                my_lsm6ds3_acc_z = ((float)acc[2] * 0.488f) / 1000.0f;
                break;
        }
    }
    return err;
}
```

Funkcia `lsm6ds3_data_read` načítava údaje z akcelerometra. Vstupné parametre určujú, aká metóda konverzie sa má použiť na načítané údaje (na základe nastaveného parametra maximálneho rozsahu). Konkrétny vzorec konverzie nájdete v príručke k údajom senzora.

/* Funkcia `lsm6ds3_get_gry` na čítanie parametrov gyroskopu z `lsm6ds3tr` */

```
esp_err_t lsm6ds3_get_gry(uint8_t fsg)
{
    esp_err_t err = ESP_OK;
    uint8_t buf[6];
    int16_t gry[3];
    err = lsm6ds3_data_read(OUTX1_G_ADDRESS, buf, 6); /*Read the values stored in the gyroscope register*/
    if (err != ESP_OK)
        return err;
    else
    {
        gry[0] = buf[1] << 8 | buf[0];
        gry[1] = buf[3] << 8 | buf[2];
        gry[2] = buf[5] << 8 | buf[4];
        switch (fsg) /*Based on setting different full-scale parameters, convert the parameters returned by the gyroscope*/
        {
            case LSM6DS3TRC_GVR_FSG_245:
                my_lsm6ds3_gry_x = ((float)gry[0] * 0.750f) / 1000.0f;
                my_lsm6ds3_gry_y = ((float)gry[1] * 0.750f) / 1000.0f;
                my_lsm6ds3_gry_z = ((float)gry[2] * 0.750f) / 1000.0f;
                break;
            case LSM6DS3TRC_GVR_FSG_500:
                my_lsm6ds3_gry_x = ((float)gry[0] * 17.50f) / 1000.0f;
                my_lsm6ds3_gry_y = ((float)gry[1] * 17.50f) / 1000.0f;
                my_lsm6ds3_gry_z = ((float)gry[2] * 17.50f) / 1000.0f;
                break;
            case LSM6DS3TRC_GVR_FSG_1000:
                my_lsm6ds3_gry_x = ((float)gry[0] * 35.00f) / 1000.0f;
                my_lsm6ds3_gry_y = ((float)gry[1] * 35.00f) / 1000.0f;
                my_lsm6ds3_gry_z = ((float)gry[2] * 35.00f) / 1000.0f;
                break;
            case LSM6DS3TRC_GVR_FSG_2000:
                my_lsm6ds3_gry_x = ((float)gry[0] * 70.00f) / 1000.0f;
                my_lsm6ds3_gry_y = ((float)gry[1] * 70.00f) / 1000.0f;
                my_lsm6ds3_gry_z = ((float)gry[2] * 70.00f) / 1000.0f;
                break;
        }
    }
    return err;
}
```

Údaje z gyroskopu sa čítajú prostredníctvom funkcie `lsm6ds3_data_read`. Vstupné parametre určujú, aká metóda konverzie sa má použiť na načítané údaje (na základe nastaveného parametra maximálneho rozsahu). Konkrétny konverzný vzorec nájdete v príručke k údajom senzora.

/* Funkcia `lsm6ds3_scan` na skenovanie a čítanie `lsm6ds3tr` */

```
esp_err_t lsm6ds3_scan()
{
    esp_err_t err = ESP_FAIL;
    uint8_t status;
    status = lsm6ds3_get_status();
    if (status & LSM6DS3TRC_STATUS_ACCELEROMETER) /*Accelerometer new data available status*/
    {
        err = lsm6ds3_get_acc(LSM6DS3TRC_ACC_FSXL_2G);
        if (err != ESP_OK)
            return err;
    }

    if (status & LSM6DS3TRC_STATUS_GYROSCOPE) /*Gyroscope new data available status*/
    {
        err = lsm6ds3_get_gry(LSM6DS3TRC_GYR_FSG_2000);
        if (err != ESP_OK)
            return err;
    }

    return err;
}
```

Načítanie aktuálneho stavu aktualizácie senzora prostredníctvom funkcie `lsm6ds3_get_status` a určenie, či sa majú načítať údaje z akcelerometra alebo gyroskopu na základe stavu.

1.3.2 Súbor `Kconfig`

Hlavnou funkciou tohto súboru je začleniť požadovanú konfiguráciu do súboru `sdkconfig`, čím sa umožní vykonávať určité úpravy parametrov prostredníctvom grafického rozhrania. Tu `0x6B` označuje 7-bitovú adresu senzora `LSM6DS3TR-C`.

```
menu "BSP LSM6DS3 Setup"
    config BSP_LSM6DS3_SENSOR_ENABLED
        bool "Enable LSM6DS3TR sensor"
        depends on BSP_I2C_ENABLED

    if BSP_LSM6DS3_SENSOR_ENABLED
        config LSM6DS3_I2C_ADDRESS
            hex "LSM6DS3TR I2C Address"
            default 0x6B
    endif
endmenu
```

1.3.3 Súbor `CMkaLists.txt`

Funkčnosť tejto ukážkovej rutiny závisí predovšetkým od ovládača `bsp_lsm6ds3tr`. Aby bolo možné úspešne vyvolať obsah priečinka `bsp_lsm6ds3tr` z iných funkcií, je potrebné nakonfigurovať súbor `CMakeLists.txt` v priečinku `bsp_lsm6ds3tr`. Podrobnosti konfigurácie sú nasledovné:

```
FILE(GLOB_RECURSE component_sources "*.c")

idf_component_register(SRCS ${component_sources}
    INCLUDE_DIRS "include"
    REQUIRES bsp_i2c)
```

V tomto súbore `CMakeLists.txt` sú najskôr definované adresáre pre zdrojové súbory a hlavičkové súbory spolu s požadovanou knižnicou ovládača (`bsp_i2c`). Následne sa tieto nastavenia zaregistrujú v systéme zostavovania pomocou príkazu `idf_component_register`, čím sa projektu umožní využívať funkčnosť ovládača `bsp_lsm6ds3tr`.

1.3.4 Hlavný priečink

www.rlx.sk

Hlavný priečink slúži ako hlavný adresár pre spustanie programu a obsahuje hlavnú spustiteľnú funkciu main.c a hlavičkový súbor main.h v priečinku include. Pridajte hlavný priečink do súboru CMakeLists.txt v rámci systému zostavovania.

Súbor main.h odkazuje predovšetkým na požadované hlavičkové súbory: funkcie využívajúce ovládač bsp_display vyžadujú zaradenie hlavičkového súboru bsp_display, zatiaľ čo funkcie využívajúce ovládač bsp_lsm6ds3tr vyžadujú hlavičkový súbor bsp_lsm6ds3tr.

Nižšie je analýza programu main.c: inicializácia systému a vykonávanie funkcií pre I2C, senzor lsm6ds3tr-c a funkcie displeja.

```
#ifndef CONFIG_BSP_I2C_ENABLED
    err = i2c_init(); /*I2C Initialization*/
    if (err != ESP_OK)
    {
        init_fail("i2c", err);
        vTaskDelay(200 / portTICK_PERIOD_MS);
    }
#endif
#ifdef CONFIG_BSP_LSM6DS3_SENSOR_ENABLED
    err = lsm6ds3_begin(); /*Lsm6ds3tr Initialization*/
    if (err != ESP_OK)
    {
        init_fail("lsm6ds3tr", err);
    }
#endif
#ifdef CONFIG_BSP_DISPLAY_ENABLED
    err = display_init(); /*Display Initialization*/
    if (err != ESP_OK)
    {
        init_fail("display", err);
    }
#endif
```

Tento kód sa nachádza vo funkcii init, ktorá slúži na ukladanie inicializačných funkcií vyžadujúcich vyvolanie a hodnotenie výsledku takejto inicializácie. Ak sa stav návratu odchyľuje od ESP_OK, kód vypíše chybovú správu a ukončí ďalšie vykonávanie.

/* Inicializácia obrazovky a funkcia zobrazenia lsm6ds3_display */

```
void lsm6ds3_display()
{
    if (lvgl_port_lock(0))
    {
        lsm6ds3_acc_data = lv_label_create(lv_scr_act()); /*Create a label object*/
        static lv_style_t acc_label_style;
        lv_style_init(&acc_label_style);
        lv_style_set_bg_opa(&acc_label_style, LV_OPA_TRANSP); /*Initialize a style*/
        lv_style_set_bg_color(&acc_label_style, LV_COLOR_BLACK); /*Set the style LVGL background color*/
        lv_obj_add_style(lsm6ds3_acc_data, &acc_label_style, LV_PART_MAIN); /*Add a style to an object*/
        lv_obj_set_style_text_color(lsm6ds3_acc_data, LV_COLOR_WHITE, LV_PART_MAIN); /*Set the style LVGL text color*/
        lv_obj_set_style_text_font(lsm6ds3_acc_data, &lv_font_montserrat_30, LV_PART_MAIN); /*Set the style LVGL text font*/
        lv_obj_align(lsm6ds3_acc_data, LV_ALIGN_CENTER, 0, -20); /*Change the alignment of an object*/
        lv_obj_set_style_bg_color(lv_scr_act(), LV_COLOR_BLACK, LV_PART_MAIN); /*Set the screen's LVGL background color*/
        lv_obj_set_style_bg_opa(lv_scr_act(), LV_OPA_COVER, LV_PART_MAIN); /*Set the screen's LVGL background transparency*/
        lv_label_set_text(lsm6ds3_acc_data, "acc_x: 0.00 m/s2 acc_y = 0.00 m/s2 acc_z = 0.00 m/s2"); /*Set a new text for a label*/

        lsm6ds3_gry_data = lv_label_create(lv_scr_act()); /*Create a label object*/
        static lv_style_t gry_label_style;
        lv_style_init(&gry_label_style);
        lv_style_set_bg_opa(&gry_label_style, LV_OPA_TRANSP); /*Initialize a style*/
        lv_style_set_bg_color(&gry_label_style, LV_OPA_TRANSP); /*Set the style LVGL background color*/
        lv_obj_add_style(lsm6ds3_gry_data, &gry_label_style, LV_PART_MAIN); /*Add a style to an object*/
        lv_obj_set_style_text_color(lsm6ds3_gry_data, LV_COLOR_WHITE, LV_PART_MAIN); /*Set the style LVGL text color*/
        lv_obj_set_style_text_font(lsm6ds3_gry_data, &lv_font_montserrat_30, LV_PART_MAIN); /*Set the style LVGL text font*/
        lv_obj_align(lsm6ds3_gry_data, LV_ALIGN_CENTER, 0, 30); /*Change the alignment of an object*/
        lv_obj_set_style_bg_color(lv_scr_act(), LV_COLOR_BLACK, LV_PART_MAIN); /*Set the screen's LVGL background color*/
        lv_obj_set_style_bg_opa(lv_scr_act(), LV_OPA_COVER, LV_PART_MAIN); /*Set the screen's LVGL background transparency*/
        lv_label_set_text(lsm6ds3_gry_data, "gry_x: 0.00 rad/s gry_y = 0.00 rad/s gry_z = 0.00 rad/s"); /*Set a new text for a label*/
    }
    lvgl_port_unlock();
}
```

Táto funkcia primárne konfiguruje obsah počiatočného zobrazenia obrazovky: nastavuje farbu pozadia a zobrazenie textu prostredníctvom ovládacieho prvku lvgl.

Funkcia lv_label_set_text nastavuje text zobrazený na ovládacom prvku.

Funkcia **lv_style_set_bg_opa** nastavuje farbu pozadia štýlu. Funkcia

lv_obj_set_style_text_color nastavuje farbu zobrazenia textu.

Funkcia **lv_obj_set_style_text_font** nastavuje veľkosť písma textu.

Funkcia **lv_obj_set_style_bg_color** nastavuje farbu pozadia.

Funkcia **lv_obj_set_style_bg_opa** nastavuje priehľadnosť pozadia.

Funkcia **lv_obj_align** nastavuje zarovnanie a posun ovládacieho prvku.

Poznámka: Pri volaní funkcií **lvgl** mimo funkcií vlákna **lvgl** je potrebné získať mutex zámok.

Funkcia **lvgl_port_lock** získava mutex zámok, funkcia **lvgl_port_unlock** ho uvoľňuje.

/* Funkcia aktualizácie zobrazenia údajov na obrazovke update_lsm6ds3_value */

```
void update_lsm6ds3_value(lsm6ds3_t data)
{
    if ((lsm6ds3_acc_data && lsm6ds3_gry_data))
    {
        char buffer_acc[80];
        char buffer_gry[80];
        sprintf(buffer_acc, "%f", (float)lsm6ds3_acc_data); //Format the data into a string
        sprintf(buffer_gry, "%f", (float)lsm6ds3_gry_data); //Format the data into a string
        lv_label_set_text(lsm6ds3_acc_data_label, buffer_acc); //Set a new text for a label
        lv_label_set_text(lsm6ds3_gry_data_label, buffer_gry); //Set a new text for a label
    }
}
```

Táto funkcia používa funkciu **snprintf** na formátovanie získaných údajov typu **float** do reťazca a následné obnovenie zobrazeného obsahu prostredníctvom funkcie **lv_label_set_text**.

Stojí za zmienku, že funkcia **snprintf** pripája na koniec formátovaného reťazca ukončovací znak. Keďže **lv_label_set_text** rozpoznáva reťazce identifikáciu tohto ukončovacieho znaku, použitie **snprintf** na formátovanie reťazcov je preferovaným prístupom.

```
void app_main(void)
{
    MAIN_INFO("-----Demo version-----");
    MAIN_INFO("-----Start the test-----");
    Init();
    #ifdef CONFIG_ESP_LSM6DS3_SENSOR_ENABLED
    #endif
    #ifdef CONFIG_ESP_DISPLAY_ENABLED
    set_lcd_brightness(100); //Set the screen backlight to maximum brightness
    lsm6ds3_display(); //Set the screen's LVGL default display page
    xTaskCreate(lsm6ds3_read_task, "read_lsm6ds3", 4096, NULL, configMAX_PRIORITIES - 5, &read_lsm6ds3); //Create a lsm6ds3 data display refresh thread
    #endif
    #endif
}
```

V rámci funkcie **app_main** sa najskôr nastaví jas podsvietenia na 100 % a následne sa inicializuje obsah zobrazenia na obrazovke. Nakoniec sa vytvorí vlákno **FreeRTOS** na spracovanie údajov a obnovovanie obrazovky.

/* **lsm6ds3_read_task**: Spracovanie údajov zo senzora a vlákno na obnovovanie obrazovky pre senzor **LSM6DS3TR-C** */

```

void lsm6ds3_read_task(void *param)
{
    while (1)
    {
        if (lsm6ds3_scan() != ESP_OK) /*The lsm6ds3 scans to determine whether the currently read data is from the gyroscope or the accelerometer*/
        {
            if (lvgl_port_lock(0))
            {
                lv_label_set_text(lsm6ds3_acc_data, "lsm6ds3 read acc data error"); /*Read failure message displayed*/
                lv_label_set_text(lsm6ds3_gry_data, "lsm6ds3 read gry data error"); /*Read failure message displayed*/
                lvgl_port_unlock();
            }
            MAIN_ERROR("lsm6ds3 scan false");
        }
        else
        {
            if (lvgl_port_lock(0))
            {
                update_lsm6ds3_value(my_lsm6ds3); /*Update the lsm6ds3 data displayed on the screen*/
                lvgl_port_unlock();
            }
            MAIN_INFO("acc_x: %.2f m/s^2", my_lsm6ds3.acc_x);
            MAIN_INFO("acc_y: %.2f m/s^2", my_lsm6ds3.acc_y);
            MAIN_INFO("acc_z: %.2f m/s^2", my_lsm6ds3.acc_z);
            MAIN_INFO("gry_x: %.2f rad/s", my_lsm6ds3.gry_x);
            MAIN_INFO("gry_y: %.2f rad/s", my_lsm6ds3.gry_y);
            MAIN_INFO("gry_z: %.2f rad/s", my_lsm6ds3.gry_z);
            vTaskDelay(1000 / portTICK_PERIOD_MS);
        }
    }
}

```

V rámci vlákna spracovania údajov zo senzora LSM6DS3TR-C a obnovovania zobrazenia obrazovky sa najskôr vytvorí cyklus while. V rámci tohto cyklu sa vyvolá funkcia lsm6ds3_scan na aktualizáciu a získanie údajov zo senzora. Po úspešnom získaní údajov funkcia update_lsm6ds3_val-ue obnoví údaje zobrazené na obrazovke. Ak sa získavanie nepodarí, na obrazovke sa zobrazí chybová správa „LSM6DS3 read acc data error“ (Chyba čítania údajov LSM6DS3) a „LSM6DS3 read gry data error“ (Chyba čítania údajov LSM6DS3). Konečné 1-sekundové oneskorenie zabezpečuje, že údaje sa obnovujú a získavajú raz za sekundu.

1.3.5 Súbor CMkaLists.txt

Aby bolo možné úspešne vyvolať obsah priečinkov bsp_display a bsp_lsm6ds3tr v hlavnej funkcii, je potrebné nakonfigurovať súbor CMakeLists.txt, ktorý sa nachádza v hlavnom priečinku. Podrobnosti konfigurácie sú nasledovné:

```

FILE(GLOB_RECURSE main ${CMAKE_SOURCE_DIR}/main/*.c)

idf_component_register(SRCS ${main}
                       INCLUDE_DIRS "include"
                       REQUIRES bsp_i2c bsp_display bsp_lsm6ds3tr)

```

Najskôr sa definujú adresáre pre zdrojové súbory a hlavičkové súbory spolu s požadovanými knižnicami ovládačov: bsp_display, bsp_lsm6ds3tr a bsp_i2c. Tieto nastavenia sa potom zaregistrujú v systéme zostavovania pomocou príkazu idf_component_register, čím sa hlavnej funkcii umožní využívať tieto funkcie ovládačov.

1.4 Postup programovania

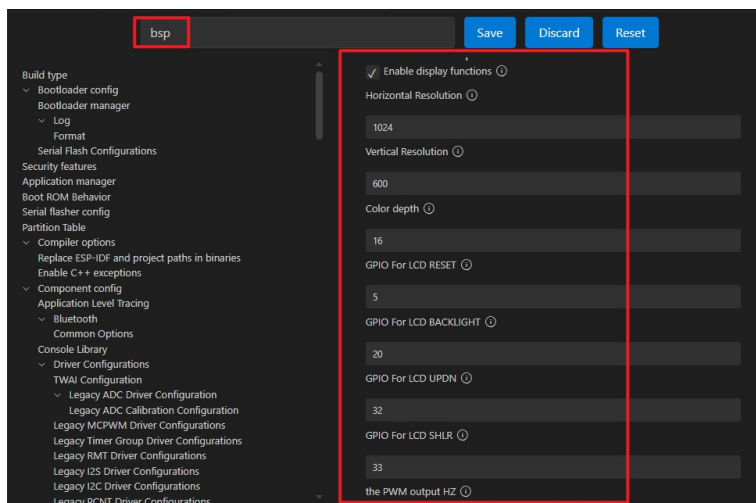
www.rlx.sk

Pripojte zariadenie P4 k počítaču prostredníctvom USB

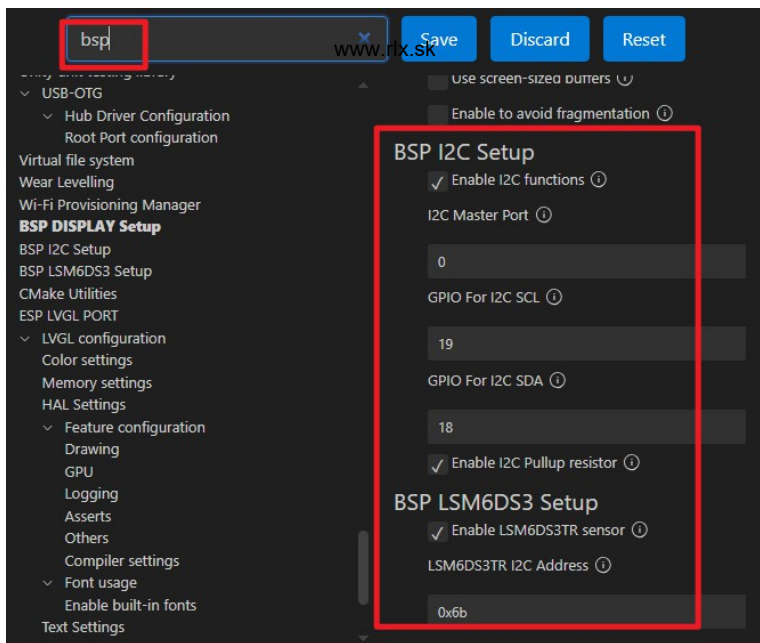


1.4.1 Po klonovaní kódu cez Git (odkaz bude potvrdený) vymažte všetky lokálne kompilačné informácie. Nakonfigurujte prostredie IDF a model čipu pre kompiláciu podľa lekcie 1 a nastavte číslo sériového portu pre programovanie.

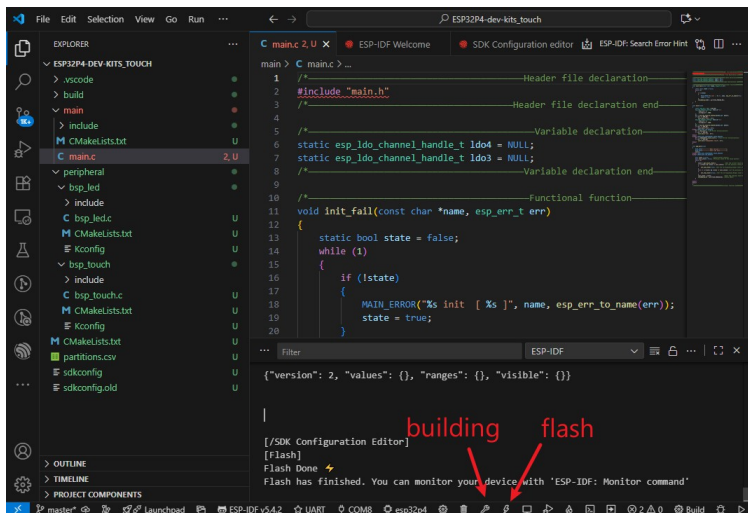
1.4.2 Nasledujúca konfigurácia SDKConfig je z veľkej časti identická s lekciou 1, jednoducho prekonfigurujte piny DS1play, IIC a LSM6DS3.



RLX COMPONENTS



1.4.3 Kliknite na Kompilovať. Po úspešnej kompilácii kliknite na Stiahnuť.



RLX COMPONENTS

Lekcia 14 – Ovládanie WS2814 RGBW

Úvod

Táto kapitola tutoriálu demonštruje ovládacie aplikácie ESP32-P4 pre LED pásiky WS2814 RGB. Prostredníctvom príkladov osvetľovacích a prechodových svetelných efektov pomáha pochopiť jeho základné schopnosti. Ako pokročilý testovací prípad poskytuje osvetlenie LED pásu čitateľom intuitívne a hlboké pochopenie schopností periférnych ovládačov ESP32-P4, čím vytvára základ pre zložitejšie projekty, ako sú maticové displeje LED pásov a ovládanie okolitého osvetlenia.

Demonštrácia projektu Efekt



Táto kapitola je rozdelená do nasledujúcich podkapitol

- 1.1 Úvod do WS2814 a RGB LED
- 1.2 Návrh hardvéru
- 1.3 Návrh softvéru
- 1.4 Stiahnutie a overenie

RLX COMPONENTS

1.1.1 Úvod do WS2814

WS2814 je radiaci čip s integrovaným ovládačom konštantného prúdu a funkciou dekódovania signálu, ktorý je zvyčajne zapuzdrený s LED diódami a tvorí programovateľné svetelné korálky. Medzi jeho kľúčové vlastnosti patria:

- ① Jednožičové ovládanie: WS2814 sa ovláda prostredníctvom jedného dátového vedenia, ktoré podporuje kaskádové pripojenia, kde je možné postupne prepojiť viacero LED diód, čo výrazne zjednodušuje zapojenie obvodu.
- ② Pohon s konštantným prúdom: Každý kanál obsahuje interný zdroj konštantného prúdu, ktorý zabezpečuje rovnomerný jas a zabráňuje nerovnomernému osvetleniu spôsobenému kolísaním napätia.
- ③ 16-bitová škála sivej: WS2814 podporuje 256 úrovní jasu (8-bit), čo umožňuje bohaté farebné efekty zobrazenia.
- ④ Odolnosť voči poruchám: Podporuje funkciu obnovenia po prerušení. V prípade poruchy jednej LED diódy ostatné LED diódy pokračujú v normálnej prevádzke, čo zvyšuje spoľahlivosť.
- ⑤ Kompatibilita napätia: WS2814 zvyčajne pracuje pri 5 V, pričom spotreba prúdu závisí od farby a jasu LED.

ESP32-P4 môže ovládať WS2814 prostredníctvom periférie RMT alebo PWM + presného časovania, čím zjednodušuje komplexné ovládanie farebného osvetlenia.

WS2814 sa bežne používa v RGB svetelných pásoch.



1.2 Hardvérový dizajn

www.rlx.sk

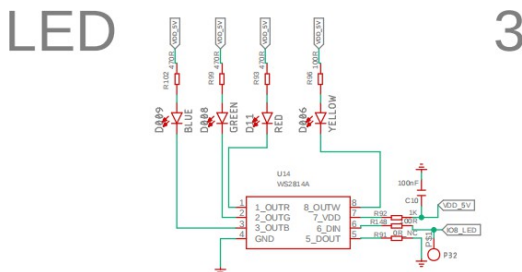
V tomto experimente sa používa vývojová doska ESP32-P4 so štyrmi sériovo zapojenými LED diódami WS2814 RGB. Zapojenie obvodu je nasledovné:

ESP32-P4 GPIO 8 → WS2814 DIN (vstup údajov)

5V napájanie → WS2814 VCC GND →

WS2814 GND

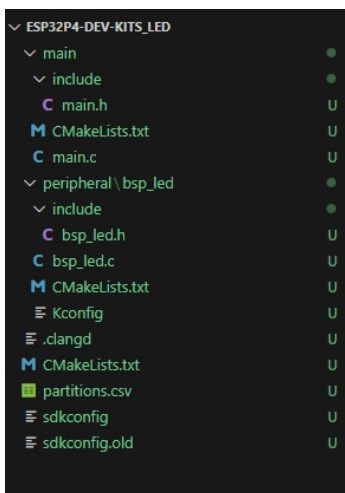
WS2814 DOUT → Ďalší WS2814 DIN (kaskádový)



1.3 Analýza programu

<https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design>

Otvorte súbor projektu v VS Code podľa predchádzajúcich pokynov.



V rámci príkladu **ESP32P4-dev-kits_led** bola v adresári **ESP32P4-dev-kits_led/peripheral** vytvorená nová zložka s názvom **bsp_led**. V rámci cesty **bsp_led** bola vytvorená zložka **include**, súbor **CMakeLists.txt** a súbor **Kconfig**. Zložka **bsp_led** je určená na ukladanie súboru ovládača **bsp_led.c**. Zložka **include** obsahuje súbor hlavičky **bsp_led.h**, zatiaľ čo súbor **CMakeLists.txt** integruje ovládač do systému zostavovania, čím umožňuje projektové využitie jeho funkcionality. Súbor **Kconfig** registruje celý ovládač spolu s definíciami pinov GPIO v súbore **sdkconfig** na platforme IDF (konfigurovateľné prostredníctvom grafického rozhrania).

RLX COMPONENTS

1.3.1 Kód ovládača RMT LED

Tu sa zameriame výlučne na základný kód, podrobný zdrojový kód nájdete v príslušných zdrojových súboroch pre tento experiment v materiáloch kódu.

Zdrojový kód ovládača LED pozostáva z dvoch súborov: bsp_led.c a bsp_led.h.

Nižšie najprv analyzujeme program bsp_led.h: definuje výstupné kanály RMT a deklaruje funkcie pre použitie LED.

/* Odkazy na hlavičkový súbor */

