****

**Embedded Systems and microcontrollers - Project**

Submitted to: Dr. Bogaraju Sharatchandra varma

Ulster University, Belfast Campus

s.bogaraju @ulster.ac.uk

|  |  |
| --- | --- |
| B Code | Name |
| B00896175 | James Hardaker |
| B00781527 | Stephen McAteer |

**Table of Contents**

Concept, Overview …………….……….…………………........................3

Audio Source, Audio Output, User Interface……………………….….4

Guitar Input Circuit Design……………………………………………………..5

LTSpice Offset Amplifier Circuit Design…………………………………..6

Input Circuit Implementation…………………………………………………7

Arduino Nano ESP32 ADC sampling implementation…………….8-9

Bluetooth, Wi-Fi or SPI Radio communication……………………….10

**Concept**

**Wireless guitar system**

**A diagram of a device connected to a device

Description automatically generated**

**Overview**

* Process ‘A’ – Audio Source
  + Takes audio signal from guitar via ADC and sends it over Bluetooth to a phone/PC and to a receiving Bluetooth audio device.
* Process ‘B’ – Audio Output
  + Receives the audio signal from the Bluetooth audio source and sends it to an audio jack output.
* Process ‘C’ – Phone/PC User Interface
  + Advanced audio functions and user feedback.

**Audio Source**

* Guitar pickups generate small voltage signal which is sent along the 3.5 mm audio cable connected to circuit.
* Circuit cleans (band pass filter) and amplifies signal to 0-5V range which is output to Arduino Analog input.
* Arduino Nano ESP32 reads as 12 bit at 32 kHZ
* Arduino Nano ESP32 writes data to A2DP sink – paired with other ESP32 (Audio output).

https://en.wikipedia.org/wiki/List\_of\_Bluetooth\_profiles#Advanced\_Audio\_Distribution\_Profile\_(A2DP)

* It should be possible to also pair with a mobile phone or PC at the same time and send the data there.

**Audio Output**

* Connect to Bluetooth Audio source and use the ESP32 as a Bluetooth audio sink.
* Using WM8960 audio codec module set as an I2S peripheral, the ESP32 will receive audio and play it back via I2S.
* Once the WM8960 receives the I2S audio, it will be sent to the DAC and then to the output to the amplifier.

**Phone/PC User Interface**

* .NET MAUI cross platform app receives audio data.
* Stetch objectives
  + Note detection (Stretch objective)
    - Using Fast Fourier Transform (FFT) split the time domain signal into frequency bins.
    - By analysing the most common frequencies we can determine which note has been played and display this on the screen.
    - Guitar tuning function.
  + By sampling the audio signal and sending a stream to Shazam API ‘songs/v2/detect’ we can know which song is being played.
  + Apply ‘Effects’ on sound – e.g. echo.

**Guitar Input Circuit Design**

The first stage of the system must be to acquire the guitar signal and feed it into the source devices ADC pin.

Initially we analyse the raw signal coming from the guitar. Below is a capture from the oscilloscope after strumming all 6 strings of the guitar with average force.

Testing shows that with the volume knob turned up to 10 we can reach voltages of around 1V in amplitude. The maximum frequency (not counting noise) that a guitar can output is 659 Hz on the high E string of the 12th fret.

The ESP32 ADC can only accept a voltage range of 0-3.3V with 12 bits of precision. This means we have to offset the voltage and amplify it to fill out the full available range. 12 bits converts to 4096 different levels. We want to take advantage of as much of the range as possible so we will amplify the signal upto that range.

We will also use a diode to prevent any negative voltage from seeping into our ADC input.

After discussion and research we decided to use an LT1167 offset amplifier.

<https://www.analog.com/media/en/technical-documentation/data-sheets/1167fc.pdf>

**LTSpice Offset Amplifier Circuit Design**

A computer screen shot of a circuit board

Description automatically generated

Below is the Voltage drop across the ADC resistor for an input of 300mv (V1).

A screen shot of a computer screen

Description automatically generated

We will see later that this circuit does not line up exactly with reality and we have had to adjust the resistor values until we are happy with the output in practice.

**Input Circuit Implementation**

<images here>

**Arduino Nano ESP32 ADC sampling implementation**

We decided to use the Arduino Nano ESP32 because it has 2 cores both clocked at 240 MHz, Bluetooth, Wi-Fi, 80 MHz timers and a 12-bit ADC. This allows for lots of choice when developing our system.

There are 2 main ways to sample an ADC input.

1. Individual samples taken directly in user code e.g. **AnalogRead()**
2. Continuous samples take via DMA Controller copying values from ADC pin sensor at a Frequency F into a memory location.

