Aliya Chambless, Tyler Jacobson, Stephen Sun
Prof. R. Iris Bahar
ENGN 1931I: Design of Robotic Systems
7 April 2021

Project 3: Signal Processing Report

1. A well-commented version of the Arduino sketches used for demoing your project (with appropriate citations). Please upload your .ino file(s) in addition to putting your code in your report Appendix.

Refer to appendix 1.

2. A discussion of your design concept and its evolution and how it is meant to integrate into your final project.

The concept for this project was based on a problem that many city-dwellers have. These residents want their window shades closed at night to help them fall asleep, preventing city lights from blaring in their room at night, yet want to be woken up with natural light in the morning. While raising and lowering the blinds every night manually could work, this still does not allow the resident to be woken up by natural light. Our original concept was a pair of “smart shades” in which a photosensor alone could lower and raise shades and a voice-command system could be used to change the position of the blinds hands-free. We decided to conduct user and precedent research to identify what problems users had with their current shades and what features they might like to see in such smart shades.

Our initial user research revealed that users wanted simplicity in using the shades. For example, the users that wanted to use the shade as an alarm did not want to have to set a second auditory alarm to make sure they woke up. To address this, we added sound capability to the alarm sequence. We also learned that users use blinds for the insulation benefits, but still wanted to enjoy natural light when they were using the room. This led to the idea of adding a motion sensor to track if a person is present in the room before closing the shades for insulation. We confirmed that voice control would be a useful addition to the product when the user is unable to get to the window, which could be for various reasons, from a zoom call to mobility issues.

For this project, we wanted to test the functionality of a minimum viable voice recognition and motor control software. This includes voice recognition that could rotate a stepper motor, rotate a stepper motor for a certain time period then rotate it back in the opposite direction, and set a timer that would rotate the motor and sound an alarm upon completion. The voice control software that we used was BitVoicer Server. This program makes use of Windows’ speech recognition and speech synthesis tools to run speech recognition on recorded audio clips. It can also read out phrases out loud that you set in the program. BitVoicer provides an Arduino library that allows the Arduino to interface with the program via the Serial port. The PC’s system microphone, or a microphone connected to the Arduino can be used for speech recognition. The library allows for two way communication, with first any audio from the Arduino being sent to BitVoicer Server, and then whatever data we set up to send on a voice command being recognized being transmitted back to the Arduino. In our case, we set up specific phrases that will trigger functions on the Arduino that each correspond to one function of the smart shades, for instance raising or lowering the blinds.

This stepper motor, together with the Arduino running the voice control code, will serve as the control system for the fully integrated "Helios" product. The stepper motor would be used to raise and lower the blinds and the alarm. A second servo motor can be used to control a tilt rod that rotates slats in the blinds to let in variable levels of light permeate into a room. Additional functionality can be easily layered into the control system as well. For example, motion sensors will be easily integrated into the final design such that when a person enters a room, the shades raise, and vice-versa.

3. A discussion of challenges encountered with the implementation of your design.

We based our alarm system on Ken Silverman's code, but he used a custom song format so we could not directly convert midi files to fit his code. We also observed that playing chords, or multiple frequencies at once, muddied the sound when played through a piezo or cheap buzzer. In order to play a loud and clear song, we converted notes using Max MSP's midiparse object and created our own timing. We also modified Ken's code to just play one frequency at a time.

The piezo was still pretty quiet, so we built a high pass filter circuit which made the sound more loud and clear. We had limited capacitor and resistor options, so we ended up using a 560 Ω resistor and a 100 microfarad capacitor. Following the formula $1/2\pi RC$we tried to raise the cutoff frequency slightly, but the only other capacitor we had was 100 picofarads, which would require a resistor in the hundreds of MΩ to get close to the current cutoff frequency.

The biggest area that we got stuck on was in the integration of the voice control and stepper code with the alarm audio processing code. We worked on these two areas separately, but when it came time to combine them together, we discovered that there was a big problem with getting the speaker to play sounds while the stepper motor was running. Once the stepper motor is initialized by creating a stepper object, the speaker's sound becomes extremely distorted and very quiet.

Apart from testing the original Arduino stepper library, we tested 2 third party libraries: AccelStepper as well as StepMotor. While both libraries claim to prevent blocking of the CPU, both libraries still caused distorted sound, and both libraries had other limitations. AccelStepper did not have a straightforward function that allows us to set how many steps to move. Instead, it requires a setting of a reference zero position and then the user provides target positions that are relative to the zero position. While this would allow for more precise control of the stepper motor, in our application we do not require this sort of precision, and simply setting the number of revolutions to turn to fully raise or lower our blinds should be sufficient. With StepMotor, the library makes use of interrupts on Timer 1 or 2 to move the stepper motor. Since the audio processing code uses Timer 2, we theoretically could use Timer 1 for controlling the stepper motor. However, even if we do use Timer 1, we are still running 2 interrupts on the Arduino at the same time. If 1 interrupt is triggered and we are running the interrupt service routine, any triggering of the other interrupt will be missed. In our testing, this caused the stepper motor to move, and no sound to be emitted from the speaker at all.

