E91B Final Project

Arduino-Based Controller for DJ Software

Julian Leland
E91B, 2010
Swarthmore College

ABSTRACT

A hardware controller, based around the Arduino Mega microcontroller (Atmel ATMega 1280 microprocessor), was designed for use with Algoriddim GmbH's dJay, a consumer-level computer DJ application. It was designed to interface with dJay through MIDI over USB, and was

Control of the following interface controls was accomplished: turntables, crossfader, volume, play/pause, seek. Control latency was found to be extremely high, on the order of 103 ms. Major problems encountered during the project included difficulties with capacitive touch sensing technologies, and extremely poor documentation. Avenues for future work are discussed.

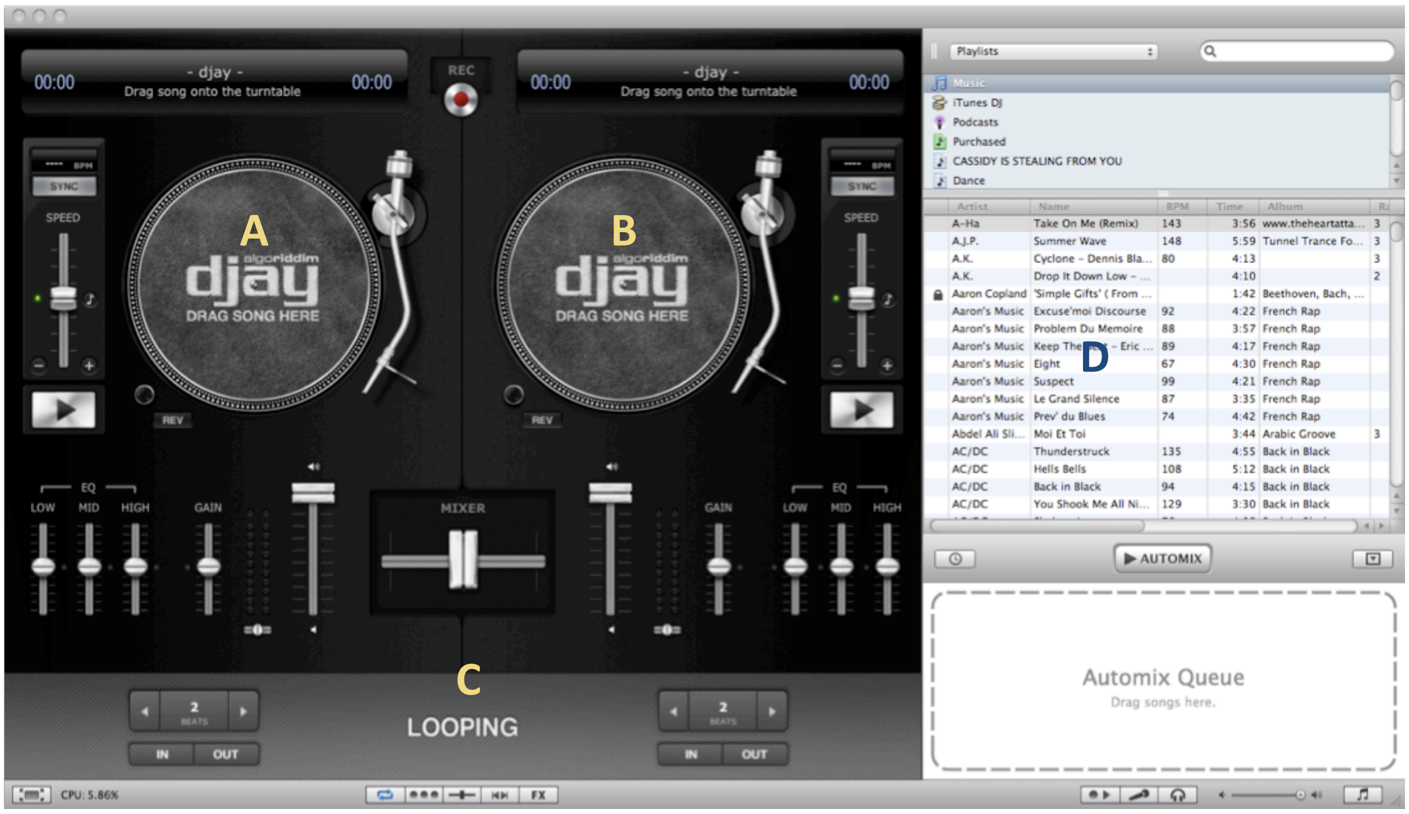
Table of Contents

Introduction

Among the many applications for microcontroller technology, one of the most compelling is software interfacing. With hardware for both digital and analog input available on the microcontrollers themselves, and a multitude of well-documented sensors easily available to the hobbyist, the ability of even simple, inexpensive microcontrollers to accurately record and measure physical events is tremendous. Recently, this has been supplemented by the development of user-friendly microcontroller platforms such as the Arduino, which decrease the difficulty of both microcontroller programming and using microcontroller inputs to control software on a host computer significantly. Ultimately, through development of simple input-output systems using a microcontroller, extremely complex tasks can be accomplished by fusing real-world sensing with the flexibility and power of software through microcontrollers.

In this project, a microcontroller was used to create a physical interface for Algoriddim GmbH's dJay. dJay is a consumer-level virtual DJ package which sources its music directly from the user's iTunes library rather than from proprietary libraries. Although not nearly as sophisticated as professional packages such as Native Instruments' Traktor, its simple interface and relatively broad feature set make it an ideal choice for casual users and amateur DJs. Control of most program functions is available through keyboard shortcuts and/or mouse control. However, controlling the program in this manner is often extremely inefficient, and for tasks such as scratching, too imprecise to be useful. Many external hardware