```
#include <ADC.h>
#include <RingBuffer.h>
#include <IntervalTimer.h>
#include <Wire.h>
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>

// System mode definitions
#define MODE_SYSTEM_DISPLAY 0
#define MODE_SYSTEM_MANUAL  1
#define MODE_SYSTEM_AUTO    2

// Display mode definitions
#define MODE_DISPLAY_MENU_MAIN    0
#define MODE_DISPLAY_MENU_FREQ    1
#define MODE_DISPLAY_MANUAL       2
#define MODE_DISPLAY_AUTO         3

// System display state definitions
#define STATE_DISPLAY_MENU_MAIN_SETUP       0
#define STATE_DISPLAY_MENU_MAIN_SELECTION   1
#define STATE_DISPLAY_MENU_FREQ_SETUP       2
#define STATE_DISPLAY_MENU_FREQ_SELECTION   3
#define STATE_DISPLAY_MENU_MANUAL           4
#define STATE_DISPLAY_MENU_AUTO             5

// System state defines
#define STATE_SYSTEM_READ_SAMPLE            0
#define STATE_SYSTEM_PROCESS_SAMPLE         1
```

```
#define STATE_SYSTEM_DISPLAY_FREQUENCY      2
#define STATE_SYSTEM_AUTO_CONTROL_MOTOR     3
#define STATE_SYSTEM_MANUAL_CONTROL_MOTOR   4

// Peak detection state defines
#define STATE_PEAK_DETECTION_SET_THRESHOLD         0
#define STATE_PEAK_DETECTION_CHECK_POSITIVE_SLOPE   1
#define STATE_PEAK_DETECTION_CHECK_NEGATIVE_SLOPE   2
#define STATE_PEAK_DETECTION_FOUND                 3

// Pin defines
#define PIN_SAMPLE       A0
#define PIN_MOTOR_ACTIVE  A7
#define PIN_MOTOR_CW      A8
#define PIN_MOTOR_CCW     A9
#define PIN_BUTTON_UP     9
#define PIN_BUTTON_SELECT 10
#define PIN_BUTTON_DOWN   11

#define SAMPLE_SIZE 1024
#define PID_STACK_SIZE 10

Adafruit_SSD1306 display(4);

const float sample_frequency = 49000; // Sample frequency from ADC (Hz) (22.3k)
const int sample_period = 25; // us
const int read_period = 100000; // us

const float Kp = 0.2;
const float Ki = 0.2;
const float Kd = 0.2;

const float target_frequency_cap_high = 2000;
const float target_frequency_cap_low = 0;

float found_frequency = 0;
float found_frequency_zero_cross = 0;
volatile float target_frequency = 0;

short sample_buffer[SAMPLE_SIZE]; // ADC samples stored in this buffer
short sample_buffer_offset[SAMPLE_SIZE];
int sample_index;

volatile byte mode_system;
volatile byte mode_display;
volatile byte state_system;
volatile byte state_display;
volatile byte mainMenuSelection = 1;

bool debug_on;

ADC *adc = new ADC();
RingBuffer *memory_buffer = new RingBuffer();

IntervalTimer timer;
int startTimerValue;

float pidStack[PID_STACK_SIZE];
```

```
int pidStackIndex = 0;

volatile bool memory_buffer_copied = false;
volatile bool motorTurningCW = false;
volatile bool motorTurningCCW = false;

void setup()
{
  Serial.begin(115200);      // Input data rate (bps)

  pinMode(PIN_MOTOR_ACTIVE, OUTPUT);
  pinMode(PIN_MOTOR_CW, OUTPUT);
  pinMode(PIN_MOTOR_CCW, OUTPUT);
  pinMode(PIN_SAMPLE, INPUT);
  pinMode(PIN_BUTTON_UP, INPUT);