Generally, option 2 would be better for our purposes because it is suited to high frequency sampling, however, we encountered various issues setting this up. Alternatively we used a continuous sample using the platform specific function provided on ESP32 – **adc1\_get\_raw()**, which is faster than **AnalogRead()**.

To continuously sample using **adc1\_get\_raw()** at a rate of 32k samples per second (sps) we need to use a timer interrupt to alert our code to make the sample.

adcTimer = timerBegin(3, 2499, true); //80,000,000 ÷ 2500 = 32000

    timerAttachInterrupt(adcTimer, &onTimer32k, true);

    timerAlarmWrite(adcTimer, 1, true);

    timerAlarmEnable(adcTimer);

onTimer32k is a function stored in the instruction ram (IRAM).

With the function **onTimer32K()** we call **adc1\_get\_raw()** and place the 12 bit sample in a buffer of type array[n]. After n samples we must send the data to our output. Sending data to the output requires a slice of the processor time. We cannot use the processor that is doing the sampling because otherwise it would not have time to do the samples continuously. To solve this problem we use the 2nd processor core which runs a wait and execute function – it waits until the buffer is full and then outputs it appropriately.

    xTaskCreatePinnedToCore(onBufferFull,"Buffer Full Task",8192,NULL, 1,&bufferFull…,0);

In order to signal the 2nd core that the buffer is full we use a semaphore.

**Data packing**

Microcontrollers generally pass data in sizes of multiple of 8bits (byte). This means that our 12 bit sample will take up 2 bytes of data to store itself completely. This is 4 bits of wasted space. However, it is possible to store 2 samples inside 24 bits which is 3 bytes and 0 wasted bits. This will save us 25% on the amount of data that has to be sent over the air.

|  |  |  |
| --- | --- | --- |
| BYTE | Nibble (Upper) | Nibble (Lower) |
| 01 | 1 | 5 |
| 02 | F | 2 |
| 03 | E | 3 |
| 04 | D | E |
| 05 | F | 1 |
| 06 | 3 | A |

4 samples

**Double Buffering**

To keep the data sampling running continuously at the same time as writing the data to the output we must provide two separate buffers.

1. Buffer1 is full
2. Swap Buffer1 with Buffer2
3. Write out Buffer1. Store new samples in Buffer2
4. Buffer2 is full
5. Swap Buffer2 with Buffer1
6. Write out Buffer2. Store new samples in Buffer1
7. Goto 1.

**Filtering Noise**

The ESP32 ADC is quite noisy. To reduce the noise, we have the option of averaging the samples at the expense of clarity. Sample[N] = Sample[N] / 2 + Sample[N-1] / 2. We will see if this is a useful system to filter noise later. We may use an analogue filter.

Below is an analysis of the noise and filtering on a zero flat signal

A graph with red and green lines

Description automatically generated

The full code for the ESP32 ADC is provided in appendix item **1.a. ADC Sample code**

**Bluetooth, Wi-Fi or SPI Radio communication**

**Bluetooth**

As per the original vision we had imagined using Bluetooth A2DP protocol to communicate between the various devices. Bluetooth was investigated as an option and several drawbacks were discovered:

* At least 90 mA of current drawn
* Latency issues reported by other applications. Low latency BT devices on amazon are quite expensive.
* Crowded radio frequency
* Not all Bluetooth devices support A2DP.
* Windows operating systems do not properly implement the A2DP protocol and are not compatible as a A2DP sink.

**UDP over Wi-Fi**

As an alternative to Bluetooth, we decided to pivot to UDP Multicast over Wi-Fi (Access Point / Station mode).

This was implemented but we also discovered a few drawbacks.

* Access Point mode draws a lot of current. Station Mode is less but still more than we would like.
* Multicast in Access Point mode causes more packet loss/delay per additional device. With a single device packet loss is acceptable
* We used bare driver similar to these examples <https://github.com/espressif/esp-idf/tree/v5.2.1/examples/protocols/sockets> to achieve the best performance but found the receiving client performance was very bad. We could only achieve packets as small as 500 samples. This was connected but there was a noticeable delay in the speaker output compared to the input from the user.
* Crowded radio frequency

**Serial Radio Communication @ 2.524 GHz**

In the end we opted to use an nrf24l01+ device which connects to the Arduino via the Serial Peripheral Interface (SPI). This device requires low power, has good range and we can select a frequency above 2.5 GHz which is not crowded. A critical downside to this approach is that it is only compatible with other radio devices which operate on the same protocol. Most devices are not compatible with this radio protocol which prevents us from including the PC/Mobile phone user interface as a member of the wireless communication network.