```
/*----- Header file declaration -----*/
#include "freertos/FreeRTOS.h" //References for FreeRTOS Function-related API Functions
#include "freertos/task.h" //References for FreeRTOS Task Function-related API Functions
#include "esp_log.h" //References for LOG Printing Function-related API Functions
#include "esp_err.h" //References for Error Type Function-related API Functions
#include "driver/rmt_tx.h" //References for RMT TX Function-related API Functions
/*----- Header file declaration end -----*/
```

/* Deklarácie funkcií a definície makier */

```
#ifdef CONFIG_BSP_LED_ENABLED

#define LED_GPIO CONFIG_LED_GPIO // LED GPIO
#define RMT_LED_STRIP_RESOLUTION_HZ 10000000 // Channel clock resolution, in Hz

typedef enum
{
    LED_BYTE_RED = 0,
    LED_BYTE_YELLOW,
    LED_BYTE_GREEN,
    LED_BYTE_BLUE,
    LED_BYTE_DONE,
} led_byte_order_t;

esp_err_t led_init(); // RMT LED Initialization Function
esp_err_t set_led_status(uint32_t bit); // Set led display status
esp_err_t set_single_led_status(led_byte_order_t color, uint8_t brightness); // Set single led display status
#endif
```

Táto štruktúra definuje farby pre LED pripojené k štyrom pinom WS2814A, čo uľahčuje následné ovládanie prostredníctvom funkcií.

Ďalej preskúmame program bsp_led.c: inicializácia radiča RMT a sprístupnenie funkcií rozhrania API.

/* Funkcia inicializácie radiča RMT led_init */

```
esp_err_t led_init()
{
    esp_err_t err = ESP_OK;
    rmt_tx_channel_config_t tx_chan_config = {
        .clk_src = RMT_CLK_SRC_DEFAULT, //Clock source of RMT TX channel*/
        .gpio_num = LED_GPIO, //GPIO number used by RMT TX channel*/
        .mem_block_symbols = 64, //The size of the memory block bytes(64 * 4 = 256)*/
        .resolution_hz = RMT_LED_STRIP_RESOLUTION_HZ, //10 MHz tick resolution(1 tick = 0.1 μs)*/
        .trans_queue_depth = 4, //max num*/
    };
    err = rmt_new_tx_channel(&tx_chan_config, &led_chan); //Create a RMT TX channel*/
    if (err != ESP_OK)
    {
        LED_ERROR("create new rmt tx channel fail");
        return err;
    }
    const rmt_simple_encoder_config_t simple_encoder_cfg = {
        .callback = encoder_callback, //Callback to call for encoding data into RMT items*/
    };
    err = rmt_new_simple_encoder(&simple_encoder_cfg, &led_encoder); //Create RMT simple callback encoder*/
    if (err != ESP_OK)
    {
        LED_ERROR("create new rmt simple encoder fail");
        return err;
    }
    err = rmt_enable(led_chan); //Enable the RMT channel*/
    if (err != ESP_OK)
    {
        LED_ERROR("enable rmt fail");
        return err;
    }
    return err;
}
```

RLX COMPONENTS

V rámci funkcie `led\_init` sa najskôr vykoná konfigurácia štruktúry `rmt\_tx\_channel\_config\_t`. Špecifikované parametre sú: zdroj hodín ~~prevzatý z~~ **pinu 31**, číslo pinov GPIO používané kanálom TX, veľkosť pamäťového bloku, rozlíšenie hodín kanála a maximálna hĺbka internej prenosovej fronty. Následne sa vytvorí nový výstupný kanál RMT, v dekodéri RMT sa zaregistruje funkcia spätného volania pre spracovanie signálu RMT a nakoniec sa aktivuje radič RMT.

Stojí za zmienku, že tu nastavujeme rozlíšenie hodín kanála na 10 MHz. Keďže riadiace signály WS2814A pracujú na úrovni nanosekúnd, je nastavené rozlíšenie 10 MHz (čo znamená, že jeden tik sa rovná 0,1 mikrosekundy).

/* Definícia signálu pre výstup RMT do čipu WS2814A */

```
/*The layout of RMT symbol stored in memory*/
static const rmt_symbol_word_t signal_zero = {
    .level0 = 1,    /*Level of the first part*/
    .duration0 = 4, /*Duration of level0*/
    .level1 = 0,    /*Level of the second part*/
    .duration1 = 9, /*Duration of level1*/
};

static const rmt_symbol_word_t signal_one = {
    .level0 = 1,    /*Level of the first part*/
    .duration0 = 6, /*Duration of level0*/
    .level1 = 0,    /*Level of the second part*/
    .duration1 = 7, /*Duration of level1*/
};

static const rmt_symbol_word_t signal_reset = {
    .level0 = 0,    /*Level of the first part*/
    .duration0 = 1500, /*Duration of level0*/
    .level1 = 0,    /*Level of the second part*/
    .duration1 = 1500, /*Duration of level1*/
};
```

Najskôr rozparsujeme obsah štruktúry:

Level0 a duration0 definujú typ úrovne a trvanie prvej časti signálu. Level1 a duration1 definujú typ úrovne a trvanie druhej časti signálu.

Podľa požiadaviek na signál čipu WS2814A: - Signál s vysokou úrovňou trvajúcou 400 ns a nízkou úrovňou trvajúcou 900 ns predstavuje hodnotu signálu 1. - Signál s vysokou úrovňou trvajúcou 600 ns a nízkou úrovňou trvajúcou 700 ns predstavuje hodnotu signálu 0. - Signál s nízkou úrovňou trvajúcou viac ako 280 ns predstavuje signál resetovania.

/* Funkcia spätného volania dekodovania radiča RMT **encoder_callback** */

```

/*Callback to call for encoding data into 801 frame*/
static size_t encoder_callback(const void *data, size_t data_size, size_t symbols_written, size_t symbols_free, ret_symbol_word_t *symbols, bool *done, void *arg)
{
    /* Check if there's enough space in the symbols buffer for a full byte (8 symbols) */
    if (symbols_free < 8) /*The maximum amount of symbols that can be written to the symbols buffer*/
    {
        return 0; /*Not enough space, return 0 to indicate no symbols were written*/
    }
    /*Calculate current position in the data stream based on symbols already written*/
    /*Each byte requires 8 symbols, so divide by 8 to get the byte position*/
    size_t data_pos = symbols_written / 8; /*Current position in encoded stream, in symbols*/
    uint8_t *data_bytes = (uint8_t *)data; /*Cast void pointer to byte array for easier access */
    if (data_pos > data_size) /*Check if we still have data bytes to encode*/
    {
        size_t symbol_pos = 0; /*Position counter within the current symbol buffer*/
        for (int bitmask = 0x00; bitmask != 0; bitmask >>= 1)
        {
            /*Encode each bit of the current byte (801 first)*/
            /*bitmask starts at 0x00 (10000000 binary) and shifts right each iteration*/
            if ((data_bytes[data_pos] & bitmask) /*Check if the current bit is set (1) or clear (0)*/)
            {
                symbols[symbol_pos++] = signal_one; /*use the symbol defined for logic high*/
            }
            else
            {
                symbols[symbol_pos++] = signal_zero; /*use the symbol defined for logic low*/
            }
        }
        return symbol_pos;
    }
    else
    {
        symbols[0] = signal_reset; /*All data bytes have been encoded, send reset symbol to indicate end of transmission*/
        *done = 1; /*Write reset symbol to mark transmission end*/
        return 1;
    }
}

```

Najskôr určite, či aktuálna vyrovnávacia pamäť symbolov má dostatočný priestor na uloženie znaku (8 symbolov). Ak táto podmienka nie je splnená, okamžite ukončíte. Ak sú požiadavky na priestor splnené, zistíte aktuálnu pozíciu v dátovom toku na základe už zapísaných symbolov. (Každý bajt vyžaduje 8 symbolov; počet zapísaných bajtov možno určiť vydelením počtu zapísaných symbolov číslom 8). Ak súčasne dáta ešte neboli zakódované, vykonajte konfiguráciu kódovania signálu (určite, či sa má na súčasnej pozícii vydať signál 0 alebo signál 1). Po dokončení kódovania dát vydajte resetovací signál na obnovenie LED displeja.

/* Funkcia LED pre nastavenie jednej LED: set_single_led_status */

```

esp_err_t set_single_led_status(led_byte_order_t color, uint8_t brightness)
{
    esp_err_t err = ESP_OK;
    /*Configure RMT transmission for single shot operation (no repetition)*/
    rmt_transmit_config_t tx_config = {
        .loop_count = 0,
    };
    /*Set the specified color component to the desired brightness and turn off all other color components*/
    switch (color)
    {
        case LED_BYTE_RED:
            led_strip_pixels[0] = 0x00; /*Yellow component off*/
            led_strip_pixels[1] = brightness; /*Red component on*/
            led_strip_pixels[2] = 0x00; /*Green component off*/
            led_strip_pixels[3] = 0x00; /*Blue component off*/
            break;
        case LED_BYTE_GREEN:
            led_strip_pixels[0] = 0x00;
            led_strip_pixels[1] = 0x00;
            led_strip_pixels[2] = brightness;
            led_strip_pixels[3] = 0x00;
            break;
        case LED_BYTE_BLUE:
            led_strip_pixels[0] = 0x00;
            led_strip_pixels[1] = 0x00;
            led_strip_pixels[2] = 0x00;
            led_strip_pixels[3] = brightness;
            break;
        case LED_BYTE_YELLOW:
            led_strip_pixels[0] = brightness;
            led_strip_pixels[1] = 0x00;
            led_strip_pixels[2] = 0x00;
            led_strip_pixels[3] = 0x00;
            break;
        default:
            break;
    }
    err = rmt_transmit(led_chan, led_encoder, led_strip_pixels, sizeof(led_strip_pixels), &tx_config); /*Transmit data by RMT TX channel*/
    if (err != ESP_OK)
    {
        LED_INFO("transmit rmt cmd fail");
        return err;
    }
    err = rmt_tx_wait_all_done(led_chan, portMUX_DELAY); /*Wait for all pending TX transactions done*/
    if (err != ESP_OK)
    {
        LED_INFO("transmit rmt fail");
        return err;
    }
    return err;
}

```

RLX COMPONENTS

Po prvé, táto funkcia má len jeden vstupný parameter: bit – 32-bitový dátový parameter, ktorý ovláda výstup štyroch LED diód na WS2814A (ako je uvedené vyššie, konkrétne hodnoty ovláda jas svetiel). Dátové bity 0-7 zodpovedajú modrému svetlu, 8-15 zelenému svetlu, 16-23 oranžovému svetlu a 24-31 červenému svetlu.

Vstupné dáta prechádzajú operáciou posunu, aby sa rozdelili na jednotlivé 8-bitové segmenty. Podobne ako pri funkcii ovládajúcej jednu LED diódu, príslušné dáta sa konfigurujú do výstupného poľa a potom sa vyvolá funkcia odoslania, aby sa preniesli. (Prípád použitia: zadanie 0x00000000 zhasne všetky svetlá).

1.3.2 Súbor Kconfig

Hlavnou funkciou tohto súboru je pridať požadovanú konfiguráciu do súboru sdkconfig, čím sa umožní modifikácia určitých nastavení parametrov prostredníctvom grafického rozhrania. Číslo 8 tu zodpovedá **GPIO_NUM_8**.

```
menu "BSP LED Setup"
  config BSP_LED_ENABLED
    bool "Enable LED config"
    default n

  if BSP_LED_ENABLED
    config LED_GPIO
      int "GPIO for LED"
      default 8
  endif
endmenu
```

1.3.3 Súbor CMkaLists.txt

Funkčnosť tejto ukážkovej rutiny závisí predovšetkým od ovládača **bsp_led**. Aby bolo možné úspešne vyvolať obsah priečinka **bsp_led** z iných funkcií, je potrebné nakonfigurovať súbor **CMakeLists.txt** v priečinku **bsp_led**. Podrobnosti konfigurácie sú nasledovné:

```
FILE(GLOB_RECURSE component_sources "*.c")

idf_component_register(SRCS ${component_sources}
                      INCLUDE_DIRS "include"
                      REQUIRES driver)
```

V tomto súbore CMakeLists.txt sú najskôr definované adresáre pre zdrojové súbory a hlavičkové súbory spolu s požadovanými knižnicami ovládačov. Následne sa tieto nastavenia zaregistrujú v systéme zostavovania pomocou príkazu `idf_component_register`, čím sa projektu umožní využívať funkčnosť ovládača **bsp_led**.

1.3.4 Hlavný priečinok

Hlavný priečinok slúži ako hlavný adresár pre vykonávanie programu a obsahuje hlavnú funkciu **main.c** a hlavičkový súbor **main.h** v priečinku `include`. Pridajte hlavný priečinok do súboru **CMakeLists.txt** systému zostavenia.

Súbor **main.h** odkazuje predovšetkým na požadované hlavičkové súbory: funkcie využívajúce ovládač **bsp_led**

vyžadujú zaradenie hlavičkového súboru **bsp_led**.

Nižšie je analýza programu `main.c`: inicializácia systému a vykonávanie špecifické pre funkciu RMT LED.

```

    init_fail(1004, err);
#ifdef CONFIG_BSP_LED_ENABLED
    err = led_init(); /*RMT LED initialization*/
    if (err != ESP_OK)
        init_fail("led", err);
    vTaskDelay(200 / portTICK_PERIOD_MS);
    set_led_status(0x00000000); /*All the LEDs are off*/
#endif

```

Tento kód sa nachádza vo funkcii `init`, ktorá slúži na ukladanie inicializačných funkcií vyžadujúcich vyvolanie a hodnotenie ich výsledkov. Ak sa stav návratu odchyľuje od **ESP_OK**, kód vygeneruje chybovú správu a zastaví ďalšie vykonávanie. Po 200 ms oneskorení sa vykoná funkcia `set_led_status`, aby sa vymazali všetky LED displeje. Tento krok zabezpečuje, že po zapnutí sú všetky LED diódy v zhasnutom stave.

```

void app_main(void)
{
    MAIN_INFO("-----Demo version-----");
    MAIN_INFO("-----Start the test-----");
    Init();
#ifdef CONFIG_BSP_LED_ENABLED
    xTaskCreate(led_task, "led", 4096, NULL, configMAX_PRIORITIES - 5, &led); /*Create a led display refresh thread*/
#endif
}

```

V rámci funkcie `app_main` vytvorte vlákno FreeRTOS na vykonanie obnovenia efektu bežiacich svetiel LED.

/* Vlákno LED bežeckého svetla `led_task` */

```

void led_task(void *param)
{
    led_byte_order_t led_byte = LED_BYTE_RED; /*Initialize the default display color*/
    while (1)
    {
        /*Set the lighting of a single lamp*/
        set_single_led_status(led_byte++, 0xFF); /*led_byte++:cycled through all available colors,0xFF = full brightness (255)*/
        vTaskDelay(1000 / portTICK_PERIOD_MS);
        if (led_byte == LED_BYTE_DONE) /*Check if we've cycled through all available colors*/
        {
            led_byte = LED_BYTE_RED; /*Reset to Red color to restart the cycle*/
        }
    }
}

```

V rámci vlákna LED bežného svetla najskôr vytvorte slučku `while`. V rámci tejto slučky vyvolajte funkciu `set_single_led_status` na aktiváciu požadovanej LED. Sekvencia definovaná v rámci našej štruktúry LED zodpovedá usporiadaniu LED zľava doprava na vývojovej doske. Preto postupné zvyšovanie premennej výpočtu dosahuje efekt postupného rozsvietenia každej LED (zhasnutie predtým rozsvietených LED). Táto sekvencia sa vykonáva raz za sekundu. Po dosiahnutí konečnej hodnoty LED sa sekvencia resetuje do počiatočného stavu červenej LED.

1.3.5 Súbor `CMkaLists.txt`

Aby bolo možné úspešne vyvolať obsah priečinka `bsp_led` v hlavnej funkcii, je potrebné nakonfigurovať súbor `CMakeLists.txt`, ktorý sa nachádza v hlavnom priečinku. Podrobnosti konfigurácie sú nasledovné:


```
FILE(GLOB_RECURSE main ${CMAKE_SOURCE_DIR}/main/*.c)
                                www.rlx.sk
idf_component_register(SRCS ${main}
                        INCLUDE_DIRS "include"
                        REQUIRES bsp_led)
```

Najskôr sa definujú adresáre pre zdrojové súbory a hlavičkové súbory spolu s požadovanou knižnicou ovládačov – konkrétne knižnicou ovládačov pre prepojenie bsp_led. Následne sa tieto nastavenia zaregistrujú v systéme zostavovania pomocou príkazu `idf_component_register`, čím sa hlavnej funkcii umožní využívať tieto schopnosti ovládačov.

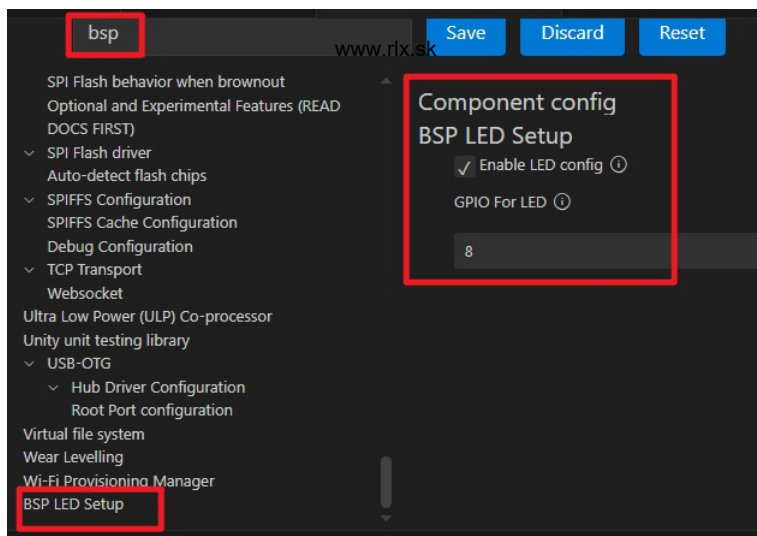
1.4 Postup programovania

Pripojte zariadenie P4 k počítaču prostredníctvom USB



1.4.1 Po klonovaní kódu prostredníctvom Git (odkaz bude potvrdený) vymažte všetky lokálne informácie o kompilácii. Nakonfigurujte prostredie IDF a model čipu pre kompiláciu podľa lekcii 1 a nastavte číslo sériového portu pre programovanie.

1.4.2 Nasledujúca konfigurácia SDKConfig je z veľkej časti identická s lekciami 1, jednoducho prekonfigurujte piny LED.



1.4.3 Kliknite na Kompilovať. Po úspešnej kompilácii kliknite na Stiahnuť.

Lekcia 15 – Ovládanie tlačidiel ADC

Úvod

Táto kapitola tutoriálu predstavuje aplikáciu vstupu ADC v ESP32-P4. Prostredníctvom rutiny ovládajúcej štyri LED diódy pomocou štyroch tlačidiel pomáha pochopiť základné funkcie detekcie analógového signálu a multiplexného ovládania. Ako bežná metóda interakcie človek-stroj, vstupy tlačidiel v kombinácii s technikou detekcie deliča napätia ADC umožňujú vstup viacerých klávesov pri obsadení iba jedného pinu ADC, čo výrazne zvyšuje využitie pinov. Tento príklad poskytuje čitateľom intuitívne pochopenie aplikácií ADC ESP32-P4 a vytvára základ pre zložitejšie senzorové a interaktívne projekty.



Táto kapitola je rozdelená do nasledujúcich podkapitol

- 1.1 Úvod do ADC a tlačidiel
- 1.2 Návrh hardvéru
- 1.3 Návrh softvéru
- 1.4 Stiahnutie a overenie

1.1 Úvod do ADC a tlačidiel

1.1.1 Úvod do ADC

Čip ESP32-P4 obsahuje viackanálový ADC (analogovo-digitálny prevodník), ktorý je schopný prevádzať analogové napätia v rozsahu od 0 do 3,3 V na digitálne hodnoty. Medzi jeho hlavné vlastnosti patria:

- ① Viackanálový vstup: Podporuje viacero kanálov ADC, čo umožňuje pripojenie viacerých senzorov alebo vstupných signálov.
- ② Vysoké rozlíšenie: ADC čipu ESP32-P4 podporuje 12-bitové rozlíšenie s výsledkami prevodu v rozsahu od 0 do 4095.
- ③ Flexibilné vzorkovanie: Konfigurovateľné vzorkovacie obdobie a režim útlmu na prispôbenie

RLX COMPONENTS

rôzne rozsahy vstupného napätia.

④ Univerzálne použitie: Bežne sa používa na detekciu deliča napätia tlačidiel, potenciometrov, teplotných senzorov, monitorovanie napätia batérie a podobné scenáre.

V experimente v tejto kapitole využijeme schopnosť ADC detekovať napätie na identifikáciu vstupov z rôznych tlačidiel prostredníctvom obvodu deliča napätia s odporom, čím budeme zodpovedajúcim spôsobom ovládať príslušné LED svetlá.

1.1.2 Prehľad tlačidiel

Princíp fungovania tlačidiel

Tlačidlá sú najbežnejším typom spínacích zariadení, ktoré pri stlačení vedú prúd a pri uvoľnení ho prerušujú. Na úsporu I/O pinov je možné použiť metódu delenia napätia rezistorom, pri ktorej sa viacero tlačidiel pripojí k rovnakému kanálu ADC:

Každé tlačidlo je zapojené do série s odporom s inou hodnotou; Keď sú

stlačené rôzne tlačidlá, ADC získava rôzne napätia;

Program určuje, ktoré tlačidlo bolo stlačené, na základe rozsahu napätia.

1.2 Hardvérová konštrukcia

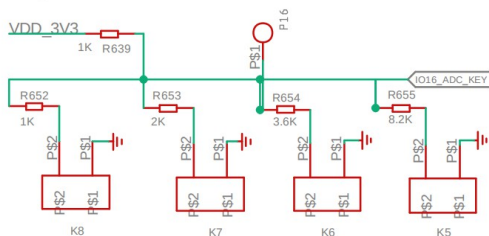
Hardvér pozostáva z jedného vstupného pinu ADC a štyroch

tlačidiel: Vstupný pin ADC: GPIO16

Obvod tlačidiel: Každé zo štyroch tlačidiel je pripojené k pinom ADC prostredníctvom obvodu deliča napätia s použitím rezistorov s rôznymi hodnotami.

Custom_key

ADC



10

Schéma

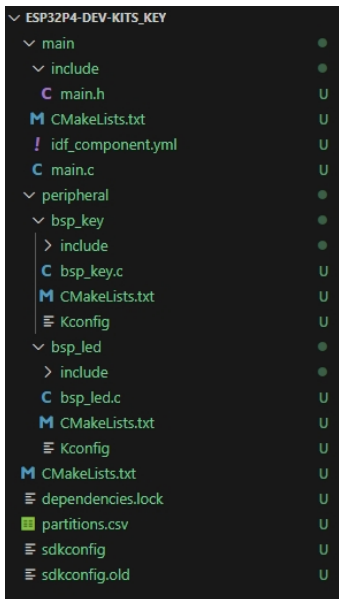
RLX COMPONENTS

1.3 Analýza programu

www.rlx.sk

<https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design>

Otvorte súbor projektu v VS Code podľa predchádzajúcich pokynov.



V rámci príkladu **ESP32P4-dev-kits_key** bola v adresári **ESP32P4-dev-kits_key\peripheral** vytvorená nová zložka s názvom **bsp_key**. V rámci cesty **bsp_key** bola vytvorená zložka **include**, súbor **CMakeLists.txt** a súbor **Kconfig**. Zložka **bsp_key** je určená na ukladanie súboru ovládača **bsp_key.c**. Zložka **include** obsahuje súbor hlavičky **bsp_key.h**, zatiaľ čo súbor **CMakeLists.txt** integruje ovládač do systému zostavovania, čím umožňuje projektové využitie jeho funkcionality. Súbor **Kconfig** načíta celú konfiguráciu ovládača, vrátane definícií pinov GPIO, do súboru **sdkconfig** v rámci platformy IDF (konfigurovateľné prostredníctvom grafického rozhrania).

1.3.1 Kód ovládača tlačidla ADC

Tu sa zameriame výlučne na základný kód; podrobný zdrojový kód nájdete v príslušných zdrojových súboroch pre tento experiment v materiáloch kódu.

Zdrojový kód ovládača tlačidla ADC pozostáva z dvoch súborov: **bsp_key.c** a **bsp_key.h**.

Nižšie najprv analyzujeme program **bsp_key.h**: definuje príslušné piny kľúča ADC a deklaruje použité funkcie.

/* Odkazy na hlavičkový súbor */