While we did consider changing to a DC motor, a stepper motor would still be best for our application, since stepper motors can provide much higher holding torque. A potential solution to our issue will be to have 2 Arduinos in communication, either through I2C or SPI, or even wirelessly with a wireless communication module. One Arduino will be involved in handling the voice and stepper motor control, while the other Arduino will run the interrupt based alarm code. The Arduino connected to voice control can send a packet to the other Arduino telling it how long of a duration to wait before triggering the alarm. When the alarm is triggered the second Arduino can tell the first Arduino to raise or lower the blinds as required, perhaps to let in natural light to help the user wake up.

4. A brief discussion of how the work was divided among team members.

Stephen worked mainly on getting the voice control program to work, and integrating it with the Arduino code. Aliya worked on the user research and modified the provided audio example code to work with a custom tune. Tyler worked on testing stepper and DC motors for use with our system, and also worked on the presentation.

5. Suggestions for improvements for this assignment (in future years).

We were initially confused about the importance and benefits of timer interrupts, and if time permits we thought that an individual signal processing lab would have been helpful between the last two projects.

```
#include <Stepper.h>
#include <BVSP.h>
#define STEPS 2038 // the number of steps in one revolution of your motor
(28BYJ-48)

// define stepper motor - connected to pin 5-8
Stepper stepper(STEPS, 5, 7, 6, 8);

// Defines the constants that will be passed as parameters to
// the BVSP.begin function
const unsigned long STATUS_REQUEST_TIMEOUT = 1000;
const unsigned long STATUS_REQUEST_INTERVAL = 2000;

// Defines the size of the receive buffer
const int RECEIVE_BUFFER_SIZE = 2;

// variable for tracking the time
unsigned long timerStart = 0;
unsigned long alarmStart = 0;

// Takes 10s for stepper to rotate 2 revolutions at 12rpm
const int blindsRaiseLowerTime = 10000;

// wait time for how long blinds should be lowered for
int waitTime = 0;
int alarmTime = 0;

// Initializes a new global instance of the BVSP class
BVSP bvsp = BVSP();

// buffer for receiving data from BitVoicer defined
// Just a byte array
byte receiveBuffer[RECEIVE_BUFFER_SIZE];

// assume blinds are raised at boot
boolean blindsLowered = false;

/// disable alarm at the start
boolean soundEnabled = false;


static const uint8_t wavlut[256] =
{
  0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, //sawtooth
wave
  16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31,
  32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47,
  48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63,
  64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79,
  80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95,
  96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111,
  112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126,
127,
  128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142,
143,
  144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158,
159,
  160, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174,
175,
  176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 190,
191,
  192, 193, 194, 195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 205, 206,
207,
  208, 209, 210, 211, 212, 213, 214, 215, 216, 217, 218, 219, 220, 221, 222,
223,
  224, 225, 226, 227, 228, 229, 230, 231, 232, 233, 234, 235, 236, 237, 238,
239,
  240, 241, 242, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252, 253, 254,
255,

};


static const PROGMEM char mysong_snd[] =
{
  // numbers correspond to keys on a pianos (MIDI note shifted down )
  //beach.snd:
  0, 8,  77, 4,  79, 4,  81, 4,
  84, 12,  77, 8,  76, 8,  86, 8
};

static const unsigned int freq[74] = //convert MIDI-like note values to
frequencies (proportional to Hertz)
{
  0,
  523,  554,  587,  622,  659,  698,  740,  784,  831,  880,  932,  988,
  1047, 1109, 1175, 1245, 1319, 1397, 1480, 1568, 1661, 1760, 1865, 1976,
  2093, 2217, 2349, 2489, 2637, 2794, 2960, 3136, 3322, 3520, 3729, 3951,
  4186, 4435, 4699, 4978, 5274, 5588, 5920, 6272, 6645, 7040, 7459, 7902,
  8372, 8870, 9397, 9956, 10548, 11175, 11840, 12544, 13290, 14080, 14917,
15804,