  pinMode(PIN_BUTTON_SELECT, INPUT);
  pinMode(PIN_BUTTON_DOWN, INPUT);
  pinMode(12, OUTPUT);

  attachInterrupt(digitalPinToInterrupt(PIN_BUTTON_UP), buttonUpLowISR, LOW);
  //attachInterrupt(digitalPinToInterrupt(PIN_BUTTON_SELECT), buttonSelectLowISR, LOW);
  attachInterrupt(digitalPinToInterrupt(PIN_BUTTON_DOWN), buttonDownLowISR, LOW);

  attachInterrupt(digitalPinToInterrupt(PIN_BUTTON_UP), buttonUpRisingISR, RISING);
  attachInterrupt(digitalPinToInterrupt(PIN_BUTTON_SELECT), buttonSelectRisingISR, RISING);
  attachInterrupt(digitalPinToInterrupt(PIN_BUTTON_SELECT), buttonDownRisingISR, RISING);

  attachInterrupt(digitalPinToInterrupt(PIN_BUTTON_UP), buttonUpFallingISR, FALLING);
  //attachInterrupt(digitalPinToInterrupt(PIN_BUTTON_SELECT), buttonSelectFallingISR, FALLING);
  attachInterrupt(digitalPinToInterrupt(PIN_BUTTON_DOWN), buttonDownFallingISR, FALLING);

  display.begin(SSD1306_SWITCHCAPVCC, 0x3C);
  delay(2000);
  display.clearDisplay();

  // ADC setup
  adc->setAveraging(4);
  adc->setResolution(12);

  display.begin(SSD1306_SWITCHCAPVCC, 0x3C);

  // Default system mode/state
  mode_system = MODE_SYSTEM_DISPLAY;
  mode_display = MODE_DISPLAY_MENU_MAIN;
  state_system = STATE_SYSTEM_READ_SAMPLE;
  state_display = STATE_DISPLAY_MENU_MAIN_SETUP;
  debug_on = false;
}

void loop()
{
  if (debug_on)
    debugOutput();

  if (mode_system == MODE_SYSTEM_DISPLAY)
  {
    switch (mode_display)
    {
```

```
case MODE_DISPLAY_MENU_MAIN:
 switch (state_display)
 {
  case STATE_DISPLAY_MENU_MAIN_SETUP:
   stateDisplayMainMenuSetup();
   break;

  case STATE_DISPLAY_MENU_MAIN_SELECTION:
   break;

  default:
   break;
 }
 break;

case MODE_DISPLAY_MENU_FREQ:
 switch (state_display)
 {
  case STATE_DISPLAY_MENU_FREQ_SETUP:
   stateDisplayFreqMenuSetup();
   break;

  case STATE_DISPLAY_MENU_FREQ_SELECTION:
   break;

  default:
   break;
 }
 break;
```

```
  case MODE_DISPLAY_MANUAL:
   switch (state_display)
   {
    case STATE_DISPLAY_MENU_MANUAL:
     stateDisplayManual();
     break;

    default:
     break;
   }
   break;

  case MODE_DISPLAY_AUTO:
   switch (state_display)
   {
    case STATE_DISPLAY_MENU_AUTO:
     stateDisplayAuto();
    break;

    default:
     break;
   }
   break;
 }
}
else if (mode_system == MODE_SYSTEM_MANUAL)
{
 switch (state_system)
```

```
  {
    case STATE_SYSTEM_DISPLAY_FREQUENCY:
      stateDisplayFrequency();
      break;

    case STATE_SYSTEM_MANUAL_CONTROL_MOTOR:
      stateManualControlMotor();
      break;

    case STATE_SYSTEM_READ_SAMPLE:
      stateReadSample();
      break;

    case STATE_SYSTEM_PROCESS_SAMPLE:
      stateProcessSample();
      break;

    default:
      break;
  }
}
else if (mode_system == MODE_SYSTEM_AUTO)
{
  switch (state_system)
  {
    // TODO: Bluetooth input state.

