Time Delay Device Files Documentation

Alex Striff

August 26, 2019

All of the documentation concerning the auxilliary files for the project is assembled here. What follows is a description of the contents of each directory.

- `adjustable-delay`: Schematics and ideas for implementing adjustable delay in the circuit. The subdirectory `Linear` contains a tenative schematic (KiCAD files and PDF) of a circuit that implements the different clock frequencies by counting, while `Arbitrary` gives what I have so far in creating an arbitrary delay circuit.
- `arduino`: The code uploaded to the Arduino Due.
- `characterization`: Contains many different parts of the code used to analyze the characterization data for the device, and includes instructions on how to program the history.
- `doc`: The top-level documentation directory (this one).
- `fifo-p3f`: The KiCAD files for the device, as well as a schematic PDF and the gerber files that I sent for manufacture.
- `figures`: The images and GNUPlot code used to create the figures for the paper.[1]
- `program-rescaling`: The code used to calculate the gain and offset resistor values to be optimal, given the constraints of E-series values and the overall resistances needed.
- `triangle`: The code used to generate the spumpus (initial history signal) for the triangle-DDE (relay-control) circuit in the paper. This is a good starting point if you want to program something different.

[1] *Spumpus* stands for spiky lumpus, where *lumpus* refers to a lumpy-looking signal. We want a strange signal for the programming to demonstrate that it is really *arbitrary.*