```
/*-----Header file declaration-----*/
#include "esp_log.h" //References for LOG Printing Function-related API Functions
#include "esp_err.h" //References for Error Type Function-related API Functions
#include "driver/gpio.h" //References for GPIO Function-related API Functions
#include "iot_button.h" //References for IOT Button Function-related API Functions
#include "button_adc.h" //References for ADC Button Function-related API Functions
/*-----Header file declaration end-----*/
```

/* Deklarácie funkcií a definície makier */

```
#ifndef IOT_BUTTON_KEY_HM100
#define KEY_GPIO_CONFIG_KEY_GPIO // The GPIO used for the ADC button channel

typedef enum
{
    Key_idle = 0,
    Key_left,
    Key_middle,
    Key_right,
    Key_up,
    Key_release,
} key_state_t; // Key Press State Structure

esp_err_t key_init(adc_oneshot_unit_handle_t handle); // ADC Key Initialization function
esp_err_t key_register_cb(); // Key press status registration callback function
void key_state_callback(key_state_t key_state);
#endif
```

Táto štruktúra definuje efekty stlačenia a uvoľnenia pre štyri tlačidlá ADC, čo uľahčuje následné ovládanie prostredníctvom funkcií.

Ďalej budeme analyzovať program bsp_key.c: inicializácia a konfigurácia tlačidiel ADC, využité funkcií spätného volania a zavedenie funkcií rozhrania API.

/* Funkcia inicializácie tlačidla ADC key_init */

```
esp_err_t key_init(adc_oneshot_unit_handle_t handle)
{
    esp_err_t err = ESP_OK;
    /*Configure button timing parameters*/
    button_config_t btn_cfg = {
        .long_press_time = 3000, /*Trigger time(ms) for long press*/
        .short_press_time = 200, /*Trigger time(ms) for short press*/
    };
    /*Configure ADC parameters for 4 buttons on the same ADC channel*/
    button_adc_config_t btn_adc_cfg[4] = {
        /*Button 0: Voltage range 1500-1900*/
        {
            .adc_handle = &handle, /*Pointer to ADC unit handle*/
            .unit_id = ADC_UNIT_1, /*Use ADC unit 1*/
            .adc_channel = ADC_CHANNEL_0, /*Use ADC channel 0*/
            .button_index = 0, /*Identifier for button 0*/
            .min = 1500, /*Minimum ADC value for button 0 detection*/
            .max = 1900, /*Maximum ADC value for button 0 detection*/
        },
        /*Button 1: Voltage range 2000-2320*/
        {
            .adc_handle = &handle,
            .unit_id = ADC_UNIT_1,
            .adc_channel = ADC_CHANNEL_0,
            .button_index = 1,
            .min = 2000, /*Minimum ADC value for button 1 detection*/
            .max = 2320, /*Maximum ADC value for button 1 detection*/
        },
        /*Button 2: Voltage range 2400-2650*/
        {
            .adc_handle = &handle,
            .unit_id = ADC_UNIT_1,
            .adc_channel = ADC_CHANNEL_0,
            .button_index = 2,
            .min = 2400, /*Minimum ADC value for button 2 detection*/
            .max = 2650, /*Maximum ADC value for button 2 detection*/
        },
        /*Button 3: Voltage range 2700-3000*/
        {
            .adc_handle = &handle,
            .unit_id = ADC_UNIT_1,
            .adc_channel = ADC_CHANNEL_0,
            .button_index = 3,
            .min = 2700, /*Minimum ADC value for button 3 detection*/
            .max = 3000, /*Maximum ADC value for button 3 detection*/
        },
    };
};
```

```
for (size_t i = 0; i < 4; i++)
{
    err = iot_button_new_adc_device(&btn_cfg, &btn_adc_cfg[i], &btns[i]); /*Create a new ADC button device*/
    if (err != ESP_OK)
        return err;
    if (btns[i] == NULL) /*Verify that the button handle was created successfully*/
        return err;
}
return err;
```

V rámci funkcie key_init sa konfigurácia začína štruktúrou button_config_t, kde parametre špecifikujú dĺžku rozpoznávania dlhého stlačenia a dĺžku rozpoznávania krátkeho stlačenia. Následne sa konfiguruje štruktúra button_adc_config_t s

parametre vrátane používaného ovládača ADC, čísla ovládača ADC, kanála na získavanie údajov ADC, poradového čísla rozpoznávania kľúčov, minimálnej hodnoty napätia ADC pre kľúč a maximálnej hodnoty napätia ADC pre kľúč. Na konfiguráciu každého zo štyroch tlačidiel sa postupne používa cyklus for, čím sa získajú vrátené ovládače.

Je dôležité poznamenať, že pri nastavovaní rozsahov napätia pre tlačidlá ADC je potrebné vyhnúť sa prekryvaniu rozsahov, pretože to zabráni správne rozpoznaniu tlačidiel.

/* Funkcia registrácie udalostí tlačidiel ADC key_register_cb() */

```
esp_err_t key_register_cb()
{
    esp_err_t err = ESP_OK;
    for (size_t i = 0; i < 4; i++)
    {
        err = !iot_button_register_cb(btns[i], BUTTON_PRESS_DOWN, NULL, button_event_cb, (void *)i); //Register the button event callback function*/
        if (err != ESP_OK)
            return err;
        err = !iot_button_register_cb(btns[i], BUTTON_PRESS_UP, NULL, button_event_cb, (void *)i);
        if (err != ESP_OK)
            return err;
    }
    return err;
}
```

Táto funkcia používa cyklus for na registráciu všetkých štyroch spätných volaní tlačidiel. Funkcia `iot_button_register_cb` sa používa na registráciu spätných volaní udalostí tlačidiel (s využitím identifikátorov tlačidiel získaných počas inicializácie).

Stojí za zmienku, že tento experiment využíva iba udalosti stlačenia a uvoľnenia tlačidla. Ak chcete použiť iné udalosti, pozrite si typ výpočtu. Okrem toho tlačidlá ADC nepodporujú kombinované udalosti klávesov, pretože hodnoty ADC generované súčasným stlačením viacerých tlačidiel sú nepredvídateľné.

/*Funkcia spätného volania udalosti tlačidla ADC button_event_cb*/

```
static void button_event_cb(void *arg, void *data)
{
    button_event_t event = iot_button_get_event(arg); //Get button event*/
    switch (event)
    {
        case BUTTON_PRESS_DOWN: //Press the button*/
            switch ((int)data) //Determine the key sequence identifier*/
            {
                case 0:
                    key_status = Key_left;
                    KEY_INFO("left btn : BUTTON_PRESS_DOWN");
                    break;
                case 1:
                    key_status = Key_right;
                    KEY_INFO("right btn : BUTTON_PRESS_DOWN");
                    break;
                case 2:
                    key_status = Key_middle;
                    KEY_INFO("middle btn : BUTTON_PRESS_DOWN");
                    break;
                case 3:
                    key_status = Key_up;
                    KEY_INFO("up btn : BUTTON_PRESS_DOWN");
                    break;
            }
            break;
        case BUTTON_PRESS_UP: //Release the button*/
            switch ((int)data)
            {
                case 0:
                    KEY_INFO("left btn : BUTTON_PRESS_UP");
                    break;
                case 1:
                    KEY_INFO("right btn : BUTTON_PRESS_UP");
                    break;
                case 2:
                    KEY_INFO("middle btn : BUTTON_PRESS_UP");
                    break;
                case 3:
                    KEY_INFO("up btn : BUTTON_PRESS_UP");
                    break;
            }
            key_status = Key_release;
            break;
        default:
            break;
    }
}
```

Najskôr sa spustí funkcia spätného volania po detegovaní udalosti stlačenia tlačidla. V rámci obslužnej rutiny udalosti stlačenia tlačidla je možné určiť identifikátor tlačidla, aby sa zistilo, ktoré tlačidlo udalosť spustilo. Pre každé zo štyroch tlačidiel sa príslušne priradí premenná stavu tlačidla. Stav uvoľnenia tlačidla nevyžaduje identifikáciu spúšťačieho tlačidla.

1.3.2 Súbor Kconfig

Hlavnou funkciou tohto súboru je pridať požadovanú konfiguráciu do súboru **sdkconfig**, čím sa umožní úprava určitých nastavení parametrov prostredníctvom grafického rozhrania. Tu sa číslo 16 vzťahuje na **GPIO_NUM_16**.

```
menu "BSP KEY Setup"
    config BSP_KEY_ENABLED
        bool "Enable KEY"
        default n
    if BSP_KEY_ENABLED
        config KEY_GPIO
            int "GPIO For KEY"
            default 16
    endif
endmenu
```

1.3.3 Súbor CMkaLists.txt

Funkčnosť tejto ukážkovej rutiny závisí predovšetkým od ovládača **bsp_key**. Aby bolo možné úspešne vyvolať obsah priečinka **bsp_key** z iných funkcií, je potrebné nakonfigurovať súbor **CMakeLists.txt** v priečinku **bsp_key**. Podrobnosti konfigurácie sú nasledovné:

```
FILE(GLOB_RECURSE component_sources "*.c")

idf_component_register(SRCS ${component_sources}
    INCLUDE_DIRS "include"
    REQUIRES driver button)
```

V tomto súbore **CMakeLists.txt** sú najskôr definované adresáre pre zdrojové súbory a hlavičkové súbory spolu s požadovanou knižnicou ovládača (knižnica tlačidiel). Následne sa tieto nastavenia zaregistrujú v systéme zostavovania pomocou príkazu **idf_component_register**, čím sa projektu umožní využívať funkčnosť ovládača **bsp_key**.

1.3.4 hlavný priečinok

Hlavný priečinok slúži ako hlavný adresár pre spúšťanie programu a obsahuje hlavnú spustiteľnú funkciu **main.c** a hlavičkový súbor **main.h** v priečinku **include**. Pridajte hlavný priečinok do súboru **CMakeLists.txt** systému zostavenia.

Súbor **main.h** odkazuje predovšetkým na požadované hlavičkové súbory: funkcie využívajúce ovládač **bsp_led** vyžadujú zaradenie hlavičkového súboru **bsp_led**, zatiaľ čo funkcie využívajúce ovládač **bsp_key** vyžadujú hlavičkový súbor **bsp_key**.

Nižšie je analýza programu **main.c**: inicializácia systému a vykonávanie kľúčových funkcií ADC.


```

err = adc_oneshot_new_unit(&init_cfg, &adc_handle); /*Create a handle to a specific ADC unit*/
if (err != ESP_OK)
    init_fail("new adc oneshot", err);

#ifdef CONFIG_BSP_LED_ENABLED
err = led_init(); /*RMT LED Initialization*/
if (err != ESP_OK)
    init_fail("led", err);
vTaskDelay(200 / portTICK_PERIOD_MS);
set_led_status(0x00000000); /*All the LEDs are off*/
#endif
#ifdef CONFIG_BSP_KEY_ENABLED
err = key_init(adc_handle); /*ADC Key Initialization*/
if (err != ESP_OK)
    init_fail("key", err);
err = key_register_cb(); /*Key press status registration callback*/
if (err != ESP_OK)
    init_fail("key register cb", err);
#endif

```

Tento kód sa nachádza vo funkcii `init`, ktorá slúži na ukladanie inicializačných funkcií vyžadujúcich vyvolanie a hodnotenie ich návratového stavu. Ak návratový stav nie je `ESP_OK`, kód vypíše chybovú správu a zastaví ďalšie vykonávanie. Po 200 ms oneskorení sa vykoná funkcia `'set_led_status'`, aby sa vymazali všetky LED displeje. Tento krok zabezpečuje, že všetky LED diódy sú po zapnutí vypnuté. Vyvolaním funkcie `'key_register_cb'` sa zaregistruje spätné volanie udalosti stlačenia klávesy.

Stojí za zmienku, že pred volaním inicializácie kľúča musíme najskôr vytvoriť nový identifikátor smerujúci na radič ADC. Tento identifikátor sa odovzdá do funkcie inicializácie kľúča na konfiguráciu.

```

void app_main(void)
{
    MAIN_INFO("-----Demo version-----");
    MAIN_INFO("-----Start the test-----");
    init();
#ifdef CONFIG_BSP_KEY_ENABLED
#ifdef CONFIG_BSP_LED_ENABLED
xTaskCreate(key_task, "key", 4096, NULL, configMAX_PRIORITIES - 5, &key); /*Create a thread for turning on the LED light by pressing a button*/
#endif
#endif
}

```

V rámci funkcie `app_main` vytvorte vlákno FreeRTOS na vykonanie detekcie a zodpovedajúceho efektu LED pre stlačenie klávesov.

`/* Vlákno na vykonanie stlačenia klávesy ADC key_task */`

```

void key_task(void *param)
{
    esp_err_t err = ESP_OK;
    while (1)
    {
        /*Process key events based on the current key_status*/
        switch (key_status)
        {
            case Key_idle:
                /*No key pressed*/
                vTaskDelay(10 / portTICK_PERIOD_MS);
                break;
            case Key_left:
                if (set_single_led_status(LED_BYTE_RED, 0xFF) != ESP_OK) /*Left button pressed - set LED to red*/
                {
                    MAIN_ERROR("Failed to set LED status");
                }
                key_status = Key_idle;
                break;
            case Key_right:
                if (set_single_led_status(LED_BYTE_BLUE, 0xFF) != ESP_OK) /*Right button pressed - set LED to blue*/
                {
                    MAIN_ERROR("Failed to set LED status");
                }
                key_status = Key_idle;
                break;
            case Key_middle:
                if (set_single_led_status(LED_BYTE_YELLOW, 0xFF) != ESP_OK) /*middle button pressed - set LED to yellow*/
                {
                    MAIN_ERROR("Failed to set LED status");
                }
                key_status = Key_idle;
                break;
        }
    }
}

```

RLX COMPONENTS

```

case key_up:
    if (set_single_led_status(LED_BYTE_GREEN, 0xFF) != ESP_OK) /*up button pressed - set LED to green*/
    {
        MAIN_ERROR("Failed to set LED status");
    }
    key_status = Key_idle;
    break;
case key_release:
    if (set_led_status(0x00000000) != ESP_OK) /*All the LEDs are off*/
    {
        MAIN_ERROR("Failed to set LED status");
    }
    key_status = Key_idle;
    break;
default:
    MAIN_ERROR("Unknown key status: %d", key_status);
    key_status = Key_idle;
    break;
}
}
}

```

www.rlx.sk

V rámci vlákna na vykonanie stlačenia klávesy ADC vytvorte slučku while. V rámci tejto slučky vyvolajte premennú stavu klávesy, aby ste určili stlačenie rôznych klávesov, a podľa toho rozsvietite príslušné LED diódy. Ak sa klávesa uvoľní, zhasnite všetky LED diódy. (Ak nedochádza k žiadnej operácii klávesy alebo po vykonaní efektov klávesy, nastavte premennú do stavu nečinnosti a zavedzte oneskorenie 10 ms, t. j. interval rozpoznávania skenovania je 10 ms.)

1.3.5 Súbor CMkaLists.txt

Aby bolo možné úspešne vyvolať obsah priečinka **bsp_key** v hlavnej funkcii, je potrebné nakonfigurovať súbor **CMakeLists.txt**, ktorý sa nachádza v hlavnom priečinku. Podrobnosti konfigurácie sú nasledovné:

```

FILE(GLOB_RECURSE main ${CMAKE_SOURCE_DIR}/main/*.c)

idf_component_register(SRCS ${main}
    INCLUDE_DIRS "include"
    REQUIRES bsp_led bsp_key)

```

Najskôr sa definujú adresáre pre zdrojové súbory a hlavičkové súbory spolu s požadovanou knižnicou ovládačov – konkrétne knižnicou ovládačov pre prepojenie **bsp_key**. Následne sa tieto nastavenia zaregistrujú v systéme zostavovania pomocou príkazu **idf_component_register**, čím sa umožní hlavnému programu využívať tieto funkcie ovládačov.

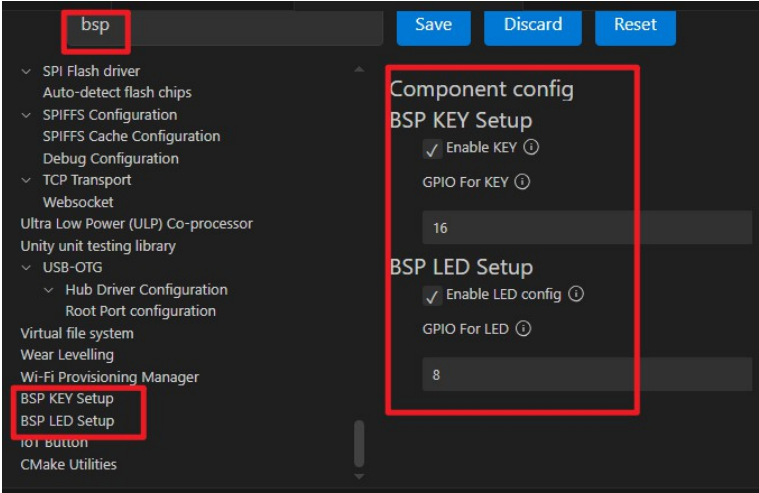
1.4 Postup programovania

Pripojte zariadenie P4 k počítaču prostredníctvom USB

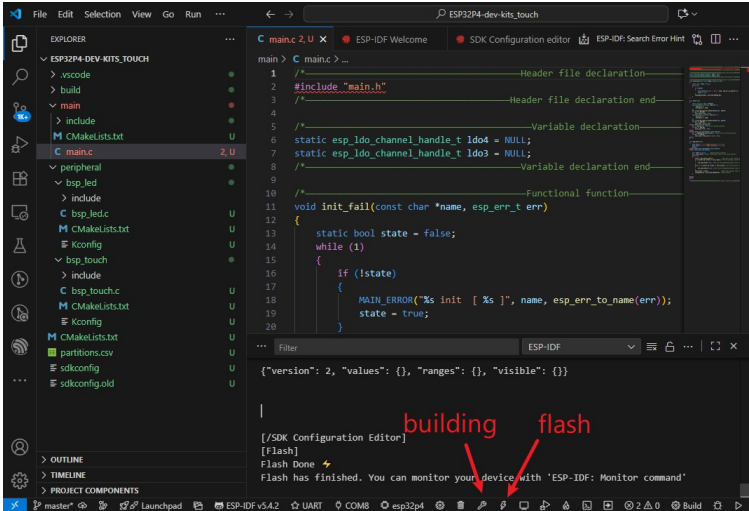


1.4.1 Po klonovaní kódu prostredníctvom Git (odkaz bude potvrdený) vymažte všetky lokálne informácie o kompilácii. Nakonfigurovajte prostredie IDF a model čipu pre kompiláciu podľa lekcie 1 a nastavte číslo sériového portu pre programovanie.

1.4.2 Nasledujúca konfigurácia SDKConfig je z veľkej časti identická s lekciou 1, jednoducho prekonfigurujte kľúčové a LED piny.



1.4.3 Kliknite na Kompilovať. Po úspešnej kompilácii kliknite na Stiahnuť.



RLX COMPONENTS

Lekcia 16 – Detektor dymu Alek

Úvod

Táto kapitola tutoriálu demonštruje použitie vstupov ADC ESP32-P4 pripojením detektora dymu MQ-2 na detekciu koncentrácie dymu v prostredí. MQ-2, bežný senzor plynu, detekuje horľavé plyny, vrátane LPG, butánu, metánu, alkoholu, vodíka a dymu. Tento experiment pomáha čitateľom pochopiť základné princípy analógového snímania ADC a snímania prostredia, čím vytvára základ pre následné projekty monitorovania IoT a bezpečnostných alarmov.

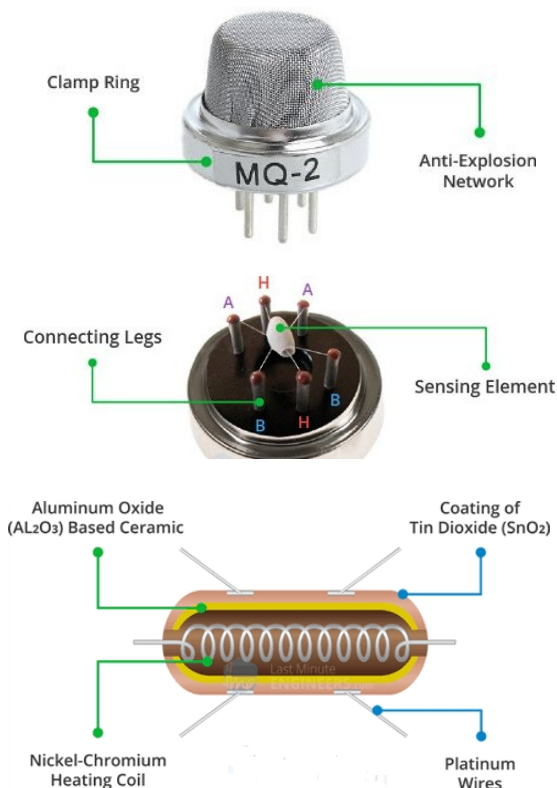
Demonštrácia projektu Efekt



Táto kapitola je rozdelená do nasledujúcich podkapitol

- 1.1 Úvod do senzora MQ-2
- 1.2 Konštrukcia hardvéru
- 1.3 Návrh programu
- 1.4 Stiahnutie a overenie

1.1 Úvod do senzora MQ-2



1.1.1 Princípy fungovania plynového senzora MQ-2

Senzor MQ-2 obsahuje vo svojej štruktúre ohrievač a plynovo citlivý rezistor (polovodičový materiál SnO_2). Jeho princíp fungovania je nasledovný:

- ① Pod vplyvom ohrievača sa molekuly kyslíka adsorbujú na povrchu materiálu citlivého na plyn, čo spôsobuje zmenu elektrického odporu;
- ② Ak sa vo vzduchu nachádzajú horľavé plyny alebo dym, molekuly kyslíka reagujú s cieľovým plynom a uvoľňujú elektróny;
- ③ Tieto elektróny opäť vstupujú do polovodiča, čím znižujú odpor rezistora citlivého na plyn a menia výstupné napätie;
- ④ Odčítaním hodnoty napätia prostredníctvom ADC je možné vypočítať zmeny koncentrácie plynu.

RLX COMPONENTS

1.1.2 Vzťah medzi simulovaným napätím a koncentráciou dymu

Čistý vzduch: Výstupné napätie je nízke (malá hodnota ADC);

Prítomnosť dymu alebo horľavých plynov: Výstupné napätie sa zvyšuje (väčšia hodnota ADC);

Prostredníctvom kalibrácie a prispôsobenia krivky je možné hodnoty ADC priradiť k relatívnym koncentráciám (ppm).

1.2 Hardvérová konštrukcia

Vstup senzora: Analógový výstupný pin MQ-2 sa pripája k GPIO17 (kanál ADC) na ESP32-P4;

Napájanie: Modul MQ-2 pracuje pri 5 V s výstupným napätím 0–3,3 V;

Výstup indikátora (voliteľné): LED dióda sa môže rozsvietiť, keď koncentrácia dymu prekročí stanovenú hranicu.

Schéma hardvérového pripojenia:

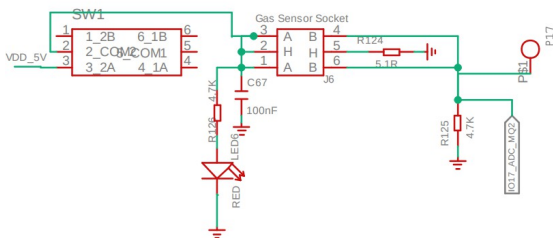
VCC → 5V GND

→ GND

AOUT → GPIO17 (vstup ADC)

GAS-MQ2
ADC

14

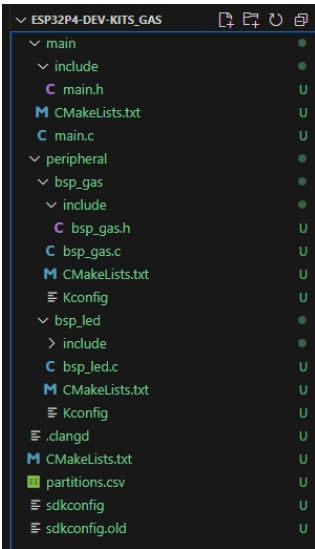


1.3 Analýza programu

<https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design>

Otvorte súbor projektu v VS Code podľa predchádzajúcich pokynov.

RLX COMPONENTS



V rámci príkladu **ESP32P4-dev-kits_gas** bola vytvorená nová zložka s názvom **bsp_gas** pod

ESP32P4-dev-kits_gas\peripheral. V adresári **bsp_gas** bol vytvorený nový priečinok **include**, súbor **CMakeLists.txt** a súbor **Kconfig**. Priečinok **bsp_gas** je určený na ukladanie súboru ovládača **bsp_gas.c**. Zložka **include** obsahuje súbor hlavičky **bsp_gas.h**, zatiaľ čo súbor **CMakeLists.txt** integruje ovládač do systému zostavovania, čím umožňuje projektové využitie jeho funkcionality. Súbor **Kconfig** načíta celý ovládač a definície pinov GPIO do súboru **sdkconfig** v rámci platformy IDF (konfigurovateľné prostredníctvom grafického rozhrania).

1.3.1 Kód ovládača ADC GAS

Tu sa zameriame výlučne na základný kód; podrobný zdrojový kód nájdete v príslušných zdrojových súboroch pre tento experiment v materiáloch kódu.

Zdrojový kód ovládača dymového senzora MQ2 pozostáva z dvoch súborov: **bsp_gas.c** a **bsp_gas.h**.

Nižšie najprv analyzujeme program **bsp_gas.h**: definuje príslušné piny pre senzor dymu a deklaruje použité funkcie.

/* Odkazy na hlavičkový súbor */