    // TODO: Bluetooth output state.
```

```
   case STATE_SYSTEM_DISPLAY_FREQUENCY:
    stateDisplayFrequency();
    break;

   case STATE_SYSTEM_AUTO_CONTROL_MOTOR:
    stateAutoControlMotor();
    break;

   case STATE_SYSTEM_READ_SAMPLE:
    stateReadSample();
    break;

   case STATE_SYSTEM_PROCESS_SAMPLE:
    stateProcessSample();
    break;

   default:
    break;
  }
 }
}

void stateDisplayMainMenuSetup()
{
 display.setTextSize(1.75);
 display.setTextColor(WHITE);
 display.setCursor(0, 0);
 display.println("Mode of Operation:");
 display.println("> 1. Manual Mode");
```

```
  display.println("  2. Automatic Mode");
  display.display();

  state_display = STATE_DISPLAY_MENU_MAIN_SELECTION;
}

void stateDisplayFreqMenuSetup()
{
  display.clearDisplay();
  display.setCursor(0, 0);
  display.println("  TARGET FREQUENCY");
  display.println("---------------------");
  display.println();
  display.println();
  display.println("Target: 0000 (Hz)");
  display.display();

  state_display = STATE_DISPLAY_MENU_FREQ_SELECTION;
}

void stateDisplayManual()
{
  display.clearDisplay();
  display.setCursor(0, 0);
  display.println("     MANUAL MODE");
  display.println("---------------------");
  display.println();
  display.print("CC Frequency: ");
  display.println(found_frequency);
  display.print("ZC Frequency: ");
  display.println(found_frequency_zero_cross);
  display.println();
  display.println("> 1. Main Menu");
  display.display();

  mode_system = MODE_SYSTEM_MANUAL;
  state_system = STATE_SYSTEM_READ_SAMPLE;
}

void stateDisplayAuto()
{
  display.clearDisplay();
  display.setCursor(0, 0);
  display.println("   AUTOMATIC MODE");
  display.println("---------------------");
  display.println();
  display.print("CC Frequency: ");
  display.println(found_frequency);
  display.print("ZC Frequency: ");
  display.println(found_frequency_zero_cross);
  display.println();
  display.println("> 1. Main Menu");
  display.display();

  mode_system = MODE_SYSTEM_AUTO;
  state_system = STATE_SYSTEM_AUTO_CONTROL_MOTOR;
}
```

```
// Reads and stores the full sample from the ADC
void stateReadSample()
{
  // Start the timers, if it's not possible, startTimerValue will be false.
  startTimerValue = timer.begin(timer_callback, sample_period);
  adc->enableInterrupts(ADC_0);
  delay(500);

  if (startTimerValue == false)
    Serial.println("Timer setup failed!");

  while(!memory_buffer_copied);

  timer.end();

  memory_buffer_copied = false;
  state_system = STATE_SYSTEM_PROCESS_SAMPLE;
}

void stateProcessSample()
{
  int i;
  int signal_threshold = 0;
  int signal_period = 0;
  byte state_peak_detection = STATE_PEAK_DETECTION_SET_THRESHOLD;
  long sum = -1;
  long sum_previous = -1;
  int posSamplesPerPeriod = 0;
  int negSamplesPerPeriod = 0;
  int numHalfPeriodCheck = 6;
  int numHalfPeriods = 0;
  int samplesPerPeriodCheck = 0;
  short lastOffsetSample = -1;
  sample_buffer_offset[0] = -1;
  boolean firstPositiveCrossingPassed = false;
  boolean zeroCrossCheck = true;
  bool posZeroCrossingCheck = false;
  bool negZeroCrossingCheck = false;

  for (i = 0; i < SAMPLE_SIZE; i++)
  {
