远程多功能探测器

湖南大學

HUNAN UNIVERSITY

班级：计科2101

学号：202108010111

姓名：庞海鑫

指导老师：刘三一

摘要

为了更好地了解和掌握单片机的相关知识，我们需要通过实践来加深对知识的理解。在本次实验中，我通过阅读STC15F2K60S2的数据手册，STC-B学习板的原理图以及相关芯片的数据手册，对串口通信、定时器、中断、IO口等知识有了更深入的理解。通过单片机编程实现了一个远程多功能探测器，实现了对温度、光照强度，震动和磁场的检测，并通过串口将检测到的数据发送到上位机或者通过WI-FI模块发送到服务器，实现了远程监控的功能。

关键词

单片机；串口通信；定时器；中断；IO口

目录

引言

单片机是一种集成电路芯片，它集中了微处理器的所有功能部件，包括CPU、RAM、ROM、I/O端口、定时器、串行通信接口、并行通信接口、中断系统等，它是一种专用集成电路，用于控制电器或机电设备。单片机的应用范围非常广泛，从家用电器到工业控制，从汽车电子到军事设备，从电子游戏机到电脑外围设备，都有单片机的身影。

基本BSP编写(部分模块)

串口通信

核心代码

```
INTERRUPT_USING(__uart1, SI0_VECTOR, 1) {
  if(RI) {
    RI = 0;
    if(uart_cfg[0].len) {
      if(uart_cfg[0].idx < uart_cfg[0].len) {
        uart_cfg[0].buf[uart_cfg[0].idx++] = SBUF;
        if(uart_cfg[0].idx == uart_cfg[0].len) {
          uart1_recv_flag = 1;
        }
      }
    }
  }
  if(TI) {
    TI = 0;
    if(uart_cfg[1].idx < uart_cfg[1].len) {
      SBUF = uart_cfg[1].buf[uart_cfg[1].idx++];
    } else {
      uart1_send_flag = 1;
    }
  }
}
INTERRUPT_USING(__uart2, SI1_VECTOR, 1) {
  while(S2CON & 0x01) {
    S2CON &= ~0x01;
    if(uart_cfg[2].len) {
      if(uart_cfg[2].idx < uart_cfg[2].len) {
        uart_cfg[2].buf[uart_cfg[2].idx++] = S2BUF;
        if(uart_cfg[2].idx == uart_cfg[2].len) {
          uart2_recv_flag = 1;
        }
      }
    }
  }
  while(S2CON & 0x02) {
    S2CON &= ~0x02;
    if(uart_cfg[3].idx < uart_cfg[3].len) {
      S2BUF = uart_cfg[3].buf[uart_cfg[3].idx++];
    } else {
      uart2_send_flag = 1;
      P3_7 = 0x0;
    }
  }
}
```

代码解释

串口通信的核心代码是中断函数，中断函数的核心代码如上所示。串口通信的核心是通过中断函数来实现的，当串口接收到数据时，会触发中断函数，中断函数会将接收到的数据存入 `uart_cfg_recv` 函数设定的缓冲区中，当缓冲区满时，会将 `uart_cfg_recv_flag` 置1，表示接收到了完整的数据。当串口发送数据时，会触发中断函数，中断函数会将 `uart_cfg_send` 函数设定的缓冲区中的数据发送出去，当缓冲区中的数据发送完毕时，会将 `uart_cfg_send_flag` 置1，表示数据发送完毕。

定时器

核心代码

```
__sysclk = clk;
TMOD = 0x00;  // T0和T1都是工作在模式0下,即16位自动重装载模式
TH0 = (65535 - __sysclk / 1000) >> 8;
TL0 = (65535 - __sysclk / 1000) & 0xff;
IE |= 0x02;
IP &= ~0x2;
```

代码解释

定时器的核心代码是中断函数，中断函数的核心代码如上所示。定时器的核心是通过中断函数来实现的，设置周期为1ms，当定时器计数到1ms时 `sys_tick` 加1，当 `sys_tick` 计数到10时会设定10ms定时器的 `flag` 为1，表示1ms定时器已经计数到10ms。100ms与1s定时器同理。