**Appendix**

**1.a. ADC Sample code**

#include <driver/adc.h>

#include "adc\_audio.h"

namespace Audio::ADC

{

  //scheduling

  TaskHandle\_t bufferFullTaskHandler;

  hw\_timer\_t \* adcTimer = NULL;

  SemaphoreHandle\_t  semaOnBufferFull;

  //buffering

  void (\*onBufferFullEvent)(void\* data, int length);

  uint8\_t\* SecondaryBuffer; //this is the ptr to the buffer that we will fill up

  uint8\_t\* PrimaryBuffer;   //this is the ptr to the buffer that we will pass through 'onBufferFullEvent'

  void onBufferFull(void \*params)

  {

    while(true)

    {

      xSemaphoreTake( semaOnBufferFull,10000000 );

      onBufferFullEvent(PrimaryBuffer,32);

    }

  }

  //runs 32k times per second. Is put into the

  //place in Instructrum RAM for better performance

  //https://docs.espressif.com/projects/esp-idf/en/stable/esp32s2/api-guides/memory-types.html

  void IRAM\_ATTR onTimer32k()

  {

    static int16\_t sampleCount = 0;  //runs from 0 -> Shared::PACKED\_SAMPLES\_PER\_PACKET repeated

    static uint16\_t xn1=0;

    //each sample is 12 bits. The smallest usable value in c is a byte or a short (2 bytes).

    //Therefore if we want to maximize the space when transmitting data we want to pack these 12 bits into 3 bytes (2\*12bits = 3 bytes)

    //we can divide up the samples into odd and even. The even samples will be split into 8/4 bits with the first 8 bits placed in the 1st byte

    //the last 4 bits will be placed in the upper 1/2 of the nexrt byte. The odd samples will also be split into 8/4 with the data placed in the remaining space

    //as below

    // FF

    // FE

    // EE

    uint16\_t bufPos =  (sampleCount \* 3) >> 1;

    uint16\_t xn = adc1\_get\_raw(ADC1\_CHANNEL\_0);

    uint16\_t value =xn;// (xn1>>1) + (xn>>1);

    xn1 =xn;

    if(sampleCount %2 == 0)

    {

      SecondaryBuffer[bufPos] = (uint8\_t)value;//store the first byte

      SecondaryBuffer[bufPos+1] =(uint8\_t)((value & 0x0F00) >> 4);//store the remaining 4 bits in the upper 1/2 of the byte

    }

    else

    {

      SecondaryBuffer[bufPos] |= (uint8\_t)((value & 0x0F00 )>>8);//store the remaining 4 bits in the lower 1/2 of the byte

      SecondaryBuffer[bufPos+1] =(uint8\_t)( value);//store the first byte

    }

    sampleCount++;

    if (sampleCount >= Shared::PACKED\_SAMPLES\_PER\_PACKET)

    {

      uint8\_t\* temp = PrimaryBuffer;

      PrimaryBuffer = SecondaryBuffer;

      xSemaphoreGiveFromISR(semaOnBufferFull, NULL);

      sampleCount = 0;

      SecondaryBuffer =temp;

    }

  }

  void Setup(void (\*pOnBufferFullEvent)(void\* data, int length))

  {

    adc1\_config\_width(ADC\_WIDTH\_BIT\_12); //12 bits

    adc1\_config\_channel\_atten(ADC1\_CHANNEL\_0,ADC\_ATTEN\_DB\_2\_5);    //0-1.25V

    analogRead(ADC1\_CHANNEL\_0);//must be called once at the start to initialize some driver stuff

    setCpuFrequencyMhz(240); //set cpu to 240mhz incase it isnt already set

    semaOnBufferFull = xSemaphoreCreateBinary();

    onBufferFullEvent = pOnBufferFullEvent;

    PrimaryBuffer =  (uint8\_t\*)malloc(Shared::PACKET\_SIZE); //create buffers of 32 bytes - this is the maximum size the radio can send in 1 packet

    SecondaryBuffer = (uint8\_t\*)malloc(Shared::PACKET\_SIZE);

    xTaskCreatePinnedToCore(onBufferFull, "Buffer Full Task", 8192, NULL, 1, &bufferFullTaskHandler,0);

    //the timers of the ESP32 are 80 MHZ. Using the CPU Divider of 2500 gives us 32k.

    adcTimer = timerBegin(3, 2499, true);

    timerAttachInterrupt(adcTimer, &onTimer32k, true);

    timerAlarmWrite(adcTimer, 1, true);

    timerAlarmEnable(adcTimer);

  }

  void Teardown()

  {

    timerStop(adcTimer);

    timerAlarmDisable(adcTimer);

    timerDetachInterrupt(adcTimer);

    timerEnd(adcTimer);

    vTaskDelete(bufferFullTaskHandler);

    free(PrimaryBuffer);

    free(SecondaryBuffer);

    onBufferFullEvent = NULL;

    vSemaphoreDelete(semaOnBufferFull);

  }

}