    sample_buffer_offset[i] = sample_buffer[i] - 3083;

    if (zeroCrossCheck)
    {
     // Positive Zero Crossing Check
     if (!posZeroCrossingCheck && sample_buffer_offset[i] >= 0 && lastOffsetSample <= 0)
     {
      if (firstPositiveCrossingPassed)
       posZeroCrossingCheck = true;
      else
       firstPositiveCrossingPassed = true;
     }

     // On positive side of period
     if (posZeroCrossingCheck)
     {
      posSamplesPerPeriod++;  //Count samples per positive side of period
```

```
    // Negative Zero Crossing Check
    if (sample_buffer_offset[i] <= 0 && lastOffsetSample >= 0)
    {
     posZeroCrossingCheck = false;   // Not on positive half of period
     negZeroCrossingCheck = true;
     numHalfPeriods++;               // Completed a full half period
     samplesPerPeriodCheck += posSamplesPerPeriod;
     posSamplesPerPeriod = 0;
    }
   }
  }

  if (negZeroCrossingCheck)
  {
   negSamplesPerPeriod++;

   if (sample_buffer_offset[i] >= 0 && lastOffsetSample <= 0)
   {
    negZeroCrossingCheck = false;
    numHalfPeriods++;
    samplesPerPeriodCheck += negSamplesPerPeriod;
    negSamplesPerPeriod = 0;

    if (numHalfPeriods >= numHalfPeriodCheck)
    {
     found_frequency_zero_cross = ( 1 / ( ((samplesPerPeriodCheck + numHalfPeriods) /
(numHalfPeriods / 2)) * 20 * 0.000001));
     zeroCrossCheck = false;
    }
   }
  }

  lastOffsetSample = sample_buffer_offset[i];
 }

 for (i = 0; i < SAMPLE_SIZE; i++)
 {
  sum_previous = sum;

  sum = doAutocorrelation(i);
  doPeakDetection(i, &state_peak_detection, &signal_threshold, &signal_period, sum, sum_previous);
 }

 // Frequency is found (Hz)
 found_frequency = sample_frequency / signal_period;
 state_system = STATE_SYSTEM_DISPLAY_FREQUENCY;

 sample_index = 0;
}

long doAutocorrelation(int i)
{
 int j;
 long sum = 0;

 for (j = 0; j < SAMPLE_SIZE - i; j++)
 {
    sum += (sample_buffer_offset[j]) * (sample_buffer_offset[j + i]);
  }

  sum /= 4096;

  return sum;
}

void doPeakDetection(int i, byte *state, int *threshold, int *signal_period, long sum, long sum_previous)
{
  switch (*state)
  {
    case STATE_PEAK_DETECTION_SET_THRESHOLD:
      *threshold = sum / 2;
      *state = STATE_PEAK_DETECTION_CHECK_POSITIVE_SLOPE;
      break;

    case STATE_PEAK_DETECTION_CHECK_POSITIVE_SLOPE:
      if ((sum > *threshold) && (sum - sum_previous) > 0)
        *state = STATE_PEAK_DETECTION_CHECK_NEGATIVE_SLOPE;
      break;

    case STATE_PEAK_DETECTION_CHECK_NEGATIVE_SLOPE:
      if ((sum - sum_previous) <= 0)
      {
        *signal_period = i;
        *state = STATE_PEAK_DETECTION_FOUND;
      }
      break;
    default:
      break;
  }
}

void stateDisplayFrequency()
{
  Serial.print("FREQUENCY: ");
  Serial.println(found_frequency);

  if (mode_system == MODE_SYSTEM_MANUAL)
  {
    mode_display = MODE_DISPLAY_MANUAL;
    state_display = STATE_DISPLAY_MENU_MANUAL;
  }
  else if (mode_system == MODE_SYSTEM_AUTO)
  {
    mode_display = MODE_DISPLAY_AUTO;
    state_display = STATE_DISPLAY_MENU_AUTO;
  }

  mode_system = MODE_SYSTEM_DISPLAY;
}


// All values here are complete guesses right now. Will need to check later.