ADC

核心代码

```
INTERRUPT_USING(__adc, 5, 1) {
  static XDATA uint8_t count = 0, curchannel = 0x3;  // rt channel
  static XDATA uint16_t sum = 0;
  sum += (ADC_RES << 2) + (ADC_RESL >> 6);  //__ADC_GET();
  ++count;
#define __ADC_AVG_CNT 2
  if(count == __ADC_AVG_CNT) {
    switch(curchannel) {
      case 0x3 :  // rt channel
        adcs.rt = __d2t[(((sum + (__ADC_AVG_CNT >> 1)) / __ADC_AVG_CNT) >> 2) - 1];
        curchannel = 0x4;  // rop channel
        break;
      case 0x4 :  // rop channel
        adcs.rop = (sum + (__ADC_AVG_CNT >> 1)) / __ADC_AVG_CNT;
        curchannel = 0x7;  // nav channel
        break;
      case 0x7 :  // nav channel
        adcs.nav = ((sum + (__ADC_AVG_CNT >> 1)) / __ADC_AVG_CNT) >> 7;
#undef __ADC_AVG_CNT
        curchannel = 0x3;  // rt channel
        break;
      default :
        break;
    }
    count = 0;
    sum = 0;
  }
  __ADC_START(curchannel);
#undef __ADC_START
}
```

代码解释

ADC的核心代码是中断函数，中断函数的核心代码如上所示。ADC的核心是通过中断函数来实现的，当ADC转换完成时，会触发中断函数，中断函数会将转换完成的数据求平均值，然后将数据存入 `adcs` 结构体中。

项目实现

项目功能描述

本项目实现了一个远程多功能探测器，实现了对温度、光照强度，震动和磁场的检测，并通过串口将检测到的数据发送到上位机或者通过WI-FI模块发送到服务器，实现了远程监控的功能。

项目实现逻辑

使用stc-b学习板上的温度传感器、光敏电阻、震动传感器、磁场传感器收集环境信息，通过串口将数据发送到上位机或者通过WI-FI模块（用ESP32模块代替）发送到服务器，实现了远程监控的功能。

stc-b学习板部分

模式选择

```
static bit mode = 0;
void _key1(void) {
  mode = !mode;
}
```

通过设置 `key1` 按键，可以选择显示/修改时间或者显示温度、光照强度，震动和磁场的检测。

显示/修改时间

```
void _key2(void) {
  if(mode == 0) {
    if(!m_time) {
      m_t_e_idx = 0;
      m_t_e_cnt = 0;
    }
    m_time = !m_time;
  } else {
    sys_reset();
  }
}
```

当 `mode` 为0时，按下 `key2` 按键可以选择显示/修改时间，当 `mode` 为1时，按下 `key2` 按键可以选择重启系统。

```
void _change(uint8_t step) {
  if(m_t_d_idx) {
    switch(m_t_e_idx) {
      case 0 :
        rtc.year = (rtc.year + step * 10) % 100;
        break;
      case 1 :
        rtc.year = (rtc.year + step) % 10;
        break;
      case 3 :
        rtc.month = (rtc.month + step * 10) % 100;
        break;
      case 4 :
        rtc.month = (rtc.month + step) % 10;
        break;
      case 6 :
        rtc.date = (rtc.date + step * 10) % 100;
        break;
      case 7 :
        rtc.date = (rtc.date + step) % 10;
        break;
      default :
        break;
    }
  } else {
    switch(m_t_e_idx) {
      case 0 :
        rtc.hour = (rtc.hour + step * 10) % 100;
        break;
      case 1 :
        rtc.hour = (rtc.hour + step) % 10;
        break;
      case 3 :
        rtc.minute = (rtc.minute + step * 10) % 100;
        break;
      case 4 :
        rtc.minute = (rtc.minute + step) % 10;
        break;
      case 6 :
        rtc.second = (rtc.second + step * 10) % 100;
        break;
      case 7 :
        rtc.second = (rtc.second + step) % 10;
        break;
      default :
        break;
    }
  }
}
void _nav_up(void) {
```

```
  if(mode == 0 && m_time && m_t_d_idx) {
    _change(1);
  }
}
void _nav_down(void) {
  if(mode == 0 && m_time && m_t_d_idx) {
    _change(9);
  }
}
void _nav_left(void) {
  if(mode == 0) {
    if(m_time) {
      if(m_t_e_idx == 0)
        m_t_e_idx = 7;
      else if(m_t_e_idx == 3)
        m_t_e_idx -= 2;
      else if(m_t_e_idx == 6)
        m_t_e_idx -= 2;
      else {
        --m_t_e_idx;
      }
    } else {
    }
  }
}
void _nav_right(void) {
  if(mode == 0) {
    if(m_time) {
      if(m_t_e_idx == 7)
        m_t_e_idx = 0;
      else if(m_t_e_idx == 1)
        m_t_e_idx += 2;
      else if(m_t_e_idx == 4)
        m_t_e_idx += 2;
      else {
        ++m_t_e_idx;
      }
    } else {
    }
  } else {
  }
}
void _nav_center(void) {
  if(mode == 0 && m_time) {
    rtc_write();
  }
}
```