```
/*-----Header file declaration-----*/
#include <math.h>
#include "freertos/FreeRTOS.h" //References for FreeRTOS Function-related API Functions
#include "freertos/task.h" //References for FreeRTOS Task Function-related API Functions
#include "esp_log.h" //References for LOG Printing Function-related API Functions
#include "esp_err.h" //References for Error Type Function-related API Functions
#include "esp_adc/adc_oneshot.h" //References for ADC Oneshot Function-related API Functions
#include "esp_adc/adc_call.h" //References for ADC Call Function-related API Functions
#include "esp_adc/adc_call_scheme.h" //References for ADC Call Scheme Function-related API Functions
/*-----Header file declaration end-----*/
```

/* Deklarácie funkcií a deklarácie definícií makriér */

```
#ifndef CONFIG_BSP_GAS_ENABLED
#define GAS_CHANNEL CONFIG_GAS_CHANNEL // ADC acquisition channel

typedef struct
{
    adc_oneshot_unit_handle_t gas_handle;
    int gas_voltage;
    float gas_vault;
    float R_L;
    float R_A;
} gas_config;

esp_err_t gas_init(adc_oneshot_unit_handle_t handle); // GAS ADC Initialization Function
float get_gas_vault(); // Obtain the ADC acquisition voltage value of GAS
float get_R0_calibration(); // Obtain the R0 value (load resistance data under clean air conditions) through data conversion
float get_gas_data(); // Obtain the gas concentration value of gas(ppm)
#endif
```

Táto štruktúra definuje parametre pre zaťažovací odpor dymového senzora, ADC rukoväť a získané hodnoty, čo uľahčuje následné ovládanie funkcií.

Ďalej budeme analyzovať program `bsp_gas.c`: inicializácia konfigurácie kanála ADC, výpočet hodnôt napätia, konverzia parametrov ppm a vystavenie funkcií rozhrania API.

/* Inicializačná funkcia ADC GAS `gas_init` */

```
esp_err_t gas_init(adc_oneshot_unit_handle_t handle)
{
    esp_err_t err = ESP_OK;

    adc_oneshot_chan_cfg_t adc_config = {
        .bitwidth = SOC_ADC_DIGI_MAX_BITWIDTH, /*ADC conversion result bits*/
        .atten = ADC_ATTEN_DB_12, /*ADC attenuation*/
    };
    gas_data.gas_handle = handle; /*Obtain the handle of the ADC controller*/
    err = adc_oneshot_config_channel(gas_data.gas_handle, GAS_CHANNEL, &adc_config); /*Set ADC oneshot mode required configurations*/
    if (err != ESP_OK)
    {
        GAS_ERROR("create adc oneshot channel fail");
        return err;
    }

    adc_cal_curve_fitting_config_t cali_config = {
        .unit_id = ADC_UNIT_1, /*Use ADC unit 1*/
        .chan = GAS_CHANNEL, /*Use ADC channel 1*/
        .atten = ADC_ATTEN_DB_12, /*ADC attenuation*/
        .bitwidth = SOC_ADC_DIGI_MAX_BITWIDTH, /*ADC conversion result bits*/
    };
    err = adc_cal_create_scheme_curve_fitting(&cali_config, &gas_cal_handle); /*Create a Curve Fitting calibration scheme*/
    if (err != ESP_OK)
    {
        GAS_ERROR("create adc scheme curve fitting fail");
        return err;
    }
    gas_data.RL = 4.7;
    return err;
}
```

V rámci funkcie `gas_init` sa najskôr vykoná konfigurácia štruktúry `adc_oneshot_chan_cfg_t`, čím sa nastaví šírka čítacieho bitu ADC. Následne sa vyvolá funkcia `adc_oneshot_config_channel` na inicializáciu kanála ADC. Následne sa nakonfiguruje štruktúra `adc_cali_curve_fitting_config_t`. Táto štruktúra podobne nastavuje parametre pre ADC handle, pričom špecifikuje relevantný kalibračný schématický diagram konfigurácie kanála. Špecifická funkcia na vytvorenie je `adc_cali_create_scheme_curve_fitting`. (Konečná hodnota `RL=4,7` je predvolená; v hardvérovom dizajne má tento rezistor skutočne hodnotu 4,7 kΩ.)

V rámci funkcie `gas_init` sa najskôr vykoná konfigurácia štruktúry `adc_one-shot_chan_cfg_t`. Pozoruhodným aspektom konfiguračných hodnôt je, že čip ESP32P4 v súčasnosti podporuje iba schému kalibrácie prispôsobenia krivky.

/* Funkcia `get_gas_voltage` na získanie aktuálnej hodnoty napätia GAS prostredníctvom ADC */

```
esp_err_t get_gas_voltage()
{
    static int vol_sum = 0;
    static uint8_t vol_reading_cnt = 0;
    esp_err_t err = ESP_OK;
    int vauale = 0;
    while (1)
    {
        err = adc_oneshot_read(gas_data.gas_handle, GAS_CHANNEL, &vauale); /*Get one ADC conversion raw result*/
        if (err != ESP_OK)
        {
            return err;
        }
        vol_sum += vauale; /*Accumulate the reading for averaging*/
        vol_reading_cnt++;
        if (vol_reading_cnt == 2) /*Process when we have collected 2 samples*/
        {
            err = adc_cal_raw_to_voltage(gas_cal_handle, vol_sum / vol_reading_cnt, &gas_data.gas_voltage); /*Convert ADC raw data to calibrated voltage*/
            if (err != ESP_OK)
            {
                GAS_ERROR("GAS adc voltage call fail");
                return err;
            }
            vol_sum = 0;
            vol_reading_cnt = 0; /*Reset accumulators for next reading cycle*/
            gas_data.gas_vauale = ((float)gas_data.gas_voltage) / 1000; /*Convert millivolts to volts (divide by 1000)*/
            break; /*Exit the while loop after successful processing*/
        }
        vTaskDelay(10 / portTICK_PERIOD_MS);
    }
    return err;
}
```



```
float get_gas_vaule()
{
    return gas_data.gas_vaule; /*Return the voltage value collected by the ADC*/
}
```

V rámci funkcie **gas_init** sa najskôr vykoná konfigurácia štruktúry **adc_one-shot_chan_cfg_t**.

Táto funkcia najprv vyvolá **adc_oneshot_read**, aby získala vzorkovanú hodnotu napätia z aktuálneho kanála ADC. Prostredníctvom akumulatívneho prístupu sčíta každé dve získané vzorkované hodnoty pred vykonaním výpočtu priemeru. Výsledná hodnota napätia sa potom odovzdá ako vstupný parameter funkcii kalibrácie prispôsobenia krivky na kalibráciu. Nakoniec sa kalibrovaná hodnota napätia prevádza z milivoltov (mv) na volty.

(V). Ak sú na odstránenie chýb potrebné parametre napätia, je možné ich získať pomocou funkcie **get_gas_value**.

/* Funkcia kalibrácie odporu plynového zaťaženia (podmienky čistého vzduchu)
get_r0_calibration */

```
float get_r0_calibration()
{
    static uint8_t vol_reading_cnt = 0;
    static float vol_sum = 0;
    while (1)
    {
        if (get_gas_voltage() == ESP_OK) /*Attempt to read current gas sensor voltage*/
        {
            vol_sum += gas_data.gas_vaule; /*Accumulate voltage value*/
            vol_reading_cnt++; /*Increment reading counter*/
        }
        if (vol_reading_cnt >= 10) /*Check if we have collected enough samples (10 readings)*/
        {
            /*
             * Calculate R0 using the voltage divider formula:
             * R0 = ((Vcc - Vavg) * RL) / Vavg
             * Where Vavg = vol_sum / vol_reading_cnt
             */
            /* This formula derives from:
             * Rs = (Vcc - Vout) * RL / Vout
             * Where Rs is sensor resistance, and in clean air Rs = R0
             */
            gas_data.R0 = ((5 - (vol_sum / vol_reading_cnt)) * gas_data.RL) / (vol_sum / vol_reading_cnt);
            vol_reading_cnt = 0; /*Reset accumulators for potential future calibrations*/
            vol_sum = 0;
            break;
        }
        vTaskDelay(100 / portTICK_PERIOD_MS);
    }
    return gas_data.R0;
}
```

Táto funkcia využíva funkciu **get_gas_voltage** na získanie hodnoty napätia, ktorá je aktuálne spätne dodávaná z pinu senzora plynu. Prostredníctvom priemerného filtrovania získa desať vzoriek a vypočíta priemernú hodnotu pre konverziu. Následne sa odvodí aktuálna hodnota zaťažovacieho odporu pomocou vzorca deliča napätia:

$$RS = (Vcc - Vout) * RL / Vout.$$

Treba poznamenať, že táto metóda je určená na použitie v podmienkach čistého vzduchu a vypočítava počiatočnú hodnotu zaťažovacieho odporu platnú pre čistý vzduch.

/*funkcia plynu na získanie aktuálnej hodnoty ppm vzduchu **get_gas_data***/

```

float get_gas_data()
{
    float RS = 0;           /*Current sensor resistance in ohms*/
    float ppm = 0;          /*Gas concentration in Parts Per Million*/
    if (get_gas_voltage() == ESP_OK) /*Read current gas sensor voltage*/
    {
        /*
         * Calculate current sensor resistance (RS) using voltage divider formula:
         * RS = ((Vcc - Vout) * RL) / Vout
         * Where:
         * - Vcc = 5V (supply voltage)
         * - Vout = gas_data.gas_vault (measured voltage across load resistor)
         * - RL = gas_data.RL (load resistor value in ohms)
         *
         * This formula derives from: Vout = Vcc * RL / (RS + RL)
         */
        RS = ((5 - gas_data.gas_vault) * gas_data.RL) / gas_data.gas_vault;
        /*
         * Convert resistance ratio to PPM using calibration curve:
         * ppm = ((11.5428 * R0) / RS)^0.6549 * 100
         * Where:
         * - R0 = gas_data.R0 (baseline resistance in clean air)
         * - RS = current sensor resistance
         *
         * This empirical formula is specific to the gas sensor type and characteristics.
         * The constants (11.5428 and 0.6549) are derived from the sensor's datasheet
         * or calibration data for a specific gas (e.g., MQ-2 for LPG, MQ-135 for air quality).
         */
        ppm = pow(((11.5428 * gas_data.R0) / RS), 0.6549f) * 100;
    }
    return ppm;
}

```

Táto funkcia využíva funkciu **get_gas_voltage** na získanie hodnoty napätia, ktorá je aktuálne spätne dodávaná z pinu senzora plynu. Vypočíta hodnotu odporu RS pomocou vzorca deliča napätia (uvedeného vyššie) a potom na spracovanú hodnotu odporu RS aplikuje vzorec na prepočet ppm. (nastaviteľné na základe konkrétnych experimentov; tu sa používa predvolený vzorec na kalibráciu koncentrácie senzora plynu). Tu R0 označuje hodnotu odporu senzora v čistom vzduchu, RS predstavuje hodnotu odporu senzora v aktuálnom prostredí plynu, 11,5428 je charakteristický parameter senzora (kalibrovaný výrobcom) a 0,6549 je parameter krivky odozvy senzora. Výraz $(11,5428 \cdot R_0) / RS$ predstavuje pomer aktuálnej koncentrácie plynu vo vzťahu k referenčnému stavu. Funkcia Pow je funkcia výpočtu mocniny v jazyku C. Keďže reakcia senzora plynu je nelineárna, na prispôbenie je potrebná exponenciálna funkcia.

1.3.2 Súbor Kconfig

Hlavnou funkciou tohto súboru je pridať požadovanú konfiguráciu do súboru **sdkconfig**, čím sa umožní modifikácia určitých nastavení parametrov prostredníctvom grafického rozhrania. Tu „1“ označuje vzorkovací kanál ADC 1.

```

menu "BSP GAS Setup"
    config BSP_GAS_ENABLED
        bool "Enable GAS"
        default n
    if BSP_GAS_ENABLED
        config GAS_CHANNEL
            int "ADC CHANNEL For GAS"
            default 1
        endif
    endif
endmenu

```

1.3.3 Súbor CMkaLists.txt

www.rlx.sk

Funkčnosť tejto ukážkovej rutiny závisí predovšetkým od ovládača **bsp_gas**. Aby ste mohli úspešne volať funkcie zo zložky **bsp_gas** v rámci iných funkcií, musíte nakonfigurovať súbor **CMakeLists.txt**, ktorý sa nachádza v zložke **bsp_gas**. Podrobnosti konfigurácie sú nasledovné:

```
FILE(GLOB_RECURSE component_sources "*.c")

idf_component_register(SRCS ${component_sources}
                      INCLUDE_DIRS "include"
                      REQUIRES driver_esp_adc)
```

V tomto súbore **CMakeLists.txt** sú najskôr definované adresáre pre zdrojové súbory a hlavičkové súbory spolu s požadovanou knižnicou ovládača (knižnica **esp_adc**). Následne sa tieto nastavenia zaregistrujú v systéme zostavovania pomocou príkazu

idf_component_register, čím sa projektu umožní využívať funkčnosť ovládača **bsp_gas**.

1.3.4 Hlavný priečinok

Hlavný priečinok slúži ako hlavný adresár pre vykonávanie programu a obsahuje hlavnú funkciu **main.c** a hlavičkový súbor **main.h** v priečinku **include**. Pridajte hlavný priečinok do súboru **CMakeLists.txt** systému zostavenia.

Súbor **main.h** odkazuje predovšetkým na požadované hlavičkové súbory: funkcie využívajúce ovládač **bsp_led** vyžadujú zaradenie hlavičkového súboru **bsp_led**, zatiaľ čo funkcie využívajúce ovládač **bsp_gas** vyžadujú hlavičkový súbor **bsp_gas**.

Nižšie je analýza programu **main.c**: inicializácia systému a inicializácia špecifická pre funkciu ADC.

```
err = adc_oneshot_new_unit(&init_cfg, &adc_handle);
if (err != ESP_OK)
    init_fail("new adc oneshot", err);
#ifdef CONFIG_BSP_LED_ENABLED
err = led_init(); /*HW LED Initialization*/
if (err != ESP_OK)
    init_fail("led", err);
vTaskDelay(200 / portTICK_PERIOD_MS);
set_led_status(0x00000000); /*All the LEDs are off*/
#endif
#ifdef CONFIG_BSP_GAS_ENABLED
err = gas_init(adc_handle); /*GAS ADC Initialization*/
if (err != ESP_OK)
    init_fail("gas", err);
float r0 = 0;
r0 = get_r0_calibration(); /*Obtain the R0 value (load resistance data under clean air conditions) through data conversion*/
MAIN_INFO("r0 = %.2f K", r0);
vTaskDelay(5000 / portTICK_PERIOD_MS);
#endif
```

Tento kód sa nachádza vo funkcii **init**, ktorá sa používa na ukladanie inicializačných funkcií vyžadujúcich vyvolanie a na vyhodnotenie ich návratového stavu. Ak návratový stav nie je **ESP_OK**, kód vytlačí chybovú správu a zastaví ďalšie vykonávanie. Po 200 ms oneskorení sa vykoná funkcia **set_led_status**, aby sa vymazali všetky LED displeje. Tento krok zabezpečuje, že všetky LED diódy sú po zapnutí v stave vypnutia. Vyvolaním funkcie **get_r0_calibration** sa nastaví predvolený počiatočný stav na podmienky čistého vzduchu pri zapnutí a kalibruje sa hodnota rezistora **R0**.

```
void app_main(void)
{
    MAIN_INFO("-----Demo version-----");
    MAIN_INFO("-----Start the test-----");
    Init();
#ifdef CONFIG_BSP_GAS_ENABLED
#ifdef CONFIG_BSP_LED_ENABLED
    xTaskCreate(gas_task, "gas", 4096, NULL, configMAX_PRIORITIES - 5, &gas); /*Create a thread for smoke alarm recognition and processing*/
#endif
#endif
}
```

RLX COMPONENTS

V rámci funkcie **app_main** vytvorte vlákno **FreeRTOS** na vykonanie monitorovania koncentrácie plynu a vizuálnych alarmových vývarov, ktoré sú vyvolané systémovým senzora.

/* Vlákno monitorovania koncentrácie plynu gas_task */

```
void gas_task(void *param)
{
    float ppm_data = 0; /*Current gas concentration in PPM*/
    float last_ppm_data = get_gas_data(); /*Previous gas concentration reading*/
    static bool state = false; /*LED blink state (true-on, false-off)*/
    uint32_t now_time = 0; /*Current time in milliseconds*/
    uint32_t last_time = 0; /*Last LED toggle time for blinking*/
    while (1)
    {
        now_time = esp_timer_get_time() / 1000; /*Get current time in milliseconds from system timer*/
        MAIN_INFO("gas_vault = %.2f", get_gas_vault()); /*Log raw gas sensor voltage value for debugging*/
        ppm_data = get_gas_data(); /*Read current gas concentration in PPM*/
        if (ppm_data >= 1000) /*Alarm condition: Gas concentration exceeds 1000 PPM threshold*/
        {
            if (now_time >= last_time)
            {
                if (now_time - last_time >= 500) /*Check if 500ms has elapsed since last LED toggle for blinking effect*/
                {
                    state = !state; /*Toggle LED state*/
                    if (state)
                    {
                        set_led_status(0x0FFFFFFF); /*Turn LED on*/
                    }
                    else
                    {
                        set_led_status(0x00000000); /*Turn LED off*/
                    }
                    last_time = now_time; /*Update last toggle time*/
                }
            }
            else
            {
                last_time = now_time; /*Handle timer overflow case (unlikely in normal operation)*/
            }
        }
        else if ((ppm_data < 1000) && (last_ppm_data >= 1000)) /*Alarm cancellation condition: Concentration drops below threshold after being above it*/
        {
            set_led_status(0x00000000); /*Turn off LED when gas concentration returns to safe levels*/
        }
        last_ppm_data = ppm_data; /*Store current reading for next iteration comparison*/
        MAIN_INFO("ppm = %.2f", ppm_data);
        vTaskDelay(20 / portTICK_PERIOD_MS);
    }
}
```

V rámci vlákna monitorovania koncentrácie plynu sa najskôr inicializujú premenné a získa sa počiatočná hodnota koncentrácie dymu na účely porovnania. Následne sa vytvorí cyklus **while**, v rámci ktorého sa vyvolá funkcia **esp_timer_get_time** na získanie aktuálneho času systému a funkcia **get_gas_data** na získanie aktuálnej hodnoty koncentrácie plynu. Ak aktuálna koncentrácia plynu presiahne **1000 ppm**, LED diódy blikajú raz za 500 ms (všetky štyri diódy svietia súčasne). To sa dosiahne odpočítaním predchádzajúceho systémového času od aktuálneho systémového času; ak je rozdiel väčší alebo rovný 500 ms, stavová premenná LED diód `state` sa invertuje. Ovládanie LED diód potom sleduje túto stavovú premennú. Ak je aktuálna koncentrácia plynu nižšia ako **1000 ppm** a predchádzajúca hodnota presiahla 1000 ppm, prah alarmu zostane nespustený a LED diódy zhasnú. Nakoniec sa aktuálna koncentrácia plynu priradí k predchádzajúcej hodnote pre následné vyhodnotenie. Oneskorenie 20 ms predstavuje 20-milisekundový interval medzi každým cyklom rozpoznávania a posudzovania.

1.3.5 CMkaLists.txt

Aby bolo možné úspešne vyvolať obsah priečinka **bsp_gas** v hlavnej funkcii, je potrebné nakonfigurovať súbor **CMakeLists.txt**, ktorý sa nachádza v hlavnom priečinku. Podrobnosti konfigurácie sú nasledovné:

```
FILE(GLOB_RECURSE main ${CMAKE_SOURCE_DIR}/main/*.c)