controllers are available for precisely this reason, but they are extremely expensive, and are limited in their flexibility. Consequently, a custom-designed controller for dJay would be an economical and versatile alternative to a commercial controller, would provide an excellent opportunity to learn more about hardware-software interfacing, and hopefully would contribute to the awesomeness of future Engineering parties.

Upon opening dJay, the user is presented with the following screen:



Artist	Name	BPM	Time	Album	Ra
A-Ha	Take On Me (Remix)	143	3:56	www.theheartatta...	3
A.J.P.	Summer Wave	148	5:59	Tunnel Trance Fo...	3
A.K.	Cyclone – Dennis Bla...	80	4:13		3
A.K.	Drop It Down Low – ...		4:10		2
Aaron Copland	'Simple Gifts' (From ...		1:42	Beethoven, Bach, ...	
Aaron's Music	Excuse'moi Discourse	92	4:22	French Rap	
Aaron's Music	Problem Du Memoire	88	3:57	French Rap	
Aaron's Music	Keep The[illegible] – Eric ...	89	4:17	French Rap	
Aaron's Music	Eight	67	4:30	French Rap	
Aaron's Music	Suspect	99	4:21	French Rap	
Aaron's Music	Le Grand Silence	87	3:35	French Rap	
Aaron's Music	Prev' du Blues	74	4:42	French Rap	
Abdel Ali Sli...	Moi Et Toi		3:44	Arabic Groove	3
AC/DC	Thunderstruck	135	4:55	Back in Black	
AC/DC	Hells Bells	108	5:12	Back in Black	
AC/DC	Back in Black	94	4:15	Back in Black	
AC/DC	You Shook Me All Ni...	129	3:30	Back in Black	

Figure 1 - dJay Interface

The dJay interface is comprised of two turntable sections (A and B), a mixer section consisting of a crossfader and a variety of effects available through the panel at the bottom of the window (C), and a music library window (D). Additionally, a window for playing samples is available.

The goals of this project were to implement external hardware control of the following functions:

- Turntables: Successful motion control of turntables, including scratching, seeking (quickly fast-forwarding/rewinding) and forward/backspin[1], was to be implemented. Additionally, touch-sensitive electronics were to be used to

[1] When using the mouse to control the turntables, a rapid "flick" of the turntable will cause forwardspin/backspin, wherein the song advances or rewinds rapidly as with a real turntable

control turntable activity, allowing turntable control only when the turntables were touched. This was intended to replicate the functionality of an actual turntable, where the turntable operates freely until the user touches the spinning record to scratch/etc.

