Riley Hicks, Ellis Ruckman
June 8, 2022
CPE 316-03
Dr. Bridget Benson

Project 3 – Digital Guitar Pedal

Behavior Description

The goal of this project was to interpret and manipulate an incoming sinusoidal signal from an electric guitar. The guitar output signal is initially biased at 0 V with a maximum amplitude of 700mV, but the first transformation circuit is a summing op-amp circuit that pulls it up to a DC bias of 1.5 V. The signal is then sent to the on-board ADC, and a mathematical filter function is performed on the digital signal to add some desired effect to the original guitar waveform. The board then outputs the newly generated digital signal to the MCP4921 DAC. To drive the speaker, the analog output signal from the DAC is sent through a LM386n-3 audio amplifier utilizing the power supply for extra voltage and current capability. In software, the type of effect produced can be altered by manipulating various multipliers of the filter function.

System Specification

Table 1: System specifications

Parameter	Value	Units
Nucleo supply voltage	5	V
Main clock frequency	80	MHz
uA741 positive supply voltage	3.3	V
uA741 negative supply voltage	-3.3	V
STM32 ADC reference voltage	3.3	V
STM32 ADC resolution	12	bits
DAC reference voltage	3.3	V
DAC resolution	12	bits
DAC output frequency	20	kHz
LM386n-3 supply voltage	12	V
Speaker impedance	8	Ω
Speaker power rating	5-7	W

System Architecture

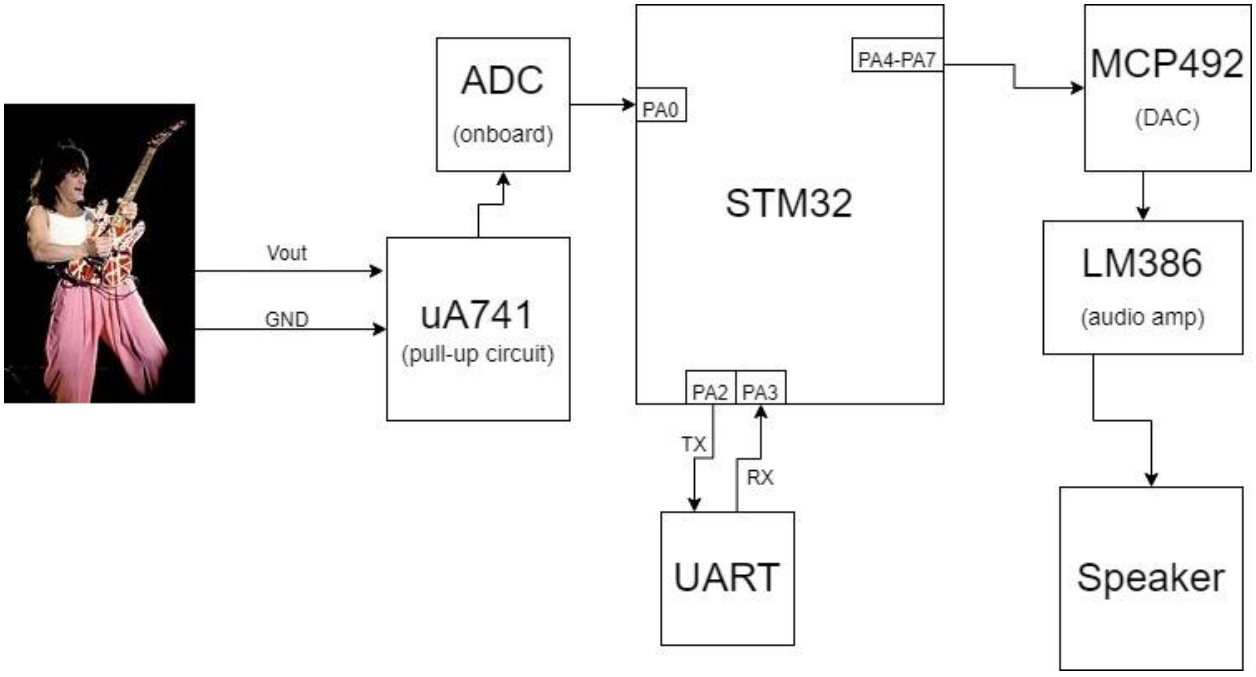

Figure 1: System architecture of the guitar pedal

Figure 1 above shows the system architecture of the guitar pedal. The main signal input comes from a guitar, and is pulled up by a circuit utilizing the uA741 op-amp. The STM32 on-board ADC converts the input waveform to digital values, which are then transformed and output to the MCP492 DAC. The analog output of the DAC feeds into the amplifier circuit containing the LM386, and the high-power output is sent to the speaker. The UART is used for debugging.

