2024 年电子科技大学盟升杯电子设计竞赛

高年级组 D 题（智能声源识别定位系统）设计报告

组号：D6

提交日期：2024 年 12 月 7 日

目　录

摘要

基于 STM32H750VBT6 的开发板实现该系统的硬件连接和软件开发部分。识别部分由携带一个咪头的语音识别模块 ASR-PRO 组成，采集并识别信号后通过 USART 串口通信传输至 H750，并由 USART 中断实现结果判断；而定位部分的麦克风阵列由位于 20cm*40cm 的矩形顶点的四个 MAX9814 麦克风放大电路组成，由 AD7606 数模转换器实现实时采集四个通道的声音信号，并在定位模式下通过 SPI 串口通信传输到 H750，随后进入 TDOA 时延定位算法得出预估坐标和区域；模式间切换由按键 K3 和 K4 控制的状态机实现；菜单和各个结果由通过 FMC 与 H750 通信的 2.8 寸 LCD 屏幕显示。

关键词

声源定位、命令识别、STM32、信号分析、TDOA 算法、状态机

1 设计任务与要求

1.1 基本要求

设计制作一个智能声源识别定位系统，能够识别不同指令，并且能够指示发出当前指令的声源的位置

（1）设计并制作智能声源识别定位系统，其用于声音信号拾取的麦克风阵列形 式可自行设计，但所使用麦克风个数不能超过 5 个，整个声源定位系统的尺寸不 能超过测量区。系统设置为“识别模式”，将声源置于 C 区域中心位置，系统 能够识别区分 4 条指令，包括“打开灯光”、“关闭灯光”、“打开空调”、“关 闭空调”。指令通过手机或蓝牙音箱播放，也可自制播放设备，识别结果通过液 晶屏幕在系统端进行显示；

（2）声源粗定位功能：系统设置为“粗定位模式”，将 1 个声源随机置于 A-E 的某个区域，播放 1 条指令（可自行指定上述 4 条指令中的 1 条），系统能够通 过液晶屏幕显示当前声源所区域（如：A 区域）；

1.2 发挥部分

（1）声源精定位功能：系统设置为“精定位模式”，将 1 个声源随机置于 A-E 区域内的细分区域（1-4 号区域），播放 1 条指令（可自行指定上述 4 条指令中 的 1 条），系统能够通过液晶屏幕显示当前声源所区域（如：A3 区域）；

（2）声源定位识别功能：系统设置为“定位识别模式”，将 1 个声源随机置于 A-E 区域内的细分区域（1-4 号区域），播放 1 条指令（由测评人员指定上述 4 条指令中的 1 条），系统能够通过液晶屏幕显示当前声源所区域及指令内容，（如： A3 区域 打开灯光）；

（3）抗干扰功能：系统设置为“定位识别模式”，将 2 个声源随机置于 A-E 区 域内的细分区域（1-4 号区域），其中 1 个声源连续播放高斯白噪声（由测评人 员统一准备，播放指令的声源音量不能明显高于播放高斯白噪声声源），另 1 个声源连续重复播放 1 条指令（由测评人员指定上述 4 条指令中的 1 条），播放 指令的时间周期不超过 1 秒，系统能够通过液晶屏幕准确显示当前声源所区域及 指令内容，（如：A3 区域 打开灯光）；

（4）多点声源定位识别功能：系统设置为“定位识别模式”，将 2 个声源随机 置于 A-E 区域内的细分区域（1-4 号区域），同时且连续重复播放 2 条指令（由 测评人员指定上述 4 条指令中的 2 条），每个声源播放指令的时间周期不超过 1 秒，系统能够通过液晶屏幕显示当前声源所区域及指令内容，（如：A3 区域 打 开灯光；C2 区域 关闭空调）；

（5）其他，使用麦克风数量更少、阵列体积更小，可加分；

2 方案设计与论证

2.1 声音信号采集方案

麦克风阵列决定采用处在 20cm*40cm 的矩形顶点的四个 MAX9814 麦克风传感器,该电路能能选择多种增益和释放比。

对于如何采集到 H750 中，初步有以下两种方案：

1. **使用 H750N 内部 ADC**：通过内部的 3 个 ADC+DMA，可以实现快速采集麦克风的信号。

   缺点：由于内部通道是轮询采集，每个通道间存在微小时间差，很难实现同步采集。

   优点：有单 ADC 采集例程代码，可以比较容易编写代码来采集。

2. **采用语音识别模块**:自带只能分析识别，安装咪头并连接至 H750 后，在“天问 PRO”软件编写简单代码可以直接通过 USART 将所识别到的命令通信给 H750 以显示。

   缺点：之前未使用过该模块，上手需要时间。并且例程当中没有 HAL 库代码。

   优点：可以实现 16 为位 8 通道最高 200kSPS 同步采集，可以提供多种过采样倍数。

由于没有经验，我们最先尝试了使用内部 ADC 的方案，发现通道间的时间差较难通过软件修正，这导致定位结果几乎全错；之后在指导老师和科协会长的指导下，采用了外接 AD7606 的采集方案，我们实现了顺利采集，但是由于经验不足和时间原因，我们没能顺利完成这个采集后的数据以合适的格式和方式进入定位算法代码中的关键步骤，导致没有定位结果的显示。

2.2 识别方案

初步有以下两种方案：

3. **手动分析麦克风阵列信号**：通过分析其中一个麦克风通道一段时间的信号频谱，通过特征分析和阈值分析来手动判定 4 种命令。

   难度：中等

4. **采用语音识别模块**:自带只能分析识别，安装咪头并连接至 H750 后，在“天问 PRO”软件编写简单代码可以直接通过 USART 将所识别到的命令通信给 H750 以显示。

   难度：较低

经综合考虑，决定采用语音识别模块。

2.3 定位算法方案

初步有以下两种方案：

1. **TDOA 互相关**：麦克风阵列接收到同一声源信号的时间会有所不同，因为每个麦克风与声源的距离不同，通过在接收的信号之间进行互相关运算，找到信号的时延差。每一对麦克风的时延差对应一个超曲面，交点即为声源的位置。

   优点：可以保证较高的定位精度，并且不依赖声源信号的功率，只依赖时间差。

   缺点：各个麦克风的时钟需要高度同步，否则时延差计算会不准确，并且在信号可能发生反射路径干扰的情况精度会大幅下降。

2. **源到达角（DOA）**：测量声源信号到达每个麦克风阵列的角度。通常使用阵列天线（麦克风阵列）来接收声源的信号。通过分析各个麦克风接收到信号的相位差，计算信号到达的角度，也可以通过计算多个麦克风阵列接收到信号的方向，形成交点来确定声源的位置

   优点：相对容易实现，尤其是在声源数量较少时。

   缺点：定位精度通常较低，尤其是在麦克风阵列较小或者信噪比较低的情况下。

   经综合考虑和仿真，决定采用 TDOA 算法。

3 理论设计与仿真

3.1 定位算法仿真

通过修改 6 次算法形式和信号预处理部分，我们基于矩形麦克风阵列的 TDOA 算法的 MATLAB 仿真实现了较高的理论精度。

仿真核心代码与结果如下：

```
                [-25, 25; 215, 265], ...
            'D', [25, 75; 215, 265], ...
            'E', [75, 125; 215, 265]);

% 随机选择一个区域内的声源位置
source_index = randi(5); % 随机选择一个声源中心
random_offset = (rand(1, 2) - 0.5) * 50; % 在该区域内随机偏移
source_position = source_positions(source_index, :) + random_offset;

% 计算每个麦克风到声源的距离
distances = sqrt(sum((mic_positions - source_position).^2, 2));

% 模拟信号传播延时
delays = distances / c; % 根据距离计算传播延迟
signals = zeros(length(t), 4);
source_signal = sin(2 * pi * 1000 * t); % 生成一个1kHz的测试信号

% 模拟各个麦克风接收到的信号
for i = 1:4
    signals(:, i) = circshift(source_signal, round(delays(i) * fs));
end

% 计算TDOA
tdoa = zeros(1, 3); % TDOA between microphone pairs
tdoa(1) = delays(2) - delays(1); % Mic2-Mic1
tdoa(2) = delays(3) - delays(1); % Mic3-Mic1
tdoa(3) = delays(4) - delays(1); % Mic4-Mic1

% 使用最小化误差的优化方法估计声源位置
objective_func = @(pos) sum((sqrt(sum((mic_positions(2:end, :) - pos).^2, 2)) - sqrt(sum((mic_positions(1, :) - pos).^2)) - tdoa' * c).^2);

% 使用fminsearch进行优化，初始位置opoint为(0, 0)
opoint = [0, 0];
estimated_position = fminsearch(objective_func, opoint);

% 获取估计的坐标
x_est = estimated_position(1);
y_est = estimated_position(2);

% 区域判定
estimated_region = 'Unknown';
for region = fieldnames(regions)'
    limits = regions.(region{1});
    if x_est >= limits(1, 1) && x_est <= limits(1, 2) && y_est >= limits(2, 1) && y_est <= limits(2, 2)
        estimated_region = region{1};
        break;
    end
end

% 显示估计结果
fprintf('真实声源位置: (%.2f, %.2f)\n', source_position(1), source_position(2));
fprintf('估计声源位置: (%.2f, %.2f)\n', x_est, y_est);
fprintf('声源所在区域: %s\n', estimated_region);
```