- Fader Section: A functioning crossfader, as well as controls for dJay's auto-crossfade feature (which automatically transitions between the right and left turntable at a button press), were to be implemented.
- Play/Pause/Reverse Controls: Controls for the play/pause and reverse buttons were to be implemented, preferably with some form of status indicator.
- Volume, Pitch and EQ Controls: Control of the volume and pitch controls for each channel, as well as the EQ sections, was to be implemented.
- Time permitting, control of other functions, such as cue point controls, looping controls, or the sample player, would be implemented.

The microcontroller used in this project was an Arduino Mega 1280. The Mega is based around the Atmel ATMega 1280 microprocessor, and has 54 digital input/output pins and 16 analog pins. It is designed to be programmed in the Arduino coding environment, which greatly simplifies many microcontroller programming tasks, albeit at the cost of reduced performance.

Communication with dJay was to be accomplished through the MIDI protocol, which dJay natively supports. Although the Arduino is capable of outputting MIDI data, it cannot be trivially configured to present to the computer as a MIDI USB device: consequently, software on the host computer would be required to successfully direct the Arduino's MIDI messages to dJay.

Technical Background

Two simple technologies form the backbone of this project: the MIDI protocol, which is used to transmit messages to dJay, and quadrature encoders, which are use to encode turntable motion.

The MIDI Protocol

The Musical Instrument Digital Interface (MIDI) protocol is a serial data transmission protocol designed for use with musical instruments. It was originally defined in 1982, and has not changed drastically since then.

Most MIDI messages are sent in 3-byte packets: for example, the message 0x90 0x82 0xFF is a complete MIDI message. The individual bytes perform the following functions:

- Byte 1: The first byte, known as the Status byte, controls the type of command sent (the high nibble) and the channel (effectively, the instrument) that the command is sent to. Valid command types include Note On, Note Off, and Controller Change (for control of sliders, pitchwheels and other effects) among others. Sixteen channels are available under the MIDI protocol. In the example given, Byte 1 consists of a Note On command (the 9) sent to Channel 1 (the 0).
- Byte 2: The second byte, known as the Note Number or Controller Number, controls which note or control is being actuated (depending on what type of command has been sent). In the example given, the command indicates that Note 130 should be played.
- Byte 3: The third byte, known as the Velocity or Controller Value, controls the value used by the actuated control. In the case of a Note On command, this affects the velocity with which the note is played: in the case of a Controller Change command, this affects the value that the control is set to. In the example given, the note being played is played with maximum velocity (127).

Quadrature Encoders

In order to determine turntable motion, rotary encoders were used. These encoders output a quadrature signal, in which two square waves are output 90º out of phase from each other, as shown in Figure 2.

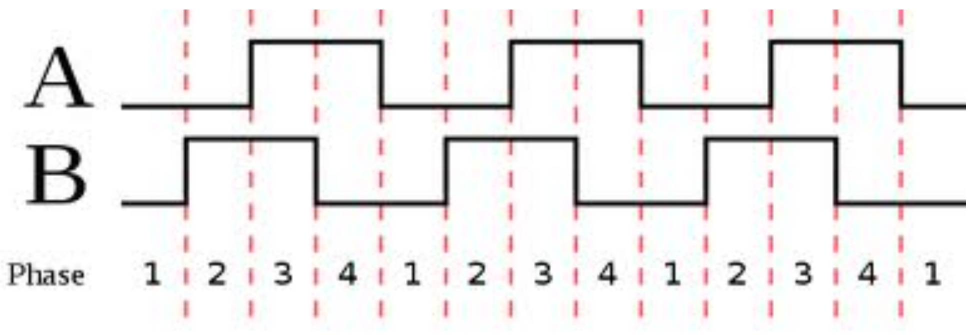


Figure 2 - Quadrature Output Signals
(From http://en.wikipedia.org/wiki/File:Quadrature_Diagram.svg)

Using quadrature signals, both rate of rotation (through period) as well as direction of rotation (through relative phase) can be determined.

