代码通读

作者: Xvsenfeng

LichuangDevBoard(类)=> 嘉立创开发板的初始化函数

InitializeI2c=> 初始化I2c驱动

InitializeSpi=> SPI驱动

InitializeButtons=> 初始化一个按钮, 使用Button类进行初始化, (跳转)

InitializeSt7789Display=> 初始化一下屏幕的底层驱动, 同时使用LcdDisplay类初始化顶部的驱动(跳转)

InitializeIot=>初始化物联网, 实际是初始化了一个Speaker 的ting子类, 加入到实际的控制器里面

重写函数GetAudioCodec=>实现音频输入输出的初始化

BoxAudioCodec(类)一个音频的控制器, 从nvs区里面读取一部分数据

...

整体的处理

1. 初始化不同的开发板
2. 在main函数里面处理监听的音频或播放的音频
3. 使用mqtt处理函数监听指令, 针对不同的指令进行显示以及设置

按钮

作用

嘉立创的开发板这里只注册了一个按钮, boot_button_, 这个按钮实现的是短按重置Wifi, 长按按下的时候是开始监听, 抬起释放

实现

> 使用库manage_component进行管理

Button(类)=>控制一个按钮, 使用button_config_t记录不同的GPIO类型的初始化, 长按短按时间, 和不同的类型, 初始化使用的是GPIO btn

iot_button_create=>根据按钮的不同种类建立不同的按钮

button_gpio_init=> 初始化按键的底层硬件配置实现

button_create_com=> 按键dev的初始化, 传入按键的读取函数, 初始化的dev加入链表

```
//button handle list head.
static button_dev_t *g_head_handle = NULL;
```

Button::OnClick=> 注册一个按下的事件

iot_button_register_cb=> 注册回调函数

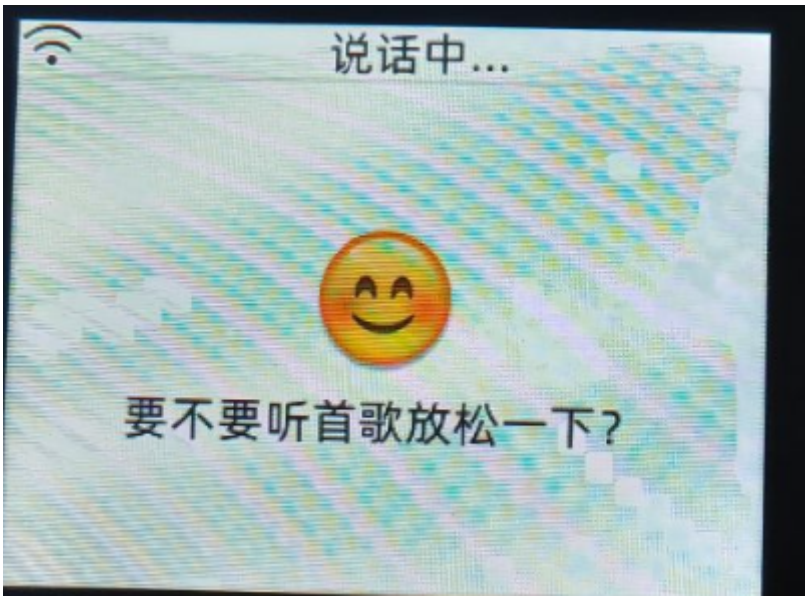
button_cb=> 处理按键的函数, 使用button_handler进行处理每一个按键

button_handler=>使用状态机管理每一个按键

显示

作用

显示提示信息, 用户的返回值等, 使用ShowNotification显示在最上面的状态栏, SetStatus也是同样的 这两个位置是一样的, 交替出现



实际实现

lcd_display.cc

使用类LcdDisplay进行管理, 继承Display

- Display

记录屏幕的大小信息以及显示页面上面的各种图标, 还有一个加锁解锁的函数, 实际使用锁的方式是创建一个DisplayLockGuard类, 构造函数是加锁, 析构函数是解锁