FIFO P3F Arduino Due Code

Alex Striff

August 26, 2018

```
/******************************************************************************
 * Time Delay Device Arduino Control
 * Edgar Perez and Alex Striff
 * Reed College Physics Department
 *
 * For more information, contact Lucas Illing <illing@reed.edu>.
 *
 * Version  Date        Author       Changes
 * -------  ----------  -----------  -------------------------------------------
 * v1.0     2016-12-15  Edgar Perez  Prototype completed.
 * v2.0     2019-07-XX  Alex Striff  Modified for publication board.
 *
 ******************************************************************************/

// Code options (feature selection)
#define SCREEN   // Uncomment to enable screen output
#define SCONTROL // Uncomment for serial control and logging
#define HISTORY  // Uncomment for FIFO initial state programming (req: SCONTROL)
#define DEBUG    // Uncomment for serial debug output

#if defined(HISTORY) && defined(SCONTROL)
#define HIST
#include <CRC32.h>
#endif

#ifdef SCREEN
#include <Wire.h>
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>
// #include <Fonts/FreeSans9pt7b.h>
#include "lato24.h"

#define DIGIT_FONT  &Lato_Regular_24 // 24 px high
#define TEXT_FONT   NULL  // 8 px high, NULL for default font in Adafruit GFX

extern TwoWire Wire1; // Use second I2C pins on the Due

#define SCREEN_WIDTH 128 // OLED display width, in pixels
```

```
#define SCREEN_HEIGHT 32 // OLED display height, in pixels
#define OLED_RESET    A0 // Reset pin (or -1 if sharing Arduino reset pin)
Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire1, OLED_RESET);
boolean screen = true;
boolean draw   = true;
char s_digits[16];

enum menu_mode {
    MENU = 0, // Main menu
    DELAY,    // Adjust the word count using the rotary encoder
    FIFO,     // Show FIFO status
    MENU_MODES,
} menu_mode = DELAY;

const PROGMEM char menu_names[][16] = {
    "Menu",
    "Delay",
    "FIFO",
};

uint32_t menu_pos = MENU + 1;
int menu_press = 0;

// Double-press parameters
int dpress = 0;
long dpress_delay = 100L;
long dpress_time = 0L;

// Repeated drawing parameters (in ms)
long last_update = 0L;
long update_interval = 100L;

#endif // SCREEN

typedef const uint8_t pin;

typedef struct {
    pin p;                      // The pin that the button is connected to.
    long debounce_delay;        // The delay (ms) for debouncing a button press.
    int state;                  // The current state of the button.
    int last_state;             // The last state of the button.
    long last_debounce_time;    // The last debounce time for the button.
} button;

typedef struct {
    pin a;          // Output A
    pin b;          // Output B
    button btn;     // Button (knob of encoder)
```

```
} rotary_encoder;

// FIFO word limits
// min_words = 1 or 2 give the same delay.
// min_words = 3 is off-by-one depending upon the clock.
// FIFO words:  0        1  2  3     4   5  ...
// Delay words: invalid  8  8  9,10  11  12 ...
const uint32_t min_words  = 4; // FIFO flags
const uint32_t max_words  = 262144; // FIFO flags
const uint32_t off_words  = 3 + 2 + 2; // ADC + DAC + FIFO.
const uint32_t init_words = 100; // FIFO + offset (user-visible)

#ifdef HIST
// History programming buffer
#define HBUF_SIZE   16 // Make hist relatively large, given SRAM limitations
byte hist[HBUF_SIZE];  // Holds (HBUF_SIZE / 2) code words (uint16_t)
#endif // HIST

// Rotary encoder pin mapping
rotary_encoder enc = {
    .a   = 48, // ENC_A pin
    .b   = 50, // ENC_B pin
    .btn = {
        .p = 52, // ENC_BTN pin
        .debounce_delay = 50,
        .state = 0,
        .last_state = 0,
        .last_debounce_time = 0
    }
};

uint32_t words; // The number of delay words for the FIFO
uint32_t words_digit = 0;

int32_t  count_min = min_words;
int32_t  count_max = max_words;
int32_t  count_mul = 1000;

// FIFO pin mappings:
// Active LOW unless stated otherwise
pin RT    = 42;
pin OE    = 45;
pin REN   = 43;
pin OR    = 40;
pin PAE   = 41;
pin HF    = 38;
pin PAF   = 39;
pin FWFT  = 37; // FWFT/SI. Active HIGH
```

```
pin IR    = 36;
pin LD    = 34;
pin MRS   = 35;
pin PRS   = 32;
pin WEN   = 31;
pin SEN   = 30;

// Clock control pin mappings:
// Active HIGH unless stated otherwise
pin nPROG  = 22;
pin CLK1   = 26; // 1CLK (single-clock operation)
pin WCLK_S = 24;
pin ADC_CLK_S = 28;
pin PROG_D = DAC1;
pin STRIG  = 33;

// FIFO flags to show in status. All active low.
enum fifo_flag {
    F_IR = 0, // Input ready
    F_OR,     // Output ready
    F_PAF,    // Partially almost full
    F_PAE,    // Partially almost empty
    F_HF,     // Half full
    F_SIZE,   // The number of flags (not on FIFO)
} fifo_flags;

const pin fifo_flag_pins[] = {
    IR,
    OR,
    PAF,
    PAE,
    HF,
};

const PROGMEM char fifo_flag_names[][16] = {
    "IR",
    "OR",
    "PAF",
    "PAE",
    "HF",
};

// DAC pin mapping
pin DAC_LDAC = 46;
pin DAC_CS   = 44;

// Serial command buffer
#ifdef SCONTROL
```

```
#define SBLEN   16
char sb[SBLEN];
#endif // SCONTROL


void setup(void)
{
    // Serial communication
    Serial.begin(115200);
    Serial.print(F(
        "====================\n"
        " FIFO_P3 Time Delay \n"
        "====================\n"
        "    Reed College    \n"
        " Physics Department \n"
        "        2019        \n"
        "====================\n"
        "    Alex Striff     \n"
        "    Edgar Perez     \n"
        "    Lucas Illing    \n"
        "====================\n"
        "Enabled features: "
        #ifdef DEBUG
        "DEBUG "
        #endif // DEBUG
        #ifdef SCREEN
        "SCREEN "
        #endif // SCREEN
        #ifdef SCONTROL
        "SCONTROL "
        #endif // SCONTROL
        #ifdef HIST
        "HIST "
        #endif // HIST
        "\n"
        "Type 'h' for help.\n\n"
        "Setting up I/O ... "
        ));

    // DAC initialization
    delay(10);
    digitalWrite(nPROG, LOW); // x. Check not flipped
    digitalWrite(DAC_LDAC, HIGH);
    delay(5);
    digitalWrite(nPROG, HIGH); // x.
    digitalWrite(DAC_LDAC, LOW);

    // Due (self) DAC initialization
```

```
analogWriteResolution(12);
pinMode(PROG_D, OUTPUT);
analogWrite(PROG_D, 1u << (12 - 2)); // Half full scale

// Clock control pins
pinMode(nPROG,     OUTPUT);
pinMode(CLK1,      OUTPUT);
pinMode(WCLK_S,    OUTPUT);
pinMode(ADC_CLK_S, OUTPUT);
pinMode(STRIG,     OUTPUT);

// Clock default state (single clock, not programming)
digitalWrite(CLK1,      HIGH);
digitalWrite(WCLK_S,    LOW);
digitalWrite(ADC_CLK_S, LOW);
digitalWrite(STRIG,     LOW);

digitalWrite(nPROG, HIGH); // Disable operation while setting up.

// FIFO pins
pinMode(REN,  OUTPUT);
pinMode(OR,   INPUT_PULLUP);
pinMode(PAE,  INPUT_PULLUP);
pinMode(HF,   INPUT_PULLUP);
pinMode(PAF,  INPUT_PULLUP);
pinMode(IR,   INPUT_PULLUP);
pinMode(OE,   OUTPUT);
pinMode(RT,   OUTPUT);
pinMode(FWFT, OUTPUT);
pinMode(LD,   OUTPUT);
pinMode(MRS,  OUTPUT);
pinMode(PRS,  OUTPUT);
pinMode(WEN,  OUTPUT);
pinMode(SEN,  OUTPUT);

// FIFO default states
digitalWrite(MRS,  HIGH);
digitalWrite(PRS,  HIGH);
digitalWrite(RT,   HIGH);
//digitalWrite(FWFT, HIGH);
digitalWrite(FWFT, LOW); // Let's try IDT mode
digitalWrite(LD,   HIGH);