Project Design

The components of this project loosely fall into “interface” and “software” categories, as Figure 3 shows.

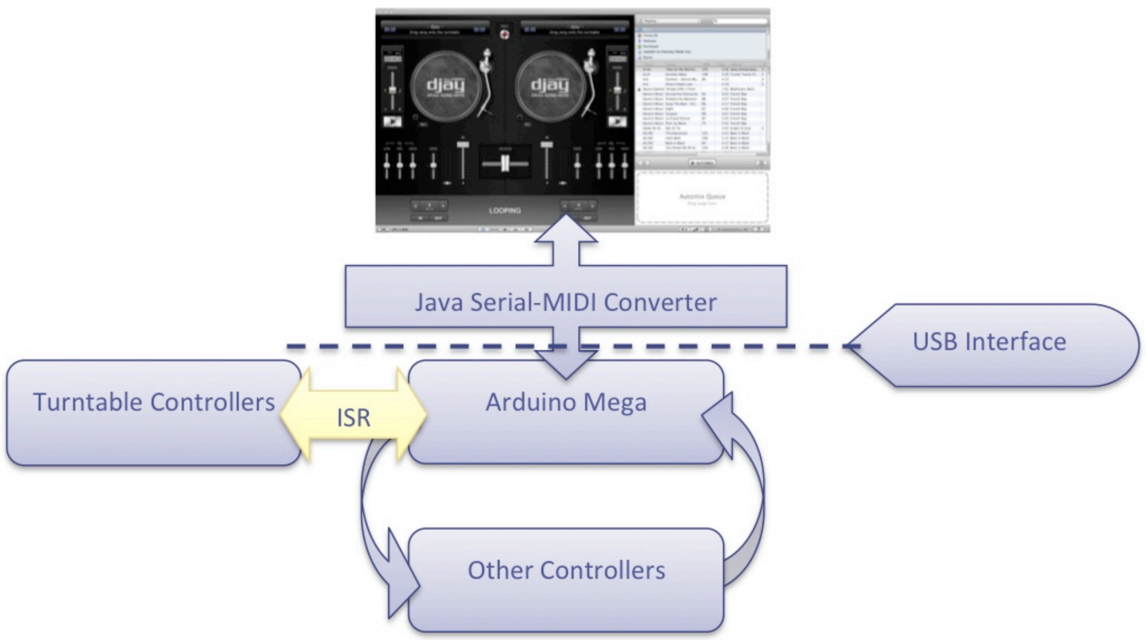
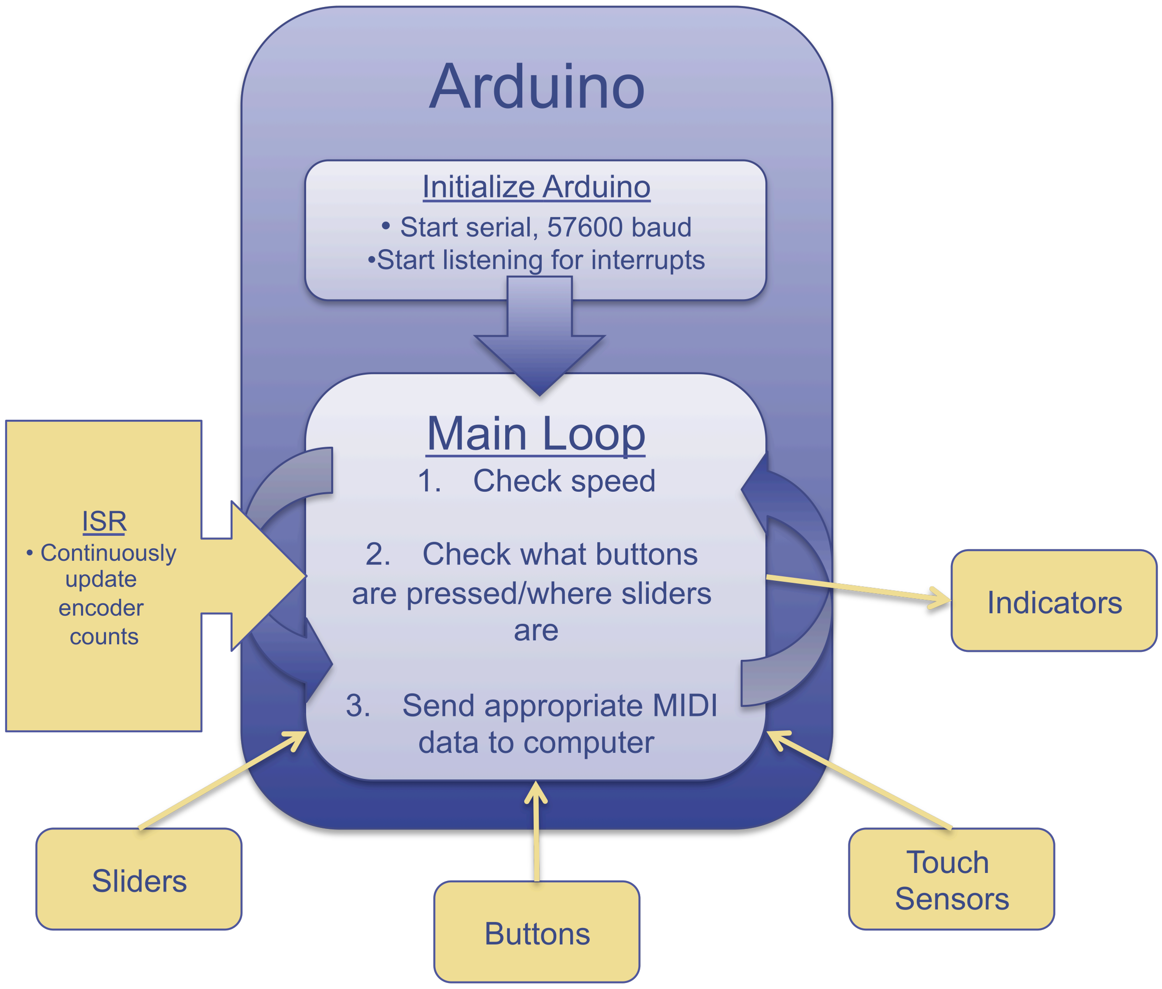