System Schematic

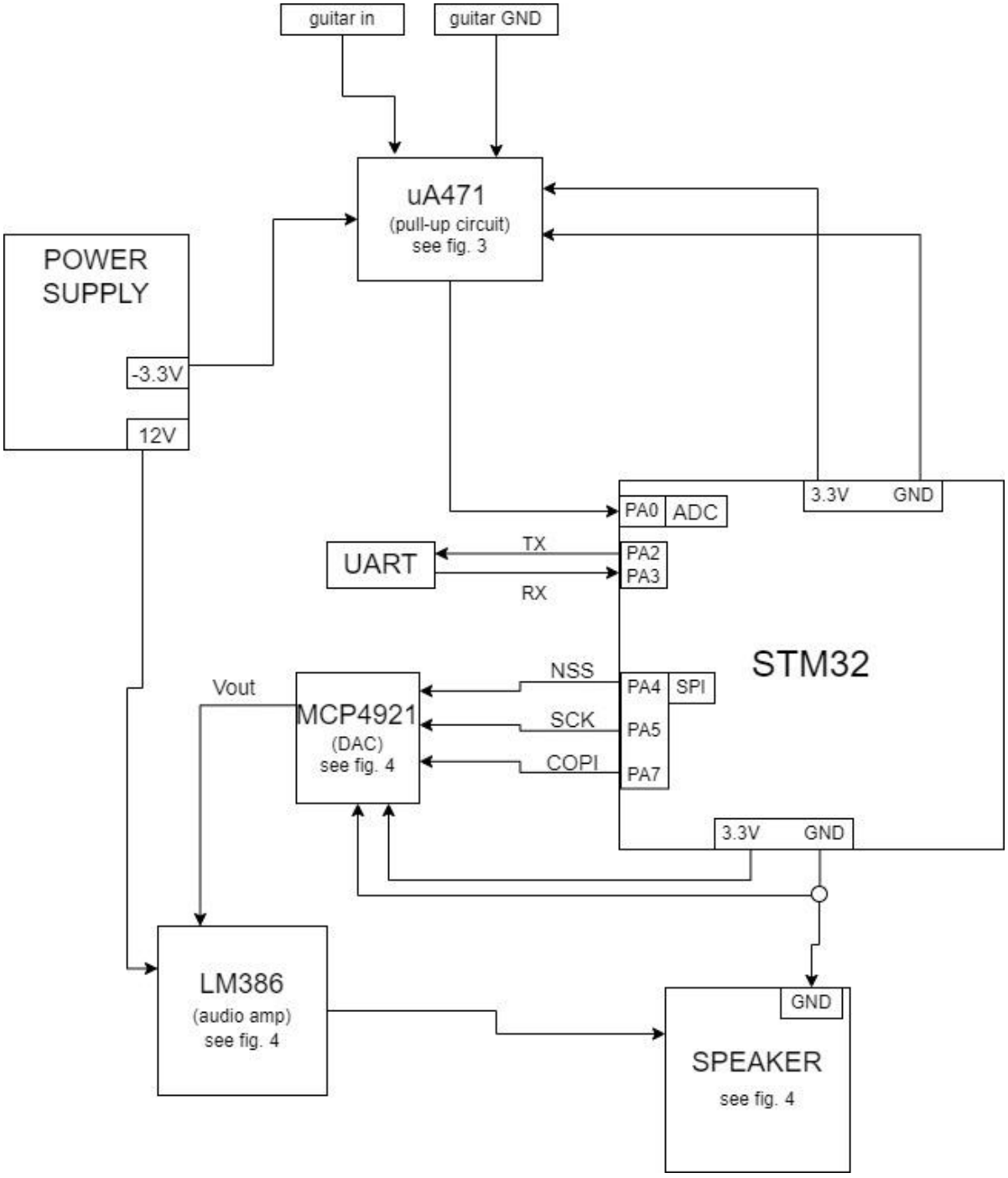

Figure 2: System schematic of the guitar pedal (overview, see Figures 3 and 4 for more detail)

Figure 2 above shows the system schematic of the guitar pedal. For more detail on the pull-up circuit, see Figure 3 below. For more detail on the DAC and amplifier circuit, see Figure 4 below. Analog input data is sent to the ADC on pin PA0, and digital data is sent to the DAC via SPI on PA4, PA5, and PA7.