digitalWrite(WEN,  HIGH);
digitalWrite(REN,  HIGH);
digitalWrite(SEN,  HIGH);
digitalWrite(OE,   LOW);

// DAC pins
```

```
delay(100);
pinMode(nPROG,    OUTPUT);
pinMode(DAC_LDAC, OUTPUT);
pinMode(DAC_CS,   OUTPUT);

// More DAC initialization
delay(10);
pinMode(DAC_CS,   OUTPUT);
digitalWrite(DAC_CS, HIGH);
delay(5);
digitalWrite(DAC_CS, LOW);
delay(5);
digitalWrite(DAC_CS, HIGH);
delay(5);
digitalWrite(DAC_CS, LOW);
delay(10);

// Rotary encoder pins
pinMode(enc.btn.p, INPUT_PULLUP);
pinMode(enc.a,     INPUT_PULLUP);
pinMode(enc.b,     INPUT_PULLUP);
read_encoder(&enc); // Read to initialize previous quadrature state.

// Screen initialization
#ifdef SCREEN
if (!display.begin(SSD1306_SWITCHCAPVCC, 0x3C)) { // Address 0x3C for 128x32
    Serial.println(F("SSD1306 allocation failed. Unable to use screen."));
    screen = false;
} else {
    screen = true;
    display.clearDisplay();
    display.cp437(true);
    display.setTextColor(WHITE);
    display.setTextSize(1);
    display.setCursor(0, 0);
    display.print(F("Time Delay Device vP3"));
    display.setCursor(0, 8);
    display.print(F("Reed College Physics"));
    display.setCursor(0, 24);
    display.print(F("2-press knob for menu"));
    display.display();

    for (uint8_t i = 0; i < SCREEN_WIDTH; i++) {
        display.drawPixel(i, 16 + (i*i / 7) % 8, WHITE);
        display.display();
    }
}
#endif // SCREEN
```

```
    // Do a master reset to initialize the FIFO
    Serial.print(F(" complete. Initializing FIFO:\n\t"));
    delay(5);
    master_reset(false);
    Serial.println();
    set_delay(init_words, true);
}


void loop(void)
{
    uint32_t narg;

    #ifdef SCONTROL
    for (int i = 0; i < SBLEN; i++)
        sb[i] = '\0';

    // TODO: Update help.
    if (Serial.available() > 0) {
        Serial.readBytesUntil(';', sb, SBLEN);

        narg = atol(sb + 2);
        switch (sb[0]) {
            case 'h': // Fallthrough.
            case 'H': // Fallthrough.
            case '?': serial_help(); break;
            case 'm':
                switch (sb[1]) {
                    case 'r': master_reset(true); break;
                    default:  unknown(); break;
                }
                break;
            case 'p':
                switch (sb[1]) {
                    case 'r': partial_reset(true); break;
                    #ifdef DEBUG
                    case 'p': prog_debug(); break;
                    case 'c': prog_clock_debug(); break;
                    #endif // DEBUG
                    #ifdef HIST
                    case 'h': prog_hist(narg); break;
                    #endif // HIST
                    default:  unknown(); break;
                }
                break;
            case 'd': set_delay(narg, true); break;
            #ifdef HIST
```

```
        case 'g':
            switch (sb[1]) {
                case 'o': start(); break;
                default:  unknown(); break;
            }
            break;
        #endif // HIST
        case 'c':
            switch (sb[1]) {
                case '1': single_clock(); break;
                case '2': dual_clock(); break;
                default:  unknown(); break;
            }
            break;
        case 's': report_status(); break;
        case 'x': pause(narg); break;
        case 'i': initialize(); break;
        case 'q': quit(); break;
        default:  unknown(); break;

    }
}
#endif // SCONTROL

#ifdef SCREEN
if (screen) {
    switch (menu_mode) {
        case MENU:
            menu_pos = enc_adjust_nodigit(&menu_pos,
                    MENU + 1, MENU_MODES - 1, true);
            if (draw) {
                display.clearDisplay();
                display.setFont(TEXT_FONT);
                for (size_t i = MENU + 1; i < MENU_MODES; i++) {
                    int16_t x = 8 + ((i - 1) / 4) * SCREEN_WIDTH;
                    int16_t y = ((i - 1) % 4) * 8;
                    display.setCursor(x, y);
                    display.print(menu_names[i]);
                }

                int16_t alt = ((menu_pos - 1) / 4) * SCREEN_WIDTH;
                int16_t mid = 2 + ((menu_pos - 1) % 4) * 8;
                display.fillTriangle(alt, mid - 2, alt, mid + 2, alt + 3,
                        mid, WHITE);
                display.display();
                draw = false;
            }
```

```
        if (read_button(&(enc.btn))) {
            menu_press = 1;
        } else if (menu_press == 1) {
            // button was pressed and now it is not
            menu_mode = (enum menu_mode) menu_pos;
            menu_press = 0;
            draw = true;
        }

        break;

    case DELAY:
        { // Surrounding block to disambiguate scope
            set_delay(enc_adjust(&words, &words_digit, count_min,
                        count_max, false, false), true);
            int16_t  x, y;
            uint16_t w, h;
            int16_t digit_x = 0;
            int16_t digit_y = 24;

            draw |= millis() - last_update >= update_interval;
            if (draw) {
                display.clearDisplay();
                sprintf(s_digits, "%06u", words + off_words);
                display.setFont(DIGIT_FONT);
                display.setCursor(digit_x, digit_y);
                display.print(s_digits);

                int16_t mid = 6 + (6 - words_digit - 1) * 15;
                int16_t base = 31;
                display.fillTriangle(mid - 3, base, mid + 3, base, mid,
                        base - 4, WHITE);

                display.setFont(TEXT_FONT);
                display.getTextBounds(s_digits, digit_x, digit_y, &x,
                        &y, &w, &h);
                display.setCursor(SCREEN_WIDTH - w + 6,
                        digit_y - 2*h + 1);
                display.print(F("Delay"));
                display.getTextBounds(s_digits, digit_x, digit_y,
                        &x, &y, &w, &h);
                display.setCursor(SCREEN_WIDTH - w + 6,
                        digit_y - h + 1);
                display.print(F("words"));

                display.display();
                draw = false;
            }
```

```
        }
        break;

    case FIFO:
        {
            draw |= millis() - last_update >= update_interval;
            if (draw) {
                display.clearDisplay();
                display.setFont(TEXT_FONT);
                for (size_t i = 0; i < F_SIZE; i++) {
                    int16_t x = 8 + (i / 4)
                        * SCREEN_WIDTH / ((F_SIZE / 4) + 1);
                    int16_t y = (i % 4) * 8;
                    display.setCursor(x, y);
                    display.print(fifo_flag_names[i]);
                    display.setCursor(x + 4*8, y);
                    // Invert to show understandable logic
                    display.print(digitalRead(fifo_flag_pins[i])
                            ? 0 : 1, DEC);
                }
                display.display();
                draw = false;
                last_update = millis();
            }

            press_menu_return(&enc);
        }
        break;

    default:
        Serial.println("Menu error!");
        break;
}

// Double-press detection
if (menu_mode != MENU) {
    int press = read_button(&(enc.btn));
    int quick = millis() - dpress_time <= dpress_delay;