Figure 3 - Project Overview

The “software” category comprises dJay as well as serial-to-MIDI converter software running on the host computer. The serial-to-MIDI converter used in this project was Mark Demer’s Serial _MIDI_Converter_V2D, available at spikenzielabs.com [1]. This software was found early on in the project, and despite some shortcomings, proved to be sufficiently functional to allow completion of the project.

The "interface" category comprises the Arduino and the control electronics, as well as the software running on the Arduino. The following block diagram shows the relationship between these components:



Arduino
Initialize Arduino
• Start serial, 57600 baud
•Start listening for interrupts
Main Loop
1. Check speed
2. Check what buttons are pressed/where sliders are
3. Send appropriate MIDI data to computer
ISR
• Continuously update encoder counts
Indicators
Sliders
Buttons
Touch Sensors

Figure 4 - Interface Block Diagram

Each section of the interface as delineated by the block diagram is further explained below, with discussion of both hardware and software components.

Circuit diagrams have not been included due to the simplicity of the circuits: most circuits are either simple buttons connected to digital pins, or potentiometers connected to analog pins. Power for the interface is provided through USB.

Arduino Initialization

Upon initialization, variables and constants for each channel are defined. The Arduino is set to communicate over serial at 57600 baud: although this is higher than specified by the MIDI protocol (which specifies 31250 baud), it does not appear to adversely affect performance, and higher baud rates are believed to be preferable to reduce control latency. Once baud rate has been set, interrupts are enabled (using the attachInterrupt() command) on digital pins 18 through 21. These interrupts are configured to be triggered by either rising or falling edges, and are used to read the position of the rotary encoders controlling the turntables.

Interrupt Service Routine

The interrupts that have just been attached to pins 18 – 21 are now active, but need interrupt service routines (ISRs) to be able to read the rotary encoders' position. The rotary encoders used are low-friction Bourns optical encoders outputting 400 edges per revolution: consequently, the frequency of interrupts can be quite high, and the ISRs must necessarily be kept simple to minimize interference with the rest of the code.