图 3-1 TDOA 仿真核心代码

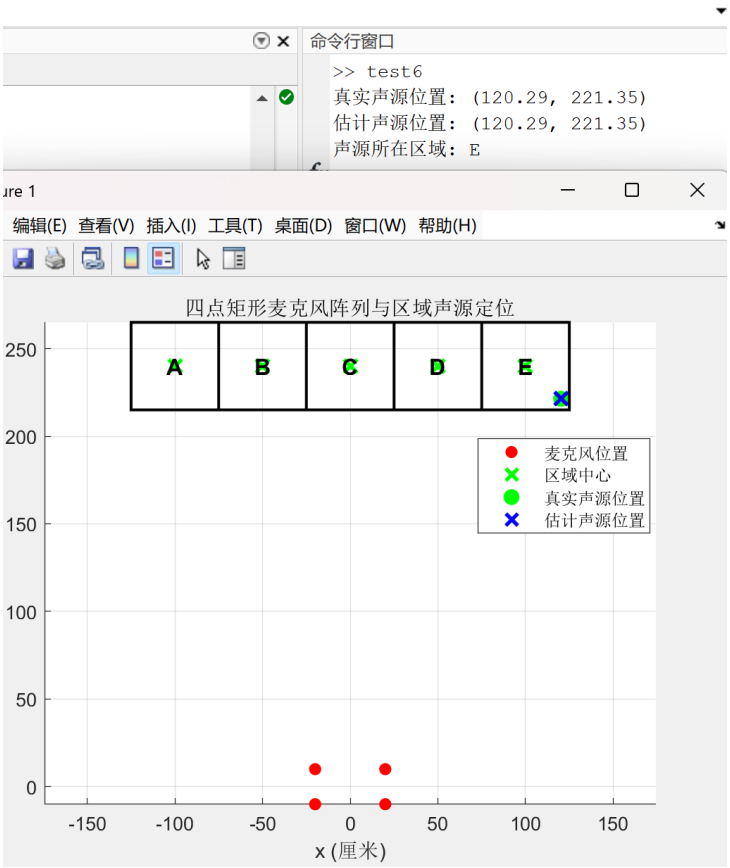


图 3-2 随机声源仿真结果 1

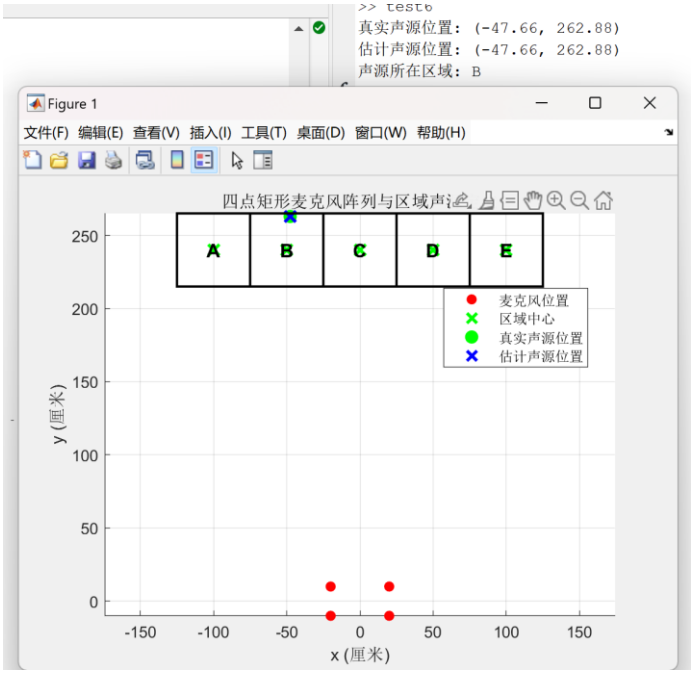


图 3-3 随机声源仿真结果 2

4 硬件设计

4.1 STM32H750VBT6 开发板

STM32H750VBT6 是一款高性能的 32 位微控制器，最高频率 400MHz，适用于高性能嵌入式应用。该开发板提供了丰富的外设接口，包括 SPI、I2C、UART 等，满足系统与外部模块的通信需求，并且拥有 3 个内部 ADC 的较多通道。选用该开发板的主要原因是其高处理能力和丰富的接口，能够满足系统中语音识别、数据采集、声源定位等复杂任务的需求。

开发板与其他模块的连接部分原理图如下：

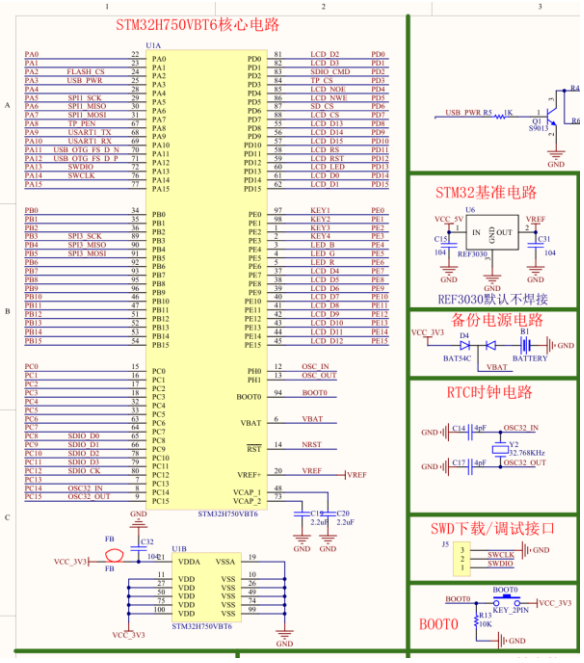


图 4-1 STM32H750VBT6 原理图

4.2 语音识别模块（ASR-PRO 核心板）

ASR-PRO 核心板采用了先进的语音识别技术，能够进行多种语音指令的识别。通过与 STM32H750VBT6 的 USART 串口通信，实时获取语音识别结果，并触发相应操作。

核心代码如下：

```
#include "asr.h"
extern "C" { void * __dso_handle = 0 ;}
#include "setup.h"
#include "myLib/asr_event.h"
#include "HardwareSerial.h"

uint32_t snid;
void ASR_CODE();

//{speak:小蝶-青新女声,vol:10,speed:10,platform:haohaodada}
//{playid:10001,voice:}
//{playid:10002,voice:}

/*描述该功能...
*/
void ASR_CODE(){
  switch (snid) {
    case 1:
     digitalWrite(4,0);
     Serial.print("1");
     break;
    case 2:
     digitalWrite(4,1);
     Serial.print("2");
     break;
    case 3:
     digitalWrite(2,0);
     Serial.print("3");
     break;
    case 4:
     digitalWrite(2,1);
     Serial.print("4");
     break;
  }

}

void hardware_init(){
  setPinFun(13,SECOND_FUNCTION);
  setPinFun(14,SECOND_FUNCTION);
  Serial.begin(9600);
  vTaskDelete(NULL);
}

void setup()
{
  set_wakeup_forever();
  //{ID:1,keyword:"命令词",ASR:"打开灯光",ASRTO:""}
  //{ID:2,keyword:"命令词",ASR:"关闭灯光",ASRTO:""}
  //{ID:3,keyword:"命令词",ASR:"打开空调",ASRTO:""}
  //{ID:4,keyword:"命令词",ASR:"关闭空调",ASRTO:""}
  setPinFun(12,FIRST_FUNCTION);
  pinMode(12,input);
  setPinFun(4,FIRST_FUNCTION);
}
```