- LcdDisplay

加入了背光控制, 构造函数的时候实现了lvgl等的较上层小时的初始化, 以及一个显示聊天信息的函数

thing.h

作用

用于管理led灯和声音的一部分属性, 可以使用一个命令进行设置, 使用函数GetDescriptorJson即可获取这个thing的名字, 描述, 状态和控制方法, 在初始化的时候发送给服务器

服务器在某些时候会发过来控制函数, 使用Invoke函数调用下面的控制函数

使用DECLARE_THING添加一个物件

实现

- Property类

可以初始化为三种状态, 通过不同的信心获取函数, 返回一个json的字符串, 有一个名字

使用GetDescriptorJson方法可以获取实际的描述类型, 使用GetStateJson获取状态

- PropertyList类

使用一个vector管理Property, 可以使用[]获取对应名字的Property

可以使用`GetDescriptorJson`把所有的Property的描述汇总

- Parameter参数类

可以更具不同的种类进行记录信息, 使用set_xx进行设置, 使用GetDescriptorJson获取描述符

- method

使用一个name和一个参数列表记录一个处理方式

- MethodList参数列表
- 基类

管理一系列的property和method

使用RegisterThing进行添加, 实际是使用一个map类型记录这一个thing的创造函数以及和名字配对

```
#define DECLARE_THING(TypeName) \
    static iot::Thing* Create##TypeName() { \
        return new iot::TypeName(); \   // 把一个新的ting创建出来
    } \
    static bool Register##TypeNameHelper = []() { \
        RegisterThing(#TypeName, Create##TypeName); \ //注册一类
        return true; \
    }();

} // namespace iot
```

thing_manager.h

使用一个vector进行管理Thing类

主函数

在主函数里面使用InputAudio或OutputAudio进行数据的处理, 之后根据使用的开发板进行数据预处理等

也有一个时钟处理函数, 使用Schedule进行注册

音频预处理

audio_process文件夹里面的, 有两个版本, 一个是只简单的处理音频, 另一个是再加上一个音频检测, 分别用于不同的处理阶段, 检测是一直开启的(除了升级的时候), 但是只有检测到数据的时候才发送给服务器

处理音频是随着时间变化的, 把所有的数据预处理一下发送个服务器

作用

1. 音频预处理audio_processor_
2. 唤醒检测wake_word_detect

等待唤醒词到来, 在speaking的时候可以用于打断说话, 可以改变状态

唤醒

使用下面的接口可以获取有语音输入时候的音频

AFE 声学前端算法框架 - ESP32-S3 - — ESP-SR latest 文档

> 关键词CONFIG_USE_AUDIO_PROCESSING
>
> ```
> config USE_AUDIO_PROCESSING
>  bool "启用语音唤醒与音频处理"
>  default y
>  depends on IDF_TARGET_ESP32S3 && USE_AFE
>  help
>      需要 ESP32 S3 与 AFE 支持
> ```

esp_srmodel_init: 加载模型

...

> esp-sr/docs/zh_CN/audio_front_end/README.rst at master · espressif/esp-sr

在InputAudio函数里面输入待处理的音频

- 处理任务

等待DETECTION_RUNNING_EVENT信号以后, 使用fetch函数获取处理的结果, 使用StoreWakeWordData记录一下这一部分的语音

获取到音频数据以后试用下面的处理函数

```
wake_word_detect_.OnWakeWordDetected([this](const std::string& wake_word) {
    Schedule([this, &wake_word]() {
        if (device_state_ == kDeviceStateIdle) {
            SetDeviceState(kDeviceStateConnecting);
            // 它建立一个任务使用 Opus 编码器将 PCM 数据编码为 Opus 格式，
            // 并将编码后的数据存储在 wake_word_opus_ 容器中
            wake_word_detect_.EncodeWakeWordData();

            if (!protocol_->OpenAudioChannel()) {
                ESP_LOGE(TAG, "Failed to open audio channel");
                SetDeviceState(kDeviceStateIdle);
                wake_word_detect_.StartDetection();
                return;
            }

            std::vector<uint8_t> opus;
            // Encode and send the wake word data to the server
            while (wake_word_detect_.GetWakeWordOpus(opus)) {
                protocol_->SendAudio(opus); //发送唤醒词的音频
            }
```

```
            // Set the chat state to wake word detected
            protocol_->SendWakeWordDetected(wake_word); // 发送一个json
            ESP_LOGI(TAG, "Wake word detected: %s", wake_word.c_str());
            keep_listening_ = true;
            SetDeviceState(kDeviceStateListening);
        } else if (device_state_ == kDeviceStateSpeaking) {
            AbortSpeaking(kAbortReasonWakeWordDetected);
        }

        // Resume detection
        wake_word_detect_.StartDetection();
    });
});
wake_word_detect_.StartDetection();
```