Each encoder uses two ISRs, one for each output channel: there are two encoders (Channel A and Channel B), giving four interrupt service routines. The general structure of an ISR is as follows:

```
void ChanA1(){
  if (digitalRead(18) == digitalRead(19)) { // When an interrupt is triggered, check the current state of the
                                            // two channels
    ++encAPos;                              // If they're equal, then rotation is CW
  }
  else {
    --encAPos;                              // Otherwise, rotation is CCW
  }
}
void ChanA2(){
  if (digitalRead(18) == digitalRead(19)) {  // If an interrupt is triggered on the other channel, check the
                                             // current state of the two channels
    --encAPos;                               // Since we're considering the other channel, if they're equal,
  }                                          // then rotation is CCW
  else {
    ++encAPos;                              // Otherwise, rotation is CW
  }
}
```

The ISR simply encodes the direction of rotation as an increase or decrease in position. Currently, no error checking is implemented to ensure that the encoder does not overflow the encPos variable: however, this has yet to be encountered, and is unlikely to ever be encountered (it would require 82 consecutive rotations in the same direction).

Main Routine

While the ISRs continue to run in the background, the main routine begins. The main routine runs indefinitely, until the system is powered down. Three primary tasks are accomplished in the main loop: measuring the current speed of the rotary encoder, determining the positions or states of the other control surfaces, and sending this information as MIDI data over USB to the computer. These tasks are accomplished identically for both channels.

Encoder Speed

Encoder speed is determined by reading encoder position, waiting a specified amount of time, and then reading encoder position again. The difference in position gives the effective rotational speed of the encoder. This is accomplished with the following code:

```
void checkSpeed(volatile int *encPos, int *curSpeed) { // Check current speed. Input is encoder position of
                                                        // channel being checked: output is current speed.
  int rawSpeed = 0;
  firstPos = *encPos;                                   // Read encoder position
  delay(25);                                            // Wait 25 ms
  secondPos = *encPos;                                  // Read encoder position again
  rawSpeed = (secondPos - firstPos);                    // Speed is difference between positions

  if (rawSpeed > 0) {
    *curSpeed = map(rawSpeed, 1, 200, 65, 127); // If turning CW, map speed into range from 65 to 127
  }                                             // These speed values correspond to the permissible range
                                                // of MIDI note values
  else if (rawSpeed < 0) {
    *curSpeed = map(rawSpeed, -1, -200, 63, 0); // If turning CCW, map speed into range from 63 to 0
  }
  else *curSpeed = 64;                          // Otherwise, speed = 64.
}
```

The measured speed is then mapped from the possible range of input values (measured to be roughly between -200 and +200) to a range of valid MIDI note velocity values (0 to 127) using the Arduino map() command. A rotational speed of zero corresponds to a MIDI value of 64.

Button State

Buttons are defined as having four distinct states: just pressed, pressed, just released, and released. These states are encoded using two variables, state and chgState. This is demonstrated in the following code, used to measure the current status of the capacitive touch sensor:

```
void checkTouch(int readChan, int *touch, int *chgTouch) { // Check touch sensor status. Input is channel
                                                            // being read, output is state and chgState variables
  if (analogRead(readChan) <= 200 && *touch == 0) {  // The capacitive sensor outputs roughly 2.5 V, so
                                                            // it is read with an analog pin
    *touch = 1;        // If we weren't touched before, but we are now, change touch = 1 and chgTouch = 1
    *chgTouch = 1;
  }
  else if(analogRead(readChan) <= 200 && *touch == 1) {
    *chgTouch = 0;  // If we were touched before, and are still touched, change chgTouch = 0, touch still = 1
  }
  else if(analogRead(readChan) > 200 && *touch == 1) {
    *touch = 0;       // If we were touched before but aren't now, change touch = 0, chgTouch = 1
    *chgTouch = 1;
  }
  else {
    *chgTouch = 0; // If we were not touched before, and still aren't touched, both variables = 0
  }
}
```

Although the above code reads analog inputs, this code is trivially rewritten to work with digital inputs. Encoding both the current state and whether or not the state has changed is important for controls which require that both a Note On and Note Off message (sent only when chgState is high), such as the touch sensor controls. Additionally, having a button’s state fully encoded simplifies programming tasks such as having status lights activate when a control is active (for example, the play/pause buttons).