图 4-2 ASR-PRO 核心代码

当然，为了后续定位当中可能会需要，我们通过编写 Python 代码，分析了原生命令信号和 2m 处实际接收到的信号。

分析代码和结果如下：

```python
import librosa
import numpy as np
import matplotlib.pyplot as plt


# 加载 mp3文件
audio_file = '4.mp3'  #1, 2, 3, 4分别为打开/关闭灯光，打开/关闭空调
y, sr = librosa.load(audio_file, sr=None)

# 绘制波形
plt.figure(figsize=(10, 6))
plt.plot( *args: np.linspace( start: 0, len(y) / sr, len(y)), y)
plt.title("guan bi kong tiao 2m ")
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")
plt.show()

# 计算信号的 RMS（Root Mean Square）值作为信号强度
rms = librosa.feature.rms(y=y)
print(f"RMS value: {np.mean(rms)}")
```

图 4-3 Python 信号分析代码

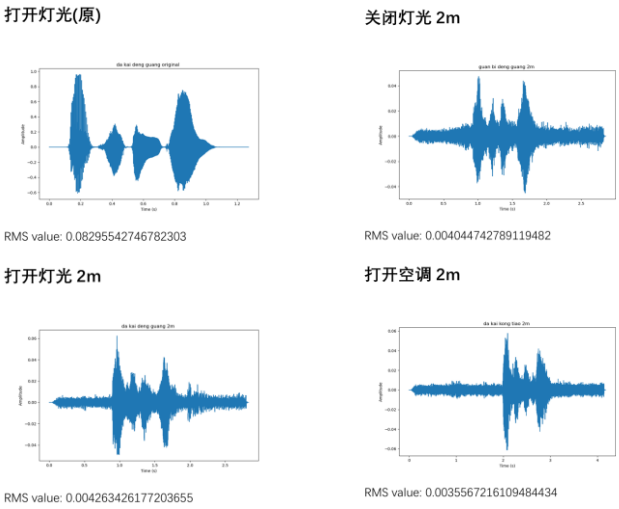


图 4-4 Python 信号分析部分结果

4.3 麦克风矩阵设计（MAX9814）

MAX9814 是一款低成本高品质的麦克风放大器，内置自动增益控制 AGC 以及低噪声麦克风偏置，可以选择 40dB, 50dB, 60Db。

麦克风的电路图与引脚分布图如下：

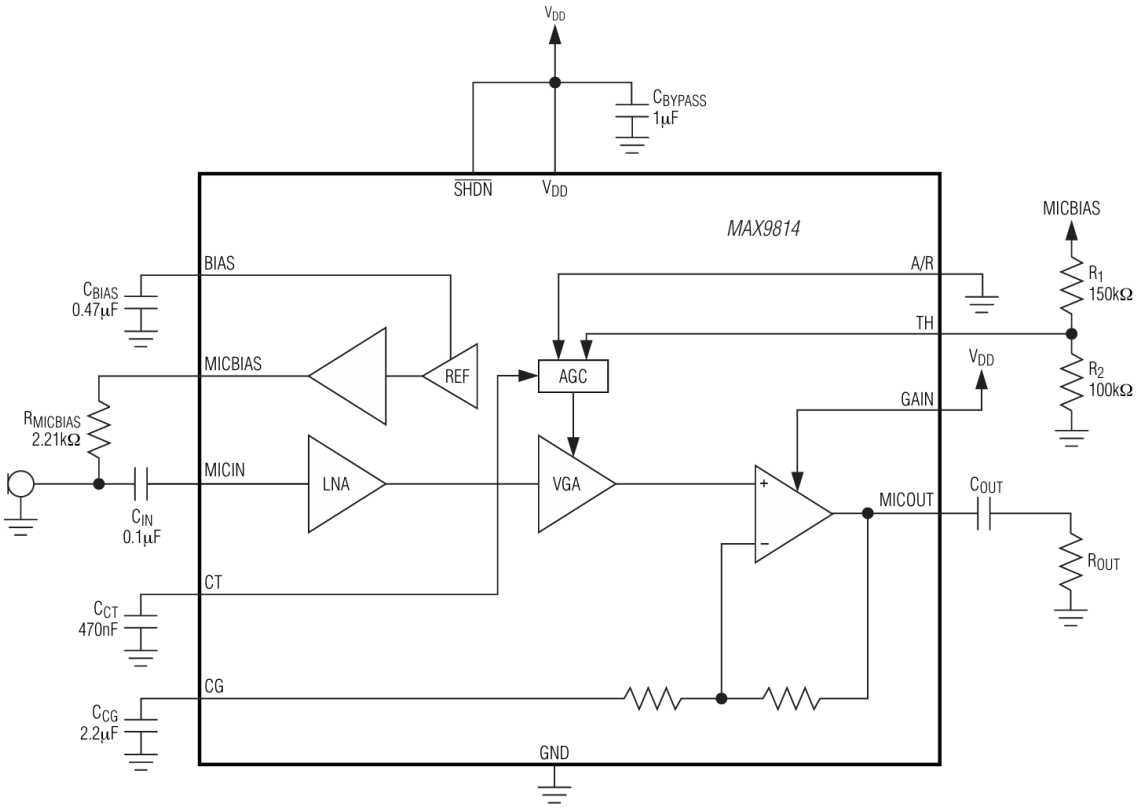


图 4-5 MAX9814 电路图

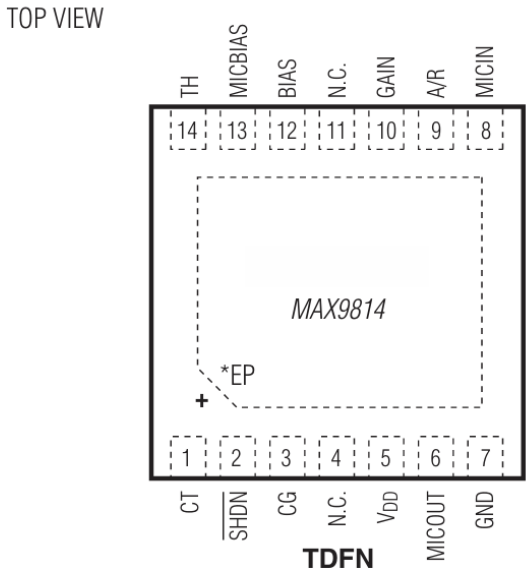


图 4-6 实际采集效果图

而麦克风矩阵由四个 MAX9814 麦克风放大电路组成，被布置了成 20x40 的矩形阵列，保证了较为均匀的声音采集效果。

麦克风阵列实物连接图如下：

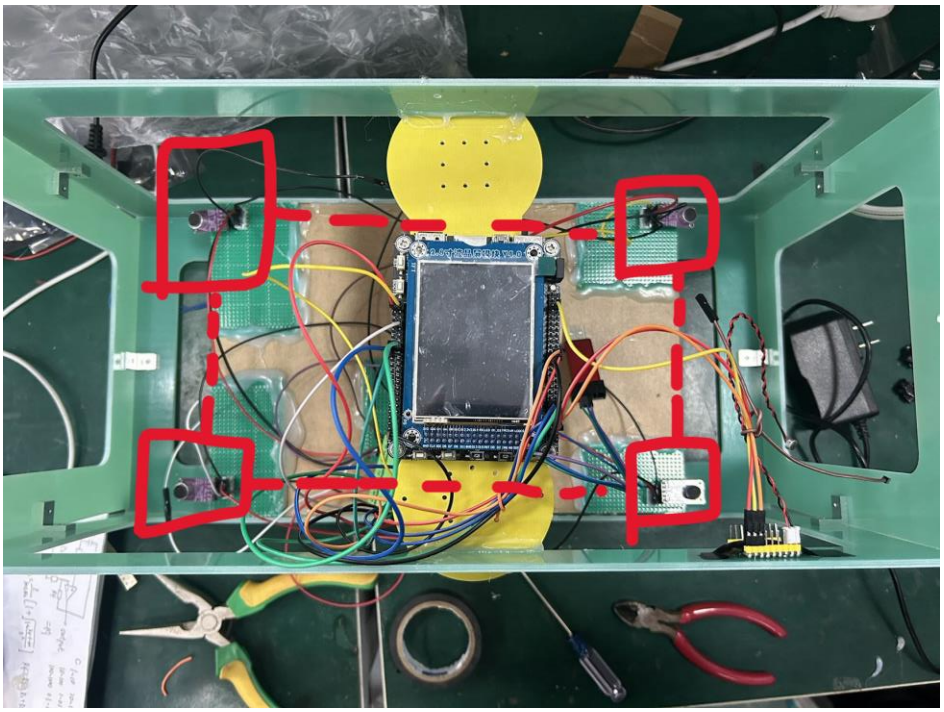


图 4-7 实际采集效果图

4.4 AD7606 数据采集模块

AD7606 是一款高精度的多通道模拟数字转换器，能够同时采集多个通道的数据。我们选择 SPI 串行接口（因为并行总线 FMC 已经被 2.8 寸 LCD 占据）与 STM32H750VBT6 进行数据传输。