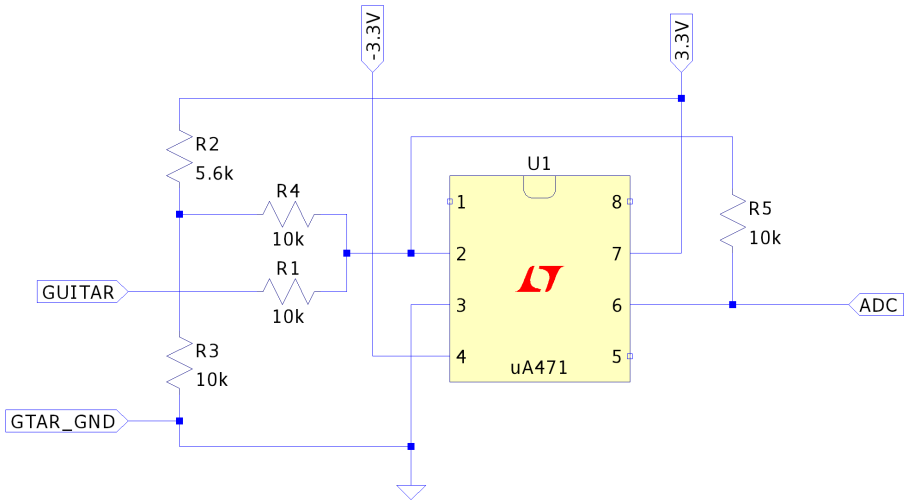

Figure 3: System schematic for the uA741 op-amp pull-up circuit

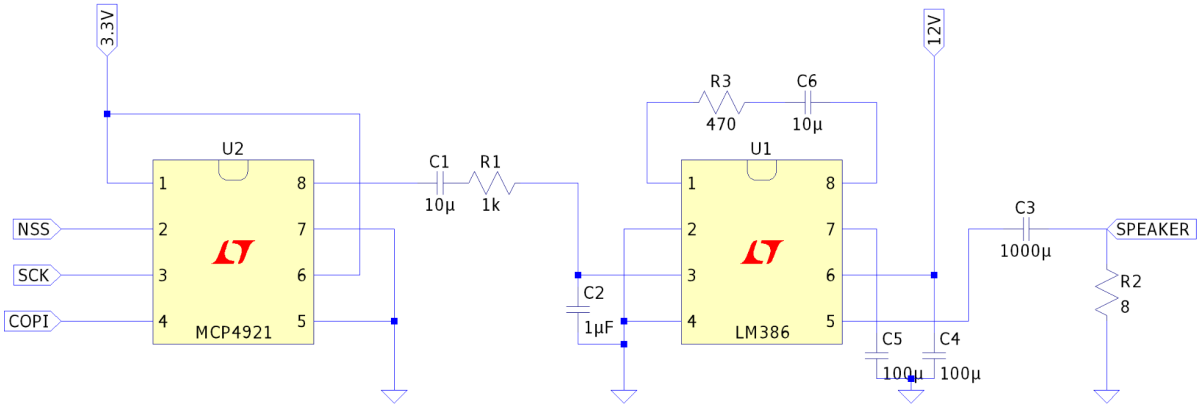

Figure 4: System schematic for the MCP4921 DAC output circuit and LM386 op-amp amplifier circuit

Figure 3 above shows the uA741 op-amp circuit in more detail. 3.3 V and -3.3 V power is provided by the Nucleo board and an external power supply. The input from the guitar is fed into a unity gain summing op-amp along with a constant voltage signal so that the output consists of the same guitar input shifted up to a DC of ~1.5 V.

Figure 4 above shows the MCP4921 DAC circuit connected to the LM386 amplifier circuit. The 12 V rail is from an external supply. Pins 2-4 of the DAC receive digital values via SPI, and the DAC gives an analog output on its pin 8. In the diagram, C1 is a decoupling capacitor before the low pass filter formed by R1 and C2. The loopback R3 and C6 determine the gain of the amplifier, and C3 is used as another decoupling capacitor to block any DC voltage from reaching the speaker.