idf_component_register(SRCS ${main}
                        INCLUDE_DIRS "include"
                        REQUIRES esp_timer bsp_led bsp_gas)
```

Najsôr sa definujú adresáre pre zdrojové súbory a hlavičkové súbory spolu s požadovanými knižnicami ovládačov, konkrétne knižnicami ovládačov pre prepojenie **bsp_gas**, **bsp_led** a **esp_timer**. Následne sa tieto nastavenia zaregistrujú v systéme zostavovania pomocou príkazu **idf_component_register**, čím sa hlavnej funkcii umožní využívať tieto funkcie ovládačov.

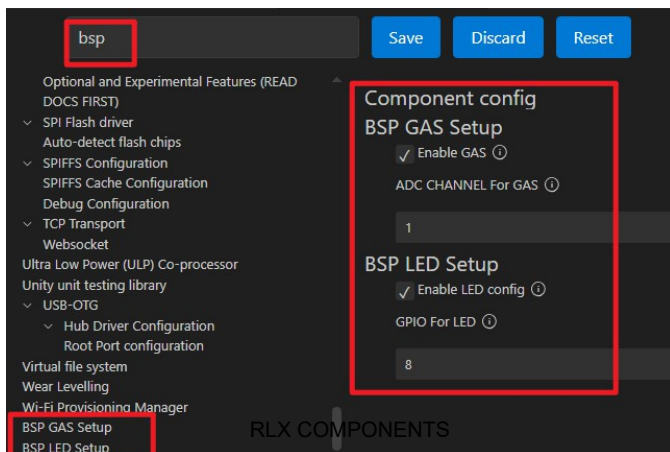
1.4 Postup programovania

Pripojte zariadenie P4 k počítaču prostredníctvom USB

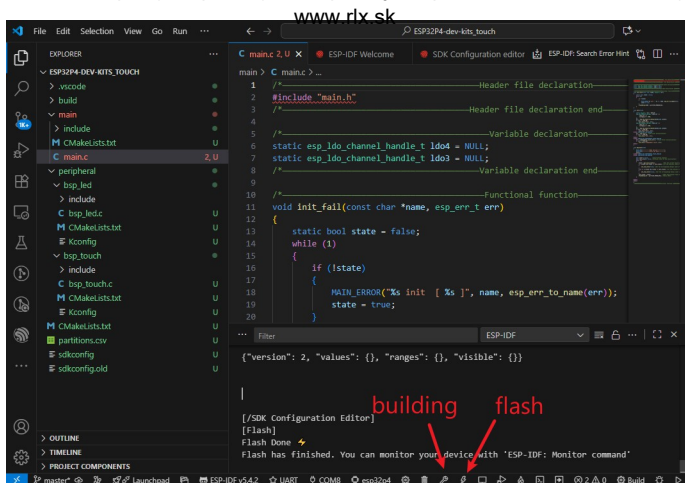


1.4.1 Po klonovaní kódu prostredníctvom Git (odkaz bude potvrdený) vymažte všetky lokálne informácie o kompilácii. Nakonfigurujte prostredie IDF a model čipu pre kompiláciu podľa lekcie 1 a nastavte číslo sériového portu pre programovanie.

1.4.2 Nasledujúca konfigurácia SDKConfig je z veľkej časti identická s lekciami 1, jednoducho prekonfigurujte kanál adc pre piny plynu a led.



1.4.3 Kliknite na Compile (Kompilovať). Po úspešnej kompilácii kliknite na Download (Stiahnuť).

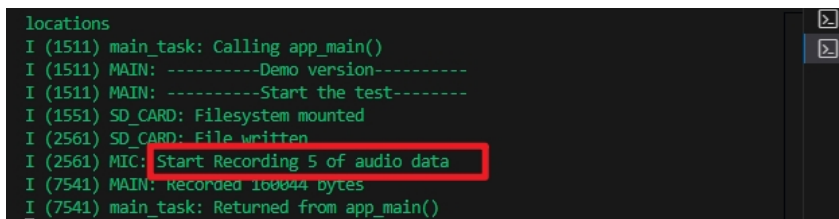


Lekcia 17 – Nahrávanie zvuku I2S

Úvod

Táto kapitola tutoriálu predstavuje aplikáciu na zachytávanie mikrofónu I2S-PDM a ukladanie zvuku pre ESP32-P4, pričom na pomoc pri pochopení základných funkcií zbernice I2S používa príklad nahrávania.

Ako typický prípad použitia zvukovej aplikácie, ukladanie nahrávok umožňuje čitateľom rýchlo pochopiť schopnosti ESP32-P4 v oblasti spracovania reči a zvuku, čím sa vytvára základ pre zložitejšie projekty, ako je rozpoznávanie reči a prehrávanie zvuku.



- 1.1 Úvod do mikrofónov I²S a PDM
- 1.2 Návrh hardvéru
- 1.3 Návrh softvéru
- 1.4 Stiahnutie a overenie

1.1 Úvod do mikrofónov I²S a PDM

1.1.1 I²S Úvod

Čip ESP32-P4 obsahuje viacero periférnych rozhraní I²S na získavanie a prehrávanie zvukových dát. Medzi jeho kľúčové vlastnosti patria:

- ① Podpora viacerých audio protokolov: vrátane štandardných režimov I²S, ľavého zarovnania, pravého zarovnania a PDM (Pulse Density Modulation).
- ② Podpora vysokej vzorkovacej frekvencie: Konfigurovateľné vzorkovacie frekvencie od 8 kHz do 192 kHz, vhodné na zachytávanie hlasu a spracovanie zvuku s vysokou vernosťou.
- ③ DMA prenos: Umožňuje priamy prenos audio dát prostredníctvom DMA, čím sa znižuje zaťaženie CPU a zvyšuje výkon v reálnom čase.
- ④ Podpora viacerých kanálov: súčasne spracováva mono, stereo a iné formáty dát, čo uľahčuje stereo nahrávanie alebo prehrávanie.
- ⑤ Flexibilná konfigurácia: Funguje ako hostiteľ alebo podriadené zariadenie a podporuje získavanie aj výstup údajov.

V tomto príklade využijeme rozhranie I²S ESP32-P4 v režime PDM mikrofónu na implementáciu získavania hlasových údajov.

1.1.2 Prehľad mikrofónu PDM

PDM (Pulse Density Modulation) je metóda digitálneho audio výstupu, ktorá sa bežne používa v mikrofónoch MEMS.

Vlastnosti PDM mikrofónov:

- ① Miniaturizácia: Kompaktná veľkosť a nízka spotreba energie, vhodné pre vstavané zariadenia.
- ② Digitálne rozhranie: Výstup priamo ako digitálny pulzný tok, čím sa eliminuje potreba analógových zosilňovačov.
- ③ Vysoká integrácia: Väčšina mikrofónov PDM obsahuje vstavané moduly na prevod analógového signálu na digitálny.
- ④ Flexibilné vzorkovanie: Dá sa zachytiť prostredníctvom rozhrania I²S-PDM, filtrovať a obnoviť na štandardné PCM dáta.

Princíp fungovania:

Keď zvukové vlny dopadajú na membránu mikrofónu, vnútorné kapacitné senzory prevádzajú akustickú energiu na elektrické signály. Tie sú potom spracované modulátorom $\Sigma\text{-}\Delta$, ktorý generuje vysokofrekvenčný signál s moduláciou hustoty impulzov (PDM).

ESP32-P4 vzorkuje tieto impulzy cez rozhranie I2S-PDM a pomocou digitálneho filtra ich rekonštruje na PCM audio dáta.

1.1.3 Úvod do súborov WAV

Súbory WAV sú bežným bezstratovým audio formátom. V hlavičke súboru sú uložené parametre, ako napríklad vzorkovacia frekvencia zvuku, počet kanálov a bitová hĺbka, zatiaľ čo časť s údajmi obsahuje PCM údaje zachytené zo vzoriek.

Jeho primárna štruktúra je nasledovná:

Blok RIFF: Identifikuje súbor ako formát WAV

blok fmt: opisuje formát zvuku (vzorkovaciu frekvenciu, šírku bitu, kanály atď.) blok

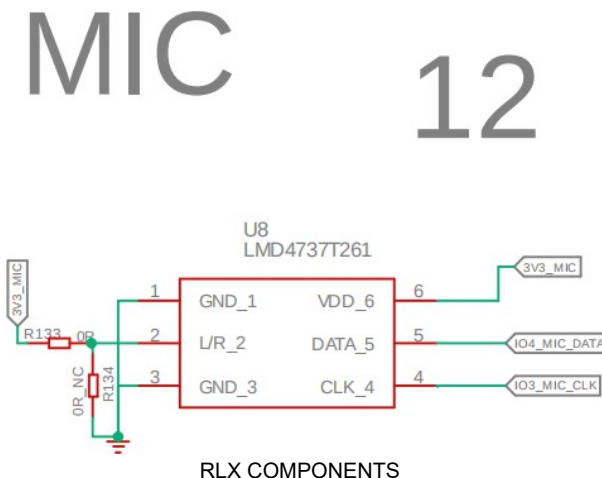
dát: ukladá skutočné zvukové dáta

Zapísaním PCM údajov zachytených mikrofónom PDM na SD kartu a ich zapuzdrením ako súbor WAV môžeme dosiahnuť štandardnú funkciu nahrávania zvuku.

1.2 Hardvérový dizajn

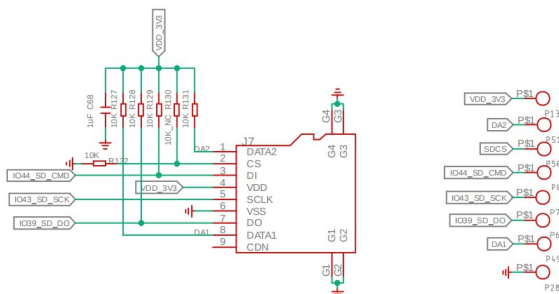
V tomto experimente je typické pripojenie medzi ESP32-P4, digitálnym mikrofónom PDM a modulom SD karty nasledovné:

Mikrofón PDM → rozhranie ESP32-P4 I2S



SD card

SPI

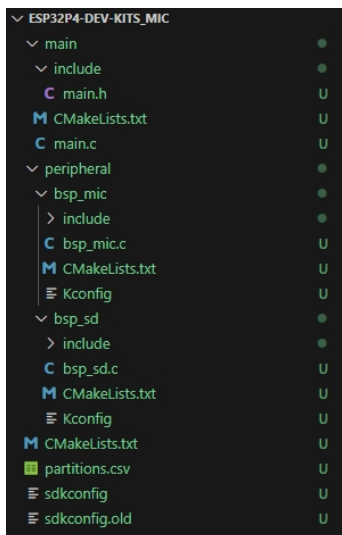


Týmto spôsobom ESP32-P4 získava zvukové dáta prostredníctvom I2S a zapisuje ich na SD kartu prostredníctvom SPI, čím vytvára súbory WAV.

1.3 Analýza programu

<https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design>

Otvorte súbor projektu v VS Code podľa predchádzajúcich pokynov.



V rámci príkladu **ESP32P4-dev-kits_mic** boli vytvorené nové priečinky **bsp_mic** a **bsp_sd** pod priečinkom **ESP32P4-dev-kits_mic/peripheral**. V rámci ciest **bsp_mic** a **bsp_sd** boli vytvorené nové priečinky **include**, súbor **CMakeLists.txt** a súbor **Kconfig**. Priečinkom **bsp_mic** obsahuje ovládačový súbor **bsp_mic.c**, zatiaľ čo priečinkom **bsp_sd** obsahuje ovládačový súbor **bsp_sd.c**. Ich príslušné priečinky **include** ukladajú súbory hlavičiek **.h**. Súbor **CMakeLists.txt** integruje ovládače do systému zostavovania, čím umožňuje projektové využitie funkcionality ovládačov. Súbor **Kconfig** načíta celú konfiguráciu ovládača, vrátane definícií pinov GPIO, do súboru **sdkconfig** v rámci platformy IDF (konfigurovateľného prostredníctvom grafického rozhrania).

RLX COMPONENTS

1.3.1 Kód ovládača SD karty

Tu sa zameriame výlučne na základný kód, pretože zdrojový kód nájdete v príslušných zdrojových súboroch pre tento experiment v materiáloch kódu.

Zdrojový kód ovládača SD karty pozostáva z dvoch súborov: **bsp_sd.c** a **bsp_sd.h**.

Nižšie najprv analyzujeme program **bsp_sd.h**: definuje príslušné piny SD karty a deklaruje použité funkcie.

/* Odkazy na súbory hlavičiek */

```
/*-----Header file declaration-----*/
#include "esp_vfs_fat.h" //References for vfs fat Function-related API Functions
#include "sdmmc_cmd.h" //References for sdmmc cmd Function-related API Functions
#include "driver/sdmmc_host.h" //References for sdmmc host Function-related API Functions
#include "esp_private/sdmmc_common.h" //References for sdmmc common Function-related API Functions
/*-----Header file declaration-----*/
```

/* Deklarácie funkcií a definície makier */

```
#ifndef CONFIG_BSP_SD_ENABLE
#define CONFIG_BSP_SD_ENABLE 1

// Maximum byte count
#define SD_MAX_COUNT 1024
// path where partition should be registered
#define SD_GPT0_CLK CONFIG10_SD_GPT0_CLK // SD SDIO CLK GPT0
#define SD_GPT0_CMD CONFIG10_SD_GPT0_CMD // SD SDIO CMD GPT0
#define SD_GPT0_DA CONFIG10_SD_GPT0_DA // SD SDIO DA GPT0

esp_err_t sd_init(); // SD Initialization Function
esp_err_t create_file(const char *filename); // Create a binary file
esp_err_t write_string_file(const char *filename, char *data); // Create and write to a text file
esp_err_t read_string_file(const char *filename); // Read to a text file
esp_err_t write_file(const char *filename, char *data, size_t size); // Create and write to a binary file
esp_err_t write_file_seek(const char *filename, void *data, size_t size, int32_t seek); // Create and write to a binary file that can offset the writing position
esp_err_t read_file(const char *filename, char *data, size_t size); // Read to a binary file
esp_err_t read_file_size(const char *filename); // Read the number of bytes in a binary file
esp_err_t format_sd_card(); // Format the SD card
void get_sd_card_info(); // Obtain information about the SD card device
#endif
```

Ďalej budeme analyzovať program **bsp_sd.c**: inicializuje a konfiguruje riadiacu zbernicu SD karty, pripája ju k súborovému systému a sprístupňuje funkcie rozhrania API.

/* Funkcia inicializácie SD karty a pripojenia súborového systému sd_init */

```
esp_err_t sd_init()
{
    esp_err_t err = ESP_OK;
    /*Configuration for FAT filesystem mounting*/
    esp_vfs_fat_sdmmc_mount_config_t mount_config = {
#ifdef CONFIG_FORMAT_IF_MOUNT_FAILED
        .format_if_mount_failed = true, /*Format SD card if mount fails*/
    else
        .format_if_mount_failed = false, /*Do not format if mount fails*/
    endif
    .max_files = 5, /*Maximum number of open files*/
    .allocation_unit_size = 16 * 1024, /*Cluster size for FAT filesystem (10KB)(Must be a power of 2)*/
    };
    /*SDMMC host controller configuration*/
    sdmmc_host_t host = {
        /*Supported bus modes (1-bit, 4-bit, 8-bit with DDR)*/
        .flags = SDMMC_HOST_FLAG_8BIT |
                SDMMC_HOST_FLAG_4BIT |
                SDMMC_HOST_FLAG_1BIT |
                SDMMC_HOST_FLAG_DDR |
                SDMMC_HOST_FLAG_DDEINIT_ARG,
        .slot = SDMMC_HOST_SLOT_0, /*Use SDMMC slot 0*/
        .max_freq_khz = SDMMC_FREQ_DEFAULT, /*Default frequency (20MHz)*/
        .io_voltage = 1.8f, /*I/O voltage level (3.3V)*/
        .driver_strength = SDMMC_DRIVER_STRENGTH_B, /*Default drive strength*/
        .current_limit = SDMMC_CURRENT_LIMIT_200mA, /*200mA current limit*/
        /*Host controller function pointers*/
        .init = &sdmmc_host_init,
        .set_bus_width = &sdmmc_host_set_bus_width,
        .get_bus_width = &sdmmc_host_get_bus_width,
        .set_bus_dir_mode = &sdmmc_host_set_bus_dir_mode,
        .set_card_clk = &sdmmc_host_set_card_clk,
        .set_ckick_always_on = &sdmmc_host_set_ckick_always_on,
        .do_transaction = &sdmmc_host_do_transaction,
        .deinit_p = &sdmmc_host_deinit_slot,
        .io_int_enable = &sdmmc_host_io_int_enable,
        .io_int_wait = &sdmmc_host_io_int_wait,
        .command_timeout_ms = 0, /*Use default timeout*/
        .get_real_freq = &sdmmc_host_get_real_freq,
        .input_delay_phase = SDMMC_DELAY_PHASE_0, /*No input delay*/
        .set_input_delay = &sdmmc_host_set_input_delay,
        .dma_aligned_buffer = 0, /*No DMA buffer*/
        .pwr_ctrl_handle = NULL, /*No power control*/
        .get_dma_info = &sdmmc_host_get_dma_info,
        .is_slot_set_to_uhs1 = &sdmmc_host_is_slot_set_to_uhs1,
    };
}
```

RLX COMPONENTS

V rámci funkcie ``sd_init`` sa konfigurácia začína štruktúrou ``esp_vfs_fat_sdmmc_mount_config_t``, ktorá nastavuje parametre pre pripojenie súborového systému SD karty. Následne sa konfiguruje štruktúra ``sdmmc_host_t`` na správu nastavení hostiteľského radiča SDMMC (podrobnosti o konkrétnych parametroch nájdete v komentároch kódu). Štruktúra ``sdmmc_slot_config_t`` konfiguruje SD kartu tak, aby využívala režim jedného vodiča na zbernici SDIO. Nakoniec sa vyvolá funkcia `esp_vfs_fat_sdmmc_mount`, aby sa inicializovala SD karta a pripojil súborový systém. Ak sa pripojenie nepodarí, zaznamená sa typ chyby.

Stojí za zmienku, že na čipe ESP32P4 sa pri prevádzke v režime Wi-Fi/Bluetooth hostiteľského zariadenia musí vybraný slot zbernice SDIO líšiť od slotu používaného v režime Wi-Fi/Bluetooth hostiteľského zariadenia.

/* Funkcia `get_sd_card_info` získava informácie o aktuálnej pripojenej SD karte

*/

```
void get_sd_card_info()
{
    bool print_scr = false;           //Flag to control SCR register printing*/
    bool print_csd = false;          //Flag to control CSD register printing*/
    const char *type;                //String describing card type*/
    SD_INFO("name: %s\n", card->cid.name); //Display card name from card identification register*/
    if (card->is_sdio)                //Determine card type and set appropriate print flags*/
    {
        type = "SDIO";              //SD Input/output card*/
        print_scr = true;            //SDIO cards have SCR register*/
        print_csd = true;            //SDIO cards have CSD register*/
    }
    else if (card->is_mmc)
    {
        type = "MMC";               //MultiMediaCard*/
        print_csd = true;            //MMC cards have CSD register*/
    }
    else
    {
        if ((card->ocr & SD_OCR_SDHC_CAP) == 0) //Standard SD card type detection*/
            type = "SDSC";           //Standard Capacity SD card (up to 2GB)*/
        else
        {
            if (card->ocr & SD_OCR_S18_RA)
                type = "SDHC/SDXC (UHS-I)"; //High/Extended Capacity with UHS-I support*/
            else
                type = "SDHC";          //High Capacity SD card (4GB-32GB)*/
        }
        print_csd = true;            //All SD cards have CSD register*/
    }
    SD_INFO("Type: %s\n", type);
    if (card->real_freq_khz == 0) //Display current and maximum operating frequencies*/
        SD_INFO("Speed: N/A\n"); //Frequency information not available*/
    else
    {
        //Convert frequency to appropriate units (KHz or MHz)*/
        const char *freq_unit = card->real_freq_khz < 1000 ? "kHz" : "MHz";
        const float freq = card->real_freq_khz < 1000 ? card->real_freq_khz : card->real_freq_khz / 1000.0;
        const char *max_freq_unit = card->max_freq_khz < 1000 ? "kHz" : "MHz";
        const float max_freq = card->max_freq_khz < 1000 ? card->max_freq_khz : card->max_freq_khz / 1000.0;
        //Display current speed, maximum capability, and ODM mode if applicable*/
        SD_INFO("Speed: %2f %s (limit: %2f %s)\n",
            freq, freq_unit,
            max_freq, max_freq_unit,
            card->is_ddr ? ", ODM" : ""); //Double Data Rate Indicator*/
    }
}
```

```
/*Calculate and display total card capacity in MegaBytes*/
SD_INFO("Size: %lluMB\n", ((uint64_t)card->csd.capacity * card->csd.sector_size / (1024 * 1024));
//Display Card Specific Data register information if enabled*/
if (print_csd)
{
    SD_INFO("CSD: ver=%d, sector-size=%d, capacity=%d read_b1_len=%d\n",
        (uint32_t)card->is_mmc ? card->csd.ver : card->csd.ver + 1,
        card->csd.sector_size, //Size of each sector in bytes*/
        card->csd.capacity, //Total number of sectors*/
        card->csd.read_block_len); //Maximum read block length*/
    //Display extended information based on card type*/
    if (card->is_mmc)
    {
        SD_INFO("EXT CSD: bus_width=%d Result %s",
            (uint32_t)(1 <= card->bus_width); //MMC bus width (1,4,8 bits)*/
    }
    else if (card->is_sdio)
    {
        //Display SD Card Status Register bus width information*/
        SD_INFO("SDR: bus_width=%d Result %s",
            (uint32_t)(card->ssr.cur_bus_width > 4 ? 1); //4-bit or 1-bit mode*/
    }
}
//Display SD Configuration Register information if enabled*/
if (print_scr)
{
    SD_INFO("SCR: sd_spec=%d, bus_width=%d\n",
        card->scr.sd_spec, //SD physical layer specification number*/
        card->scr.bus_width); //Supported bus width*/
}
```

Toto volanie funkcie inicializuje identifikátor „card“ získaný z SD karty, prečíta relevantné informácie o SD karte a vytlačí informácie do protokolu.

/* Funkcia formátovania SD karty format_sd_card */

```
esp_err_t format_sd_card()
{
    esp_err_t err = ESP_OK;
    err = esp_vfs_fat_sdcard_format(sd_mount_point, card); /*Format FAT filesystem*/
    if (err != ESP_OK)
    {
        SD_ERROR("Failed to format FATFS (%s)", esp_err_to_name(err));
        return err;
    }
    return err;
}
```

Táto funkcia formátuje špecifikovanú SD kartu pomocou esp_vfs_fat_sdcard_format.

/* Operácie so súbormi na SD karte */

create_file: Vytvorí binárny súbor.

write_string_file: Zapiše do textového

súboru. read_string_file: Prečíta z

textového súboru. write_file: Zapiše

binárny súbor.

write_file_seek: Zapiše binárny súbor (s posunom, čo umožňuje zapisovať za počiatočnou pozíciou súboru).

read_file: Číta binárny súbor.

read_file_size: Číta veľkosť dát binárneho súboru.

read_write_file: Číta binárny súbor a zapisuje jeho obsah do iného binárneho súboru.

1.3.2 Súbor Kconfig

Hlavnou funkciou tohto súboru je pridať požadovanú konfiguráciu do súboru sdkconfig, čím sa umožní modifikácia určitých nastavení parametrov prostredníctvom grafického rozhrania. Čísla tu zodpovedajú príslušným číslam pinov GPIO.

Je dôležité poznamenať, že ak je povolená konfigurácia FORMAT_IF_MOUNT_FAILED, SD karta bude naformátovaná v prípade, že inicializácia a pripojenie zlyhajú.

```
menu "BSP SD Setup"
    config BSP_SD_ENABLED
        bool "Enable SD CARD"
        default n

    if BSP_SD_ENABLED
        config FORMAT_IF_MOUNT_FAILED
            bool "Format the card if mount failed"
            default n
            help
                If this config item is set, format_if_mount_failed will be set to true and the card will be formatted if
                the mount has failed.

        config SD_GPIO_CLK
            int "SDIO CLK GPIO number"
            default 43

        config SD_GPIO_CMD
            int "SDIO CMD GPIO number"
            default 44

        config SD_GPIO_D0
            int "SDIO D0 GPIO number"
            default 39
    endif
endmenu
```

RLX COMPONENTS

1.3.3 Súbor CMkaLists.txt

www.rlx.sk

Funkčnosť tejto ukážkovej rutiny závisí predovšetkým od ovládača **bsp_sd**. Aby bolo možné úspešne volať funkcie zo zložky **bsp_sd** v rámci iných funkcií, je potrebné nakonfigurovať súbor **CMakeLists.txt**, ktorý sa nachádza v zložke **bsp_sd**. Podrobnosti konfigurácie sú nasledovné:

```
FILE(GLOB_RECURSE component_sources "*.c")

idf_component_register(SRCS ${component_sources}
                       INCLUDE_DIRS "include"
                       REQUIRES fatfs)
```

V tomto súbore **CMakeLists.txt** sú najskôr definované adresáre pre zdrojové súbory a hlavičkové súbory spolu s požadovanou knižnicou ovládača (knižnica fatfs). Následne sa tieto nastavenia zaregistrujú v systéme zostavovania pomocou príkazu **idf_component_register**, čím sa projektu umožní využívať funkčnosť ovládača **bsp_sd**.

1.3.4 Kód ovládača mikrofónu

Tu sa zameriame výlučne na základný kód; podrobný zdrojový kód nájdete v príslušných zdrojových súboroch pre tento experiment v materiáloch kódu.

Zdrojový kód ovládača mikrofónu pozostáva z dvoch súborov: **bsp_mic.c** a **bsp_mic.h**.

Nižšie najprv analyzujeme program **bsp_mic.h**: definuje piny mikrofónu a deklaruje príslušné funkcie.

/* Odkazy na súbory hlavičiek */

```
/*-----Header file declaration-----*/
#include "esp_log.h" //References for LOG Printing Function-related API Functions
#include "esp_err.h" //References for Error Type Function-related API Functions
#include "driver/i2s_pdm.h" //References for I2S PDM Function-related API Functions
#ifdef CONFIG_BSP_SD_ENABLED
#include "bsp_sd.h"
#endif
/*-----Header file declaration end-----*/
```

/* Deklarácie funkcií a definície makier */

```
#ifdef CONFIG_BSP_MIC_ENABLED

#define MIC_GPIO_CLK CONFIG_MIC_GPIO_CLK // Microphone CLK GPIO
#define MIC_GPIO_DATA CONFIG_MIC_GPIO_DATA // Microphone DATA GPIO
#define MIC_SAMPLE_RATE CONFIG_MIC_SAMPLE_RATE // Microphone recording sampling rate
#define BYTE_RATE (MIC_SAMPLE_RATE * (16 / 8)) * 1 // The data volume recorded by the microphone in one second

esp_err_t mic_init(); // Microphone Initialization Function
#ifdef CONFIG_BSP_SD_ENABLED
esp_err_t mic_readwav_to_sd(const char *filename, size_t rec_seconds, size_t *out_size); // Microphone recording saved to SD card
#endif
#endif
```

Ďalej budeme analyzovať program **bsp_mic.c**: konfigurácia mikrofónu pre režim I2S PDM a sprístupnenie funkcií rozhrania API.

/* Funkcia inicializácie mikrofónu mic_init */

www.rlx.sk

```
esp_err_t mic_init()
{
    esp_err_t err = ESP_OK;
    /*I2S channel configuration for receiver (microphone)*/
    i2s_chan_config_t rx_chan_cfg = {
        .id = I2S_NUM_0, /*Use I2S controller 0*/
        .role = I2S_ROLE_MASTER, /*I2S acts as master (generates clock signals)*/
        .dma_desc_num = 6, /*Number of DMA descriptors (affects buffer management)*/
        .dma_frame_num = 256, /*Number of frames per DMA descriptor*/
        .auto_clear_after_cb = false, /*Don't auto-clear DMA buffer after callback*/
        .auto_clear_before_cb = false, /*Don't auto-clear DMA buffer before callback*/
        .allow_pd = false, /*Don't allow power down during operation*/
        .intr_priority = 0, /*Interrupt priority (0 = default)*/
    };
    /*Create new I2S channel (receive channel only, no transmit channel)*/
    err = i2s_new_channel(&rx_chan_cfg, NULL, &rx_chan);
    if (err != ESP_OK)
        return err;
    /*PDM (Pulse Density Modulation) receiver configuration*/
    i2s_pdm_rx_config_t pdm_rx_cfg = {
        /*Clock configuration*/
        .clk_cfg = {
            .sample_rate_hz = MIC_SAMPLE_RATE, /*Sampling frequency (e.g., 16kHz, 44.1kHz)*/
            .clk_src = I2S_CLK_SRC_DEFAULT, /*Default clock source (usually APB1 or PLL)*/
            .mclk_multiple = I2S_MCLK_MULTIPLE_256, /*Master clock multiple (256 x sample rate)*/
            .dn_sample_mode = I2S_PDM_DSR_BS, /*Downsampling mode (Bs for PDM to PCM conversion)*/
            .bclk_div = 8, /*Bit clock divider*/
        },
        /*Slot configuration (data format)*/
        .slot_cfg = {
            .data_bit_width = I2S_DATA_BIT_WIDTH_16BIT, /*Data bit width (16-bit for PDM)*/
            .slot_bit_width = I2S_SLOT_BIT_WIDTH_AUTO, /*Auto-calculate slot bit width*/
            .slot_mode = I2S_SLOT_MODE_MONO, /*Mono audio (single channel)*/
            .slot_mask = I2S_PDM_SLOT_LEFT, /*Use left slot for mono data*/
            .hp_en = true, /*Enable high-pass filter*/
            .hp_cut_off_freq_hz = 35.5, /*High-pass filter cutoff frequency (35.5Hz)*/
            .amplify_num = 1, /*Amplification factor (1 = no amplification)*/
        },
        /*GPIO pin configuration*/
        .gpio_cfg = {
            .clk = MIC_GPIO_CLK, /*GPIO pin for PDM clock signal*/
            .din = MIC_GPIO_DATA, /*GPIO pin for PDM data input*/
            .invert_flags = {
                .clk_inv = false, /*Don't invert clock polarity*/
            },
        },
    };
}

err = i2s_channel_init_pdm_rx_mode(rx_chan, &pdm_rx_cfg); /*Initialize I2S channel in PDM receiver mode*/
if (err != ESP_OK)
    return err;
err = i2s_channel_enable(rx_chan); /*Enable the I2S channel to start receiving audio data*/
if (err != ESP_OK)
    return err;
return err;
}
```

V rámci funkcie `mic_init` sa konfigurácia začína štruktúrou `i2s_chan_config_t`, ktorá nastavuje parametre pre radič I2S používaný mikrofónom (v tomto prípade sa používa radič I2S 0). Následne sa vyvolá funkcia `i2s_new_channel`, aby sa zaregistroval prijímací kanál na radiči I2S 0. Štruktúra `i2s_pdm_rx_config_t` konfiguruje príjem formátu PDM. Šírku dátového bitu môžeme zmeniť úpravou parametrov `data_bit_width` a `slot_bit_width` v poli `slot_cfg`. Úprava parametrov `slot_mode` a `slot_mask` v rámci `slot_cfg` zmení režim prijímania mikrofónu na stereo, mono atď. Nakoniec vyvolajte funkciu `i2s_channel_init_pdm_rx_mode`, aby ste inicializovali prijímač pomocou formátu PDM, a potom vyvolajte `i2s_channel_enable`, aby ste aktivovali prijímač.

Poznámka: V tomto kurze čítame dáta v mono režime, konkrétne z ľavého kanála.

/* Funkcia `generate_wav_header` na konfiguráciu hlavičky súboru vo formáte WAV */