AD7606 的电路图及实物图如下：

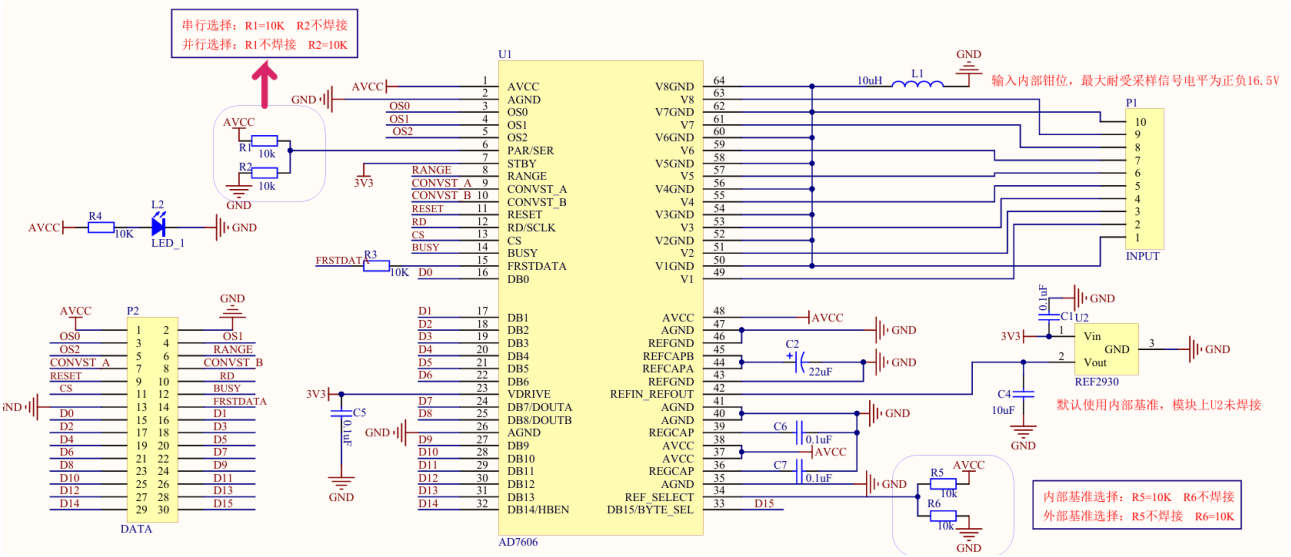


图 4-8 AD7606 的电路图

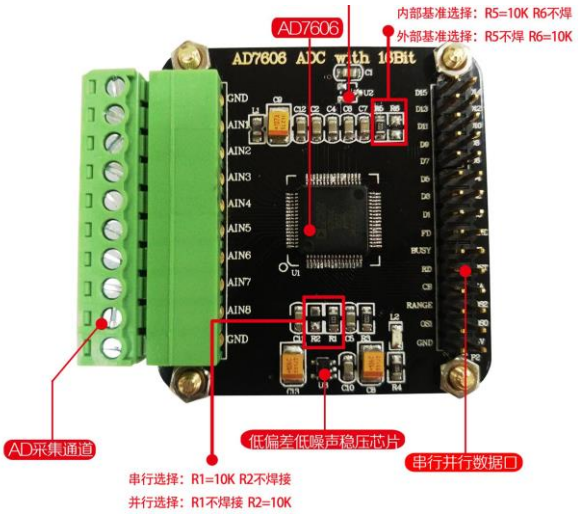


图 4-9 AD7606 实物图

5 软件设计

5.1 系统架构

系统软件架构基于 STM32H750VBT6 微控制器，开发语言是基于 KEIL5 的嵌入式 C 语言，采用模块化设计。主要包括：

1. **AD7606 采集信号：**实现对麦克风矩阵信号的实时同步采集。
2. **语音识别模块：**负责处理语音输入。
3. **声源定位模块：**基于二维 TDOA 互相关算法计算声源位置。

4. **模式切换模块：**实现四种工作模式的切换。
5. **显示模块：**通过 LCD 实时显示系统信息。

5.2 AD7606 采集

通过编写 ad7606.h 和 ad7606.c 文件半手动配置 HAL 库 SPI 以及 AD7606 的初始化函数等。

【设计思路】

1. STM32H750VBT6 通过 SPI 接口与 AD7606 进行数据交换，配置 SPI 为主模式，读取 AD7606 的采集数据用于传递给接下来的定位算法代码当中，按键按下触发模式切换，状态机进入相应状态。

AD7606 的初始化及数据读取代码示例如下：

```
void MX_SPI1_Init(void)
{
    SPI1_Handler.Instance = SPI1;
    SPI1_Handler.Init.Mode = SPI_MODE_MASTER;
    SPI1_Handler.Init.Direction = SPI_DIRECTION_2LINES;  // 全双工模式
    SPI1_Handler.Init.DataSize = SPI_DATASIZE_16BIT;
    SPI1_Handler.Init.CLKPolarity = SPI_POLARITY_HIGH;      // CPOL = 1
    SPI1_Handler.Init.CLKPhase = SPI_PHASE_2EDGE;           // CPHA = 1
    SPI1_Handler.Init.NSS = SPI_NSS_SOFT;                   // 软件控制 NSS
    SPI1_Handler.Init.BaudRatePrescaler = SPI_BAUDRATEPRESCALER_8;
    SPI1_Handler.Init.FirstBit = SPI_FIRSTBIT_MSB;
    SPI1_Handler.Init.TIMode = SPI_TIMODE_DISABLE;
    SPI1_Handler.Init.CRCCalculation = SPI_CRCCALCULATION_DISABLE;
    SPI1_Handler.Init.CRCPolynomial = 0x0;
    SPI1_Handler.Init.NSSPMode = SPI_NSS_PULSE_ENABLE;
    SPI1_Handler.Init.NSSPolarity = SPI_NSS_POLARITY_LOW;
    SPI1_Handler.Init.FifoThreshold = SPI_FIFO_THRESHOLD_01DATA;
    SPI1_Handler.Init.TxCRCInitializationPattern = SPI_CRC_INITIALIZATION_ALL_ZERO_PATTERN;
    SPI1_Handler.Init.RxCRCInitializationPattern = SPI_CRC_INITIALIZATION_ALL_ZERO_PATTERN;
    SPI1_Handler.Init.MasterSSIdleness = SPI_MASTER_SS_IDLENESS_00CYCLE;
    SPI1_Handler.Init.MasterInterDataIdleness = SPI_MASTER_INTERDATA_IDLENESS_00CYCLE;
    SPI1_Handler.Init.MasterReceiverAutoSusp = SPI_MASTER_RX_AUTOSUSP_DISABLE;
    SPI1_Handler.Init.MasterKeepIOState = SPI_MASTER_KEEP_IO_STATE_DISABLE;
    SPI1_Handler.Init.IOSwap = SPI_IO_SWAP_DISABLE;

    if (HAL_SPI_Init(&SPI1_Handler) != HAL_OK)
    {
        while(1);  // 初始化错误处理
    }
    __HAL_SPI_ENABLE(&SPI1_Handler);
}

//--------------------------------------------------------------------
// void HAL_SPI_MspInit(SPI_HandleTypeDef *hspi)
//--------------------------------------------------------------------
//
// 函数功能: SPI底层驱动，时钟使能，引脚配置
// 入口参数: SPI_HandleTypeDef *hspi: SPI句柄
// 返 回 值: 无
// 注意事项: 此函数会被HAL_SPI_Init()调用
//
//--------------------------------------------------------------------
void HAL_SPI_MspInit(SPI_HandleTypeDef* hspi)
```