Software Architecture

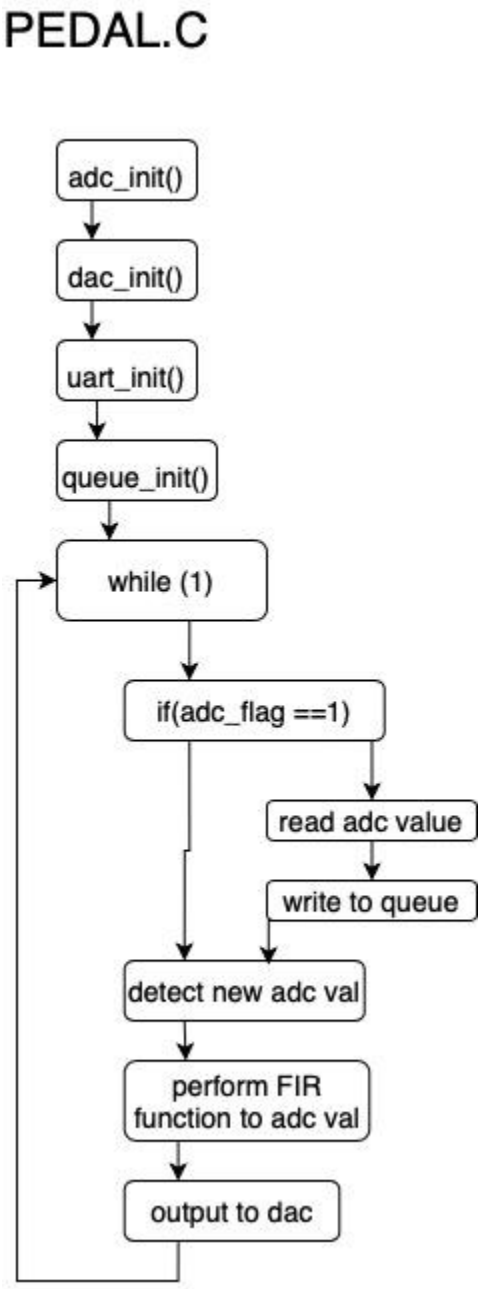

Figure 5: Software architecture of the guitar pedal

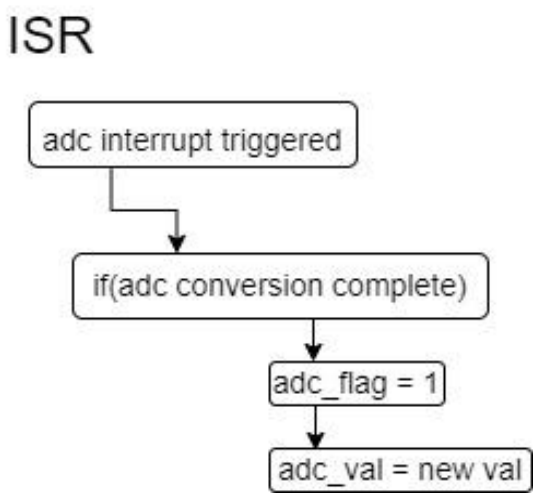

Figure 6: Software architecture of the ADC ISR

Figure 5 above shows the software architecture for the guitar pedal. First, all requisite peripherals are initialized via their xxx_init functions, and then a data queue is created via a call to queue_init. Afterwards, the ADC is manipulated to begin its first conversion. In the main loop of the program, the software checks if data is ready from the ADC, and if so, saves the data in the queue. Next, in every loop iteration, the software calls dac_write to send a certain output value to the DAC. The specific output value is a mathematical manipulation of one or more data values in the queue.

Figure 6 above show the ISR architecture for the ADC specifically. When the ADC finishes a conversion, it triggers an interrupt. In the corresponding ISR, if the conversion complete flag is set, the new data is stored into a global variable, and the conversion flag is cleared. A global flag is set so the code running in the main loop can detect when new data is ready.

Bill of Materials

Table 2: Bill of materials for the function generator

#	Part #	Description	Supplier	Quantity	Unit cost	Total cost
1	20601	Breadboard	Jameco	1	$4.95	$4.95
2	NUCLEO-L476RG	Development board	ST	1	$13.72	$13.72
3	2207401	MCP4921 DAC	Jameco	1	$2.69	$2.69
4	497-6784-5-ND	uA741 op-amp	Digikey	1	$0.46	$0.46
5	N/A	LM386n-3 audio amp	Radioshack	1	$3.94	$3.94
6	2056-FR77-8OHM-ND	8Ω speaker	Digikey	1	$8.79	$8.79
7	SC1085-ND	1/4-inch TS jack	Digikey	1	$3.68	$3.68
8	N/A	1/10/100 uF capacitor	Digikey	4	$0.26	$1.04
9	N/A	1000 uF capacitor	Radioshack	1	$0.87	$0.87
10	N/A	Resistor (various)	Digikey	7	$0.10	$0.70
11	2260738	M-M zipwire	Jameco	1	$4.95	$4.95
12	2260754	M-F zipwire	Jameco	1	$4.95	$4.95

Ethics Implications

An important aspect to consider for the construction of a guitar effect pedal is the quality of the sound and the durability of the device. Any sort of undesired noise or distortion can lead to user dissatisfaction. The amp should be capable of altering the signal while not affecting the original input's quality as it is played out of the speaker. Pedal effects also must be engineered effectively so as not to sound choppy or unpleasant. In the case of our project, more work could be put towards reducing unnecessary noise and a smoother reverb effect. Durability must be considered in the device design as

well. These devices are capable of having a large amount of current sent through it, and the components must be able to withstand these high values to assure the longevity of the pedal.

Since speakers require a fairly large power draw to produce adequate volume, it is important to consider the ethics of using such a high-power device. Care can be taken to minimize the passive power draw of the speaker and the amplifier circuit. For example, by placing a decoupling capacitor before the speaker, any DC bias is eliminated and is not dissipated across the speaker's resistance as it would be without such a capacitor. For safety purposes, the requisite high voltage and/or high current supplied for the speaker such be well insulated and kept separate from other low voltage signals. If the power supply lines are allowed to contact other parts of the device, it could experience an overcurrent event, causing a fire hazard for the end user.

Appendices

References

STM32 datasheet
STM32 reference manual
MCP4921 DAC datasheet
uA741 op-amp datasheet
LM386 datasheet

Code

– main.c (main function) –

```c
int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    // MX_USART2_UART_Init();

    // Start pedal routine
    pedal();

    while (1) {
        ;
    }
}
```

– pedal.h –

```c
/*
 * pedal.h
 *
 *  Created on: May 27, 2022
 *      Author: rileyhicks
 */

#ifndef INC_PEDAL_H_
#define INC_PEDAL_H_

void pedal(void);
```

```
#endif /* INC_PEDAL_H_ */
```

– pedal.c –

```
/*
 * pedal.c
 *
 *  Created on: May 27, 2022
 *      Author: rileyhicks
 */

#include <stdio.h>

#include "pedal.h"
#include "adc.h"
#include "dac.h"
#include "queue.h"
#include "uart.h"

#define CNT_MAX 100

void pedal(void){
    // Initialize peripherals
    adc_init();
    dac_init();
    uart_init();

    struct cq queue;
    cq_init(&queue, 1000);

    // Start first conversion
    adc_start_conv();

    size_t cnt = 0;

    while(1){
        // Update input buffer if the ADC has new data
        if (adc_flag == 1){
            adc_flag = 0;
            if (++cnt > CNT_MAX) {
                cq_append(&queue, adc_val);
                cnt = 0;
            }

            adc_start_conv(); // Start new conversion
        }
```

```
        uint16_t out;

        // No feedback
        out = cq_get(&queue, 1);

        // Constant delay
        /*
        out = cq_get(&queue, 100);
        */

        // Float reverb
        /*
        out = 0.60 * cq_get(&queue, 1) +
              0.20 * cq_get(&queue, 60) +
              0.20 * cq_get(&queue, 120);
        */

        // Int reverb
        /*
        out = 3 * (cq_get(&queue, 1) / 4) +
                   cq_get(&queue, 120) / 4;
        */

        // Int reverb and delay
        /*
        out = cq_get(&queue, 100) / 2 +
              cq_get(&queue, 200) / 4 +
              cq_get(&queue, 300) / 4;
        */

        dac_write(out);
    }
}
```