处理状态变化的函数, 用于处理一下APP的状态, 这个是根据AFE_VAD_SPEECH参数判断当前是不是有人在说话

```
wake_word_detect_.OnVadStateChange([this](bool speaking) {
    Schedule([this, speaking]() {
        if (device_state_ == kDeviceStateListening) {
            if (speaking) {
                voice_detected_ = true;
            } else {
                voice_detected_ = false; // 这个变量目前使用在led的控制函数里面
            }
            auto led = Board::GetInstance().GetLed();
            led->OnStateChanged();
        }
    });
});
```

其他板子

不加载模型, 只是把数据个AFE处理以后使用发送到服务器

```
audio_processor_.OnOutput([this](std::vector<int16_t>&& data) {
    background_task_->Schedule([this, data = std::move(data)]() mutable {
        opus_encoder_->Encode(std::move(data), [this](std::vector<uint8_t>&&
opus) {
            Schedule([this, opus = std::move(opus)]() {
                protocol_->SendAudio(opus);
            });
        });
    });
});
```

> 在状态是listening 的时候直接把数据发送出去

音频驱动封装

在文件夹audio_codecs里面是各种的音频驱动封装, 在这个文件把所有的芯片包装为一个类, 构造函数的时候进行初始化, 提供音量控制, 开启关闭输入输出通道和读取写数据函数

输出通道在长时间没有输出的时候和升级的时候关闭, 说话的时候重启

默认输入是一直开启的

音频格式处理

发送的数据使用OpusEncoderWrapper把pcm格式的数据转化为opus

实现

BoxAudioCodec<=AudioCodec

- AudioCodec类

记录当前的声音, 输入输出的通道等基础信息

还有读取以及写入的函数

- OpusEncoderWrapper

用于编解码, 把pcm的数据格式转换为opus

多媒体文件格式（五）： PCM / WAV 格式 - 灰色飘零 - 博客园

Opus 音频编码格式 · 陈亮的个人博客

- OpusResampler

重采样, 用于转换音频的频率, 这里设置了三个

```
OpusResampler input_resampler_;
OpusResampler reference_resampler_;
OpusResampler output_resampler_;
```

在输入的分辨率不使16000的时候使用第一个, 输入通道是两个的时候使用第二个, 加入一个参考通道, 再把数据处理以后进行拼接, 编码以后建立一个时钟发送数据, 或给process处理

输出的时候使用在OutputAudio函数里面, 建立一个Schedule播放音频

实际调用

```
// 这里注册的是音频经过AFE处理以后得回调函数
audio_processor_.OnOutput([this](std::vector<int16_t>&& data) {
    background_task_->Schedule([this, data = std::move(data)]() mutable {
        opus_encoder_->Encode(std::move(data), [this](std::vector<uint8_t>&&
opus) {
            Schedule([this, opus = std::move(opus)]() {
                protocol_->SendAudio(opus);
            });
        });
    });
});
```

在函数AudioProcessor::AudioProcessorTask()里面被调用, 第一层 background_task_->Schedul是使用后台处理程序进行处理, 对音频进行格式转换

最后处理结束进入最后一个回调函数

```
[this, opus = std::move(opus)]() {
    protocol_->SendAudio(opus);
}
```