图 5-1 SPI 初始化

```
//-------------------------------------------------------------------
// void GPIO_AD7606_Configuration(void)
//-------------------------------------------------------------------
//
// 函数功能: AD7606引脚配置函数
// 入口参数: 无
// 返 回 值: 无
// 注意事项: 无
//
//-------------------------------------------------------------------
void GPIO_AD7606_Configuration(void)
{
    GPIO_InitTypeDef GPIO_InitStruct = {0};

    __HAL_RCC_GPIOA_CLK_ENABLE();
    __HAL_RCC_GPIOB_CLK_ENABLE();
    __HAL_RCC_GPIOC_CLK_ENABLE();

    // 配置SPI引脚
    GPIO_InitStruct.Pin = GPIO_PIN_5|GPIO_PIN_6|GPIO_PIN_7;
    GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_MEDIUM;
    GPIO_InitStruct.Alternate = GPIO_AF5_SPI1;
    HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

    // 配置其他控制引脚
    GPIO_InitStruct.Pin = GPIO_PIN_4;  // CS 引脚
    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
    GPIO_InitStruct.Pull = GPIO_PULLDOWN;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
    HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_4, GPIO_PIN_RESET);  // 拉低CS

    // RANGE配置
    GPIO_InitStruct.Pin = GPIO_PIN_0|GPIO_PIN_1|GPIO_PIN_2;  // RANGE引脚
    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
    GPIO_InitStruct.Pull = GPIO_PULLUP; // 保证高电平
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
    HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);

    // 配置CONVST和RESET引脚
    GPIO_InitStruct.Pin = GPIO_PIN_8;  // CONVST 引脚
    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
    GPIO_InitStruct.Pull = GPIO_PULLUP;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
    HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

    GPIO_InitStruct.Pin = GPIO_PIN_9;  // RESET 引脚
    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
    GPIO_InitStruct.Pull = GPIO_PULLDOWN;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
    HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
}
```

图 5-2 AD7606 引脚配置

```
//-------------------------------------------------------------------
// void AD7606_ReadData(s16 * DB_data)
//-------------------------------------------------------------------
//
// 函数功能: 读取数据
// 入口参数: s16 * DB_data: 结构体指针，该指针为指向结构体数组的首地址
// 返 回 值: 无
// 注意事项: 无
//
//-------------------------------------------------------------------
void AD7606_ReadData(s16 * DB_data)
{
    uint16_t dummy_tx_buffer[8] = {0x0000, 0x0000, 0x0000, 0x0000,
                                   0x0000, 0x0000, 0x0000, 0x0000};
    HAL_SPI_TransmitReceive(&SPI1_Handler,
                            (uint8_t *)dummy_tx_buffer,
                            (uint8_t *)DB_data,
                            8,
                            1000);
}
```

图 5-3 AD7606 读取数据

5.2 定位算法：二维 TDOA 互相关算法

二维 TDOA 互相关算法通过计算不同麦克风阵列之间的时间差，精确定位声源的方向，相关代码文件为 localization.c 和 localization.h。

算法思想：

1. 获取从各麦克风阵列采集的音频信号。核心代码如下：

```
arm_cfft_instance_f32 fft_instance;  // FFT 实例

// 麦克风和区域定义
static Position mic_positions[4] = {
    {-20.0f, -10.0f},
    {20.0f, -10.0f},
    {-20.0f, 10.0f},
    {20.0f, 10.0f}
};

static Region regions[5] = {
    {-125.0f, -75.0f, 215.0f, 265.0f},
    {-75.0f, -25.0f, 215.0f, 265.0f},
    {-25.0f, 25.0f, 215.0f, 265.0f},
    {25.0f, 75.0f, 215.0f, 265.0f},
    {75.0f, 125.0f, 215.0f, 265.0f}
};

// FFT 变量
static arm_cfft_instance_f32 fft_instance;  // FFT 实例
static float32_t fft_input[FFT_SIZE * 2];   // 输入数据缓冲区（实部和虚部）
static float32_t magnitude[FFT_SIZE];       // 幅度数据缓冲区

// 定位结果
static LocalizationResult loc_result = {0}; // 定位结果
static bool result_ready = false;           // 定位结果是否准备好

// 信号阈值
static float signal_threshold = 1.0f;      // 信号阈值

// 设置信号阈值
void Localization_SetThreshold(float threshold) {
    signal_threshold = threshold; // 设置信号阈值
}

// 定义高斯滤波器
float gauss_filter[5] = {0.2f, 0.2f, 0.2f, 0.2f, 0.2f};

// 信号处理函数（高斯滤波）
void Signal_Process(const uint16_t* raw_data, float* processed_data) {
    // 处理信号，应用高斯滤波器
    for (int i = 0; i < CHANNEL_NUM; i++) {
        float filtered_value = 0.0f;
        for (int j = -2; j <= 2; j++) {
            int index = i + j;
            if (index >= 0 && index < CHANNEL_NUM) {
                filtered_value += raw_data[index] * gauss_filter[j + 2];
            }
        }
        processed_data[i] = filtered_value;  // 将滤波后的信号存储
    }
}
```

图 5-4 获取从各麦克风阵列采集的音频信号

2. 通过互相关算法计算时间差（TDOA）。核心代码如下：

```
// 定位计算函数（根据时差定位 TDOA）
void Calculate_Position(float* x, float* y) {
    // 计算 TDOA（时差）
    float tdoa[3] = {
        (mic_positions[1].x - mic_positions[0].x) / SPEED_OF_SOUND,
        (mic_positions[2].x - mic_positions[0].x) / SPEED_OF_SOUND,
        (mic_positions[3].x - mic_positions[0].x) / SPEED_OF_SOUND
    };

    float min_error = 1e6f;
    *x = 0.0f;
    *y = 0.0f;

    // 遍历可能的位置，找到最小误差的位置
    for (float px = -150.0f; px <= 150.0f; px += 1.0f) {
        for (float py = 200.0f; py <= 300.0f; py += 1.0f) {
            float error = 0.0f;
            for (int i = 1; i < 4; i++) {
                float dist1 = sqrtf(powf(px - mic_positions[0].x, 2) + powf(py - mic_positions[0].y, 2));
                float dist2 = sqrtf(powf(px - mic_positions[i].x, 2) + powf(py - mic_positions[i].y, 2));
                error += powf((dist2 - dist1) - tdoa[i - 1], 2);
            }
            if (error < min_error) {
                min_error = error;
                *x = px;
                *y = py;
            }
        }
    }
}
```

图 5-4 定位计算

3. 判断声源坐标和区域。核心代码如下：

```
// 区域判断函数（根据坐标判断区域）
const char* Determine_Region(float x, float y) {
    for (int i = 0; i < 5; i++) {
        if (x >= regions[i].xmin && x <= regions[i].xmax &&
            y >= regions[i].ymin && y <= regions[i].ymax) {
            switch (i) {
                case 0: return "A"; // 区域 A
                case 1: return "B"; // 区域 B
                case 2: return "C"; // 区域 C
                case 3: return "D"; // 区域 D
                case 4: return "E"; // 区域 E
                default: return "Unknown"; // 未知区域
            }
        }
    }
    return "Unknown"; // 未找到匹配的区域
}
```

图 5-5 推算声源位置

但是由于采集部分和定位部分没能衔接好，没能实现定位。

5.3 状态机设计

通过按键切换模式，系统支持四种独立运行的模式（实际只完成了两种模式）。每个模式都有独立的功能和操作流程，不同模式之间互不干扰。

【设计思路】

1. 每种模式对应一个状态机。

2. 按键按下触发模式切换，状态机进入相应状态。
3. 由 USART 中断触发每“识别模式”当中的结果显示。

状态机的核心代码及模式切换效果图如下：

```
// 更新模式函数
void UpdateMode(void)
{
    u8 key = KEY_Scan(0); // 参数为0，不支持连按

    switch (key)
    {
        case KEY3_PRES:
            currentMode = MODE_K3;
            needUpdateDisplay = 1;
            break;

        case KEY4_PRES:
            currentMode = MODE_K4;
            needUpdateDisplay = 1;

            messageDisplayFlag = 1;
            messageStartTime = HAL_GetTick(); // 记录开始时间
            break;

        case KEY1_PRES:
            currentMode = MODE_K1;
            needUpdateDisplay = 1;
            break;

        case KEY2_PRES:
            // 如果需要处理 K4，可以在这里添加
            break;

        default:

            break;
    }
}
```