– adc.h –

```
#ifndef ADC_H
#define ADC_H

#include <stdint.h>

// Pin definitions (CHANGE HERE)
#define ADC_GPIO GPIOA
#define ADC_ADC ADC1 // ADC used (e.g. ADC1, ADC2, etc.)
```

```
#define ADC_GPIO_CLK_MASK RCC_AHB2ENR_GPIOAEN // Used in RCC -> AHB2ENR |= xxx
#define ADC_ADC_CLK_MASK RCC_AHB2ENR_ADCEN // Used in RCC -> AHB2ENR |= xxx
#define ADC_PIN_IN 0
#define ADC_ISER_NUM 0 // Used as index in NVIC -> ISER[x]
#define ADC_ISER_IRQN ADC1_2_IRQn

// Voltage definitions (CHANGE HERE)
#define ADC_RES 12 // Resolution bits
#define ADC_VREF 3300 // Reference voltage in mV

// Global variables
extern volatile int adc_flag; // 1: adc_val contains new conversion, 0: no new conversion
extern volatile uint16_t adc_val;

// Function definitions
void adc_init(void);
void adc_start_conv(void);
uint16_t adc_dtoa(uint16_t dig);

#endif
```

– adc.c –

```
#include <stdint.h>

#include "adc.h"
#include "main.h"

#define ADC_MODER_AN (0b11 << ADC_PIN_IN * 2)

#define IRQN_MASK 0x1f

#define ADC_CHANNEL 5

volatile int adc_flag;
volatile uint16_t adc_val;

void adc_init(void) {
    // Enable the clock for the respective GPIO and and ADC ports
    RCC -> AHB2ENR |= ADC_GPIO_CLK_MASK;
    RCC -> AHB2ENR |= ADC_ADC_CLK_MASK;

    // Enable analog mode
    ADC_GPIO -> MODER |= ADC_MODER_AN;
```

```
// Connect analog pin(s) to the ADC
ADC_GPIO -> ASCR |= 1 << ADC_PIN_IN;

// Set ADC clock mode (no clock division)
ADC123_COMMON -> CCR &= ~ADC_CCR_CKMODE;
ADC123_COMMON -> CCR |= ADC_CCR_CKMODE_0;

// Power up ADC and ADC voltage regulator
ADC_ADC -> CR &= ~ADC_CR_DEEPPWD;
ADC_ADC -> CR |= ADC_CR_ADVREGEN;
HAL_Delay(1);

// Temporarily disable ADC
ADC_ADC -> CR &= ~ADC_CR_ADEN;

// Start calibration
ADC_ADC -> CR &= ~ADC_CR_ADCALDIF;
ADC_ADC -> CR |= ADC_CR_ADCAL;

// Wait for calibration to finish
while (ADC_ADC -> CR & ADC_CR_ADCAL) {
    ;
}

// Configure ADC channel in single-ended mode
ADC_ADC -> DIFSEL &= ~(1 << ADC_CHANNEL);

// Enable ADC
ADC_ADC -> ISR |= ADC_ISR_ADRDY;
ADC_ADC -> CR |= ADC_CR_ADEN;
while (!(ADC_ADC -> ISR & ADC_ISR_ADRDY)) {
    ;
}
ADC_ADC -> ISR |= ADC_ISR_ADRDY;

// Configure ADC conversion sequence (1 conversion, ADC_CHANNEL)
ADC_ADC -> SQR1 &= ~(ADC_SQR1_L_Msk | ADC_SQR1_SQ1_Msk);
ADC_ADC -> SQR1 |= ADC_CHANNEL << ADC_SQR1_SQ1_Pos;

// Enable ADC interrupts
ADC_ADC -> IER |= ADC_IER_EOC;
ADC_ADC -> ISR &= ~ADC_ISR_EOC;
```

```
    NVIC -> ISER[ADC_ISER_NUM] |= 1 << (ADC_ISER_IRQN & IRQN_MASK);
}

void adc_start_conv(void) {
  ADC_ADC -> CR |= ADC_CR_ADSTART;
}

uint16_t adc_dtoa(uint16_t dig) {
    if (dig > (1 << ADC_RES) - 1) {
        dig = (1 << ADC_RES) - 1;
    }
    return ((unsigned long) dig * ADC_VREF) / (1 << ADC_RES);
}

void ADC1_2_IRQHandler(void){
    if (ADC_ADC -> ISR & ADC_ISR_EOC){
        adc_flag = 1;
       adc_val = ADC_ADC -> DR;
    }
}
```