void stateAutoControlMotor()
{
```

```
float difference = target_frequency - found_frequency;

// Clockwise
if (difference > 5)
{
  if (difference > 200)
  {
    state_system = STATE_SYSTEM_READ_SAMPLE;
    return;
  }
  digitalWrite(PIN_MOTOR_ACTIVE, HIGH);
  runMotor(difference, 1);
}
else if (difference < -5)
{
  if (difference < -200)
  {
    state_system = STATE_SYSTEM_READ_SAMPLE;
    return;
  }
  digitalWrite(PIN_MOTOR_ACTIVE, HIGH);
  runMotor(difference * -1, 2);
}
else
{
  int i;

  for (i = 0; i < PID_STACK_SIZE; i++)
  {
      pidStack[i] = 0;
    }

    mode_system = MODE_SYSTEM_DISPLAY;
    mode_display = MODE_DISPLAY_MENU_MAIN;
    state_display = STATE_DISPLAY_MENU_MAIN_SETUP;

    display.clearDisplay();
    display.display();
  }

  found_frequency = 0;
  digitalWrite(PIN_MOTOR_ACTIVE, LOW);
  state_system = STATE_SYSTEM_READ_SAMPLE;
}

void runMotor(float difference, byte dir)
{
  float motorDriveValue = 0;

  //Push new value onto stack;
  float newStack[PID_STACK_SIZE];
  newStack[0] = difference;

  int i;
  for (i = 1; i < PID_STACK_SIZE; i++)
  {
    newStack[i] = pidStack[i - 1];
  }
```

```
Serial.print("DIFFERENCE: ");
Serial.println(difference);
float stackSum = 0;
Serial.print("STACK: [");
for (i = 0; i < PID_STACK_SIZE; i++)
{
  pidStack[i] = newStack[i];
  stackSum += pidStack[i];
  Serial.print(pidStack[i]);
  Serial.print(", ");
}
Serial.println("]");

motorDriveValue = (Kp * difference) + (Ki * stackSum) + (Kd * (pidStack[0] - pidStack[1]));
Serial.print("DRIVE VALUE: ");
Serial.println(motorDriveValue);

// Clockwise
if (dir == 1)
{
  analogWrite(PIN_MOTOR_CW, (int)(motorDriveValue));
  delay(1000);
}
else if (dir == 2) // Counter-Clockwise
{
  analogWrite(PIN_MOTOR_CCW, (int)(motorDriveValue));
  delay(1000);
}
```

```
}

void stateManualControlMotor()
{
  while(motorTurningCW)
  {
    digitalWrite(PIN_MOTOR_CW, HIGH);
  }

  while(motorTurningCCW)
  {
    digitalWrite(PIN_MOTOR_CCW, HIGH);
  }
}

/*
void stateManualControlMotor()
{
  int state_clockwise = digitalRead(button_clockwise);
  int state_counter_clockwise = digitalRead(button_counter_clockwise);

  if (state_clockwise == HIGH && state_counter_clockwise == HIGH)
  {
    state_system = STATE_SYSTEM_READ_SAMPLE;
    return;
  }

  // Turn clockwise
  if (state_clockwise == HIGH)
```

```
{
  digitalWrite(PIN_MOTOR_DIR, HIGH);
  digitalWrite(PIN_MOTOR_ACTIVE, HIGH);

  while (state_clockwise == HIGH)
  {
    delay(15);

    state_clockwise = digitalRead(button_clockwise);
    state_counter_clockwise = digitalRead(button_counter_clockwise);

    if (state_counter_clockwise == HIGH)
    {
      digitalWrite(PIN_MOTOR_CW, LOW);
      state_system = STATE_SYSTEM_READ_SAMPLE;
      return;
    }
  }

  digitalWrite(PIN_MOTOR_ACTIVE, LOW);
}

// Turn counter-clockwise
if (state_counter_clockwise == HIGH)
{
  digitalWrite(PIN_MOTOR_DIR, LOW);
  digitalWrite(PIN_MOTOR_ACTIVE, HIGH);

  while (state_counter_clockwise == HIGH)
  {
   delay(15);