    if ((dpress == 0 && press)
        || (dpress == 1 && !press && quick)
        || (dpress == 2 && press && quick)) {
        dpress_time = millis();
        dpress++;
    } else if (dpress == 3 && !press && quick) {
        dpress = 0;
        menu_mode = menu_mode != MENU ? MENU : DELAY;
        draw = true;
    } else if (!quick) {
```

```
                dpress = 0;
            }
        }
    }
    #endif // SCREEN
}

#ifdef SCREEN
void press_menu_return(rotary_encoder *e)
{
    if (read_button(&(e->btn))) {
        menu_press = HIGH;
    } else if (menu_press == HIGH) { // Button was pressed and now it is not
        menu_press = LOW;
        menu_mode = MENU;
        menu_press = LOW;
        draw = true;
    }
}
#endif // SCREEN

#ifdef SCONTROL
void unknown(void)
{
    Serial.println(F("Unknown Command. Type 'h' for help."));
    serial_help();
}
#endif // SCONTROL


#ifdef SCONTROL
void serial_help(void)
{
    Serial.print(F(
    "Time Delay Device Serial Commands\n"
    "=================================\n"
    "mr\tPerforms a master reset of FIFO memory.\n"
    "pr\tPerforms a partial reset of FIFO memory.\n"
    "d N\tSets delay to N words, where N is a decimal number between "));
    Serial.print(min_words + off_words, DEC);
    Serial.print(F(" and "));
    Serial.print(max_words + off_words, DEC);
    Serial.print(F(
    ".\n"
    "\tE.g.: 'd 9' and 'd 101' produce delays of 9 and 101 words, respectively.\n"
    "\tValues outside the possible range will be clamped, so 'd 0' is the same as 'd "
    ));
    Serial.print(min_words + off_words, DEC);
```

```
    Serial.print(F(
    "'.\n"
    "\t<time delay> = <delay words> * <clock period>.\n"
    "ph N;<data>\tPrograms N words of initial history into the FIFO.\n"
    "\tAfter the semicolon, 2N bytes of big-endian data must follow.\n"
    "\tThe data are 12-bit unsigned code words representing analog signals,\n"
    "\twhere 0x0000 gives -2.5V and 0x0FFF gives +2.5V.\n"
    "s\tReports FIFO status.\n"
    "x N\tPauses for N seconds (neglecting serial delays; not precise).\n"
    "i\tInitializes the program.\n"
    "q\tQuits the program.\n"
    "h H ?\tShows this help.\n"
    "\n"
    ));
}
#endif // SCONTROL


void partial_reset(boolean start)
{
    long t1, t2;
    Serial.print(F("Partial Reset ... "));
    t1 = micros();
    digitalWrite(nPROG, LOW);
    delay(1);

    digitalWrite(REN, HIGH);
    digitalWrite(WEN, HIGH);
    digitalWrite(RT,  HIGH);
    delayMicroseconds(3);
    digitalWrite(PRS, LOW);
    delayMicroseconds(3);
    digitalWrite(PRS, HIGH);
    delayMicroseconds(3);
    digitalWrite(WEN, LOW);
    if (start) digitalWrite(nPROG, HIGH);

    t2 = micros() - t1;
    delayMicroseconds(1);
    Serial.print("done (");
    Serial.print(t2);
    Serial.println(" us).");
}


void master_reset(boolean start)
{
    long t1, t2;
```

```
    Serial.print(F("Master Reset ... "));
    t1 = micros();
    digitalWrite(nPROG, LOW);
    //digitalWrite(FWFT, HIGH);
    digitalWrite(FWFT, LOW); // Let's try IDT mode
    digitalWrite(LD,    HIGH);
    digitalWrite(MRS,   LOW);
    delayMicroseconds(1);
    digitalWrite(MRS, HIGH);
    if (start) digitalWrite(nPROG, HIGH);

    t2 = micros() - t1;
    delayMicroseconds(1);
    Serial.print(F("done ("));
    Serial.print(t2);
    Serial.println(F(" us)."));
    Serial.print(F("Device in FWFT with Serial Loading.\n"));
}


#ifdef SCONTROL
void report_status(void)
{
    Serial.print(F(
        "FIFO Status\n"
        "===========\n"));

    for (size_t i = 0; i < F_SIZE; i++) {
        Serial.print(F("~"));
        Serial.print(fifo_flag_names[i]);
        Serial.print(F("\t"));
        Serial.println(digitalRead(fifo_flag_pins[i]), DEC);
    }
    Serial.println();

    #ifdef DEBUG
    Serial.println(words_digit);
    Serial.println(words + off_words);
    Serial.println(ipow(3, 0));
    Serial.println(ipow(3, 1));
    Serial.println(ipow(3, 5));
    #endif // DEBUG
}
#endif // SCONTROL


#ifdef SCONTROL
void pause(uint32_t n)
```

```
{
    for (uint32_t i = 0; i < n; i++) {
        Serial.println(n - i);
        delay(1000);
    }
    Serial.println("0.");
}
#endif // SCONTROL


void set_delay(uint32_t n, boolean start)
{
    unsigned int m;
    int bits;

    n = clamp(n - off_words, min_words, max_words);
    if (n == words && start) return;
    words = n;
    m = max_words - n;

    Serial.print(F("Sending a "));
    Serial.print(n);
    Serial.print(F("-word delay ... "));
    digitalWrite(nPROG, 0);
    delayMicroseconds(10);

    digitalWrite(WCLK_S, 0);
    digitalWrite(LD, 0);
    digitalWrite(SEN, 0);
    delayMicroseconds(2);

    // The following number is defined because PAF triggers at the given
    // number of words AWAY FROM the end, not AT the given word like PAE.
    for (int i = 0; i < 36 ; i++) {
        bits = i == 0 ? 1 : bitRead(m, i - 18);
        digitalWrite(FWFT, bits);
        delayMicroseconds(1);
        digitalWrite(WCLK_S, 1);
        delayMicroseconds(1);
        digitalWrite(WCLK_S, 0);
    }

    digitalWrite(LD, 1);
    digitalWrite(SEN, 1);
    digitalWrite(nPROG, 1);
    Serial.print(F("done. Resetting: "));
    partial_reset(start);
}
```

```
void set_fifo_delay(uint32_t n, boolean start)
{
    set_delay(n + off_words, start);
}


void initialize(void)
{
    digitalWrite(WEN, 0);
    digitalWrite(REN, 0);
    Serial.println(F("Initiated program."));
}


void quit(void)
{
    digitalWrite(WEN, 1);
    digitalWrite(REN, 1);
    Serial.println(F("Quit program."));
}


int read_encoder(rotary_encoder *e)
{
    static int8_t enc_states[4][4] = {
        { 0, -1,  1,  0},
        { 1,  0,  0, -1},
        {-1,  0,  0,  1},
        { 0,  1, -1,  0}
    };
    static uint8_t old_AB = 0;
    int8_t direction;
    uint8_t AB = digitalRead(e->a) << 1 | digitalRead(e->b);

    direction = enc_states[old_AB][AB];
    old_AB = AB;

    return direction;
}


int32_t clamp(int32_t x, int32_t min, int32_t max)
{
    return x <= min ? min : x >= max ? max : x;
}
```

```
int ipow(int base, unsigned int exp)
{
    int result = 1;

    for (; exp; base *= base, exp >>= 1)
        if (exp & 1)
            result *= base;

    return result;
}


uint32_t enc_adjust(uint32_t *n, uint32_t *digit, uint32_t n_min,
        uint32_t n_max, boolean reverse, boolean dreverse)
{
    int     direction;
    int32_t count = *n;
    direction = read_encoder(&enc);

    if (direction) {
        if (read_button(&(enc.btn))) {
            *digit = clamp(*digit + direction * (dreverse ? -1 : 1), 0, 5);
            #ifdef DEBUG
            Serial.print(F("Button press: "));
            Serial.println(*digit);
            #endif // DEBUG
        } else {
            count = clamp(count + direction * (reverse ? -1 : 1)
                    * ipow(10, *digit), n_min, n_max);
            #ifdef DEBUG
            Serial.print(F("Count: "));
            Serial.println(count);
            #endif // DEBUG
        }
        #ifdef SCREEN
        draw = true;
        dpress = 0;
        #endif // SCREEN
    }

    return (uint32_t) count + off_words;
}


uint32_t enc_adjust_nodigit(uint32_t *n, uint32_t n_min, uint32_t n_max,
        boolean reverse)
{
```