```
static void generate_wav_header(char *wav_header, uint32_t wav_size, uint32_t sample_rate)
{
    uint32_t file_size = wav_size + 44 - 8;
    uint32_t byte_rate = BYTE_RATE;
    const char set_wav_header[] = {
        'R', 'I', 'F', 'F', /*ChunkID*/
        file_size, file_size >> 8, file_size >> 16, file_size >> 24, /*ChunkSize*/
        'W', 'A', 'V', 'E', /*Format*/
        'f', 'f', 'f', 'f', /*Subchunk1ID*/
        0x10, 0x00, 0x00, 0x00, /*Subchunk1Size (16 for PCM)*/
        0x01, 0x00, /*AudioFormat (1 for PCM)*/
        0x01, 0x00, /*NumChannels (1 channel)*/
        sample_rate, sample_rate >> 8, sample_rate >> 16, sample_rate >> 24, /*SampleRate*/
        byte_rate, byte_rate >> 8, byte_rate >> 16, byte_rate >> 24, /*ByteRate*/
        0x02, 0x00, /*BlockAlign*/
        0x10, 0x00, /*BitsPerSample (16 bits)*/
        'd', 'a', 't', 'a', /*Subchunk2ID*/
        wav_size, wav_size >> 8, wav_size >> 16, wav_size >> 24, /*Subchunk2Size*/
    };
    memcpy(wav_header, set_wav_header, sizeof(set_wav_header));
}
```

Táto funkcia konfiguruje hlavičku súboru WAV (štandardná hlavička pre súbory WAV) zadávaním vzorkovacej frekvencie a celkového množstva údajov.

/* Funkcia nahrávania a ukladania mikrofónu mic_readwav_to_sd */

```
void mic_readwav_to_sd(const char *filename, int32_t rec_seconds, int32_t *out_size)
{
    esp_err_t err = ESP_OK;
    size_t flash_w_size = 0; /*Total bytes written to SD card (audio data only)*/
    size_t bytes_read = 0; /*Bytes read from I2S in each iteration*/
    if (rec_seconds > 3600 || filename == NULL || out_size == NULL)
    {
        MIC_ERROR("Exceeding the maximum recording duration");
        return ESP_ERR_INVALID_ARG;
    }
    if (sizeof(i2s_readbuf) < 16000)
    {
        MIC_ERROR("I2S buffer too small");
        return ESP_ERR_INVALID_SIZE;
    }
    size_t rec_size = rec_seconds * BYTE_RATE; /*Calculate total audio data size (excluding wav header)*/
    char wav_header[44];
    generate_wav_header(wav_header, rec_size, MIC_SAMPLE_RATE); /*Generate wav file header (44 bytes standard wav header)*/
    err = write_file(filename, wav_header, 44); /*Write wav header to file (creates file and writes header)*/
    if (err != ESP_OK)
    {
        return err;
    }
    MIC_ERROR("Start Recording %d of audio data", rec_seconds);
    while (flash_w_size < rec_size) /*Record audio data in chunks and append to file*/
    {
        size_t bytes_to_read = MIN(16000, rec_size - flash_w_size);
        err = i2s_channel_read_ch1(i2s_readbuf, bytes_to_read, &bytes_read, portMUX_BLOCK); /*Read 16000 bytes (1600) of raw audio data from I2S microphone*/
        if (err != ESP_OK)
        {
            MIC_ERROR("read mic data fail");
            return err;
        }
        err = write_file_seek(filename, i2s_readbuf, bytes_read, flash_w_size + 44); /*Write audio data chunk to file at correct position (after 44-byte header)*/
        if (err != ESP_OK)
        {
            MIC_ERROR("write sd of audio data fail");
            return err;
        }
        flash_w_size += bytes_read; /*Update progress counter*/
    }
    *out_size = rec_size + 44; /*Set output parameter to total file size (audio data + wav header)*/
    return ESP_OK;
}
```

Táto funkcia má tri vstupné parametre:

filename – názov súboru uloženého na SD karte (vrátane cesty súborového systému)

rec_seconds – dĺžka nahrávky (v sekundách; zohľadnite dĺžku nahrávky na základe pamäte SD karty, maximálne 3600 sekúnd)

out_size – celková veľkosť údajov, ktoré sa majú prijať z nahrávky

Funkcia najskôr overí vstupné parametre a skontroluje, či je vyrovnávacia pamäť nahrávania správne nakonfigurovaná (vyžaduje sa veľkosť vyrovnávacej pamäte dostatočná na minimálne 1 sekundu nahrávania). Následne vypočíta celkový objem dát potrebných na nahrávanie na základe zadaného trvania. Vytvorí sa vyrovnávacia pamäť hlavičky súboru WAV, ktorá sa nakonfiguruje pomocou funkcie **generate_wav_header** a zapíše sa do príslušného súboru na SD karte. Nakoniec sa vytvorí slučka. Ak je splnená podmienka, že množstvo prečítaných údajov je menšie ako požadované množstvo, ktoré sa má prečítať, slučka sa vykoná. Funkcia **min** vypočíta, koľko údajových bodov sa má prečítať v aktuálnej iterácii. Funkcia **i2s_channel_read** sa volá na čítanie zvukových údajov zachytených mikrofónom, pričom sa zakaždým načítava 16 000 dátových bodov. Po úspešnom načítaní sa údaje zapisujú do súboru (preskočí sa prvých 44 dátových bodov z odpovedajúcej hlavičky WAV).

1.3.5 Súbor Kconfig

Hlavnou funkciou tohto súboru je začleniť požadovanú konfiguráciu do súboru **sdkconfig**, čo umožňuje vykonávať určité úpravy parametrov prostredníctvom grafického rozhrania. Číslo konfigurácie GPIO tu zodpovedá príslušnému číslu pinu GPIO. **MIC_SAMPLING_RATE** označuje vzorkovaciu frekvenciu.

```
menu "BSP MIC Setup"
    config BSP_MIC_ENABLED
        bool "Enable MIC"
        default n

    if BSP_MIC_ENABLED
        config MIC_GPIO_CLK
            int "GPIO For MIC CLK"
            default 3

        config MIC_GPIO_DATA
            int "GPIO For MIC DATA"
            default 4

        config MIC_SAMPLE_RATE
            int "PDM sample rate"
            default 16000
    endif
endmenu
```

1.3.6 Súbor CMkaLists.txt

Funkčnosť tejto ukážkovej rutiny závisí predovšetkým od ovládača **bsp_mic**. Aby bolo možné úspešne vyvolať obsah priečinka **bsp_mic** z iných funkcií, je potrebné nakonfigurovať súbor **CMakeLists.txt** v priečinku **bsp_mic**. Podrobnosti konfigurácie sú nasledovné:

```
FILE(GLOB_RECURSE component_sources "*.c")

idf_component_register(SRCS ${component_sources}
                       INCLUDE_DIRS "include"
                       REQUIRES driver_bsp_sd)
```

V tomto súbore **CMakeLists.txt** sú najskôr definované adresáre pre zdrojové súbory a hlavičkové súbory spolu s požadovanou knižnicou ovládačov (knižnica **bsp_sd**). Následne sa tieto nastavenia zaregistrujú v systéme zostavovania pomocou príkazu **idf_component_register**, čím sa projektu umožní **využívať** funkcie ovládača **bsp_mic**.

1.3.7 Hlavný priečinok

Hlavný priečinok slúži ako základný adresár pre vykonávanie programu a obsahuje hlavnú funkciu **main.c** a hlavičkový súbor **main.h** v priečinku **include**. Pridajte hlavný priečinok do súboru **CMakeLists.txt** systému zostavovania.

Súbor **main.h** odkazuje predovšetkým na požadované hlavičkové súbory: funkcie využívajúce ovládač **bsp_sd** vyžadujú zaradenie hlavičkového súboru **bsp_sd**, zatiaľ čo funkcie využívajúce ovládač **bsp_mic** vyžadujú hlavičkový súbor **bsp_mic**.

Nižšie je analýza programu **main.c**: inicializácia systému a inicializácia funkčnosti SD karty a mikrofónu.


```

#ifdef CONFIG_BSP_SD_ENABLED
    err = sd_init(); /*SD Initialization*/
    if (err != ESP_OK)
        init_fail("sd", err);
    vTaskDelay(500 / portTICK_PERIOD_MS);
#endif
#ifdef CONFIG_BSP_MIC_ENABLED
    err = mic_init(); /*Microphone Initialization*/
    if (err != ESP_OK)
        init_fail("mic", err);
    vTaskDelay(500 / portTICK_PERIOD_MS);
#endif

```

www.rlx.sk

Tento kód sa nachádza vo funkcii `init`, ktorá slúži na ukladanie inicializačných funkcií vyžadujúcich vyvolanie a hodnotenie ich výsledkov. Ak sa stav návratu odchyľuje od **ESP_OK**, kód vygeneruje chybovú správu a zastaví ďalšie vykonávanie. Oneskorenie 500 ms nakonfigurované pre inicializáciu mikrofónu slúži na odfiltrovanie šumu generovaného počas inicializácie I2S prijímacej kanály.

```

void app_main(void)
{
    MAIN_INFO("-----Demo version-----");
    MAIN_INFO("-----Start the test-----");
    init();
#ifdef CONFIG_BSP_MIC_ENABLED
#ifdef CONFIG_BSP_SD_ENABLED
    size_t Recorded_size = 0;
    mic_readwav_to_sd("/sdcard/test.wav", 5, &Recorded_size); /*Microphone recording saved to SD card(5s)*/
    MAIN_INFO("Recorded %d bytes", Recorded_size);
#endif
#endif
}

```

V rámci funkcie **app_main** priamo vyvolajte funkciu **mic_readwav_to_sd**, aby ste nahrali 5 sekúnd zvuku a uložili ho na SD kartu. Názov súboru je predvolene **test.wav** (môže sa podľa potreby zmeniť; ak sú potrebné dlhšie názvy súborov, aktivujte podporu dlhých názvov súborov v konfigurácii SDK).

1.3.8 Súbor **CMkaLists.txt**

Aby ste mohli úspešne vyvolať obsah priečinka **bsp_sd** v rámci hlavnej funkcie, musíte nakonfigurovať súbor **CMakeLists.txt**, ktorý sa nachádza v hlavnom priečinku. Podrobnosti konfigurácie sú nasledovné:

```

FILE(GLOB_RECURSE main ${CMAKE_SOURCE_DIR}/main/*.c)

idf_component_register(SRCS ${main}
                       INCLUDE_DIRS "include"
                       REQUIRES bsp_mic bsp_sd)

```

Najskôr sa definujú adresáre pre zdrojové súbory a hlavičkové súbory spolu s požadovanými knižnicami ovládačov – konkrétne knižnice ovládačov potrebné na prepojenie **bsp_sd** a **bsp_mic**.

Následne sa tieto nastavenia zaregistrujú v systéme zostavovania pomocou príkazu **idf_component_register**, čím sa hlavnej funkcii umožní využívať tieto funkcie ovládača.

1.4 Postup programovania

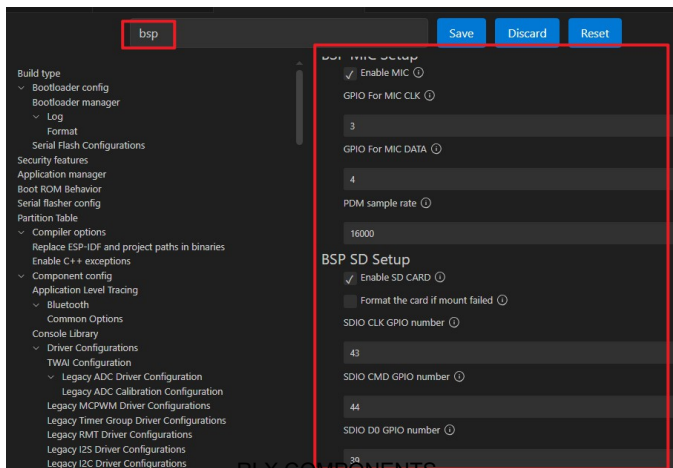
www.rlx.sk

Pripojte zariadenie P4 k počítaču prostredníctvom USB



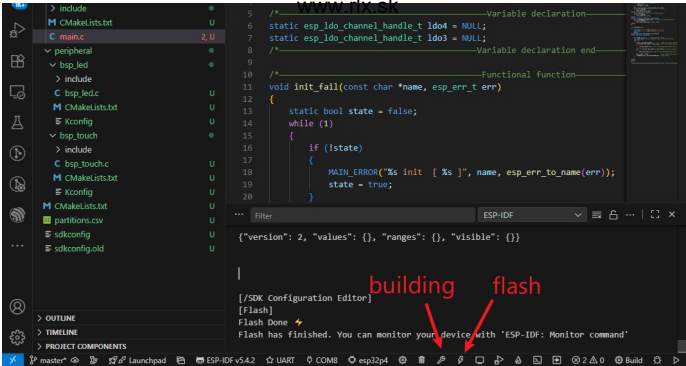
1.4.1 Po klonovaní kódu prostredníctvom Git (odkaz bude potvrdený) vymažte všetky lokálne kompilačné informácie. Nakonfigurujte prostredie IDF a model čipu pre kompiláciu podľa lekcii 1 a nastavte číslo sériového portu pre programovanie.

1.4.2 Nasledujúca konfigurácia SDKConfig je z veľkej časti identická s lekciami 1, stačí len prekonfigurovať piny LED.



RLX COMPONENTS

1.4.3 Kliknite na Kompilovať. Po úspešnej kompilácii kliknite na Stiahnuť.

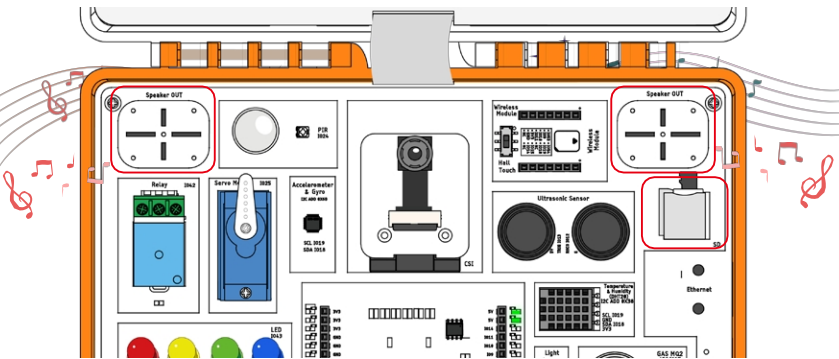


Lekcia 18 – Prehrávanie zvuku I2S

Úvod

Táto kapitola tutoriálu predstavuje aplikáciu audio rozhrania I2S pre ESP32-P4. Prehrávaním súborov WAV z SD karty pomáha čitateľom pochopiť princípy prenosu audio dátového toku a základné použitie periférie I2S. Prehrávanie zvuku je jednou z najbežnejších funkcií čipov radu ESP32 v multimediálnych a hlasových projektoch. Štúdiom tejto kapitoly sa čitatelia naučia, ako konfigurovať periférne zariadenie I2S, analyzovať súbory WAV a implementovať výstup zvuku. To vytvára základ pre zložitejšie projekty zahŕňajúce rozpoznávanie reči, syntézu zvuku alebo prehrávanie hudby.

Demonštrácia projektu



RLX COMPONENTS

- 1.1 Úvod do I2S a súborov WAV
- 1.2 Návrh hardvéru
- 1.3 Návrh softvéru
- 1.4 Stiahnutie a overenie

1.1 Úvod do súborov I2S a WAV

1.1.1 Úvod do I2S

I2S (Inter-IC Sound) je štandard sériového zbernice rozhrania pre prenos digitálnych audio dát medzi audio zariadeniami. Čip ESP32-P4 integruje radič I2S, ktorý podporuje rôzne audio formáty, vrátane režimov master/slave, stereo/mono a 16/24/32-bitového vzorkovania. Je široko používaný v scenároch, ako sú reproduktory, hlasoví asistenti, nahrávanie a Bluetooth audio.

Kľúčové vlastnosti rozhrania I2S zahŕňajú:

- ① Štandardizované rozhranie: Podporuje štandardný dátový formát I2S-STD, čím zabezpečuje kompatibilitu s bežnými periférnymi zariadeniami, ako sú audio dekodéry, DAC a zosilňovacie čipy.
- ② Vysoko verný prenos: Využívajú synchronizovaný prenos audio dát na dosiahnutie vysokokvalitného audio výstupu pri 44,1 kHz, 48 kHz a ešte vyšších vzorkovacích frekvenciách.
- ③ Hardvérové vyrovnávanie FIFO: Obsahuje internú vyrovnávaciu pamäť FIFO na zníženie zaťaženia procesora, podporuje nepretržitý prenos DMA a zabezpečuje plynulé prehrávanie zvuku bez prerušovania.
- ④ Podpora viacerých režimov: Podporuje režimy Master aj Slave, čo umožňuje flexibilné prispôsobenie rôznym architektúram zvukových obvodov.
- ⑤ Programovateľná konfigurácia: Softvérovo konfigurovateľné parametre vrátane šírky dátového bitu, časovania ľavého/pravého kanála, vzorkovacej frekvencie a polarita hodín, aby zodpovedali rôznym audio čipom.

Funkcia I2S ESP32-P4 poskytuje robustný hardvérový základ pre audio aplikácie. V tejto kapitole sa budeme venovať implementácii a konfigurácii I2S prostredníctvom praktického prehrávania súborov WAV z SD karty.

1.1.2 Sprievodca prehrávaním súborov WAV

Súbory WAV sú bežný nekomprimovaný formát, ktorý ukladá surové audio dáta pomocou

PCM (pulzná kódová modulácia). Vďaka svojej jednoduchšej štruktúre a bezstratovej kvalite zvuku sa formát WAV často používa na experimentálne prehrávaním zvuku vo vstavaných systémoch.

① Vzorkovacia frekvencia a kvalita zvuku

Vzorkovacia frekvencia určuje čistotu zvuku. Bežné vzorkovacie frekvencie sú 8 kHz (hlas), 16 kHz (telefónia) a 44,1 kHz (hudba). Vyššie vzorkovacie frekvencie poskytujú lepšiu kvalitu zvuku, ale aj väčšie veľkosti dát.

② Princíp prehrávania

Proces prehrávania I2S pre súbory WAV zahŕňa tieto kroky: Načítanie

hlavičky súboru WAV z SD karty a analýza parametrov;

Inicializácia rozhrania I2S, nastavenie zodpovedajúcej vzorkovacej frekvencie a

šírky bitu; Čítanie blokov zvukových dát v slučke a ich prenos do I2S FIFO

prostredníctvom DMA;

Periférne zariadenie I2S automaticky vysiela dáta ľavého a pravého kanála, čím ovláda externý DAC alebo zosilňovač na produkciu zvuku.

1.2 Hardvérový dizajn

Obvod audio výstupu vývojovej dosky ESP32-P4 sa skladá z nasledujúcich komponentov:

Dátové linky I2S:

BCLK (bitový takt)

WS (výber ľavého/pravého kanála)

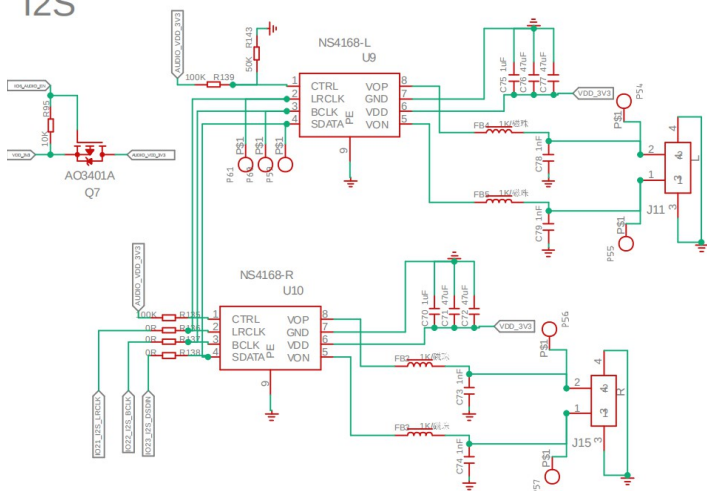
DATA (audio dátová linka)

Externý modul DAC / audio zosilňovač: napr. MAX98357A alebo ES8388, používaný na prevod digitálnych signálov na analógový audio výstup;

Modul SD karty: Komunikuje s ESP32-P4 prostredníctvom rozhrania SPI na ukladanie audio súborov WAV;

Rozhranie reproduktora alebo slúchadiel: Pripojí sa k terminálu audio výstupu.

I2S



Schematický diagram

1.3 Analýza programu

<https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design>

Otvorte súbor projektu v VS Code podľa predchádzajúcich pokynov.

ESP32P4-DEV-KITS_AUDIO

- main
 - include
 - main.h
- CMakeLists.txt
- main.c
- peripheral
 - bsp_audio
 - include
 - bsp_audio.h
 - bsp_audio.c
 - CMakeLists.txt
 - Kconfig
 - bsp_sd
 - include
 - bsp_sd.c
 - CMakeLists.txt

V príklade **ESP32P4-dev-kits_audio** bol vytvorený nový priečinok **bsp_audio** pod adresárom

ESP32P4-dev-kits_audio\peripheral. V priečinku **bsp_audio** bol vytvorený priečinok **include**, súbor **CMakeLists.txt** a súbor **Kconfig**. Zložka **bsp_audio** ukladá súbor ovládača **bsp_audio.c**, zložka **include** obsahuje súbory hlavičiek **.h** a súbor **CMakeLists.txt** integruje ovládač do systému zostavovania, čím umožňuje projektu prístup k jeho funkcionalite. Súbor **Kconfig** načítava celý ovládač spolu s definíciami pinov GPIO do súboru **sdkconfig** v rámci platformy IDF (konfigurovateľný prostredníctvom grafického rozhrania).

RLX COMPONENTS

1.3.1 Kód ovládača reproduktora

Tu vysvetlíme len základný kód. Podrobný zdrojový kód nájdete v príslušnom zdrojovom kóde pre tento experiment v materiáloch kódu.

Zdrojový kód ovládača reproduktora pozostáva z dvoch súborov: **bsp_audio.c** a **bsp_audio.h**.

Nižšie najprv analyzujeme program **bsp_audio.h**: definuje výstupné piny reproduktora a deklaruje použité funkcie.

/* Odkazy na hlavičkové súbory */

```
/*-----Header file declaration-----*/
#include "esp_log.h" //References for LOG Printing Function-related API Functions
#include "esp_err.h" //References for Error Type Function-related API Functions
#include "driver/gpio.h" //References for GPIO Function-related API Functions
#include "driver/i2s_std.h" //References for I2S STD Function-related API Functions
#ifdef CONFIG_BSP_SD_ENABLED
#include "bsp_sd.h"
#endif
/*-----Header file declaration end-----*/
```

/* Deklarácie funkcií a definície makier */

```
#ifdef CONFIG_AUDIO_ENABLED

#define AUDIO_GPIO_LRCLK CONFIG_AUDIO_GPIO_LRCLK // Audio LRCLK GPIO
#define AUDIO_GPIO_BCLK CONFIG_AUDIO_GPIO_BCLK // Audio BCLK GPIO
#define AUDIO_GPIO_SDATA CONFIG_AUDIO_GPIO_SDATA // Audio DATA GPIO
#define AUDIO_GPIO_CTRL CONFIG_AUDIO_GPIO_CTRL // Audio CTRL GPIO

esp_err_t audio_init(); // Audio Initialization Function
esp_err_t audio_ctrl_init(); // Audio ctrl Initialization Function
esp_err_t set_audio_ctrl(const state_t); // Control mute
#ifdef CONFIG_BSP_SD_ENABLED
esp_err_t audio_play_sd(const char *fp); // The speaker plays the wav file stored on the SD card
#endif
#endif
```

Ďalej budeme analyzovať program **bsp_audio.c**:

Inicializácia a konfigurácia reproduktora pre štandardný režim I2S.

Inicializácia a konfigurácia pinov výberu kanálov čipu NS4168.

Zverejnenie funkcií rozhrania API.

/* Funkcia inicializácie štandardného režimu I2S audio_init */

```
esp_err_t audio_init()
{
    esp_err_t err = ESP_OK;
    i2s_chan_config_t chan_cfg = {
        .id = I2S_NUM_1, //Use I2S controller 1*/
        .role = I2S_ROLE_MASTER, //I2S acts as master (generates clock signals)*/
        .dma_desc_num = 0, //Number of DMA descriptors for buffer management*/
        .dma_frame_num = 256, //Number of frames per DMA descriptor*/
        .auto_clear = true, //Automatically clear DMA buffer on underflow*/
        .intr_priority = 0, //Interrupt priority level*/
    }; //I2S channel configuration for transmitter (audio output)*/
    err = i2s_new_channel(&chan_cfg, &tx_chan, NULL); //Create new I2S channel (transmit channel only, no receive channel)*/
    if (err != ESP_OK)
        return err;
    i2s_std_config_t std_cfg = {
        .clk_cfg = {
            .sample_rate_hz = 16000, //Audio sample rate: 16000*/
            .clk_src = I2S_CLK_SRC_DEFAULT, //Default clock source*/
            .mclk_multiple = I2S_MCLK_MULTIPLE_256, //Master clock multiplier*/
        }, //Clock configuration*/
        .slot_cfg = {
            .data_bit_width = I2S_DATA_BIT_WIDTH_16BIT, //16-bit audio samples*/
            .slot_bit_width = I2S_SLOT_BIT_WIDTH_AUTO, //Auto-calculate slot width*/
            .slot_mode = I2S_SLOT_MODE_STEREO, //Stereo audio (2 channels)*/
            .slot_mask = I2S_STD_SLOT_BOTH, //Enable both left and right channels*/
            .ws_width = I2S_DATA_BIT_WIDTH_16BIT, //Word select signal width*/
            .ws_pol = false, //Word select polarity (normal)*/
            .bit_shift = true, //Enable bit shift in data frame*/
            .left_align = true, //Left-aligned data to slot*/
            .big_endian = false, //Little-endian byte order*/
            .bit_order_lsb = false, //MSB first bit order*/
        }, //Audio slot/data format configuration*/
        .gpio_cfg = {
            .mclk = I2S_GPIO_UNUSED, //Master clock not used*/
            .bclk = AUDIO_GPIO_BCLK, //Bit clock pin*/
            .ws = AUDIO_GPIO_LRCLK, //Word select (left/right clock) pin*/
            .sout = AUDIO_GPIO_SDATA, //Serial data output pin*/
            .din = I2S_GPIO_UNUSED, //Data input not used (output only)*/
            .invert_flags = {
                .mclk_inv = false, //Don't invert master clock*/
                .bclk_inv = false, //Don't invert bit clock*/
                .ws_inv = false, //Don't invert word select*/
            }, //Signal inversion flags*/
        }, //GPIO pin configuration for I2S signal*/
    }; //Standard I2S configuration for audio playback*/
}
```