   state_counter_clockwise = digitalRead(button_counter_clockwise);
   state_clockwise = digitalRead(button_clockwise);

   if (state_clockwise == HIGH)
   {
    digitalWrite(PIN_MOTOR_CCW, LOW);
    state_system = STATE_SYSTEM_READ_SAMPLE;
    return;
   }
  }

  digitalWrite(PIN_MOTOR_ACTIVE, LOW);
 }

 state_system = STATE_SYSTEM_READ_SAMPLE;
}
*/


void debugOutput()
{
 String mode_system_debug = "ERROR";
 String mode_display_debug = "ERROR";
 String system_state_debug = "ERROR";
 String display_state_debug = "ERROR";
```

```
if (mode_system == MODE_SYSTEM_DISPLAY)
{
  mode_system_debug = "DISPLAY";

  switch (mode_display)
  {
    case MODE_DISPLAY_MENU_MAIN:
      mode_display_debug = "MENU_MAIN";
      switch (state_display)
      {
        case STATE_DISPLAY_MENU_MAIN_SETUP:
          display_state_debug = "MENU_MAIN_SETUP";
          break;

        case STATE_DISPLAY_MENU_MAIN_SELECTION:
          display_state_debug = "MENU_MAIN_SELECTION";
          break;

        default:
          break;
      }
      break;

    case MODE_DISPLAY_MENU_FREQ:
      mode_display_debug == "MENU_FREQ";
      switch (state_display)
      {
        case STATE_DISPLAY_MENU_FREQ_SETUP:
```

```
      display_state_debug = "MENU_FREQ_SETUP";
      break;

    case STATE_DISPLAY_MENU_FREQ_SELECTION:
      display_state_debug = "MENU_FREQ_SELECTION";
      break;
  }
  break;

case MODE_DISPLAY_MANUAL:
  mode_display_debug = "MANUAL";
  switch (state_display)
  {
    case STATE_DISPLAY_MENU_MANUAL:
      display_state_debug = "MENU_MANUAL";
      break;

    default:
      break;
  }
  break;

case MODE_DISPLAY_AUTO:
  mode_display_debug = "AUTO";
  switch (state_display)
  {
    case STATE_DISPLAY_MENU_AUTO:
      display_state_debug = "MENU_AUTO";
      break;
```

```
   default:
    break;
  }
  break;
}

switch (state_system)
{
 // TODO: Button input state

 case STATE_SYSTEM_DISPLAY_FREQUENCY:
  system_state_debug = "DISPLAY_FREQUENCY";
  break;

 case STATE_SYSTEM_MANUAL_CONTROL_MOTOR:
  system_state_debug = "MANUAL_CONTROL_MOTOR";
  break;

 case STATE_SYSTEM_READ_SAMPLE:
  system_state_debug = "READ_SAMPLE";
  break;

 case STATE_SYSTEM_PROCESS_SAMPLE:
  system_state_debug = "PROCESS_SAMPLE";
  break;

 default:
  break;
  }
}
else if (mode_system == MODE_SYSTEM_MANUAL)
{
  mode_system_debug = "MANUAL";

  switch (state_system)
  {
    // TODO: Button input state

    case STATE_SYSTEM_DISPLAY_FREQUENCY:
      system_state_debug = "DISPLAY_FREQUENCY";
      break;

    case STATE_SYSTEM_MANUAL_CONTROL_MOTOR:
      system_state_debug = "MANUAL_CONTROL_MOTOR";
      break;

    case STATE_SYSTEM_READ_SAMPLE:
      system_state_debug = "READ_SAMPLE";
      break;

    case STATE_SYSTEM_PROCESS_SAMPLE:
      system_state_debug = "PROCESS_SAMPLE";
      break;

    default:
      break;
  }
```

```
}
else if (mode_system == MODE_SYSTEM_AUTO)
{
  mode_system_debug = "AUTOMATIC";

  switch (state_system)
  {
    // TODO: Bluetooth input state.