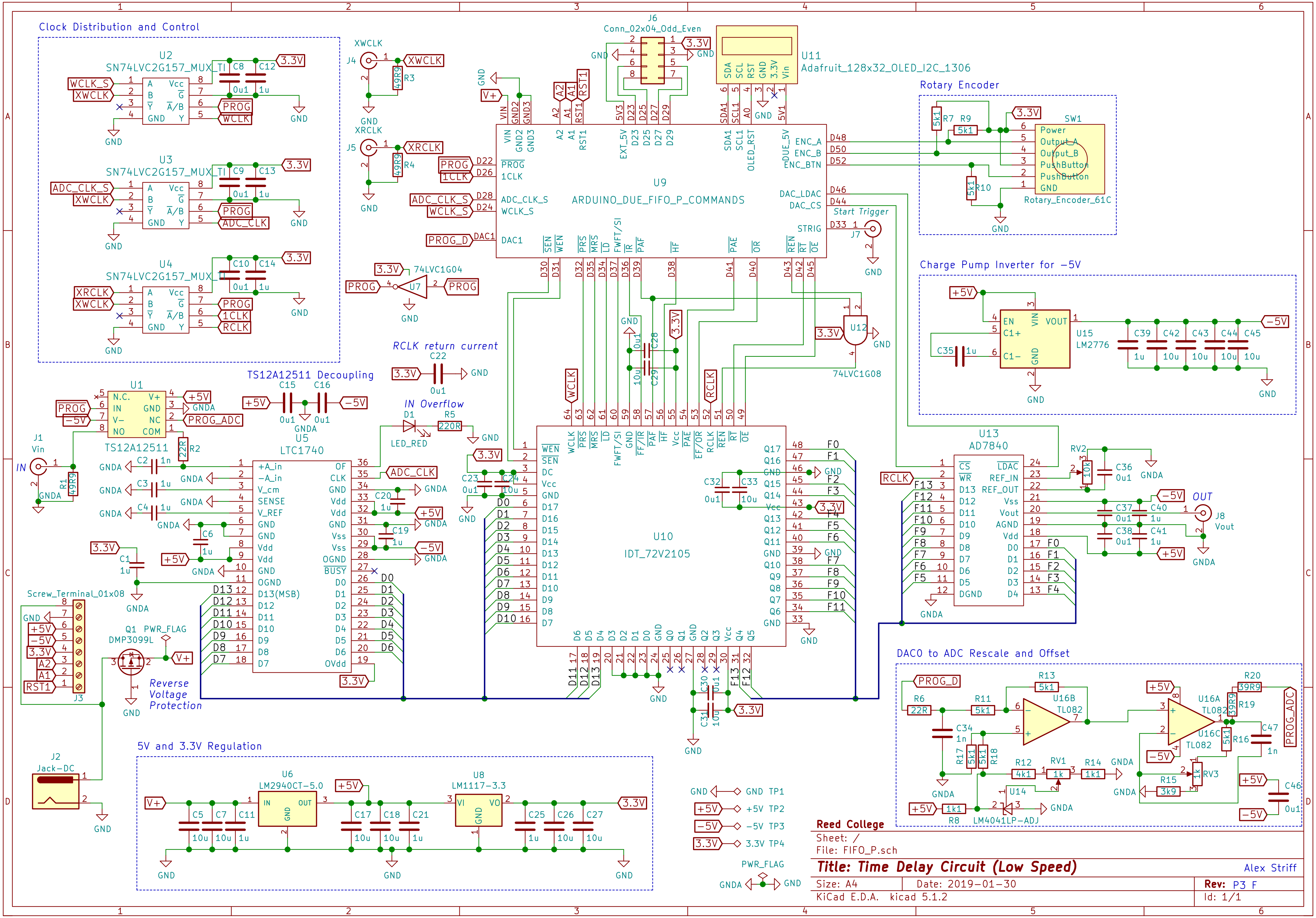
```
    int     direction;
    int32_t count = *n;
    direction = read_encoder(&enc) * (reverse ? -1 : 1);

    if (direction) {
        count = clamp(count + direction, n_min, n_max);
        #ifdef DEBUG
        Serial.print("ND Count: ");
        Serial.println(count);
        #endif // DEBUG
        #ifdef SCREEN
        draw = true;
        dpress = 0;
        #endif // SCREEN
    }

    return (uint32_t) count;
}


int read_button(button *button)
{
    int reading = digitalRead(button->p);

    if (reading != button->last_state)
        button->last_debounce_time = millis();
    if (millis() - button->last_debounce_time > button->debounce_delay)
        button->state = reading;

    return button->last_state = reading;
}


#ifdef HIST
int prog_hist(uint32_t n)
{
    // Reads n code words (uint16_t values for DAC) and programs them into the
    // FIFO memory. Little-endian.
    uint32_t nwords = n;
    n *= 2; // Words to bytes.
    uint32_t bufs = 0u;
    uint32_t bytes = 0u;
    uint32_t col = 8;
    uint32_t cols = 2;
    uint32_t row = col * cols;
    char hex[16];
    byte msb, lsb, sbyte;
    uint16_t data;
```

```
uint32_t read_bytes;
uint32_t buf_remaining;
uint32_t adc_off = 3u;
uint32_t adc_ins = 0u;
CRC32 crc;

// Enter programming mode.
digitalWrite(nPROG, LOW);
digitalWrite(STRIG, LOW);
Serial.print(F("Programming "));
Serial.print(nwords, DEC);
Serial.print(F(" words of history...\n\n"));
master_reset(false);
//set_fifo_delay(nwords, false);
set_delay(nwords, false);

// Program the data one hist[] buffer at a time.
while (bytes < n) {
    while (!Serial.available());

    buf_remaining = min(HBUF_SIZE, n - bytes);
    read_bytes = Serial.readBytes(hist, buf_remaining);
    bytes += read_bytes;
    boolean last = read_bytes < HBUF_SIZE;

    if (read_bytes != buf_remaining) {
        Serial.print(F("Error reading history data! (Received "));
        Serial.print(bytes, DEC);
        Serial.print(F(" bytes, but expected "));
        Serial.print(n, DEC);
        Serial.println(F(" bytes.)"));
        return 0;
    }

    digitalWrite(ADC_CLK_S, LOW);
    digitalWrite(WCLK_S, LOW);
    for (uint32_t j = 0; j < (read_bytes / row) + (last ? 1 : 0); j++) {
        uint32_t end_bytes = last ? read_bytes % row : row;

        // The actual programming
        for (uint32_t k = 0; k < end_bytes; k++) {
            // This may be decreased, depending upon signal speed
            delayMicroseconds(100);
            sbyte = hist[row*j + k];
            crc.update(sbyte);
            if (k % 2 == 0) {
                lsb = sbyte;
            } else {
```

```
                msb = sbyte;
                // Programming circuitry inverts, so reinvert
                data = 0x0FFF - ((msb << 8) | lsb);
                //data = ((msb << 8) | lsb); // Inverted
                analogWrite(PROG_D, data);
                // This may be decreased, depending upon signal speed
                delayMicroseconds(1000);
                // Write ADC output N-3 to the FIFO before latching in
                // the next in put to the ADC.
                if (adc_ins >= adc_off) digitalWrite(WCLK_S, HIGH);
                // These microsecond delays are probably unnecessary.
                // Check FIFO clock hold times.
                delayMicroseconds(10);
                if (adc_ins >= adc_off) digitalWrite(WCLK_S, LOW);
                delayMicroseconds(10);
                digitalWrite(ADC_CLK_S, HIGH);
                delayMicroseconds(10);
                digitalWrite(ADC_CLK_S, LOW);
                if (adc_ins < adc_off) adc_ins++;
            }
        }
        // The last 3 (adc_off) history data words remain in the ADC.
        // They will be clocked in after the switch to normal operation
        // (external WCLK).

        // Printing a hexdump back
        sprintf(hex, "%06X:\t", bufs * HBUF_SIZE + j * row);
        Serial.print(hex);
        for (uint32_t k = 0; k < end_bytes; k++) {
            sprintf(hex, "%02X", hist[row*j + k]);
            Serial.print(hex);
            if (k % col == col - 1) Serial.print(' ');
        }
        if (last) { // Pad last row
            for (uint32_t k = 0; k < 2 * (row - end_bytes); k++)
                Serial.print(' ');
        }
        Serial.print('\t');
        for (uint32_t k = 0; k < end_bytes; k++) {
            byte b = hist[row*j + k];
            Serial.print(b < 32 ? '.' : (char) b);
        }
        Serial.println();
    }

    bufs++;
}
Serial.print(F("CRC32: "));
```

```
    Serial.println(crc.finalize(), HEX);
    Serial.println(F(
    "\nDone programming history."
    "Waiting for start command ('go')."));

    return 1;
}
#endif // HIST


#ifdef HIST
void start()
{
    // Start normal delay operation after programming the history.
    digitalWrite(nPROG, HIGH);
    digitalWrite(STRIG, HIGH);
    Serial.print(F("Started."));
}
#endif // HIST


#ifdef SCONTROL
void single_clock(void)
{
    digitalWrite(CLK1, HIGH);
}
#endif // SCONTROL


#ifdef SCONTROL
void dual_clock(void)
{
    digitalWrite(CLK1, LOW);
}
#endif // SCONTROL


void prog_debug(void)
{
    uint32_t nwords = 1 << 13;
    digitalWrite(nPROG, LOW);
    digitalWrite(STRIG, LOW);
    Serial.print(F("DEBUG: Programming "));
    Serial.print(nwords, DEC);
    Serial.print(F(" words of history...\n\n"));
    master_reset(false);
    set_fifo_delay(nwords, false);
    digitalWrite(nPROG, LOW);
```

```
    delayMicroseconds(10);
    // Does not account for first three words into ADC, but OK for debugging
    for (uint16_t triangle = 0u ;; triangle++) {
    //for (uint16_t triangle = 0u; triangle < nwords; triangle++) {
        // High 4 bits are discarded. Subtraction inverts.
        analogWrite(PROG_D, ((1<<12) - 1) - 4 * triangle);
        digitalWrite(WCLK_S, HIGH);
        digitalWrite(WCLK_S, LOW);
        digitalWrite(ADC_CLK_S, HIGH);
        digitalWrite(ADC_CLK_S, LOW);
    }
    Serial.println(F(
    "\nDEBUG: Done programming history."
    "Waiting for start command ('go')."));
}


void prog_clock_debug(void)
{
    digitalWrite(WCLK_S, HIGH);
    digitalWrite(WCLK_S, LOW);
    digitalWrite(ADC_CLK_S, HIGH);
    digitalWrite(ADC_CLK_S, LOW);
}

// vim:ts=4:sts=4:sw=4:et:
```