图 5-6 按键对应配置模式

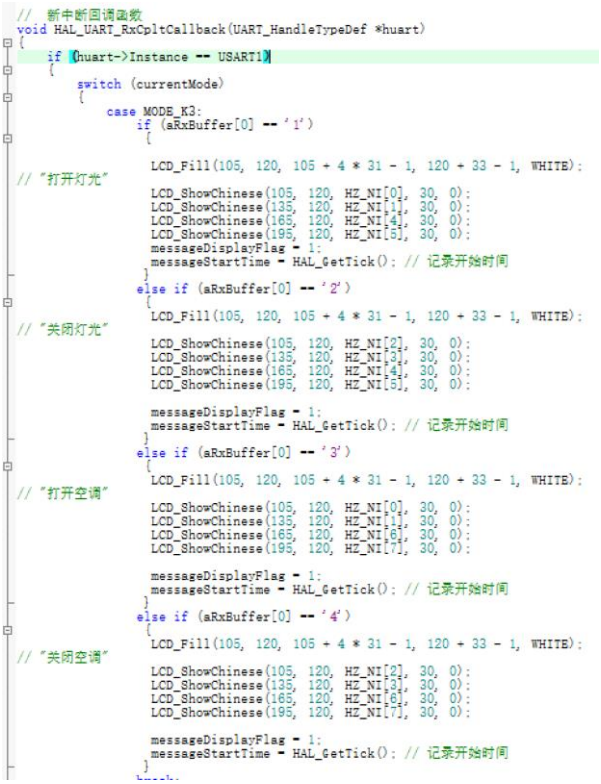
```
// 更新显示函数
void UpdateDisplay(void)
{
    if (needUpdateDisplay)
    {
        switch (currentMode)
        {

            case MODE_NONE:
// "请选择模式"
                LCD_Fill(90, 35, 90 + 5 * 31 - 1, 35 + 33 - 1, YELLOW);
                LCD_ShowChinese(90, 35, HZ_NI[8], 30, 0);
                LCD_ShowChinese(120, 35, HZ_NI[9], 30, 0);
                LCD_ShowChinese(150, 35, HZ_NI[10], 30, 0);
                LCD_ShowChinese(180, 35, HZ_NI[11], 30, 0);
                LCD_ShowChinese(210, 35, HZ_NI[12], 30, 0);
// "识别"
                LCD_Fill(0, 100, 160 - 1, 170 - 1, GREEN);
                LCD_ShowChinese(45, 120, HZ_NI[13], 30, 0);
                LCD_ShowChinese(75, 120, HZ_NI[14], 30, 0);
// "粗定位"
                LCD_Fill(160, 100, 320 - 1, 170 - 1, BRED);
                LCD_ShowChinese(200, 120, HZ_NI[15], 30, 0);
                LCD_ShowChinese(230, 120, HZ_NI[17], 30, 0);
                LCD_ShowChinese(260, 120, HZ_NI[18], 30, 0);
// "精定位"
                LCD_Fill(0, 170, 160 - 1, 240 - 1, GRAYBLUE);
                LCD_ShowChinese(35, 190, HZ_NI[16], 30, 0);
                LCD_ShowChinese(65, 190, HZ_NI[17], 30, 0);
                LCD_ShowChinese(95, 190, HZ_NI[18], 30, 0);
// "定位识别"
                LCD_Fill(160, 170, 320 - 1, 240 - 1, CYAN);
                LCD_ShowChinese(190, 190, HZ_NI[17], 30, 0);
                LCD_ShowChinese(210, 190, HZ_NI[18], 30, 0);
                LCD_ShowChinese(240, 190, HZ_NI[13], 30, 0);
                LCD_ShowChinese(270, 190, HZ_NI[14], 30, 0);
                  break;

              case MODE_K3:
                LCD_Fill(90, 35, 90 + 5 * 31 - 1, 35 + 33 - 1, WHITE);
                LCD_Fill(0, 100, 320, 240, WHITE);
                LCD_Fill(105, 35, 105 + 4 * 31 - 1, 35 + 33 - 1, GREEN);
// "识别模式"
                LCD_ShowChinese(105, 35, HZ_NI[13], 30, 0);
                LCD_ShowChinese(135, 35, HZ_NI[14], 30, 0);
                LCD_ShowChinese(165, 35, HZ_NI[11], 30, 0);
                LCD_ShowChinese(195, 35, HZ_NI[12], 30, 0);
// "等待命令"
                LCD_ShowChinese(105, 120, HZ_NI[19], 30, 0);
                LCD_ShowChinese(135, 120, HZ_NI[20], 30, 0);
                LCD_ShowChinese(165, 120, HZ_NI[21], 30, 0);
                LCD_ShowChinese(195, 120, HZ_NI[22], 30, 0);
                  break;

              case MODE_K4:
                LCD_Fill(90, 35, 90 + 5 * 31 - 1, 35 + 33 - 1, WHITE);
                LCD_Fill(0, 100, 320, 240, WHITE);
                LCD_Fill(90, 35, 90 + 5 * 31 - 1, 35 + 33 - 1, BRED);
// "粗定位模式"
                LCD_ShowChinese(90, 35, HZ_NI[18], 30, 0);
                LCD_ShowChinese(120, 35, HZ_NI[17], 30, 0);
                LCD_ShowChinese(150, 35, HZ_NI[15], 30, 0);
                LCD_ShowChinese(180, 35, HZ_NI[11], 30, 0);
                LCD_ShowChinese(210, 35, HZ_NI[12], 30, 0);
// "等待命令"
                LCD_ShowChinese(105, 120, HZ_NI[19], 30, 0);
                LCD_ShowChinese(135, 120, HZ_NI[20], 30, 0);
                LCD_ShowChinese(165, 120, HZ_NI[21], 30, 0);
                LCD_ShowChinese(195, 120, HZ_NI[22], 30, 0);
                  break;
```

图 5-7 状态机更新



```
// 新中断回调函数
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
    if (huart->Instance == USART1)
    {
        switch (currentMode)
        {
            case MODE_K3:
                if (aRxBuffer[0] == '1')
                {

                    LCD_Fill(105, 120, 105 + 4 * 31 - 1, 120 + 33 - 1, WHITE);
// "打开灯光"
                    LCD_ShowChinese(105, 120, HZ_NI[0], 30, 0);
                    LCD_ShowChinese(135, 120, HZ_NI[1], 30, 0);
                    LCD_ShowChinese(165, 120, HZ_NI[4], 30, 0);
                    LCD_ShowChinese(195, 120, HZ_NI[5], 30, 0);
                    messageDisplayFlag = 1;
                    messageStartTime = HAL_GetTick(); // 记录开始时间
                }
                else if (aRxBuffer[0] == '2')
                {
                    LCD_Fill(105, 120, 105 + 4 * 31 - 1, 120 + 33 - 1, WHITE);
// "关闭灯光"
                    LCD_ShowChinese(105, 120, HZ_NI[2], 30, 0);
                    LCD_ShowChinese(135, 120, HZ_NI[3], 30, 0);
                    LCD_ShowChinese(165, 120, HZ_NI[4], 30, 0);
                    LCD_ShowChinese(195, 120, HZ_NI[5], 30, 0);

                    messageDisplayFlag = 1;
                    messageStartTime = HAL_GetTick(); // 记录开始时间
                }
                else if (aRxBuffer[0] == '3')
                {
                    LCD_Fill(105, 120, 105 + 4 * 31 - 1, 120 + 33 - 1, WHITE);
// "打开空调"
                    LCD_ShowChinese(105, 120, HZ_NI[0], 30, 0);
                    LCD_ShowChinese(135, 120, HZ_NI[1], 30, 0);
                    LCD_ShowChinese(165, 120, HZ_NI[6], 30, 0);
                    LCD_ShowChinese(195, 120, HZ_NI[7], 30, 0);

                    messageDisplayFlag = 1;
                    messageStartTime = HAL_GetTick(); // 记录开始时间
                }
                else if (aRxBuffer[0] == '4')
                {
                    LCD_Fill(105, 120, 105 + 4 * 31 - 1, 120 + 33 - 1, WHITE);
// "关闭空调"
                    LCD_ShowChinese(105, 120, HZ_NI[2], 30, 0);
                    LCD_ShowChinese(135, 120, HZ_NI[3], 30, 0);
                    LCD_ShowChinese(165, 120, HZ_NI[6], 30, 0);
                    LCD_ShowChinese(195, 120, HZ_NI[7], 30, 0);

                    messageDisplayFlag = 1;
                    messageStartTime = HAL_GetTick(); // 记录开始时间
                }
                break;
```