– dac.h –

```
#ifndef DAC_H
#define DAC_H

#include <stdint.h>

// Pin definitions (CHANGE HERE)
#define DAC_GPIO GPIOA
#define DAC_SPI SPI1
#define DAC_GPIO_CLK_MASK RCC_AHB2ENR_GPIOAEN // Used in RCC -> AHB2ENR |= xxx
#define DAC_SPI_CLK_MASK RCC_APB2ENR_SPI1EN // Used in RCC -> APB2ENR |= xxx
#define DAC_PIN_CS 4 // All four pins must be in the lower or upper half of the port
#define DAC_PIN_SCLK 5
#define DAC_PIN_CIPO 6
#define DAC_PIN_COPI 7
#define DAC_AFR_HL 0 // 1: use AFRH (pins are in the upper half), 0: use AFRL (lower half)

// Voltage definitions (CHANGE HERE)
#define DAC_RES 12 // Resolution bits
#define DAC_VREF 3300 // Reference voltage in mV
```

```
// Function definitions
void dac_init(void);
void dac_write(uint16_t val);
uint16_t dac_atod(uint16_t an);

#endif
```

– dac.c –

```
#include <stdint.h>

#include "dac.h"
#include "main.h"

#define DAC_MODER_MASK ((0b11 << DAC_PIN_CS * 2) | (0b11 << DAC_PIN_SCLK * 2) |\
                        (0b11 << DAC_PIN_CIPO * 2) | (0b11 << DAC_PIN_COPI * 2))
#define DAC_MODER_AF ((0b10 << DAC_PIN_CS * 2) | (0b10 << DAC_PIN_SCLK * 2) |\
                      (0b10 << DAC_PIN_CIPO * 2) | (0b10 << DAC_PIN_COPI * 2))

#define DAC_AFRH_MASK ((0b1111 << (DAC_PIN_CS - 8) * 4) | (0b1111 << (DAC_PIN_SCLK - 8) * 4)
|\
                       (0b1111 << (DAC_PIN_CIPO - 8) * 4) | (0b1111 << (DAC_PIN_COPI - 8) *
4))
#define DAC_AFRL_MASK ((0b1111 << DAC_PIN_CS * 4) | (0b1111 << DAC_PIN_SCLK * 4) |\
                       (0b1111 << DAC_PIN_CIPO * 4) | (0b1111 << DAC_PIN_COPI * 4))
#define DAC_AFRH_SPI ((5 << (DAC_PIN_CS - 8) * 4) | (5 << (DAC_PIN_SCLK - 8) * 4) |\
                      (5 << (DAC_PIN_CIPO - 8) * 4) | (5 << (DAC_PIN_COPI - 8) * 4))
#define DAC_AFRL_SPI ((5 << DAC_PIN_CS * 4) | (5 << DAC_PIN_SCLK * 4) |\
                      (5 << DAC_PIN_CIPO * 4) | (5 << DAC_PIN_COPI * 4))

#define DAC_CONFIG 0x3000

void dac_init(void) {
    // Enable the clock for the respective GPIO and SPI ports
    RCC -> AHB2ENR |= DAC_GPIO_CLK_MASK;
    RCC -> APB2ENR |= DAC_SPI_CLK_MASK;

    // Enable alternate function mode
    DAC_GPIO -> MODER &= ~DAC_MODER_MASK;
    DAC_GPIO -> MODER |= DAC_MODER_AF;

    // Set alternate function
```

```
    DAC_GPIO -> AFR[DAC_AFR_HL] &= ~(DAC_AFR_HL ? DAC_AFRH_MASK : DAC_AFRL_MASK);
    DAC_GPIO -> AFR[DAC_AFR_HL] |= DAC_AFR_HL ? DAC_AFRH_SPI : DAC_AFRL_SPI;

    // Set SPI control registers
    // Set SPI as controller, clock rate fclk/4, hardware CS, full duplex, MSB first
    DAC_SPI -> CR1 = SPI_CR1_MSTR | SPI_CR1_BR_0;
    // Enable chip select, create CS pulse, 16-bit data frames
    DAC_SPI -> CR2 = (SPI_CR2_SSOE | SPI_CR2_NSSP | 0xf << SPI_CR2_DS_Pos);

    // Enable SPI
    DAC_SPI -> CR1 |= SPI_CR1_SPE;
}

void dac_write(uint16_t val) {
    // Wait for transmit empty
    while (!(DAC_SPI -> SR & SPI_SR_TXE)) {
        ;
    }

    // Write val
    DAC_SPI -> DR = (val | DAC_CONFIG);
}

uint16_t dac_atod(uint16_t an) {
    if (an < 0) {
        an = 0;
    } else if (an > DAC_VREF - 1) {
        an = DAC_VREF - 1;
    }
    return ((unsigned long) an * (1 << DAC_RES)) / (DAC_VREF);
}
```

– queue.h –

```
/*
 * queue.h
 *
 *  Created on: May 27, 2022
 *      Author: rileyhicks
 */

#ifndef INC_QUEUE_H_
#define INC_QUEUE_H_
```

```
#include <stdint.h>
#define QUEUEOUTPUT 20000

struct cq {
    int headidx;
    int size;
    uint16_t *arr;
};

void cq_init(struct cq *queue, int size);
void cq_append(struct cq *queue, uint16_t val);
uint16_t cq_get(struct cq *queue, uint16_t idx);
uint16_t cq_max(struct cq *queue);
uint16_t cq_min(struct cq *queue);
uint16_t cq_avg(struct cq *queue);

#endif /* INC_QUEUE_H_ */
```