Clock Distribution and Control
U2 SN74LVC2G157_MUX_TI
U3 SN74LVC2G157_MUX_TI
U4 SN74LVC2G157_MUX_TI
U9 ARDUINO_DUE_FIFO_P_COMMANDS
J6 Conn_02x04_Odd_Even
U11 Adafruit_128x32_OLED_I2C_1306
Rotary Encoder
SW1 Rotary_Encoder_61C
Start Trigger
U7 74LVC1G04
U12 74LVC1G08
Charge Pump Inverter for −5V
U15 LM2776
RCLK return current
TS12A12511 Decoupling
IN Overflow
D1 LED_RED
U1 TS12A12511
U5 LTC1740
U10 IDT_72V2105
U13 AD7840
Screw_Terminal_01x08
Q1 DMP3099L
PWR_FLAG
Reverse Voltage Protection
J2 Jack-DC
5V and 3.3V Regulation
U6 LM2940CT-5.0
U8 LM1117-3.3
DAC0 to ADC Rescale and Offset
U16A TL082
U16B TL082
U16C TL082
U14 LM4041LP-ADJ
GND TP1
+5V TP2
−5V TP3
3.3V TP4
Reed College
Sheet: /
File: FIFO_P.sch
Title: Time Delay Circuit (Low Speed)
Alex Striff
Size: A4
Date: 2019-01-30
Rev: P3 F
KiCad E.D.A. kicad 5.1.2
Id: 1/1