  16744, 17740, 18795, 19912, 21096, 22351, 23680, 25088, 26580, 28160, 29834,
31609,
  33488,
};

#define NCHANS 4
static volatile uint16_t sampleIdx;
static volatile uint16_t sampleInc;

int speakerPin = 11;  // using Timer2, interrupts are associated with pins 3
and 11
// pin 11 = OC2A,  pin 3 = OC2B


void setup() {
  // Starts serial communication at 115200 baudrate
  Serial.begin(115200);
  // Sets the Arduino serial port that will be used for
  // communication, how long it will take before a status request
  // times out and how often status requests should be sent to
  // BitVoicer Server.
  bvsp.begin(Serial, STATUS_REQUEST_TIMEOUT,
            STATUS_REQUEST_INTERVAL);

  // Defines the function that will handle the frameReceived
  // event which is when data is received from bitvoicer
  bvsp.frameReceived = BVSP_frameReceived;
  // enable the builtin led and the speaker pin for output
  pinMode(LED_BUILTIN, OUTPUT);
  pinMode(speakerPin, OUTPUT);

  // enable interrupts
  interrupts();


  //Timer 2 setup
  TCCR2A =   (1 << COM2A1) | (1 << COM2A0) //COM2A: controls OC2A/Arduino pin
11 output mode
            | (1 << COM2B1) | (1 << COM2B0)   //COM2B: controls OC2B/Arduino
pin 3 output mode
            |           (0 << WGM21) | (1 << WGM20); //WGM: Phase correct PWM
(doubles period)
  TCCR2B = (0 << WGM22)
            | (1 << CS20);                 //Timer2:prescaler=1
  TIMSK2 = (1 << TOIE2);                   //Enable Timer2 Overflow Interrupt
  TIFR2  = (1 << TOV2);                    //Clear TOV2 / clear pending
interrupts


}

#define NOTETIME 48   // smallest play time for a note
unsigned long previousMillis = 0;
unsigned int chordTime =  NOTETIME;
int songIndex = 0;
int j = 512;
uint8_t pauseFlag = 0;
uint8_t filterFlag = 0;



// Function to raise the blinds, currently rotates 1 revolution at 12rpm (5s
per revolution)
void raiseBlinds() {
  // Check if blinds are lowered, if blinds are already up do nothing
  if (blindsLowered) {
      stepper.setSpeed(12);
      stepper.step(STEPS);
      blindsLowered = false;
  }
}
// Function to lower the blinds, currently rotates 1 revolution at 12rpm (5s
per revolution)
void lowerBlinds() {
  // Check if blinds are not lowered. If blinds are already down do nothing
  if (!blindsLowered) {
      stepper.setSpeed(12);
      stepper.step(-STEPS);
      blindsLowered = true;
  }
}

// Function that lowers blinds for certain amount of time. Gets the current
time
// and stores it in timerStart
void lowerBlindsTime() {
  digitalWrite(LED_BUILTIN, HIGH);
  stepper.setSpeed(12);
  stepper.step(-STEPS * 2);
  blindsLowered = true;
  timerStart = millis();
}


// Function that raises blinds for certain amount of time. Sets timerStart to
0
// which stops the code from checking if its time to raise blinds
void raiseBlindsTime() {
  digitalWrite(LED_BUILTIN, LOW);
  stepper.setSpeed(12);
  stepper.step(STEPS * 2);
  blindsLowered = false;
  timerStart = 0;
}
// Function to enable the alarm. Sets alarmStart to 0
// which stops the code from checking if its time to ring alarm
void enableAlarm() {
  digitalWrite(LED_BUILTIN, HIGH);
  soundEnabled = true;
  alarmStart = 0;
}


void loop() {
  // Checks if the status request interval has elapsed and if it
  // has, sends a status request to BitVoicer Server
  bvsp.keepAlive();
  // Checks if there is data available at the serial port buffer
  // and processes its content according to the specifications
  // of the BitVoicer Server Protocol
  bvsp.receive();
  // If the timer was started, check if its time to raise the blinds
  if (timerStart != 0) {
      if (millis() >= timerStart + blindsRaiseLowerTime + waitTime) {
      raiseBlindsTime();
      }
  }
  // If alarm was started, check if its time to ring the alarm
  if (alarmStart != 0) {
      if (millis() >= alarmStart + alarmTime) {
      enableAlarm();
      }
  }
  // Code for processing sound
  unsigned char ch;
  unsigned long currentMillis = millis();   // check if it's time to update
chord
  int f0, dl;
  unsigned long deltaTime;

  deltaTime = currentMillis - previousMillis;
  if (!pauseFlag && (deltaTime >= NOTETIME))
  {
      // check if new command from keyboard was received and update waveform.
      // Serial.available waits for ENTER before updating waveform parameters.
      if (Serial.available())
      {
      ch = Serial.read();
      if (ch == '[') {
      j -= 16;
      Serial.println("decrease pitch");