```

err = i2s_channel_init_std_mode(tx_chan, &std_cfg); /*Initialize I2S channel in standard mode for audio output*/
if (err != ESP_OK)
    return err;

err = i2s_channel_enable(tx_chan); /*Enable the I2S channel to start audio transmission*/
if (err != ESP_OK)
    return err;

return err;

```

www.rlx.sk

Vo funkcii **audio_init** najskôr nakonfigurujeme štruktúru **i2s_chan_config_t**, aby sme nastavili parametre pre radič I2S používaný reproduktorom (tu používame radič I2S 1). Potom pomocou funkcie **i2s_new_channel** zaregistrujeme prenosový kanál na radiči I2S 1. Štruktúra **i2s_std_config_t** konfiguruje prenos v štandardnom režime. Šírku dátového bitu môžeme zmeniť úpravou parametrov **data_bit_width** a **slot_bit_width** v poli **slot_cfg**. Ak chcete povoliť stereo alebo mono výstup, upravte nastavenia **slot_mode** a **slot_mask** v poli **slot_cfg**. Nakoniec zavolajte **i2s_channel_init_std_mode**, aby ste inicializovali prenosový kanál pomocou štandardného režimu, a potom zavolajte **i2s_channel_enable**, aby ste aktivovali prenosový kanál.

Poznámka: Naša vývojová doska má dva reproduktory, takže ju konfiguruje pre stereofónny režim s dvojkanálovým výstupom.

/* audio_ctrl_init: Inicializačná funkcia pre pin CTRL čipu ns4168 */

```

esp_err_t audio_ctrl_init()
{
    esp_err_t err = ESP_OK;
    const gpio_config_t gpio_cfg = {
        .pin_bit_mask = 1ULL << AUDIO_GPIO_CTRL, /* GPIO pin: set with bit mask, each bit maps to a GPIO */
        .mode = GPIO_MODE_OUTPUT, /* GPIO mode: set input/output mode */
        .pull_up_en = false, /* GPIO pull-up */
        .pull_down_en = false, /* GPIO pull-down */
        .intr_type = GPIO_INTR_DISABLE, /* GPIO interrupt type */
    };
    err = gpio_config(&gpio_cfg); /*Configure GPIO*/
    if (err != ESP_OK)
        return err;

    return err;
}

```

Táto funkcia volá inicializačnú štruktúru GPIO, aby nakonfigurovala pin CTRL ako výstupný režim.

/* Funkcia ovládania stlmenia reproduktora **set_Audio_ctrl** */

```

esp_err_t set_Audio_ctrl(bool state)
{
    esp_err_t err = ESP_OK;
    bool status = !state; /*Invert the state (the sound is on when the voltage is low, and off when the voltage is high)*/
    err = gpio_set_level(AUDIO_GPIO_CTRL, status); /*Set the Corresponding Output Level of GPIO*/
    return err;
}

```

Táto funkcia volá **gpio_set_level** na výstup ovládania vysokej/nízkej úrovne. Premenná status určuje výstupnú úroveň. Inverzná operácia je spôsobená MOSFET použitým v návrhu obvodu. Nízka úroveň aktivuje MOSFET, zatiaľ čo vysoká úroveň ovláda čip NS4168 na aktiváciu zvuku (dvojkanálový). Keď je MOSFET vypnutý (vysoká úroveň), zostáva v nízkom stave a stlmí zvuk.

/*funkcia overenia hlavičky súboru wav **validate_wav_header***/


```

bool validate_wav_header(FILE *file)
{
    if (file == NULL)
    {
        AUDIO_ERROR("File pointer is NULL");
        return false;
    }
    long original_position = ftell(file); /*Store current file position to restore later*/
    if (original_position == -1)
    {
        AUDIO_ERROR("Cannot get current file position");
        return false;
    }
    if (fseek(file, 0, SEEK_SET) != 0) /*Rewind to beginning of file*/
    {
        AUDIO_ERROR("Cannot seek to file beginning");
        return false;
    }
    uint8_t header[44]; /*Read and validate WAV header*/
    size_t bytes_read = fread(header, 1, 44, file);
    if (bytes_read != 44)
    {
        AUDIO_ERROR("Cannot read complete WAV header (%d bytes)", bytes_read);
        fseek(file, original_position, SEEK_SET);
        return false;
    }
    if (memcmp(header, "RIFF", 4) != 0) /*Validate RIFF chunk descriptor*/
    {
        AUDIO_ERROR("Invalid RIFF header");
        fseek(file, original_position, SEEK_SET);
        return false;
    }
    if (memcmp(header + 8, "WAVE", 4) != 0) /*Validate WAVE format*/
    {
        AUDIO_ERROR("Invalid WAVE format");
        fseek(file, original_position, SEEK_SET);
        return false;
    }
    if (memcmp(header + 12, "fmt ", 4) != 0) /*Validate fmt subchunk*/
    {
        AUDIO_ERROR("Invalid fmt subchunk");
        fseek(file, original_position, SEEK_SET);
        return false;
    }

    uint16_t audio_format = *(uint16_t *) (header + 20); /*Check audio format (should be 1 for PCM)*/
    if (audio_format != 1)
    {
        AUDIO_ERROR("Unsupported audio format: %d (only PCM supported)", audio_format);
        fseek(file, original_position, SEEK_SET);
        return false;
    }
    uint16_t num_channels = *(uint16_t *) (header + 22); /*Check number of channels (support mono and stereo)*/
    if (num_channels != 1 && num_channels != 2)
    {
        AUDIO_ERROR("Unsupported number of channels: %d", num_channels);
        fseek(file, original_position, SEEK_SET);
        return false;
    }
    uint32_t sample_rate = *(uint32_t *) (header + 24); /*Check sample rate (support common rates)*/
    if (sample_rate != 8000 && sample_rate != 16000 && sample_rate != 22050 && sample_rate != 44100 && sample_rate != 48000)
    {
        AUDIO_ERROR("Uncommon sample rate: %u Hz", sample_rate);
        fseek(file, original_position, SEEK_SET);
        return false;
    }
    uint16_t bits_per_sample = *(uint16_t *) (header + 34); /*Check bits per sample (support 8, 16, 24, 32)*/
    if (bits_per_sample != 8 && bits_per_sample != 16 && bits_per_sample != 24 && bits_per_sample != 32)
    {
        AUDIO_ERROR("Unsupported bits per sample: %d", bits_per_sample);
        fseek(file, original_position, SEEK_SET);
        return false;
    }
    if (memcmp(header + 36, "data", 4) != 0) /*Validate data subchunk*/
    {
        AUDIO_ERROR("Invalid data subchunk");
        fseek(file, original_position, SEEK_SET);
        return false;
    }
    /*Get file size from RIFF header for additional validation*/
    uint32_t file_size = *(uint32_t *) (header + 4) + 8; // RIFF块大小 + 8字节头部
    uint32_t data_size = *(uint32_t *) (header + 40);
    AUDIO_INFO("WAV File Info: %d channels, %u Hz, %d bits, %u bytes data, %u bytes total", num_channels, sample_rate, bits_per_sample, data_size, file_size);
    /*Restore original file position*/
    fseek(file, original_position, SEEK_SET);
    return true;
}

```

Táto funkcia prečíta prvých 44 bajtov súboru WAV, skontroluje štandardnú hlavičku WAV a vytlačí informácie o súbore.

/* Funkcia **Audio_play_wav_sd** na prehrávanie súborov WAV uložených na SD karte cez reproduktor */

RLX COMPONENTS

```

esp_err_t Audio_play_wav_sd(const char *filename)
{
    esp_err_t err = ESP_OK;
    if (filename == NULL)
    {
        return ESP_ERR_INVALID_ARG;
    }
    FILE *fh = fopen(filename, "rb");
    if (fh == NULL)
    {
        AUDIO_ERROR("Failed to open file");
        return ESP_ERR_INVALID_ARG;
    }
    if (!validate_wav_header(fh)) /*Validate WAV header*/
    {
        AUDIO_ERROR("Invalid WAV file format: %s", filename);
        fclose(fh);
        return ESP_ERR_INVALID_ARG;
    }
    if (fseek(fh, 44, SEEK_SET) != 0) /*Skip 44-byte WAV header*/
    {
        AUDIO_ERROR("Failed to seek file");
        fclose(fh);
        return ESP_FAIL;
    }
    /*Buffer configuration*/
    const size_t SAMPLES_PER_BUFFER = 512;
    const size_t INPUT_BUFFER_SIZE = SAMPLES_PER_BUFFER * sizeof(int16_t);
    const size_t OUTPUT_BUFFER_SIZE = SAMPLES_PER_BUFFER * 2 * sizeof(int16_t);
    /* Allocate buffers */
    int16_t *input_buf = heap_caps_malloc(INPUT_BUFFER_SIZE, MALLOC_CAP_SPIRAM);
    int16_t *output_buf = heap_caps_malloc(OUTPUT_BUFFER_SIZE, MALLOC_CAP_SPIRAM);
    if (input_buf == NULL || output_buf == NULL)
    {
        AUDIO_ERROR("Failed to allocate audio buffers");
        if (input_buf)
            free(input_buf);
        if (output_buf)
            free(output_buf);
        fclose(fh);
        return ESP_ERR_NO_MEM;
    }
}

```

www.rlx.sk

```

size_t samples_read = 0;
size_t bytes_to_write = 0;
size_t bytes_written = 0;
size_t total_samples = 0;
int32_t volume_data = 0;
set_audio_ctrl(true); /*Enable audio hardware*/
while (1)
{
    samples_read = fread(input_buf, sizeof(int16_t), SAMPLES_PER_BUFFER, fh);
    if (samples_read == 0)
    {
        break;
    }
    for (size_t i = 0; i < samples_read; i++) /*convert mono to stereo*/
    {
        volume_data = input_buf[i] * 10; /*Linear multiplication*/
        if (volume_data > 32767)
            volume_data = 32767;
        else if (volume_data < -32768)
            volume_data = -32768;
        output_buf[i * 2] = (int16_t)volume_data; /*Left channel*/
        output_buf[i * 2 + 1] = (int16_t)volume_data; /*Right channel*/
    }
    bytes_to_write = samples_read * 2 * sizeof(int16_t);
    bytes_written = 0;
    err = i2s_channel_write(tx_chan, output_buf, bytes_to_write, portMAX_DELAY); /*I2S write data*/
    if (err != ESP_OK || bytes_written != bytes_to_write)
    {
        AUDIO_ERROR("I2S write failed: %s, written: %d/%d", esp_err_to_name(err), bytes_written, bytes_to_write);
        break;
    }
    total_samples += samples_read;
}
/*Cleanup*/
set_audio_ctrl(false);
free(input_buf);
free(output_buf);
fclose(fh);
AUDIO_INFO("Audio playback completed: %d samples", total_samples);
return err;
}

```

1. Táto funkcia najskôr volá **validate_wav_header** na overenie platnosti hlavičky súboru wav. Potom pomocou **fseek** posunie počiatočnú pozíciu súboru (preskočí hlavičku wav a prvých 44 bajtov) a vytvorí dva vyrovnávacie pamäte: jednu na ukladanie údajov načítaných z SD karty a druhú na prenos údajov v štandardnom režime I2S. Funkcia **set_Audio_ctrl** sa volá na otvorenie zvukového kanála (dvojkanálového). Funkcia **fread** číta údaje zo súboru na SD karte do vyrovnávacej pamäte, ktorá sa potom priradí k prenosovej vyrovnávacej pamäti. Keďže nahrávky z mikrofónu sú mono údaje a štandardný režim I2S prenáša zvukové údaje vo formáte ľavý kanál, pravý kanál, ľavý kanál, prečítajú sa údaje (identické údaje) pre oba kanály.

RLX COMPONENTS

2. Operácia *10 aplikovaná na načítané údaje je spôsobená nízkym akustickým tlakom zaznamenaným mikrofónom. Tu sa na zosilnenie audio údajov používa lineárna metóda zosilnenia (táto metóda zosilňuje aj šum v pozadí; nepoužíva sa pre audio zaznamenané mikrofónom). Keďže nastavená šírka audio údajov je 16 bitov, pri lineárnom zosilnení je potrebné spracovanie orezania na audio vlne, aby sa zabezpečilo, že zosilnené údaje neprekročia šírku bitu.

3. Odoslite zosilnené dáta prostredníctvom funkcie `i2s_channel_write`. Vypočítajte zostávajúce dáta, ktoré sa majú prečítať (aby ste určili, kedy ukončiť; každá operácia čítania spracováva 512 dátových bodov) a celkové odoslané dáta (aby ste overili, či boli počas ladenia odoslané všetky dáta; ak sú neúplné, zaznamenajte chybovú správu a ukončíte slučku).

4. Po dokončení prenosu (alebo zlyhaní) nakonfigurujte pin ovládania zvuku do režimu stlmenia a uvoľnite vyrovnávaciu pamäť a operácie so súbormi (aby sa zabránilo pretečeniu zásobníka).

1.3.2 Súbor Kconfig

Hlavnou funkciou tohto súboru je pridať požadované konfigurácie do súboru `sdkconfig`, čo umožňuje vykonávať určité úpravy parametrov prostredníctvom grafického rozhrania. Čísla tu zodpovedajú príslušným číslam pinov GPIO.

```
menu "BSP AUDIO Setup"
    config BSP_AUDIO_ENABLED
        bool "Enable AUDIO"
        default n

    if BSP_AUDIO_ENABLED
        config AUDIO_GPIO_LRCLK
            int "GPIO For AUDIO LRCLK"
            default 21

        config AUDIO_GPIO_BCLK
            int "GPIO For AUDIO BCLK"
            default 22

        config AUDIO_GPIO_SDATA
            int "GPIO For AUDIO SDATA"
            default 23

        config AUDIO_GPIO_CTRL
            int "GPIO For AUDIO CTRL"
            default 6
    endif
endmenu
```

1.3.3 Súbor CMkaLists.txt

Funkčnosť tohto príkladu závisí predovšetkým od ovládača `bsp_audio`. Aby ste mohli úspešne volať funkcie zo zložky `bsp_audio` v rámci iných funkcií, musíte nakonfigurovať súbor `CMakeLists.txt`, ktorý sa nachádza v zložke `bsp_audio`. Konfigurácia je nasledovná:

```
FILE(GLOB_RECURSE component_sources "*.c")

idf_component_register(SRCS ${component_sources}
    INCLUDE_DIRS "include"
    REQUIRES driver bsp_sd)
```

V tomto súbore `CMakeLists.txt` sú najskôr definované adresáre pre zdrojové súbory a hlavičkové súbory spolu s požadovanou knižnicou ovládačov (knižnica `bsp_sd`). Potom sa tieto nastavenia zaregistrujú do systému zostavovania pomocou príkazu `idf_component_register`, čím sa projektu umožní využívať funkcie ovládača `bsp_audio`.

1.3.4 Hlavný priečinok

Hlavný priečinok slúži ako hlavný adresár pre vykonávanie programu. Obsahuje hlavnú funkciu `main.c` a hlavičkový súbor `main.h`, ktorý sa nachádza v priečinku `include`. Pridajte hlavný priečinok do súboru **CMakeLists.txt** systému zostavenia.

Súbor `main.h` odkazuje predovšetkým na požadované súbory hlavičiek: funkcie využívajúce ovládač **bsp_sd** vyžadujú súbor hlavičky **bsp_sd**, zatiaľ čo funkcie využívajúce ovládač **bsp_audio** vyžadujú súbor hlavičky **bsp_audio**.

Nižšie je analýza programu `main.c`: Inicializácia systému a inicializácia funkčnosti SD karty a funkčnosti reproduktora.

```
init_fail("I2S", err);
#ifdef CONFIG_BSP_SD_ENABLED
    err = sd_init(); /*SD Initialization*/
    if (err != ESP_OK)
        init_fail("sd", err);
    vTaskDelay(500 / portTICK_PERIOD_MS);
#endif
#ifdef CONFIG_BSP_AUDIO_ENABLED
    err = audio_ctrl_init(); /*Audio CTRL Initialization*/
    if (err != ESP_OK)
        init_fail("audio ctrl", err);
    set_audio_ctrl(false);
    err = audio_init(); /*Audio Initialization*/
    if (err != ESP_OK)
        init_fail("audio", err);
    vTaskDelay(500 / portTICK_PERIOD_MS);
#endif
}
```

Tento kód sa nachádza vo funkcii `init`, ktorá ukladá inicializačné funkcie, ktoré sa majú volať, a vyhodnocuje ich stav návratu. Ak stav návratu nie je `ESP_OK`, kód vytlačí chybovú správu a zastaví vykonávanie. Tu konfigurácia najskôr nastaví piny na ovládanie zvuku a vykoná operáciu stlmenia. Inicializačné oneskorenie 500 ms je implementované na odfiltrovanie šumu generovaného počas inicializácie I2S pri odosielaní kanálov.

```
void app_main(void)
{
    MAIN_INFO("-----Demo version-----");
    MAIN_INFO("-----Start the test-----");
    init();
#ifdef CONFIG_BSP_AUDIO_ENABLED
#ifdef CONFIG_BSP_SD_ENABLED
    Audio_play_wav_sd("/sdcard/test.wav"); /*Play the WAV file stored on the SD card that was recorded by the microphone*/
#endif
#endif
}
```

Vo funkcii `app_main` priamo vyvolajte funkciu `Audio_play_wav_sd` na prehrávanie súboru WAV uloženého na SD karte. Názov súboru je `test.wav` (možno ho podľa potreby zmeniť; ak sú potrebné dlhé názvy súborov, aktivujte podporu dlhých názvov súborov v `sdkconfig`).

1.3.5 Súbor CMakeLists.txt

Aby ste mohli úspešne vyvolať obsah priečinkov `bsp_sd` a `bsp_audio` v hlavnej funkcii, musíte nakonfigurovať súbor `CMakeLists.txt`, ktorý sa nachádza v hlavnom priečinku. Podrobnosti konfigurácie sú nasledovné:

RLX COMPONENTS

```
FILE(GLOB_RECURSE main ${CMAKE_SOURCE_DIR}/main/*.c)

idf_component_register(SRCS ${main}
                       INCLUDE_DIRS "include"
                       REQUIRES bsp_audio bsp_sd)
```

www.rlx.sk

Najskôr sa definujú adresáre pre zdrojové súbory a hlavičkové súbory spolu s požadovanými knižnicami ovládačov – konkrétne knižnice ovládačov potrebné na prepojenie **bsp_sd** a **bsp_audio**. Potom sa tieto nastavenia zaregistrujú v systéme zostavovania pomocou príkazu **idf_component_register**, čím sa hlavnej funkcii umožní využívať tieto funkcie ovládačov.

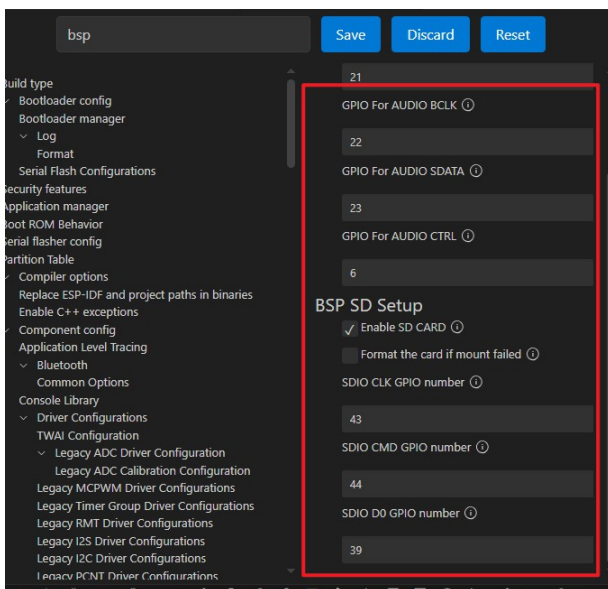
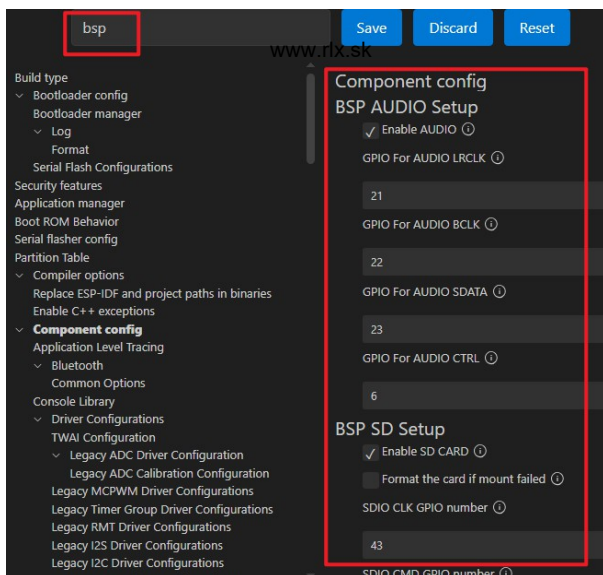
1.4 Postup programovania

Pripojte zariadenie P4 k počítaču prostredníctvom USB



1.4.1 Po klonovaní kódu prostredníctvom Git (odkaz bude potvrdený) vymažte všetky lokálne informácie o kompilácii. Nakonfigurujte prostredie IDF a model čipu pre kompiláciu podľa lekcie 1 a nastavte číslo sériového portu pre programovanie.

1.4.2 Nasledujúca konfigurácia SDKConfig je z veľkej časti identická s lekciami 1, stačí len prekonfigurovať piny LED.



1.4.3 Kliknite na Compile (Kompilovať). Po úspešnej kompilácii kliknite na Download (Stiahnuť).

RLX COMPONENTS

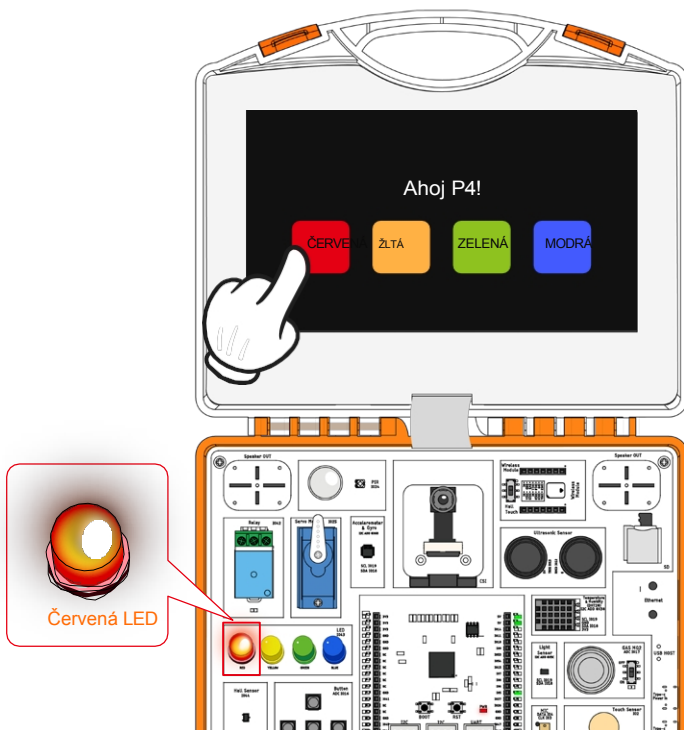
Lekcia 19 – Ovládanie LED dotykom LVGL www.rlx.sk

Úvod

Na základe predchádzajúcich kapitol „Ovládanie výstupu GPIO LED svetiel“ a „LVGL displej“ táto kapitola ďalej rozširuje scenáre ovládania aplikácií pre displej P4. V tejto kapitole sa čitatelia naučia, ako implementovať ovládanie interakcie človek-stroj na vývojovej doske ESP32-P4 pomocou knižnice grafického rozhrania LVGL:

Ovládanie zapnutia/vypnutia LED svetiel ťuknutím na tlačidlá na dotykovej obrazovke; Ovládanie základných ovládacích prvkov a mechanizmov reakcie na udalosti LVGL; Porozumenie kombinovanej aplikácii GPIO a GUI.

Demonštrácia projektu



RLX COMPONENTS

Tento projekt integruje vývoj grafického rozhrania s riadením základného hardvéru, čo predstavuje významný pokrok od „zapnutia www.itefix.sk dotykovému inteligentnému ovládaniu“. Vytvára základ pre ďalšie projekty v oblasti inteligentných domovov a interakcie človeka s počítačom.

1.1 Ciele projektu

Porozumieť základným komponentom a prevádzkovým mechanizmom grafického rozhrania LVGL;

Ovládanie používania tlačidiel (lv_btn) a štítkov (lv_label); Naučiť sa ovládať výstupy GPIO prostredníctvom dotykových udalostí na implementáciu prepínania LED;

Ovládanie logiky návrhu funkcií spätného volania udalostí a metód pre synchronizované zobrazenie stavu.

1.2 Analýza programu

<https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design>

Otvorte súbor projektu v VS Code podľa predchádzajúcich pokynov.



Do príkladu **ESP32P4-dev-kits_lvgl_touch** bola pridaná zložka **bsp_display**. Boli vykonané úpravy súboru **bsp_display.c** nachádzajúceho sa v adresári **bsp_display** spolu s aktualizáciami súboru **CMakeLists.txt** a súborov **Kconfig**. Bol zavedený nový kód súvisiaci s funkciou dotykového displeja.

RLX COMPONENTS

1.2.1 Kód ovládača dotykovej obrazovky

Tu vysvetlíme len základný kód. Podrobný zdrojový kód nájdete v príslušnom zdrojovom kóde pre tento experiment v materiáloch s kódom.

Úpravy zdrojového kódu ovládača dotykovej obrazovky sa týkajú dvoch súborov: bsp_display.c a bsp_display.h.

Nižšie najprv analyzujeme program bsp_display.h: Pridať relevantné definície a deklarácie funkcií pre dotkové piny obrazovky.

/* Odkazy na súbory hlavičiek */