program-rescaling

August 26, 2019

1 Offset and gain calculations for Due DAC to ADC opamps

1.1 Voltages that determine required offset and gain

```
In [1]: V_DL   = 3.3 * 1/6; # V. Due DAC0 minimum voltage
        V_DH   = 3.3 * 5/6; # V. Due DAC0 maximum voltage
        V_AL   = -2.5;      # V. ADC input minimum voltage
        V_AH   = 2.5;       # V. ADC input maximum voltage
        offset = ((V_AH + V_AL) - (V_DH + V_DL)) / 2.; # V
        gain   = (V_AH - V_AL) / (V_DH - V_DL); # V/V
        println("Offset:\t", offset, " V\nGain:\t", gain, " V");
```

```
Offset:         -1.65 V
Gain:       2.2727272727272725 V
```

1.2 Choice of resistor tolerance range (the E-series to use)

```
In [2]: E3  = [1.0, 2.2, 4.7]; # Very common
        E24 = [
            1.0, 1.1, 1.2, 1.3, 1.5,
            1.6, 1.8, 2.0, 2.2, 2.4,
            2.7, 3.0, 3.3, 3.6, 3.9,
            4.3, 4.7, 5.1, 5.6, 6.2,
            6.8, 7.5, 8.2, 9.1
        ]; # 5% tolerance
```

```
In [3]: E_series = E24;
```

Evaluates f on all possible combinations of resistors in a given E-series.

```
In [4]: function combinations(E, f)
            span = 1:length(E)
            [[f(E[i], E[j]), E[i], E[j]]
                for i in span for j in span] # [[f(x, y), x, y]]
        end
```

```
Out[4]: combinations (generic function with 1 method)
```

Finds the nearest E-series value to the value given.

```
In [5]: function nearest(E, x)
            p = floor(log10(x));
            y = x / 10^p;
            best = first(sort(E, lt = (a, b) -> abs(y - a) < abs(y - b)));
            best * 10^p;
        end
```

```
Out[5]: nearest (generic function with 1 method)
```

1.3 Offset component selection calculations

Improved calculation of the output voltage of the `LM4041-ADJ` where V_{ref} depends upon V_O (`V_Onom` here).

```
In [6]: V_Onom = abs(offset); # V
        V_Y = 1.240; # V
        ΔV_O = V_Onom - V_Y; # V
        ΔV_ref_ΔV_O = -1.55e-3; # V/V. Absolute worst case: -3 mV/V
        V_ref = V_Y + ΔV_O * ΔV_ref_ΔV_O; # V
        R2_to_R1 = (V_Onom / V_ref) - 1.; # Ω/Ω
        VO(ratio) = V_ref * (ratio + 1.);
        println("V_ref = ", V_ref, " V");
```

```
V_ref = 1.2393645 V
```

Selection parameters.

```
In [7]: ratio_tolerance = 1.; # %
        RV_off = 1.0e3; # Ω
        println("We want a resistor ratio of ", R2_to_R1, " Ω/Ω and ",
            ratio_tolerance, "% nominal tolerance.");
```

```
We want a resistor ratio of 0.33132746661696366 Ω/Ω and 1.0% nominal tolerance.
```

Calculate the best resistors to choose to give the best nominal ratio.

```
In [8]: offset_ratio(x, y, off) = (x + 0.5 * off) / (y + 0.5 * off);
        # Divide RV_off by 1e3 so that it is at the same scale as the E-series values.
        off_ratios     = combinations(E_series,
             # [[Ω/Ω, Ω, Ω]]. [[ratio, R2, R1]]
             (x, y) -> offset_ratio(x, y, RV_off / 1e3));
        off_ratios_tol = filter(x -> ratio_tolerance / 100. > abs(1. - x[1] / R2_to_R1),
            off_ratios); # [[Ω/Ω, Ω, Ω]]
        off_ratios_s   = sort(off_ratios_tol, lt = (x, y) -> isless(abs(x[1] - R2_to_R1),
                abs(y[1] - R2_to_R1))); # [[Ω/Ω, Ω, Ω]]
```

```
        if (length(off_ratios_s) > 0)
            R1 = off_ratios_s[1][3] * 1e3; # Ω
            R2 = off_ratios_s[1][2] * 1e3; # Ω
            display(off_ratios_s);
            println("Best offset resistor values:\n",
                "R2: ", R2 / 1e3, " kΩ\n",
                "R1: ", R1 / 1e3, " kΩ");
        else
            println("No combinations give a ", ratio_tolerance, "% offset tolerance.");
        end
```

```
3-element Array{Array{Float64,1},1}:
 [0.333333, 1.1, 4.3]
 [0.333333, 2.4, 8.2]
 [0.333333, 2.7, 9.1]


Best offset resistor values:
R2: 1.1 kΩ
R1: 4.3 kΩ
```