– queue.c –

```
/*
 * queue.c
 *
 *  Created on: May 27, 2022
 *      Author: rileyhicks
 */

#include <stdlib.h>

#include "queue.h"

void cq_init(struct cq *queue, int size){
    queue->headidx = 0;
    queue->size = size;
    queue->arr = malloc(sizeof(uint16_t) * size);
}

void cq_append(struct cq *queue, uint16_t val){
    queue->arr[queue->headidx] = val;
    queue->headidx ++;
    if(queue->headidx == queue->size){
        queue->headidx = 0;
    }
```

```
}

uint16_t cq_get(struct cq *queue, uint16_t idx){
    if(idx >= (queue->size)){
        return -1;
    }
    if (idx > queue->headidx) {
        idx -= queue->headidx;
        return (queue->arr[queue->size - idx]);
    } else {
        return (queue->arr[queue->headidx - idx]);
    }
}


uint16_t cq_max(struct cq *queue){
    uint16_t max = queue->arr[0];
    for(int i = 0; i < queue->size; i++){
        if(queue->arr[i] > max) max = queue->arr[i];
    }
    return max;
}

uint16_t cq_min(struct cq *queue){
    uint16_t min = queue->arr[0];
    for(int i = 0; i < queue->size; i++){
        if(queue->arr[i] < min) min = queue->arr[i];
    }
    return min;
}

#define AVG_COUNT 40

uint16_t cq_avg(struct cq *queue){
    uint32_t avg = 0;
    for (int i = 1; i < AVG_COUNT; ++i) {
        avg += cq_get(queue, i);
    }
    return avg/AVG_COUNT;
}
```

– uart.h –

```c
#ifndef UART_H
#define UART_H

#include <stdint.h>

// Pin definitions (CHANGE HERE)
#define UART_GPIO GPIOA
#define UART_USART USART2 // USART used (e.g. USART2, USART3, etc.)
#define UART_GPIO_CLK_MASK RCC_AHB2ENR_GPIOAEN // Used in RCC -> AHB2ENR |= xxx
#define UART_USART_CLK_MASK RCC_APB1ENR1_USART2EN // Used in RCC -> APB1ENR1 |= xxx
#define UART_PIN_TX 2
#define UART_PIN_RX 3
#define UART_AFR_HL 0 // 1: use AFRH (pins are in the upper half), 0: use ARFL (lower half)
#define UART_AFR_NUM 7
#define UART_ISER_NUM 1 // Used as index in NVIC -> ISER[x]
#define UART_ISER_IRQN USART2_IRQn

// Clock rate definitions (CHANGE HERE)
#define UART_MAIN_CLK 80000000
#define UART_BAUD 115200

// Escape code definitions
// Use strings for param1 and param2, or NULL to omit one or either and use the default
#define UART_ESC_CURS_HOME 'H' // Move cursor to the given position
#define UART_ESC_CURS_FORCE UART_ESC_CURS_HOME
// param1: cursor row (default: 1)
// param2: cursor column (default: 1)

#define UART_ESC_CURS_UP 'A' // Move cursor up by the given number of rows
// param1: number of rows (default: 1)

#define UART_ESC_CURS_DOWN 'B' // Move cursor down by the given number of rows
// param1: number of rows (default: 1)

#define UART_ESC_CURS_RIGHT 'C' // Move cursor right by the given number of rows
// param1: number of rows (default: 1)

#define UART_ESC_CURS_LEFT 'D' // Move cursor left by the given number of rows
// param1: number of rows (default: 1)

#define UART_ESC_CURS_SAVE 's' // Save current cursor position
```

```
#define UART_ESC_CURS_RESTORE 'u' // Restore current cursor position

#define UART_ESC_CURS_ASAVE '7' // Save current cursor position and character attributes

#define UART_ESC_CURS_ARESTORE '8' // Restore current cursor position and character
attributes

#define UART_ESC_ATTR_SET 'm' // Set character attributes
// param1: semicolon-separated list of attributes from the macros below (default: reset)
// (e.g. for bright green text: UART_ESC_ATTR_BRIGHT ";" UART_ESC_ATTR_FG_GREEN)
#define UART_ESC_ATTR_RESET "0" // Reset all attributes
#define UART_ESC_ATTR_BRIGHT "1" // Bright (sometimes bold)
#define UART_ESC_ATTR_DIM "2"
#define UART_ESC_ATTR_UDSCORE "4" // Underscore
#define UART_ESC_ATTR_BLINK "5"
#define UART_ESC_ATTR_REV "7" // Swap foreground and background colors
#define UART_ESC_ATTR_HIDE "8" // Hidden
#define UART_ESC_ATTR_FG_BLACK "30" // Foreground color
#define UART_ESC_ATTR_FG_RED "31"
#define UART_ESC_ATTR_FG_GREEN "32"
#define UART_ESC_ATTR_FG_YELLOW "33"
#define UART_ESC_ATTR_FG_BLUE "34"
#define UART_ESC_ATTR_FG_MAGENTA "35"
#define UART_ESC_ATTR_FG_CYAN "36"
#define UART_ESC_ATTR_FG_WHITE "37"
#define UART_ESC_ATTR_BG_BLACK "40" // Background color
#define UART_ESC_ATTR_BG_RED "41"
#define UART_ESC_ATTR_BG_GREEN "42"
#define UART_ESC_ATTR_BG_YELLOW "43"
#define UART_ESC_ATTR_BG_BLUE "44"
#define UART_ESC_ATTR_BG_MAGENTA "45"
#define UART_ESC_ATTR_BG_CYAN "46"
#define UART_ESC_ATTR_BG_WHITE "47"

// Global variables
extern volatile int uart_rx_flag; // 1: uart_rx_char contains new character from UART rx, 0:
no new char
extern volatile char uart_rx_char;

// Function definitions
void uart_init(void);
void uart_write_char(uint8_t c);
void uart_write_str(char *str);
```

```
void uart_write_esc(uint8_t cmd, char *param1, char *param2);

#endif
```