```
/*-----Header file declaration-----*/
#include "esp_log.h" //References for LOG Printing Function-related API Functions
#include "esp_err.h" //References for Error Type Function-related API Functions
#include "esp_idc_regulator.h" //References for IDC Function-related API Functions
#include "esp_lcd_exp7000.h" //References for lcd exp7000 Function-related API Functions
#include "esp_lcd_panel_io.h" //References for lcd panel io Function-related API Functions
#include "esp_lcd_panel_ops.h" //References for lcd panel ops Function-related API Functions
#include "esp_lcd_panel_io.h" //References for lcd panel io Function-related API Functions
#include "esp_lvgl_port.h" //References for LVGL port Function-related API Functions
#include "esp_lcd_touch_gt911.h" //References for Touch GT911 Function-related API Functions
#include "driver/gpio.h" //References for GPIO Function-related API Functions
#include "driver/i2c.h" //References for I2C Pin Function-related API Functions
#include "lvgl.h" //References for LVGL Function-related API Functions
#ifdef CONFIG_BSP_I2C_ENABLED
#include "bsp_i2c.h"
#endif
/*-----Header file declaration end-----*/
```

/* Deklarácie funkcií a definície makier */

```
#ifdef CONFIG_BSP_TOUCH_ENABLED

#define Touch_GPIO_RST CONFIG_TOUCH_GPIO_RST // TOUCH Reset GPIO
#define Touch_GPIO_INT CONFIG_TOUCH_GPIO_INT // TOUCH INT GPIO
esp_err_t touch_init(void); // Touch Initialization Function

#endif
```

Do existujúceho rámca bol pridaný ďalší obsah týkajúci sa funkcie dotykového displeja.

Ďalej budeme analyzovať program bsp_display.c: Boli implementované nové volania funkcií konfigurácie a nastavenia inicializácie pre dotkové piny displeja.

/* Funkcia inicializácie dotykového displeja touch_init */

```
esp_err_t touch_init(void)
{
    esp_err_t err = ESP_OK;
    esp_lcd_panel_io_i2c_config_t io_config = {
        .dev_addr = ESP_LCD_TOUCH_IO_I2C_GT911_ADDRESS_BACKUP, //Primary I2C device address attempt*/
        .control_phase_bytes = 1, //1 byte for control phase*/
        .dc_bit_offset = 0, //No DC bit offset (not used for touch)*/
        .lcd_cmd_bits = 16, //16-bit command width*/
        .flags = {
            .disable_control_phase = 1, //Disable control phase for touch controllers*/
        },
        .scl_speed_hz = 400000, //I2C clock speed: 400kHz (Fast Mode)*/
    }; //I2C configuration for touch panel controller (GT911)*/
    esp_lcd_touch_config_t tp_cfg = {
        .x_max = 0, //Maximum X coordinate (horizontal resolution)*/
        .y_max = 0, //Maximum Y coordinate (vertical resolution)*/
        .rst_gpio_num = Touch_GPIO_RST, //Reset GPIO pin number*/
        .int_gpio_num = Touch_GPIO_INT, //Interrupt GPIO pin number*/
        .levels = {
            .reset = 0, //Active low reset level*/
            .interrupt = 0, //Active low interrupt level*/
        },
        .flags = {
            .swap_xy = false, //Don't swap X and Y coordinates*/
            .mirror_x = false, //Don't mirror X axis*/
            .mirror_y = false, //Don't mirror Y axis*/
        },
    }; //Touch panel configuration parameters*/
    //Initialize I2C communication with touch controller*/
    err = esp_lcd_new_panel_io_i2c((i2c_master_bus_handle_t)i2c_bus_handle, &io_config, &tp_cfg);
    if (err != ESP_OK)
        return err;
    err = esp_lcd_touch_new_i2c_gt911(tp_cfg, &tp_cfg, &tp); //Initialize GT911 touch controller with primary address*/
    if (err != ESP_OK)
        //If primary address fails, try backup I2C address*/
    {
        io_config.dev_addr = ESP_LCD_TOUCH_IO_I2C_GT911_ADDRESS; //Backup I2C device address*/
        err = esp_lcd_new_panel_io_i2c((i2c_master_bus_handle_t)i2c_bus_handle, &io_config, &tp_cfg);
        if (err != ESP_OK)
            return err;
        err = esp_lcd_touch_new_i2c_gt911(tp_cfg, &tp_cfg, &tp); //Initialize GT911 touch controller with backup address*/
        if (err != ESP_OK)
            return err;
    }
    return err;
}
```

V rámci funkcie `touch\_init` sa najskôr vykoná konfigurácia parametrov pre každú členskú premennú štruktúry `esp\_lcd\_panel\_io\_i2c` v súbore `lvgl_sdk`. To zahŕňa konfiguráciu parametrov, ako je 7-bitová adresa I2C a šírka bitov príkazu pre dotykový ovládací čip GT911. Následne sa na konfiguráciu vyvolá štruktúra `esp\_lcd\_touch\_config\_t`. To zahŕňa definovanie parametrov súvisiacich s rozlíšením obrazovky, resetovacími pinmi, prerušovacími pinmi a ich aktivačnými signálmi. Následne sa vyvolá funkcia `esp_lcd_new_panel_io_i2c`, aby sa priradila rukoväť získaná skôr z ovládača I2C k dotykovému obrazovke. Nakoniec funkcia `esp_lcd_touch_new_i2c_gt911` inicializuje konfiguráciu dotykového čipu GT911 a vráti ovládacie rukoväť.

Poznámka: Ak zlyhá aktuálna konfigurácia adresy GT911, prejdeme na alternatívnu adresu pre rekonfiguráciu (GT911 má dve 7-bitové adresy určené úrovňou INT pinu pri zapnutí).

/* Pridané dotykové ovládanie do funkcie lvgl_init */

```
#ifndef CONFIG_BSP_TOUCH_ENABLED
const lvgl_port_touch_cfg_t touch_cfg = {
    .disp = my_lvgl_disp,
    .handle = tp, /*LCD touch IO handle*/
};
my_touch_indev = lvgl_port_add_touch(&touch_cfg); /*Add LCD touch as an input device*/
if (my_touch_indev == NULL)
{
    err = ESP_FAIL;
    DISPLAY_ERROR("LVGL touch port add fail");
}
#endif
return err;
```

Kód pridaný na konci tejto funkcie konfiguruje IO handle získaný počas inicializácie dotykového displeja s predtým nakonfigurovaným lvgl display handle. Potom používa funkciu `lvgl_port_add_touch` na pridanie dotykového funkcionality do lvgl.

1.2.2 Súbor Kconfig

Hlavnou funkciou tohto súboru je pridať požadované konfigurácie do súboru `sdkconfig`, čo umožňuje vykonávať určité úpravy parametrov prostredníctvom grafického rozhrania. Novo pridané konfiguračné parametre súvisiace s dotykou definujú prerušovací pin a resetovací pin.

```
config BSP_TOUCH_ENABLED
bool "Enable touch functions"
depends on BSP_I2C_ENABLED
```

```
if BSP_TOUCH_ENABLED
    config TOUCH_GPIO_RST
        int "GPIO For Touch RST"
        default 40

    config TOUCH_GPIO_INT
        int "GPIO For Touch INT"
        default 41
endif
```

1.2.3 Súbor CMkaLists.txt

Funkčnosť tohto príkladu spočíva predovšetkým v ovládači bsp_display. Aby ste mohli úspešne vyvolať obsah priečinka bsp_display v hlavnej funkcii, musíte nakonfigurovať súbor CMakeLists.txt, ktorý sa nachádza v priečinku bsp_display. Podrobnosti konfigurácie sú nasledovné:

```
FILE(GLOB_RECURSE component_sources "*.c")

idf_component_register(SRCS ${component_sources}
                      INCLUDE_DIRS "include"
                      REQUIRES driver_esp_lcd_ek79007 esp_lcd_touch_gt911 lvgl esp_lvgl_port bsp_i2c)
```

V tomto súbore CMakeLists.txt najskôr definujeme adresáre pre zdrojové súbory a hlavičkové súbory spolu s požadovanými knižnicami ovládačov (knižnica ovládača pre čip ovládača displeja ek79007, knižnica ovládača lvgl, novo pridaná knižnica ovládača pre čip dotykového ovládača gt911 a naša knižnica ovládača bsp_i2c). Potom pomocou príkazu idf_component_register zaregistrujeme tieto nastavenia v systéme zostavovania, aby projekt mohol využívať funkcie ovládača bsp_display.

1.2.4 hlavný priečinok

Hlavný priečinok slúži ako hlavný adresár pre spúšťanie programu. Obsahuje hlavnú spustiteľnú funkciu main.c a hlavičkový súbor main.h, ktorý sa nachádza v priečinku include. Pridajte hlavný priečinok do súboru CMakeLists.txt systému zostavenia.

Súbor main.h odkazuje predovšetkým na požadované hlavičkové súbory: funkcie využívajúce ovládač bsp_display vyžadujú hlavičkový súbor bsp_display, zatiaľ čo funkcie využívajúce ovládač bsp_led vyžadujú hlavičkový súbor bsp_led.

Nižšie je analýza programu main.c: Inicializácia systému a spustenie funkcií pre zobrazenie, dotykové ovládanie a LED funkcie.

```
/*main.c*/
#ifdef CONFIG_BSP_LED_ENABLED
    err = led_init(); /*RMT LED Initialization*/
    if (err != ESP_OK)
        init_fail("led", err);
    vTaskDelay(200 / portTICK_PERIOD_MS);
    set_led_status(0x00000000); /*All the LEDs are off*/
#endif

#ifdef CONFIG_BSP_DISPLAY_ENABLED
    #if defined(CONFIG_BSP_TOUCH_ENABLED) && defined(CONFIG_BSP_I2C_ENABLED)
        err = touch_init(); /*touch Initialization*/
        if (err != ESP_OK)
            init_fail("display touch", err);
    #endif
    err = display_init(); /*Display Initialization*/
    if (err != ESP_OK)
        init_fail("display", err);
#endif
}
```

Tento kód sa nachádza vo funkcii init, ktorá ukladá inicializačné funkcie, ktoré sa majú volať, a vyhodnocuje ich návratové hodnoty. Ak návratový stav nie je ESP_OK, kód vytlačí chybovú správu a zastaví vykonávanie.

/* Inicializácia obrazovky a funkcia zobrazenia display_test */

```
void display_test()
{
    if (lvgl_port_lock(0))
    {
        test_screen = lv_obj_create(NULL);
        lv_obj_set_style_bg_color(test_screen, LV_COLOR_BLACK, LV_PART_MAIN); /*Set the screen's LVGL background color*/
        lv_obj_set_style_bg_opa(test_screen, LV_OPA_COVER, LV_PART_MAIN); /*Set the screen's LVGL background transparency*/
        lv_obj_t *label = lv_label_create(test_screen); /*Create a label object*/
        lv_label_set_text(label, "Hello P4!"); /*Set a new text for a label. Memory will be allocated to store the text by the label.*/
        static lv_style_t label_style;
        lv_style_init(&label_style); /*Initialize a style*/
        lv_style_set_bg_opa(&label_style, LV_OPA_TRANSP); /*Set the style LVGL background color*/
        lv_obj_add_style(label, &label_style, LV_PART_MAIN); /*Add a style to an object*/
        lv_obj_align(label, LV_ALIGN_CENTER, 0, 0); /*Change the alignment of an object*/
        lv_obj_set_style_text_color(label, LV_COLOR_WHITE, LV_PART_MAIN); /*Set the style LVGL text color*/
        lv_obj_set_style_text_font(label, &lv_font_montserrat_42, LV_PART_MAIN); /*Set the style LVGL text font*/
        lv_obj_t *btn001;
        lv_color_t btn_color[] = {LV_COLOR_RED, LV_COLOR_YELLOW, LV_COLOR_GREEN, LV_COLOR_BLUE};
        const char *button_text[] = {"RED", "YELLOW", "GREEN", "BLUE"};

        for (int i = 0; i < 4; i++)
        {
            buttons[i] = lv_btn_create(test_screen); /*Create button object*/
            lv_obj_set_size(buttons[i], 100, 100); /*Set the size of an object*/
            lv_obj_set_pos(buttons[i], (i * 4 + 4 + 200), 200); /*Set the position of an object relative to the set alignment*/
            lv_obj_set_style_radius(buttons[i], 15, LV_STATE_DEFAULT); /*Set rounded corners for display*/
            lv_obj_t *label = lv_label_create(buttons[i]); /*Create a label object*/
            lv_label_set_text(label, button_text[i]); /*Set a new text for a label. Memory will be allocated to store the text by the label.*/
            lv_obj_set_style_text_font(label, &lv_font_montserrat_20, LV_PART_MAIN); /*Set the style LVGL text font*/
            lv_obj_center(label); /*Align an object to the center on its parent*/
            lv_obj_set_style_bg_color(buttons[i], btn_color[i], LV_STATE_DEFAULT); /*Set the style LVGL background color*/
            lv_obj_set_style_text_color(buttons[i], LV_COLOR_BLACK, LV_STATE_DEFAULT); /*Set the style LVGL text color*/
            lv_obj_set_style_border_width(buttons[i], 2, LV_STATE_DEFAULT); /*Set the border width of the object*/
            lv_obj_set_style_bg_opa(buttons[i], LV_OPA_COVER, LV_STATE_DEFAULT); /*Set the border color of the object*/
            lv_obj_set_style_bg_color(buttons[i], btn_color[i], LV_STATE_PRESSED); /*Set the style LVGL background color - Pressed State*/
            lv_obj_set_style_text_color(buttons[i], LV_COLOR_BLACK, LV_STATE_PRESSED); /*Set the style LVGL text color - Pressed State*/
            lv_obj_set_style_border_color(buttons[i], LV_COLOR_GRAY, LV_STATE_PRESSED); /*Set the border color of the object - Pressed State*/

            lv_obj_add_event_cb(buttons[i], button_event_handler, LV_EVENT_ALL, (void *)(&btn_color[i])); /*Add event handling and pass the LED index as user data*/
        }

        lv_scr_load(test_screen); /*Set the current active screen*/
        lvgl_port_unlock();
    }
}
```

Táto funkcia primárne konfiguruje obsah počiatočného zobrazenia obrazovky: nastavuje farbu pozadia a zobrazenie textu prostredníctvom ovládacích prvkov lvgl. (Mení polohu pôvodného zobrazenia „Hello P4“, pridáva štyri tlačidlá a konfiguruje ich podľa potreby)

Funkcia `lv_label_set_text` nastavuje textový obsah zobrazený ovládacím prvkom.

Funkcia `lv_style_set_bg_opa` nastavuje farbu pozadia štýlu.

Funkcia `lv_obj_set_style_text_color` nastavuje farbu zobrazenia textu. Funkcia `lv_obj_set_style_text_font` nastavuje veľkosť písma textu.

Funkcia `lv_obj_set_style_bg_color` nastavuje farbu pozadia. Funkcia `lv_obj_set_style_bg_opa` nastavuje priehľadnosť pozadia. Funkcia `lv_obj_align` nastavuje zarovnanie ovládacieho prvku.

Funkcia `lv_btn_create` vytvára ovládacie prvky tlačidiel (s efektmi stlačenia, uvoľnenia a kliknutia).

Funkcia `lv_obj_set_size` nastavuje veľkosť ovládacieho prvku.

Funkcia `lv_obj_set_pos` nastavuje súradnice x/y ovládacieho prvku

Funkcia `lv_obj_set_style_radius` nastavuje polomer zobrazenia ovládacieho prvku

Funkcia `lv_obj_set_style_border_width` nastavuje šírku okraja ovládacieho prvku v pixeloch

Funkcia `lv_obj_set_style_border_color` nastavuje farbu okraja ovládacieho prvku

Tu používame cyklus `for` na jednotné konfigurovanie zobrazenia tlačidiel, vrátane textu, farby, ohraničenia, polohy atď. Konkrétne efekty nájdete v komentároch kódu. Posledným kľúčovým krokom je pridanie funkcie spätného volania na spracovanie udalostí klávesov Lvgl. (Posledný parameter odovzdaný počas registrácie je argumentom funkcie spätného volania; tu odovzdávame index klávesu pre následné ovládanie LED.)

Poznámka: Pri volaní funkcií `lvgl` mimo funkcie vlákna `lvgl` je potrebné získať mutex zámok. Na získanie zámku použite `lvgl_port_lock` a na jeho uvoľnenie `lvgl_port_unlock`.

/* Funkcia spracovania spätnej volania udalosti tlačidla `button_event_handler` */

RLX COMPONENTS

```

static void button_event_handler(lv_event_t *e)
{
    lv_event_code_t code = lv_event_get_code(e);           /*Get the event code of an event*/
    lv_obj_t *btn = lv_event_get_target(e);               /*Get the LVGL object caused by the event. It's the same even if the event is bubbled*/
    int led_index = (int)(intptr_t)lv_event_get_user_data(e); /*Get the user data passed when the event was registered on the object*/

    if (code == LV_EVENT_PRESSED)
    {
        lv_obj_add_state(btn, LV_STATE_PRESSED); /*Add one or more states to the object*/
        set_single_led_status(led_index, 0xFF); /*Light up according to the color corresponding to the key press*/
        MAIN_INFO("BTN[%d] pressed", led_index);
    }
    else if (code == LV_EVENT_RELEASED)
    {
        lv_obj_clear_state(btn, LV_STATE_PRESSED); /*Add one or more states to the object*/
        set_single_led_status(led_index, 0x00);
        MAIN_INFO("BTN[%d] released", led_index);
    }
}

```

Tu získavame údaje `lv_event_t` získané z spätnej volania. Používame funkciu `lv_event_get_code` na určenie typu udalosti, funkciu `lv_event_get_target` na identifikáciu stlačenej klávesy a funkciu `lv_event_get_user_data` na získanie príslušného vstupného parametra nakonfigurovaného pre túto klávesu.

Na základe udalosti určíme, či ide o udalosť stlačenia tlačidla. Ak áno, nastavíme tlačidlo do stlačeného stavu a riadime príslušnú LED diódu tak, aby sa rozsvietila podľa vstupného parametra. Počas udalosti uvoľnenia zrušíme stlačený stav tlačidla a riadime príslušnú LED diódu tak, aby sa vypala na základe vstupného parametra.

```

void app_main(void)
{
    MAIN_INFO("-----Demo version-----");
    MAIN_INFO("-----Start the test-----");
    Init();
#ifdef CONFIG_BSP_DISPLAY_ENABLED
    set_lcd_blight(100); /*Set the screen backlight to maximum brightness*/
    if (lvgl_port_lock(0)) /*Take LVGL mutex*/
    {
        display_test(); /*Set the screen's LVGL default display page*/
        lvgl_port_unlock(); /*Give LVGL mutex*/
    }
#endif
}

```

V funkcii `app_main` najskôr zapnite podsvietenie s jasom nastaveným na 100 % a potom inicializujte obsah zobrazovania na obrazovke. Všetky operácie dotykových tlačidiel sa vykonávajú v rámci funkcií spätného volania.

1.3.5 Súbor `CMkaLists.txt`

Aby ste mohli úspešne vyvolať obsah priečinka `bsp_display` v rámci hlavnej funkcie, musíte nakonfigurovať súbor `CMakeLists.txt`, ktorý sa nachádza v hlavnom priečinku. Podrobnosti konfigurácie sú nasledovné:

```

FILE(GLOB_RECURSE main ${CMAKE_SOURCE_DIR}/main/*.c)

idf_component_register(SRCS ${main}
                       INCLUDE_DIRS "include"
                       REQUIRES bsp_display bsp_led)

```

Najskôr sa definujú adresáre pre zdrojové súbory a hlavičkové súbory spolu s požadovanými knižnicami ovládačov – konkrétne knižnice ovládačov potrebné na prepojenie `bsp_display` a `bsp_led` (`bsp_i2c` je už prepojený v priečinku `bsp_display`). Potom sa tieto nastavenia zaregistrujú v systéme zostavenia pomocou príkazu `idf_component_register`, čím sa umožní hlavnej funkcii využívať tieto funkcie ovládačov.

1.4 Postup programovania

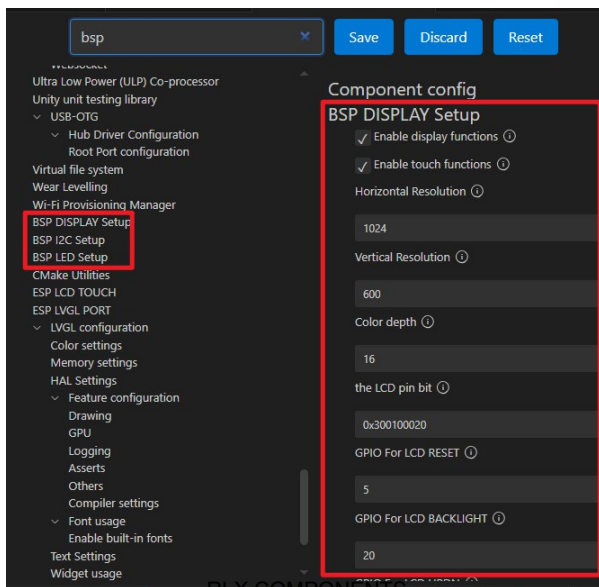
www.rlx.sk

Pripojte zariadenie P4 k počítaču prostredníctvom USB

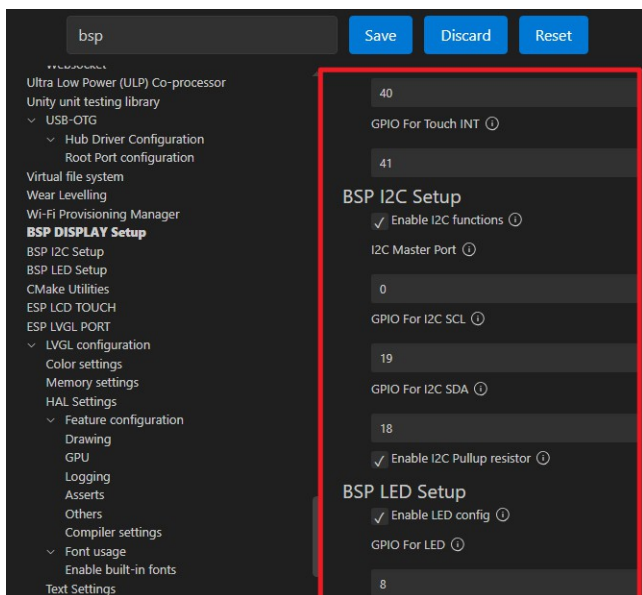
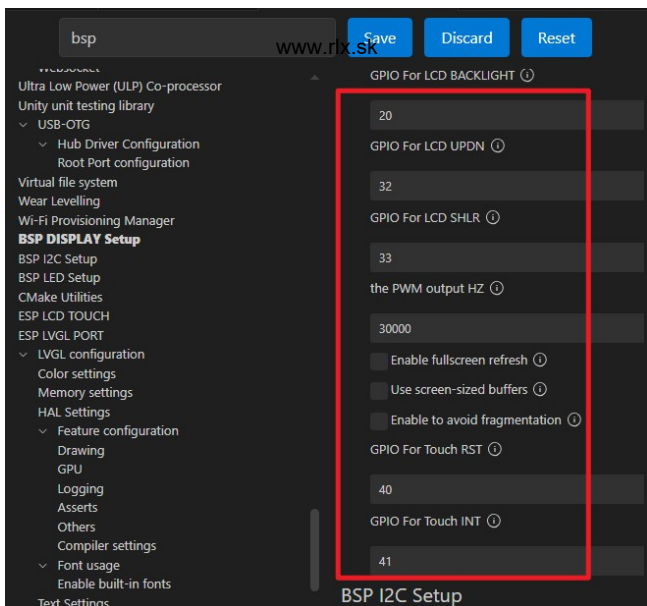


1.4.1 Po klonovaní kódu cez Git (odkaz bude potvrdený) vymažte všetky lokálne kompilačné informácie. Nakonfigurujte prostredie IDF a model čipu pre kompiláciu podľa lekcie 1 a nastavte číslo sériového portu pre programovanie.

1.4.2 Nasledujúca konfigurácia SDKConfig je z veľkej časti identická s lekciou 1, jednoducho prekonfigurujte piny LED.



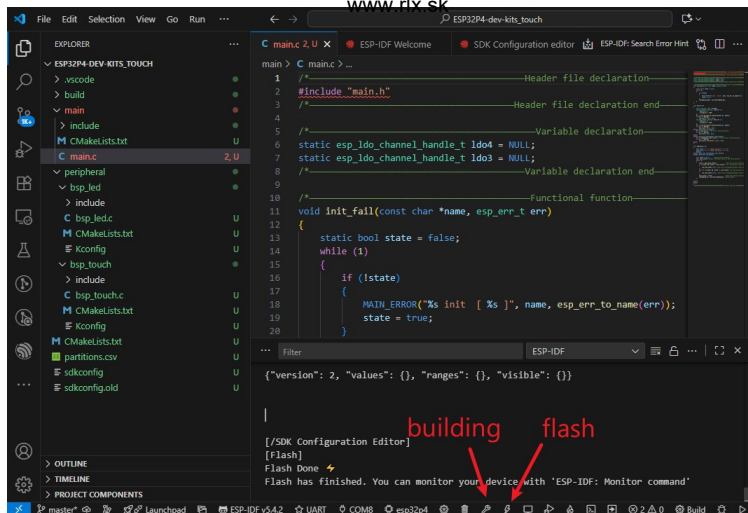
RLX COMPONENTS



RLX COMPONENTS

1.4.3 Kliknite na Kompilovať. Po úspešnej kompilácii kliknite na Stiahnuť.

www.rlx.sk



www.rlx.sk



USNADNITE SI VÝROBU

RLX COMPONENTS