      }
      else if (ch == ']') {
      j += 16;
      Serial.println("increase pitch");
      }
      else if (ch == 'f') {
      filterFlag = !filterFlag;
      Serial.print("FIR filter =");
      Serial.println(filterFlag);
      }
      else if ((ch == 10) || (ch == 13)) {
      Serial.print("."); // ASCII 10 and 13 are LF and CR
      }
      else if (ch == 't') {
      if (!soundEnabled) {
      soundEnabled = true;
      Serial.println("sound enabled");
      }
      else if (soundEnabled) {
      soundEnabled = false;
      Serial.println("sound disabled");
      }
      }
      else Serial.println("bad char");
      }


      // read 4 notes (that make up chord) plus note time from program memory
      // read the MIDI notes from the songlist (4 notes in 1 chord)
      f0 = pgm_read_byte(&mysong_snd[songIndex  ]);
      dl = pgm_read_byte(&mysong_snd[songIndex + 1]);

      chordTime = dl * NOTETIME;
      songIndex = (songIndex + 2) % sizeof(mysong_snd);  // start reading
songs from beginning when done
      pauseFlag = 1;

      // convert note to frequency and adjust for pitch (shift by 12 to
account for scaled integer and precision of table
      // indicates how fast you should be traversing the waveform array.  A
high frequency sound means you need to take large
      // steps (increments) through the waveform array.  A low frequency sound
means you need to take small step through the waveform.
      sampleInc = (freq[f0] * (long)j) >> 12;
  }

  if (deltaTime >= (NOTETIME + chordTime))
  {
      //force slight pause after each note
      sampleInc = 0;

      pauseFlag = 0;
      previousMillis = currentMillis;
  }
}

// Handles the frameReceived event
// BitVoicer supports payloads up to 1023 bytes - payloadSize is passed from
// BitVoicer to Arduino as a parameter here, along with dataType
void BVSP_frameReceived(byte dataType, int payloadSize)
{
  // Checks if the received frame contains binary data
  // 0x01 = Binary data (byte array)
  // Can check for and use other data types here
  if (dataType == DATA_TYPE_BINARY)
  {
      // If 2 bytes were received, process the command.
      // getReceivedBytes returns the no. of bytes copied ot the buffer
      if (bvsp.getReceivedBytes(receiveBuffer, RECEIVE_BUFFER_SIZE) ==
      RECEIVE_BUFFER_SIZE)
      {
      // If first byte is 1, raise blinds
      if (receiveBuffer[0] == 0x01) {
      digitalWrite(LED_BUILTIN, LOW);
      raiseBlinds();
      }
      // If first byte is 2, lower blinds
      else if (receiveBuffer[0] == 0x02) {
      digitalWrite(LED_BUILTIN, HIGH);
      lowerBlinds();
      }
      // If first byte is 3, lower the blinds for certain time
      else if (receiveBuffer[0] == 0x03) {
      // Retrieve the wait time from the 2nd byte
      waitTime = (int) receiveBuffer[1];
      if (!blindsLowered) {
      lowerBlindsTime();
      }
      // If the first byte is 3, set alarm
      else if (receiveBuffer[0] == 0x04) {
      // record the time that we started the alarm
      // Once current time >= alarmStart + alarmTime,
      // its time to trigger the alarm
      alarmStart = millis();
      // Retrieve the alarm time from 2nd byte
      alarmTime = (int) receiveBuffer[1];
      }
      }
      }
  }
}

#define FILTERTAPS 8          // The number of taps in the FIR filter
uint8_t values[FILTERTAPS] = {0, 0, 0, 0, 0, 0, 0, 0}; // to have a nice start
up, fill the array with 0's

// Interrupt service routine
// interrupt every 512 cycles (16,000,000/512Hz=31250)
ISR (TIMER2_OVF_vect)
{
  if (soundEnabled) {
      uint8_t temp;
      uint8_t out;
      static uint8_t k = 0; // pointer to circular buffer

      // the interrupt is at a fix time period and outputs a specific voltage
level for the speaker.  This voltage represents the addition of 4
      // waveforms (notes) at 4 different frequencies.
      // The voltage is interpreted as a PWM duty cycle on the OC2A pin (pin
11).
      // It is the same waveform (e.g., sine wave), sampled across a full
period repeatedly, but
      // at a sampling rate that corresponds to the frequency of the
particular note.
      values[k] = (wavlut[(sampleIdx >> 8)] >> 2);

      // OCR2A is proportional to the output voltage.  It represents the sum
of the 4 waves at an instant for this sample

      OCR2A = values[k];

      // updated index into waveform lookup table for 4 notes of the chord
      sampleIdx += sampleInc;
  }
}
```