总结

发送的数据音频经过统一的采样率, 通过UDP发送出去, 发送之前需要进行一次aes加密

Setting

这部分记录了Wifi连接信息和用户分配到的信息

非易失性存储库 - ESP32 - — ESP-IDF 编程指南 v5.4 文档

记录在非易失区的数据, 使用nvs进行记录一系列的数据, nvs实际的数据使用键值对的形式进行存储

BackgroundTask

用于处理上层程序注册的任务

在任务分配的时候, esp32s3使用的是heap_caps_malloc在SPIRAM里面进行分配

- Schedule把任务插入到任务列表里面,
- WaitForCompletion: 等待所有任务完成
- BackgroundTaskLoop: 处理任务的循环函数

MQTT

实际使用的时候使用MQTT发送的只有文字, 发送音频的时候使用的是UDP协议, 发送的数据需要使用AES加密, 加密的秘钥是在握手的时候获取的

- Mqtt

记录的是连接时候保持连接的时长

- EspMqtt

mqtt的底层建立以及创建连接

实际建立两个链接,一个是mqtt的用于处理文字, 另一个udp的用于处理音频数据

解析

使用加密算法

ESP32学习笔记（47）——加密算法AES/MD5/SHA_esp32 aes-CSDN博客

协议处理

```
protocol_->OnIncomingJson([this, display](const cJSON* root) {
    // Parse JSON data
    auto type = cJSON_GetObjectItem(root, "type");
    if (strcmp(type->valuestring, "tts") == 0) {
        auto state = cJSON_GetObjectItem(root, "state");
        if (strcmp(state->valuestring, "start") == 0) {
            // 数据来了需要播放
            Schedule([this]() {
                aborted_ = false;
                if (device_state_ == kDeviceStateIdle || device_state_ == kDeviceStateListening) {
                    SetDeviceState(kDeviceStateSpeaking);
                }
            });
        } else if (strcmp(state->valuestring, "stop") == 0) {
            Schedule([this]() {
                // 停止播放
                if (device_state_ == kDeviceStateSpeaking) {
                    background_task_->WaitForCompletion();
                    if (keep_listening_) {
                        protocol_->SendStartListening(kListeningModeAutoStop);
                        SetDeviceState(kDeviceStateListening);
                    } else {
                        SetDeviceState(kDeviceStateIdle);
                    }
                }
            });
        } else if (strcmp(state->valuestring, "sentence_start") == 0) {
            // 获取到助手显示的文字
            auto text = cJSON_GetObjectItem(root, "text");
            if (text != NULL) {
                ESP_LOGI(TAG, "<< %s", text->valuestring);
                display->SetChatMessage("assistant", text->valuestring);
            }
        }
    } else if (strcmp(type->valuestring, "stt") == 0) {
        // 用户的输入Sound=>text
        auto text = cJSON_GetObjectItem(root, "text");
        if (text != NULL) {
            ESP_LOGI(TAG, ">> %s", text->valuestring);
            display->SetChatMessage("user", text->valuestring);
        }
    } else if (strcmp(type->valuestring, "llm") == 0) {
        auto emotion = cJSON_GetObjectItem(root, "emotion");
        if (emotion != NULL) {
            // 设置表情
            display->SetEmotion(emotion->valuestring);
        }
    } else if (strcmp(type->valuestring, "iot") == 0) {
        // 处理一下命令
        auto commands = cJSON_GetObjectItem(root, "commands");
        if (commands != NULL) {
```

```
            auto& thing_manager = iot::ThingManager::GetInstance();
            for (int i = 0; i < cJSON_GetArraySize(commands); ++i) {
                auto command = cJSON_GetArrayItem(commands, i);
                thing_manager.Invoke(command);
            }
        }
    }
});
```

通信的数据格式

[通信协议：WebSocket 连接 - 飞书云文档]