    // TODO: Bluetooth output state.

    case STATE_SYSTEM_DISPLAY_FREQUENCY:
      system_state_debug = "DISPLAY_FREQUENCY";
      break;

    case STATE_SYSTEM_AUTO_CONTROL_MOTOR:
      system_state_debug = "AUTO_CONTROL_MOTOR";
      break;

    case STATE_SYSTEM_READ_SAMPLE:
      system_state_debug = "READ_SAMPLE";
      break;

    case STATE_SYSTEM_PROCESS_SAMPLE:
      system_state_debug = "PROCESS_SAMPLE";
      break;

    default:
      break;
    }
  }

  Serial.print("MODE_SYSTEM: ");
  Serial.println(mode_system_debug);
  delay(200);
  Serial.print("SYSTEM_STATE: ");
  Serial.println(system_state_debug);
  delay(200);
  Serial.print("MODE_DISPLAY: ");
  Serial.println(mode_display_debug);
  delay(200);
  Serial.print("DISPLAY_STATE: ");
  Serial.println(display_state_debug);
  delay(200);
  Serial.println();
}

// This function will be called with the desired frequency
// start the measurement
void timer_callback(void)
{
  adc->startSingleRead(PIN_SAMPLE);
}

void adc0_isr()
{
  uint8_t pin = ADC::sc1a2channelADC0[ADC0_SC1A&ADC_SC1A_CHANNELS]; // the bits 0-4 of
ADC0_SC1A have the channel
```

```
//memory_buffer->write(adc->readSingle());

// add value to correct buffer
if(pin == PIN_SAMPLE && !memory_buffer->isFull() && !memory_buffer_copied)
{
  memory_buffer->write(adc->readSingle());
}
else if (!memory_buffer_copied)
{
    for (sample_index = 0; sample_index < SAMPLE_SIZE; sample_index++)
    {
      sample_buffer[sample_index] = memory_buffer->read();
    }

    memory_buffer_copied = true;
}
else // clear interrupt anyway
{
  ADC0_RA;
}

// Restore ADC config if it was in use before being interrupted by the analog timer
if (adc->adc0->adcWasInUse)
{
  // Restore ADC config and restart conversion
  //adc->setResolution(adc->adc0->adc_config.res, ADC_0);  // Don't change res if not necessary
  ADC0_CFG1 = adc->adc0->adc_config.savedCFG1;
  ADC0_CFG2 = adc->adc0->adc_config.savedCFG2;
```

```
    ADC0_SC2 = adc->adc0->adc_config.savedSC2 & 0x7F;
    ADC0_SC3 = adc->adc0->adc_config.savedSC3 & 0xF;
    ADC0_SC1A = adc->adc0->adc_config.savedSC1A & 0x7F;
  }
}



void buttonUpRisingISR()
{
  if (state_system == STATE_SYSTEM_MANUAL_CONTROL_MOTOR)
  {
    motorTurningCW = true;
    return;
  }
}

void buttonUpFallingISR()
{
  if (state_system == STATE_SYSTEM_MANUAL_CONTROL_MOTOR)
  {
    motorTurningCW = false;
    return;
  }
}

void buttonUpLowISR()
{
  if (state_system == STATE_SYSTEM_MANUAL_CONTROL_MOTOR)
  {
    return;
  }

  if (state_display == STATE_DISPLAY_MENU_MAIN_SELECTION && mainMenuSelection != 1)
  {
    mainMenuSelection = 1;
    display.clearDisplay();
    display.setCursor(0, 0);
    display.println("Mode of Operation:");
    display.println("> 1. Manual Mode");
    display.println("  2. Automatic Mode");
    display.display();
    return;
  }

  if (state_display == STATE_DISPLAY_MENU_FREQ_SELECTION && target_frequency <