— uart.c —

```
#include <stdint.h>

#include "main.h"
#include "uart.h"

#define UART_MODER_MASK ((0b11 << UART_PIN_TX * 2) | (0b11 << UART_PIN_RX * 2))
#define UART_MODER_AF ((0b10 << UART_PIN_TX * 2) | (0b10 << UART_PIN_RX * 2))

#define UART_AFRH_MASK ((0b1111 << (UART_PIN_TX - 8) * 4) | (0b1111 << (UART_PIN_RX - 8) *
4))
#define UART_AFRL_MASK ((0b1111 << UART_PIN_TX * 4) | (0b1111 << UART_PIN_RX * 4))
#define UART_AFRH_USART ((UART_AFR_NUM << (UART_PIN_TX - 8) * 4) | \
                         (UART_AFR_NUM << (UART_PIN_RX - 8) * 4))
#define UART_AFRL_USART ((UART_AFR_NUM << UART_PIN_TX * 4) | (UART_AFR_NUM << UART_PIN_RX *
4))

#define IRQN_MASK 0x1f

#define UART_ESC_CHAR 0x1b

volatile int uart_rx_flag;
volatile char uart_rx_char;

void uart_init(void) {
    // Enable the clock for the respective GPIO and UART ports
    RCC -> AHB2ENR |= UART_GPIO_CLK_MASK;
    RCC -> APB1ENR1 |= UART_USART_CLK_MASK;

    // Enable alternate function mode
    UART_GPIO -> MODER &= ~UART_MODER_MASK;
    UART_GPIO -> MODER |= UART_MODER_AF;

    // Set alternate function
    UART_GPIO -> AFR[UART_AFR_HL] &= ~(UART_AFR_HL ? UART_AFRH_MASK : UART_AFRL_MASK);
    UART_GPIO -> AFR[UART_AFR_HL] |= UART_AFR_HL ? UART_AFRH_USART : UART_AFRL_USART;
```

```
    // Set USART control registers
    // rx not empty interrupt, no parity bit, 8 data bits, oversampling 16x, LSB first, 1
stop bit
    UART_USART -> CR1 = USART_CR1_RXNEIE;

    // Set USART baud rate
    UART_USART -> BRR = UART_MAIN_CLK / UART_BAUD;

    // Enable USART (rx and tx)
    UART_USART -> CR1 |= USART_CR1_UE | USART_CR1_RE | USART_CR1_TE;

    // Enable USART interrupts
    NVIC -> ISER[UART_ISER_NUM] |= 1 << (UART_ISER_IRQN & IRQN_MASK);
}

void uart_write_char(uint8_t c) {
    while (!(UART_USART -> ISR & USART_ISR_TXE)) {
        ;
    }

    UART_USART -> TDR = c;
}

void uart_write_str(char *str) {
    for (; *str; ++str) {
        uart_write_char(*str);
    }
}

void uart_write_esc(uint8_t cmd, char *param1, char *param2) {
    switch (cmd) {
    case UART_ESC_CURS_HOME: // '[' with param1 and param2 separated by ';'
        uart_write_char(UART_ESC_CHAR);
        uart_write_char('[');

        if (param1) {
            uart_write_str(param1);
        }

        if (param2) {
            uart_write_char(';');
            uart_write_str(param2);
        }
```

```
        uart_write_char(cmd);
        break;

    case UART_ESC_CURS_UP: // '[' with param1 only
    case UART_ESC_CURS_DOWN:
    case UART_ESC_CURS_RIGHT:
    case UART_ESC_CURS_LEFT:
    case UART_ESC_ATTR_SET:
        uart_write_char(UART_ESC_CHAR);
        uart_write_char('[');

        if (param1) {
            uart_write_str(param1);
        }

        uart_write_char(cmd);
        break;

    case UART_ESC_CURS_SAVE: // '[' with no params
    case UART_ESC_CURS_RESTORE:
        uart_write_char(UART_ESC_CHAR);
        uart_write_char('[');
        uart_write_char(cmd);
        break;

    case UART_ESC_CURS_ASAVE: // no '[' with no params
    case UART_ESC_CURS_ARESTORE:
        uart_write_char(UART_ESC_CHAR);
        uart_write_char(cmd);
        break;
    }
}

/*
void USART2_IRQHandler(void) {
    if (UART_USART -> ISR & USART_ISR_RXNE) {
        uart_rx_flag = 1;
        uart_rx_char = UART_USART -> RDR;
    }
}
*/
```