设备状态

```
// 设备的状态
enum DeviceState {
    kDeviceStateUnknown,
    kDeviceStateStarting,
    kDeviceStateWifiConfiguring,
    kDeviceStateIdle, //音频播放结束
    kDeviceStateConnecting, //
    kDeviceStateListening, //发送音频
    kDeviceStateSpeaking, //播放音频
    kDeviceStateUpgrading,
    kDeviceStateFatalError
};
```

```
void Application::SetDeviceState(DeviceState state) {
    if (device_state_ == state) {
        return;
    }

    device_state_ = state;
    ESP_LOGI(TAG, "STATE: %s", STATE_STRINGS[device_state_]);
    // The state is changed, wait for all background tasks to finish
    background_task_->WaitForCompletion();

    auto display = Board::GetInstance().GetDisplay();
    auto led = Board::GetInstance().GetLed();
    led->OnStateChanged();
    switch (state) {
        case kDeviceStateUnknown:
        case kDeviceStateIdle:
            display->SetStatus("待命");
            display->SetEmotion("neutral");
#ifdef CONFIG_USE_AUDIO_PROCESSING
            audio_processor_.Stop();
#endif
            break;
        case kDeviceStateConnecting:
            display->SetStatus("连接中...");
            break;
        case kDeviceStateListening:
```

```
            display->SetStatus("聆听中...");
            display->SetEmotion("neutral");
            ResetDecoder();
            opus_encoder_->ResetState();
#if CONFIG_USE_AUDIO_PROCESSING
            audio_processor_.Start();
#endif
            UpdateIotStates();
            break;
        case kDeviceStateSpeaking:
            display->SetStatus("说话中...");
            ResetDecoder();
#if CONFIG_USE_AUDIO_PROCESSING
            audio_processor_.Stop();
#endif
            break;
        default:
            // Do nothing
            break;
    }
}
```

WiFi连接(不需要更改)

在文件wifi_configuration_ap.cc文件里面

void WifiConfigurationAp::Start()是这个层序的起点, 用于建立一个服务器进行WiFi连接

首先注册两个事件处理, 使用函数`IpEventHandler`处理当 DHCP 客户端成功从 DHCP 服务器获取 IPV4 地址或 IPV4 地址发生改变时，将引发此事件。此事件意味着应用程序一切就绪，可以开始任务

其他的事件处理函数为WifiEventHandler

使用form_submit处理获取到的对文件/submit的POST请求, 对这部分的数据进行解析获取网络, 之后尝试进行连接ConnectToWifi

联网信息

使用ssid_manager文件, 使用一个列表进行记录, 实际使用的时候从nvs分区里面读取数据

KEY

```
#define CONFIG_BAIDU_ACCESS_KEY     "tLfDnkpqLoE0dvjlSjOc1B6y"
#define CONFIG_BAIDU_SECRET_KEY     "jh85lHQQ1kaN5HT9AjI1OCnOT48YHkMz"
```

iot处理

```
else if (strcmp(type->valuestring, "iot") == 0) {
            auto commands = cJSON_GetObjectItem(root, "commands");
            if (commands != NULL) {
                auto& thing_manager = iot::ThingManager::GetInstance();
                for (int i = 0; i < cJSON_GetArraySize(commands); ++i) {
                    auto command = cJSON_GetArrayItem(commands, i);
                    thing_manager.Invoke(command);
                }
            }
        }
```

```
enum ValueType {
    kValueTypeBoolean,
    kValueTypeNumber,
    kValueTypeString
};
```

```
{
    "type": "iot",
    "commands": [
        {
                "name" :  "Speaker",
                "method": "SetVolume",
                "parameters": {
                    "volume": 100
                 }
        }
    ]
 }
```

```
[
    {"name":"Speaker",
     "description":"当前 AI 机器人的扬声器",
     "properties":{"volume":
                    {
                        "description":"当前音量值",
                         "type":"number"}
                  },
     "methods":{
         "SetVolume":{
             "description":"设置音量",
             "parameters":{
                 "volume":{
                     "description":"0到100之间的整数",
                     "type":"number"
                 }
             }
         }
     }
    }
]
```

```
22
23
24 {
25     "session_id":"",
26     "type":"iot",
27     "states":[
28         {
29             "name":"Speaker",
30             "state":{"volume":70}
31         }
32     ]
33 }
```