通过导航键可以选择修改时间，通过 `nav_up` 和 `nav_down` 可以修改时间的某一位，通过 `nav_left` 和 `nav_right` 可以选择修改时间的哪一位，通过 `nav_center` 可以将修改后的时间写入RTC中。

显示温度、光照强度，震动和磁场的检测

```
void _100ms(void) {
  if(m_s_hall) {
    m_s_h_cnt = m_s_c_max;
    m_s_hall = 0;
  } else {
    if(m_s_h_cnt) {
      --m_s_h_cnt;
    }
  }
  if(m_s_vib) {
    m_s_v_cnt = m_s_c_max;
    m_s_vib = 0;
  } else {
    if(m_s_v_cnt) {
      --m_s_v_cnt;
    }
  }
  if(mode == 0) {
    if((!m_time) & m_t_update) {
      rtc_read();
    } else {
      if(m_t_d_idx) {
        _set_display(display_num_decoding[rtc.year / 10], display_num_decoding[rtc.year % 10], 0
          display_num_decoding[rtc.month / 10], display_num_decoding[rtc.month % 10], 0x40,
          display_num_decoding[rtc.date / 10], display_num_decoding[rtc.date % 10]);
      } else {
        _set_display(display_num_decoding[rtc.hour / 10], display_num_decoding[rtc.hour % 10], 0
          display_num_decoding[rtc.minute / 10], display_num_decoding[rtc.minute % 10], 0x40,
          display_num_decoding[rtc.second / 10], display_num_decoding[rtc.second % 10]);
      }
      if(m_time) {
        if(m_t_e_cnt < (m_t_e_c_max >> 1)) {
          display_seg[m_t_e_idx] = 0;
        } else if(m_t_e_cnt == m_t_e_c_max) {
          m_t_e_cnt = 0;
        }
        ++m_t_e_cnt;
      }
    }
    m_t_update = !m_t_update;
  } else if(mode == 1) {
    _set_display(display_num_decoding[adc.rop / 100], display_num_decoding[(adc.rop / 10) % 10],
      display_num_decoding[adc.rop % 10], 0x40, display_num_decoding[adc.rt / 10], display_num_d
      0x40, (m_s_h_cnt > 0 ? 0x30 : 0x00) | (m_s_v_cnt > 0 ? 0x06 : 0x00));
  }
}
```

100ms 定时器中断函数中，通过 m_s_h_cnt 和 m_s_v_cnt 来判断是否检测到震动和磁场，通过 m_t_update 来判断是否需要更新时间，通过 m_t_e_cnt 来判断是否需要闪烁时间的某一位。

串口部分

```
static uint8_t uartsbuf[2][4];
void _set_sbuf(uint8_t idx) {
  uartsbuf[idx][0] = adc.rop;
  uartsbuf[idx][1] = adc.rop >> 8;
  uartsbuf[idx][2] = adc.rt;
  uartsbuf[idx][3] = (m_s_v_cnt > 0 ? 0x02 : 0x00) | (m_s_h_cnt > 0 ? 0x01 : 0x00);
}
static uint8_t uart2sbuf[4];
void _uart1_recv(void) {
  uart_send(DevUART1, uartsbuf[0], 4);
}
void _uart2_recv(void) {
  display_led |= 0x01;
  uart_send(DevUART2, uartsbuf[1], 4);
}
```

当串口接收到数据时，会触发中断函数，中断函数会将接收到的数据存入 uart_cfg_recv 函数设定的缓冲区中，当缓冲区满时，调用 _uart1_recv 或者 _uart2_recv 函数，将数据发送出去。

ESP32部分

```
#include <Arduino.h>
#include <HTTPClient.h>
#include <WiFi.h>

void wifi_connect() {
  WiFi.begin("km", "12345678");
  uint8_t cnt = 0;
  while(WiFi.status() != WL_CONNECTED && cnt++ < 10) {
    delay(300);
  }
}

String uri = "http://47.115.204.164:8080/put?";

uint8_t buf[4];
char tmpbuf[10] = {};
void put() {
  Serial.println("put");
  HTTPClient http;
  uint8_t cnt = sprintf(tmpbuf, "%03u%02d%d%d", *(uint16_t *)buf, buf[2], buf[3] & 0x1, (buf[3]
  if(cnt > 0) {
    http.begin(uri+ String(tmpbuf, cnt));  // HTTP begin
    Serial.println(uri + String(tmpbuf, cnt));
    int httpCode = http.GET();
  }
  http.end();
}
uint8_t s2flag = 0;
void s2recv(void) {
  while(Serial2.available() < 4)
    ;
  Serial2.readBytes(buf, 4);
  s2flag = 1;
}
void setup() {
  Serial.begin(115200);  // open the serial port at 115200 bps;
  wifi_connect();
  Serial2.begin(115200);
  Serial2.onReceive(s2recv);
  Serial.println("start");
}
void loop() {
  if(s2flag) {
    put();
    s2flag = 0;
  }
  delay(1000);
```

```
  Serial2.print('1');
}
```