target_frequency_cap_high)
  {
    target_frequency++;
    display.clearDisplay();
    display.setCursor(0, 0);
    display.println("  TARGET FREQUENCY");
    display.println("---------------------");
    display.println();
    display.println();
    display.print("Target: ");
    display.print(target_frequency);
    display.println( " (Hz)");
    display.display();
    return;
  }
}

void buttonSelectRisingISR()
{
  if (state_display == STATE_DISPLAY_MENU_MAIN_SELECTION)
  {
    display.clearDisplay();
    display.display();

    if (mainMenuSelection == 1)
    {
      mode_system = MODE_SYSTEM_MANUAL;
      state_system = STATE_SYSTEM_READ_SAMPLE;
      mode_display = MODE_DISPLAY_MANUAL;
      return;
    }

    if (mainMenuSelection == 2)
    {
      state_system = STATE_SYSTEM_READ_SAMPLE;
      mode_display = MODE_DISPLAY_MENU_FREQ;
      state_display = STATE_DISPLAY_MENU_FREQ_SETUP;
      return;
    }
  }
  if (state_display == STATE_DISPLAY_MENU_FREQ_SELECTION)
  {
    mode_system = MODE_SYSTEM_AUTO;
    state_system = STATE_SYSTEM_READ_SAMPLE;
    mode_display = MODE_DISPLAY_AUTO;
    return;
  }

  if (mode_display == MODE_DISPLAY_AUTO || mode_display == MODE_DISPLAY_MANUAL)
  {
    if (mode_display == MODE_DISPLAY_MANUAL)
    {
      motorTurningCW = false;
      motorTurningCCW = false;
    }

    display.clearDisplay();
    display.display();
    mode_system = MODE_SYSTEM_DISPLAY;
    state_system = STATE_SYSTEM_READ_SAMPLE;
    mode_display = MODE_DISPLAY_MENU_MAIN;
    state_display = STATE_DISPLAY_MENU_MAIN_SETUP;
    return;
  }
}

/*
void buttonSelectLowISR()
{
  if (state_system == STATE_SYSTEM_MANUAL_CONTROL_MOTOR || state_system ==
STATE_SYSTEM_AUTO_CONTROL_MOTOR)
  {
    return;
  }
}
*/

void buttonDownRisingISR()
{
  if (state_system == STATE_SYSTEM_MANUAL_CONTROL_MOTOR)
  {
    motorTurningCCW = true;
    return;
  }
}

void buttonDownFallingISR()
{
  if (state_system == STATE_SYSTEM_MANUAL_CONTROL_MOTOR)
  {
    motorTurningCCW = false;
    return;
  }
}

void buttonDownLowISR()
{
  if (state_system == STATE_SYSTEM_MANUAL_CONTROL_MOTOR)
  {
    return;
  }

  if (state_display == STATE_DISPLAY_MENU_MAIN_SELECTION && mainMenuSelection != 2)
  {
    mainMenuSelection = 2;
    display.clearDisplay();
    display.setCursor(0, 0);
    display.println("Mode of Operation:");
    display.println("  1. Manual Mode");
    display.println("> 2. Automatic Mode");
    display.display();
    return;
  }

  if (state_display == STATE_DISPLAY_MENU_FREQ_SELECTION && target_frequency > target_frequency_cap_low)
  {
    target_frequency--;
    display.clearDisplay();
    display.setCursor(0, 0);
    display.println("  TARGET FREQUENCY");
    display.println("---------------------");
    display.println();
    display.println();
    display.print("Target: ");
    display.print(target_frequency);
    display.println( " (Hz)");
    display.display();
    return;
  }
}
```