图 5-8 USART 中断

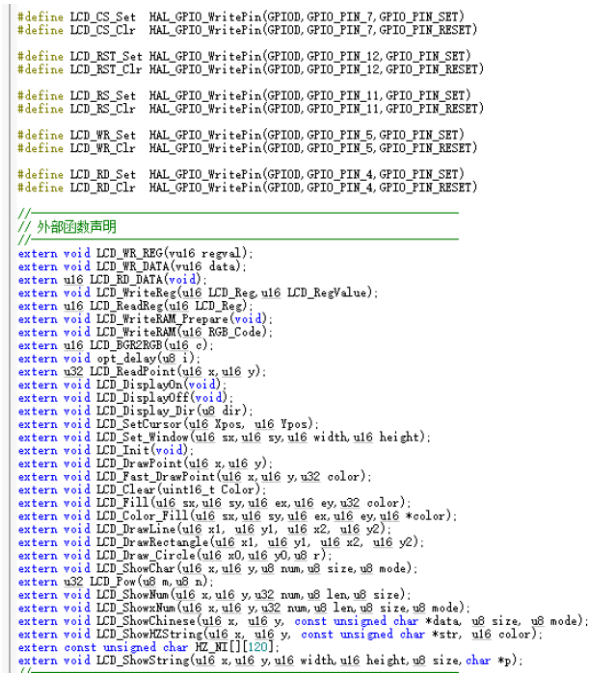
5.4 LCD 显示菜单设计

LCD 显示模块通过 FMC 与 STM32H750VBT6 连接，实时显示系统状态和模式信息。相关代码在 lcd.c 和 lcd.h。

设计思路：

1. LCD 显示系统菜单、当前模式、识别结果等。
2. 显示内容随模式切换而变化。

LCD 部分代码如下：



```
#define LCD_CS_Set  HAL_GPIO_WritePin(GPIOD,GPIO_PIN_7,GPIO_PIN_SET)
#define LCD_CS_Clr  HAL_GPIO_WritePin(GPIOD,GPIO_PIN_7,GPIO_PIN_RESET)

#define LCD_RST_Set HAL_GPIO_WritePin(GPIOD,GPIO_PIN_12,GPIO_PIN_SET)
#define LCD_RST_Clr HAL_GPIO_WritePin(GPIOD,GPIO_PIN_12,GPIO_PIN_RESET)

#define LCD_RS_Set  HAL_GPIO_WritePin(GPIOD,GPIO_PIN_11,GPIO_PIN_SET)
#define LCD_RS_Clr  HAL_GPIO_WritePin(GPIOD,GPIO_PIN_11,GPIO_PIN_RESET)

#define LCD_WR_Set  HAL_GPIO_WritePin(GPIOD,GPIO_PIN_5,GPIO_PIN_SET)
#define LCD_WR_Clr  HAL_GPIO_WritePin(GPIOD,GPIO_PIN_5,GPIO_PIN_RESET)

#define LCD_RD_Set  HAL_GPIO_WritePin(GPIOD,GPIO_PIN_4,GPIO_PIN_SET)
#define LCD_RD_Clr  HAL_GPIO_WritePin(GPIOD,GPIO_PIN_4,GPIO_PIN_RESET)

//-----------------------------------------------------------
// 外部函数声明
//-----------------------------------------------------------
extern void LCD_WR_REG(vu16 regval);
extern void LCD_WR_DATA(vu16 data);
extern u16 LCD_RD_DATA(void);
extern void LCD_WriteReg(u16 LCD_Reg,u16 LCD_RegValue);
extern u16 LCD_ReadReg(u16 LCD_Reg);
extern void LCD_WriteRAM_Prepare(void);
extern void LCD_WriteRAM(u16 RGB_Code);
extern u16 LCD_BGR2RGB(u16 c);
extern void opt_delay(u8 i);
extern u32 LCD_ReadPoint(u16 x,u16 y);
extern void LCD_DisplayOn(void);
extern void LCD_DisplayOff(void);
extern void LCD_Display_Dir(u8 dir);
extern void LCD_SetCursor(u16 Xpos, u16 Ypos);
extern void LCD_Set_Window(u16 sx,u16 sy,u16 width,u16 height);
extern void LCD_Init(void);
extern void LCD_DrawPoint(u16 x,u16 y);
extern void LCD_Fast_DrawPoint(u16 x,u16 y,u32 color);
extern void LCD_Clear(uint16_t Color);
extern void LCD_Fill(u16 sx,u16 sy,u16 ex,u16 ey,u32 color);
extern void LCD_Color_Fill(u16 sx,u16 sy,u16 ex,u16 ey,u16 *color);
extern void LCD_DrawLine(u16 x1, u16 y1, u16 x2, u16 y2);
extern void LCD_DrawRectangle(u16 x1, u16 y1, u16 x2, u16 y2);
extern void LCD_Draw_Circle(u16 x0,u16 y0,u8 r);
extern void LCD_ShowChar(u16 x,u16 y,u8 num,u8 size,u8 mode);
extern u32 LCD_Pow(u8 m,u8 n);
extern void LCD_ShowNum(u16 x,u16 y,u32 num,u8 len,u8 size);
extern void LCD_ShowxNum(u16 x,u16 y,u32 num,u8 len,u8 size,u8 mode);
extern void LCD_ShowChinese(u16 x, u16 y, const unsigned char *data, u8 size, u8 mode);
extern void LCD_ShowHZString(u16 x, u16 y, const unsigned char *str, u16 color);
extern const unsigned char HZ_NI[][120];
extern void LCD_ShowString(u16 x,u16 y,u16 width,u16 height,u8 size,char *p);
//-----------------------------------------------------------
```

图 5-9 LCD 引脚配置和一些函数

5.5 主函数

main.c 的 int main (void)当中包括初始化和一些开启函数以及 while（1）

设计思路：

1. 使用 CubeMX 配置大部分外设和引脚的初始化代码。
2. 由于使用状态机思路，while（1）的短短三个函数的调用便足矣。

main.c 部分代码如下：

```c
int main(void)
{
  /* USER CODE BEGIN 1 */
  HAL_Init();                   // 初始化HAL库
  /* USER CODE END 1 */

  /* MPU Configuration--------------------------------------------------------*/
  MPU_Config();

  /* MCU Configuration--------------------------------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();

  /* USER CODE BEGIN Init */
  MPU_Memory_Protection();  // MPU内存保护

  /* USER CODE END Init */

  /* Configure the system clock */
  SystemClock_Config();

  /* USER CODE BEGIN SysInit */
  SysTick_clkconfig(400);   // SysTick参数配置
  /* USER CODE END SysInit */

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_DMA_Init();
  MX_USART1_UART_Init();
  MX_FMC_Init();
  MX_TIM2_Init();
  MX_SPI1_Init();
  /* USER CODE BEGIN 2 */
  //代码开始区
  //Init区
  KEY_Init();
  LCD_Init();               // LCD初始化
  MX_SPI1_Init();

  //LCD
  BACK_COLOR=GREEN;         // 背景色
  POINT_COLOR=BLACK;        // 笔画颜色

  // 初始化 AD7606 和定位模块
  AD7606_Init();
  AD7606_StartConvst();     // 启动转换
  AD7606_Delay(1);          // 等待转换启动
  Localization_SetThreshold(2.0f); // 根据实际需要调整阈值

  //启用进程或中断
  HAL_UART_Receive_IT(&huart1, aRxBuffer, 1);   // 启动 UART 接收中断
```

图 5-10 int main （void）中的代码

```c
  /* USER CODE BEGIN WHILE */


  //代码开始区
  while(1){

    UpdateMode();
    UpdateDisplay();
    CheckMessageTimeout();


  }                          // while end

  //代码结束区


    /* USER CODE END WHILE */
```