当串口接收到数据时，会触发中断函数，中断函数会将接收到的数据存入 `buf` 中，并将 `s2flag` 置1，表示接收到了完整的数据。当 `s2flag` 为1时，调用 `put` 函数，将数据发送到服务器。

服务器部分（示例）

```
package main

import (
        "fmt"
        "log"
        "net/http"
        "strconv"
)

func main() {
        rop := ""
        temp := ""
        hall := false
        vib := false
        http.HandleFunc("/put", func(w http.ResponseWriter, r *http.Request) {
                rop = r.URL.RawQuery[:3]
                temp = r.URL.RawQuery[3:5]
                hall = r.URL.RawQuery[5] == '1'
                vib = r.URL.RawQuery[6] == '1'
                fmt.Println("rop=", rop, " temp=", temp, hall, vib)
                fmt.Fprintf(w, "")
        })
        http.HandleFunc("/get", func(w http.ResponseWriter, r *http.Request) {
                w.Header().Add("rop", rop)
                w.Header().Add("temp", temp)
                w.Header().Add("hall", strconv.FormatBool(hall))
                w.Header().Add("vib", strconv.FormatBool(vib))
                fmt.Fprintf(w, "")
        })
        fmt.Println("start")
        log.Fatal(http.ListenAndServe(":8080", nil))
}
```

当服务器接收到 `/put` 请求时，会将请求中的数据存入 `rop` 、 `temp` 、 `hall` 和 `vib` 中。当服务器接收到 `/get` 请求时，会将 `rop` 、 `temp` 、 `hall` 和 `vib` 中的数据发送给客户端。