Calculate the margins of adjustability when a potentiometer is inserted between the two resistors.

```
In [9]: VO_simple(ratio) = 1.233 * (ratio + 1.);

In [10]: ratio_min = R2 / (R1 + RV_off);
         ratio_max = (R2 + RV_off) / R1;
         println("Low:\t", VO(ratio_min), " V");
         println("High:\t", VO(ratio_max), " V");
         println("S Low:\t", VO_simple(ratio_min), " V");
         println("S High:\t", VO_simple(ratio_max), " V");
         V_O_low   = VO(ratio_min) - V_Onom; # V. Negative margin away from V_Onom
         V_O_high  = VO(ratio_max) - V_Onom; # V. Positive margin away from V_Onom
         V_O_total = V_O_high - V_O_low;     # V. Total argin away from V_Onom
         println("Offset margins **after amplification** for a ",
             RV_off / 1e3, " kΩ potentiometer:\n",
             "Low margin:\t",   V_O_low   * 1e3 * gain," mV\t(",
             100. * V_O_low   / V_Onom, "%)\n",
             "High margin:\t",  V_O_high  * 1e3 * gain, " mV\t(",
             100. * V_O_high  / V_Onom, "%)\n",
             "Total margin:\t", V_O_total * 1e3 * gain, " mV\t(",
             100. * V_O_total / V_Onom, "%)");
```

```
Low:        1.4965910943396226 V
High:        1.844635534883721 V
S Low:        1.4889056603773585 V
S High:        1.8351627906976746 V
```

```
Offset margins **after amplification** for a 1.0 kΩ potentiometer:
Low margin:        -348.65660377358483 mV        (-9.297509433962265%)
High margin:        442.3534883720931 mV        (11.796093023255818%)
Total margin:        791.010092145678 mV        (21.093602457218083%)
```

1.3.1 LM4041-ADJ resistor selection

```
In [11]: Vcc_ref = 5.; # V
         V_R_ref = Vcc_ref - V_Onom; # V
         R_load = R1 + RV_off + R2; # Ω
         I_load = V_Onom / R_load; # A
         I_Q = 2.9e-3; # A
         I_t = I_Q + I_load;
         R_ref = nearest(E_series, V_R_ref / I_t); # Ω
         I_ref = V_R_ref / R_ref; # A
         println("Best resistor for the LM4041-ADJ (I_load = ", I_load * 1e3, " mA):\n",
             "R_ref:\t", R_ref / 1e3, " kΩ\n",
             "I_ref:\t", I_ref * 1e3, " mA\t(", 100. * (I_ref - I_t) / I_t, "%)\n",
             "Power:\t", V_R_ref * I_ref * 1e3, " mW");
```

```
Best resistor for the LM4041-ADJ (I_load = 0.2578125 mA):
R_ref:        1.1 kΩ
I_ref:        3.045454545454546 mA        (-3.558094552651698%)
Power:        10.202272727272728 mW
```

1.4 Gain component selection calculations

Selection parameters.

```
In [12]: div_tolerance = 1.; # %
         RV_gain = 1.0e3 # Ω
         #div_nom = 1. / gain;
         div_nom = 5.0 / 2.2;
         println("We want a resistor divider with a gain of ", div_nom, " V/V and ",
             div_tolerance, "% nominal tolerance.");
```

```
We want a resistor divider with a gain of 2.2727272727272725 V/V and 1.0% nominal tolerance.
```

Calculate the best resistors to choose to give the best nominal resistor divider.

```
In [13]: # For noninverting gain configuration
         offset_divider(x, y, off) = 1. + (y + 0.5 * off) / (x + 0.5 * off);
         # Divide RV_gain by 1e3 so that it is at the same scale as the E-series values.
         gain_divs = combinations(E_series, # [[Ω/Ω, Ω, Ω]]. [[ratio, Rb, Ra]]
             (x, y) -> offset_divider(x, y, RV_gain / 1e3));
         gain_divs_tol = filter(x -> div_tolerance / 100. > abs(1. - x[1] / div_nom),
```

```
    gain_divs); # [[Ω/Ω, Ω, Ω]]
gain_divs_s   = sort(gain_divs_tol, # [[Ω/Ω, Ω, Ω]]
    lt = (x, y) -> isless(abs(x[1] - div_nom), abs(y[1] - div_nom)));
if (length(gain_divs_s) > 0)
    Ra = gain_divs_s[1][3] * 1e3; # Ω
    Rb = gain_divs_s[1][2] * 1e3; # Ω
    display(gain_divs_s);
    println("Best gain resistor values:\n",
        "Ra: ", Ra / 1e3, " kΩ\n",
        "Rb: ", Rb / 1e3, " kΩ");
else
    println("No combinations give a ", div_tolerance, "% gain tolerance.");
end
```

```
13-element Array{Array{Float64,1},1}:
 [2.27273, 3.9, 5.1]
 [2.27083, 4.3, 5.6]
 [2.26829, 3.6, 4.7]
 [2.27778, 1.3, 1.8]
 [2.28, 2.0, 2.7]
 [2.28125, 2.7, 3.6]
 [2.26316, 3.3, 4.3]
 [2.26087, 1.8, 2.4]
 [2.28571, 1.6, 2.2]
 [2.25714, 3.0, 3.9]
 [2.28846, 4.7, 6.2]
 [2.25, 1.1, 1.5]
 [2.25, 1.5, 2.0]


Best gain resistor values:
Ra: 5.1 kΩ
Rb: 3.9 kΩ
```

Calculate the margins of adjustability when a potentiometer is inserted between the two resistors.

```
In [14]: div_min    = Rb / (Ra + Rb + RV_gain);
         div_max    = (Rb + RV_gain) / (Ra + Rb + RV_gain);
         div_low    = div_min   - div_nom; # V. Negative margin away from V_Onom
         div_high   = div_max   - div_nom; # V. Positive margin away from V_Onom
         div_total = div_high - div_low; # V. Total margin away from V_Onom
         println("Gain margins for a ", RV_gain / 1e3, " kΩ potentiometer:\n",
             "Low margin:\t",   div_low    * 1e3, " mV\t(",
             100. * div_low    / div_nom, "%)\n",
             "High margin:\t",  div_high   * 1e3, " mV\t(",
             100. * div_high   / div_nom, "%)\n",
```

```
            "Total margin:\t", div_total * 1e3, " mV\t(",
            100. * div_total / div_nom, "%)");
```

```
Gain margins for a 1.0 kΩ potentiometer:
Low margin:        -1882.7272727272723 mV        (-82.83999999999999%)
High margin:        -1782.7272727272725 mV        (-78.44%)
Total margin:        99.99999999999987 mV        (4.399999999999994%)
```