图 5-11 while（1）

6 系统实现与测试

6.1 调试每个外设是否接通

已经通过视频形式记录，所有外设均接通并正常工作。

6.2 效果展示

系统效果图如下：

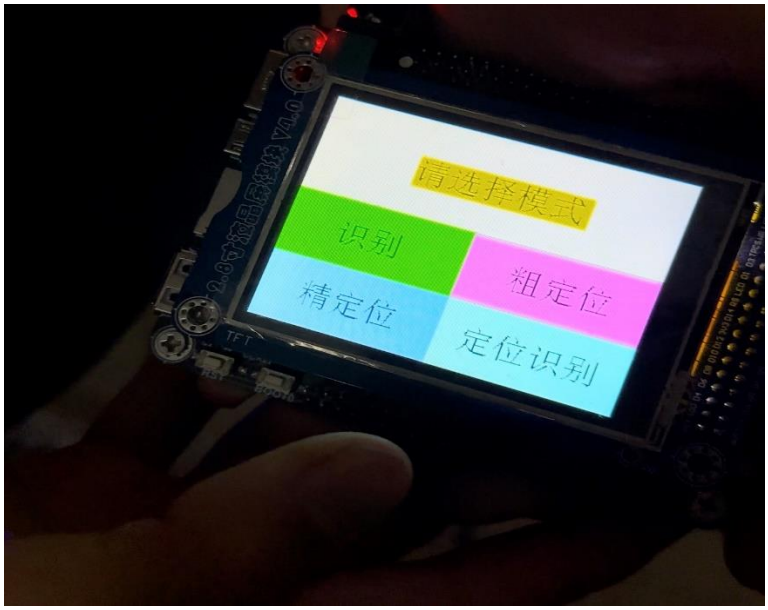


图 6-2 主菜单

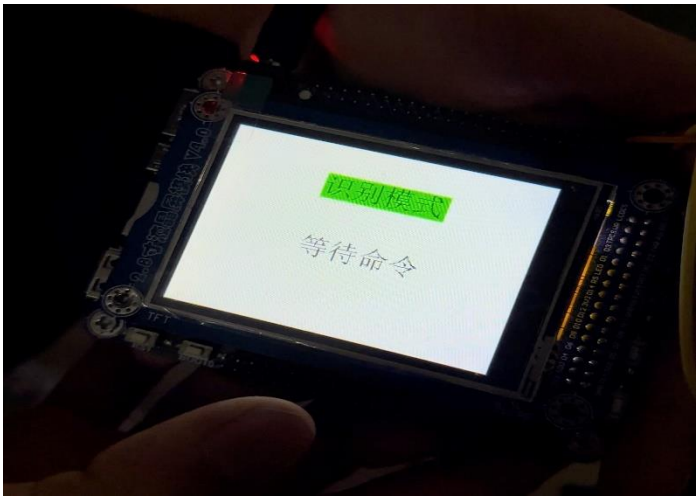


图 6-2 识别模式下等待命令

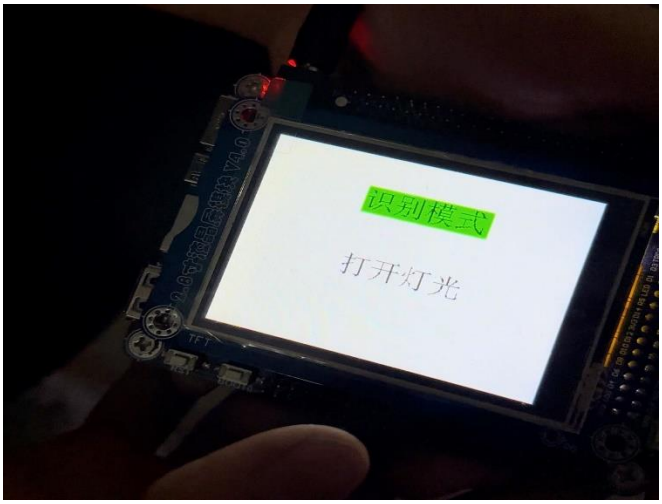


图 6-3 识别到“打开灯光”

很遗憾定位模式代码有点问题，导致功能没能顺利实现。

6.3 实物图

实物图如下：

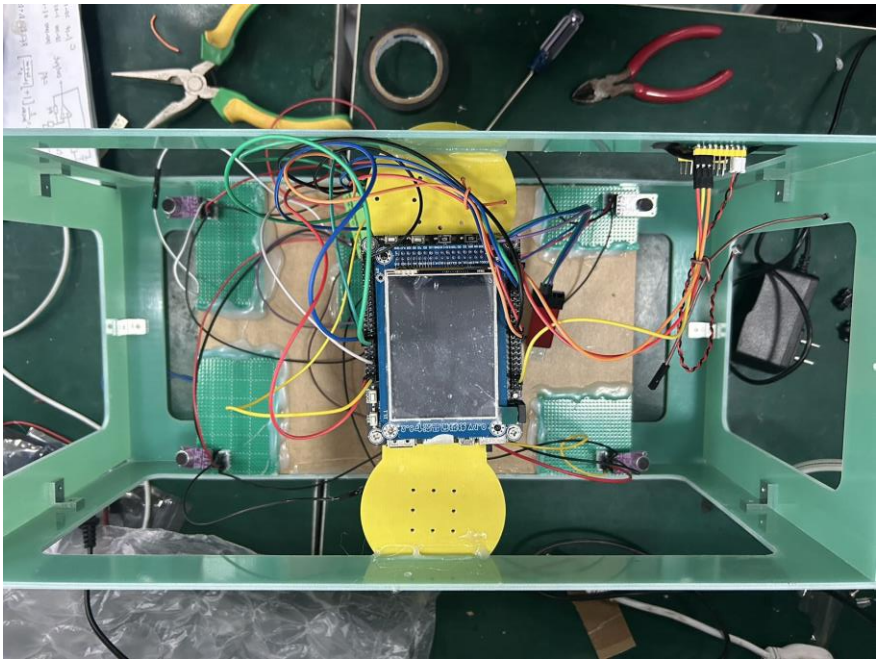


图 6-4 实物图

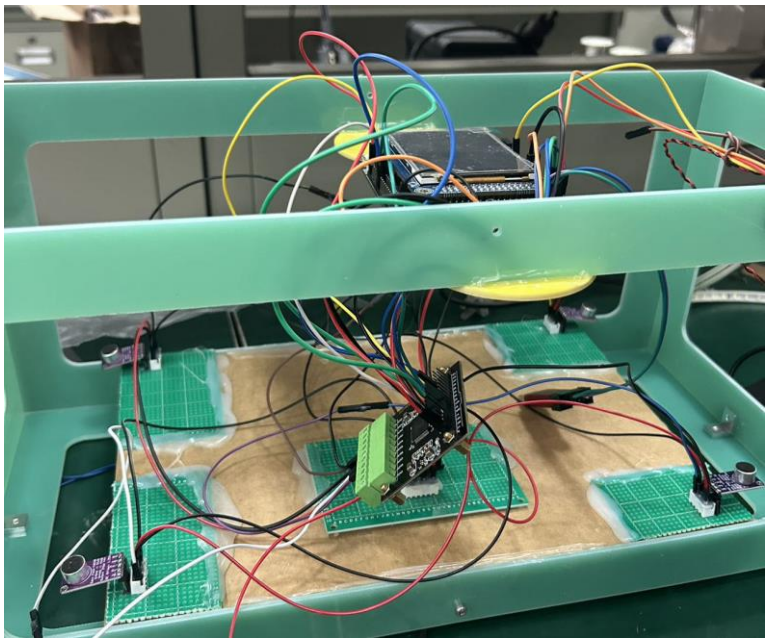


图 6-5 实物图

其中，地下木板上中间的洞洞板为四个麦克风电路取共电和共地模块，四个顶角上的洞洞板为每个麦克风与别的部分接线的模块；最外层支架采用玻纤板。

7 总结与展望

我们只完成了基本任务中的“识别模式”，没能完成“粗精定位”模式。这是我们三位同学第一次尝试电子设计类的竞赛项目，由于基础不够支撑我们完成每一个步骤，我们只完成了整个硬件部分和大部分软件部分，目前只有粗（精）定位模式的代码有点问题，但是工程文件中整体的逻辑和框架是有的。我们经历了一个多月以来几乎从零开始学习有一定工程量的嵌入式工程，无数次熬夜终究换来成功完赛的结果，我们没有因为不会某个部分而放弃，而是把能投入的时间全部都投入了 D 题当中，在有限的时间里边学习边进行项目是很刺激的过程，我们遇到的困难主要是因为对 ADC，定时器以及 DMA 没能完全掌握，导致没能成功通过相应中断将采集到的数据引入到定位算法当中；此外，尽管我们已经采用了模块化编程和逐步集成的方法，即对每个模块单独进行测试和调试并等到确保无误后再进行整体集

成，但还是有标准库和 HAL 库混用，CubeMX 和无 ioc 文件的例程混用等情况出现，这导致了我们在代码调试方面的困难。我们希望能在假期有充足的时间来继续完成这次没能成功完成的部分，给自己的努力一个满意的答卷。感谢电子科协及会长，感谢指导老师和盟升公司。

8 主要参考文献

[1] 《STM32H750VBT6 开发手册》
[2] 《ASR-PRO 核心板技术手册》
[3] 《AD7606 数据手册》

等