上位机部分（示例）

```
use iced::widget::{self, row, Column};
use iced::{
    executor, window, Application, Command, Element, Length, Settings, Size, Subscription, Theme
};
use iced_futures::futures::channel::mpsc::{self, Receiver, Sender};
use iced_futures::subscription;
#[derive(Debug, Clone)]
pub enum UpdateMessage {
    SetSource(String),
}
#[derive(Debug, Clone)]
pub enum AppMessage {
    Scan,
    Connect(Sender<UpdateMessage>),
    SourceSelected(String),
    Update(String, String, bool, bool),
}
pub struct App {
    // available_ports: Vec<String>,
    sender: Option<Sender<UpdateMessage>>,
    available_sources: Vec<String>,
    choosed_source: Option<String>,
    info: Vec<(String, String, bool, bool)>,
}
const WIN_INIT_SIZE: iced::Size<u32> = Size {
    width: 300,
    height: 245,
};

impl Application for App {
    type Executor = executor::Default;
    type Message = AppMessage;
    type Theme = Theme;
    type Flags = ();

    fn new(_: Self::Flags) -> (Self, Command<AppMessage>) {
        (
            Self {
                //None,
                available_sources: Vec::new(),
                sender: None,
                choosed_source: None,
                info: Vec::new(),
            },
            window::resize(WIN_INIT_SIZE),
        )
    }
```

```
fn title(&self) -> String {
    String::from("Rating")
}

fn update(&mut self, message: AppMessage) -> Command<AppMessage> {
    match message {
        AppMessage::Scan => {
            self.available_sources = match serialport::available_ports() {
                Ok(v) => v.iter().map(|p| p.port_name.clone()).collect(),
                Err(_) => Vec::<String>::new(),
            };
            Command::none()
        }
        AppMessage::Connect(sender) => {
            self.sender = Some(sender);
            Command::none()
        }
        AppMessage::SourceSelected(s) => {
            self.choosed_source = Some(s.clone());
            if let Some(sender) = &mut self.sender {
                if let Err(e) = sender.try_send(UpdateMessage::SetSource(s)) {
                    println!("send failed: {:?}", e)
                } else {
                    println!("send ok")
                }
            }
            Command::none()
        }
        AppMessage::Update(rop, temp, hall, vib) => {
            println!(
                "update: rop={}, temp={}, hall={}, vib={}",
                rop, temp, hall, vib
            );
            self.info.push((rop, temp, hall, vib));
            if self.info.len() > 10 {
                self.info.remove(0);
            }
            Command::none()
        }
    }
}
fn subscription(&self) -> Subscription<AppMessage> {
    struct SomeSub;
    enum Source {
        Net,
        Serial(Box<dyn serialport::SerialPort>),
        None,
    }
    enum WorkState {
        SetSource,
        Normal,
```

```
    TrySend(AppMessage),
}
enum State {
    Starting,
    Working(Receiver<UpdateMessage>, Source, WorkState),
}
Subscription::batch([
    //timer
    iced::time::every(std::time::Duration::from_millis(1000)).map(|_| AppMessage::Scan),
    subscription::channel(
        std::any::TypeId::of::<SomeSub>(),
        1000,
        |mut output| async move {
            let mut state = State::Starting;

            loop {
                use iced_futures::futures::sink::SinkExt;
                match &mut state {
                    State::Starting => {
                        // Create channel
                        let (sender, receiver) = mpsc::channel(1000);

                        // Send the sender back to the application
                        // let _ = output.send(AppMessage::Ready(sender)).await;
                        if let Err(e) = output
                            .send(
                                // AppMessage::Ready(sender)
                                AppMessage::Connect(sender),
                            )
                            .await
                        {
                            // log::warn(format!("send ready failed: {:?}", e).as_str())
                        } else {
                            // We are ready to receive messages
                            state = State::Working(
                                receiver,
                                Source::None,
                                WorkState::SetSource,
                            );
                        }
                    }
                    State::Working(recv, src, state) => match state {
                        WorkState::SetSource => {
                            // Read next input sent from `Application`
                            use iced_futures::futures::StreamExt;
                            match recv.select_next_some().await {
                                UpdateMessage::SetSource(s) => {
                                    if s == "Net" {
                                    } else {
                                        match serialport::new(s, 115200)
                                            .timeout(std::time::Duration::from_millis(50
```

```
                        .open()
                    {
                        Ok(p) => {
                            *src = Source::Serial(p);
                            *state = WorkState::Normal;
                            println!("open serial ok");
                        }
                        Err(_) => {}
                    }
                }
            }
        }
    }
    WorkState::Normal => match src {
        Source::Serial(port) => match port.write(&[0x01]) {
            Ok(_) => {
                let mut buf = [0 as u8; 4];
                match port.read_exact(buf.as_mut()) {
                    Ok(_) => {
                        let rop = format!("{}{}", buf[1], buf[0]);
                        let temp = format!("{}", buf[2]);
                        let hall = buf[0] & 0x01 == 0x01;
                        let vib = buf[0] & 0x02 == 0x02;
                        println!(
                            "rop={}, temp={}, hall={}, vib={}",
                            rop, temp, hall, vib
                        );
                        *state = WorkState::TrySend(
                            AppMessage::Update(rop, temp, hall, vib)
                        );
                    }
                    Err(_) => {
                        *state = WorkState::SetSource;
                    }
                }
            }
            Err(_) => {
                *state = WorkState::SetSource;
            }
        },
        Source::Net => {}
        Source::None => {}
    },
    WorkState::TrySend(msg) => {
        if let Err(_) = output.feed(msg.clone()).await {
            println!("send failed: {:?}", msg);
            *state = WorkState::SetSource;
        } else {
            tokio::time::sleep(std::time::Duration::from_millis(500)
                .await;
            *state = WorkState::Normal;
```

```
                                }
                            }
                        },
                    }
                }
            },
        )
        .into(),
    ])
}
fn view(&self) -> Element<AppMessage> {
    let ch = widget::pick_list(
        &self.available_sources,
        self.choosed_source.clone(),
        AppMessage::SourceSelected,
    )
    .placeholder("choose a source")
    .width(Length::Fill);
    let col = self
        .info
        .iter()
        .map(|(rop, temp, hall, vib)| {
            row![
                row!["rop=", widget::text(rop)].spacing(5),
                row!["temp=", widget::text(temp)].spacing(5),
                row!["hall=", widget::text(hall)].spacing(5),
                row!["vib=", widget::text(vib)].spacing(5),
            ]
            .spacing(5)
            .into()
        })
        .collect::<Vec<_>>();
    Column::with_children(vec![
        ch.into(),
        Column::with_children(col).spacing(5).into(),
    ])
    .into()
}
}
fn main() {
    let _ = App::run(Settings::default());
}
```

每隔1s扫描串口，将扫描到的串口发送给 `App` ，当 `App` 选择了串口后，会将串口发送给 `SomeSub` ，`SomeSub` 会不断地从串口读取数据，并将数据发送给 `App` 。

参考文献

学习通示例代码，stc-b学习板原理图，stc15f2k60s2数据手册，ESP32数据手册，使用框架相关文档等