Among the most important buttons in the project are the capacitive touch sensors used to activate the turntables and the seek controls. In order to activate both precise turntable control (scratching) as well as seek control without requiring use of two hands/removal of the hand from the turntable, the turntable was required to have two distinct touch sensitive surfaces. The turntable assembly was designed so that the center hub of the turntable (where it attaches to the encoder) was insulated from the turntable platter, which was a conductive disk. Conductive brushes were placed in contact with both surfaces, and connected to the input lines of two different QProx QT113 capacitive touch-sensing chips, as shown in Figure 5.

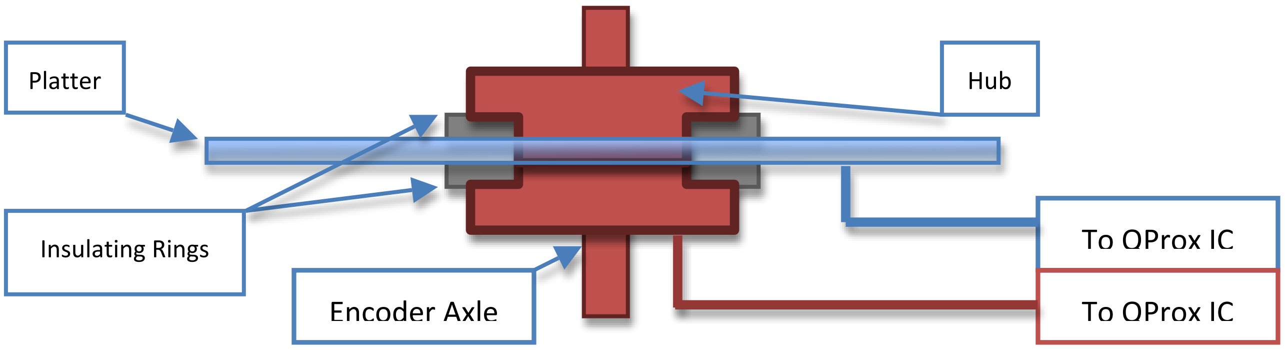


Figure 5 - Platter Design for Capacitive Sensing

This design successfully allowed independent sensing of touches on both surfaces, although not reliably.

Slider Position

Linear position measurement, needed for the crossfader, the volume controls, and other sliders and range-limited dials in the interface, is accomplished using resistive sensors. Generally, linear position is determined by measuring the voltage at the potentiometer wiper, mapping that voltage onto the range 0 – 127 using the map() command, and sending that value in the last byte of a controller change command. The following code is used with the crossfader: it both measures the current position and sends the appropriate MIDI message.

```
void faderPos() {
voltage = analogRead(4);                    // Read voltage on Pin 4
  curFader = map(voltage, 5, 550, 0, 127);  // Map voltage from permissible range onto 0 - 127 range
  Serial.print(0xB2, BYTE);                 // Controller Change, Channel 3
  Serial.print(0x08, BYTE);                 // Controller 0x08
  Serial.print(curFader, BYTE);                         // Controller position
}
```

It should be noted that this code completely disables software control of the crossfader by constantly reporting position: this is unfortunately necessary when using a mechanical slider. Although a mechanical slider can be used to control a software equivalent, the reverse is not currently possible: changes in the position of the on-screen slider cannot be transmitted to the mechanical slider. Consequently, if a slider's position is allowed to change in software, the next change in the mechanical slider's position will result in a "jump" in slider position due to the discontinuity between the software position and the position reported by hardware.

One alternative to mechanical sliders that was investigated was the SoftPot membrane linear potentiometer, manufactured by Spectra Symbol. The SoftPot uses a thin conductive film suspended (.006") above a thin resistive film. Touching the conductive film to the resistive film creates a voltage divider circuit, with the voltage measured at the conductive film being proportional to the position of the touching point along the length of the slider. When the sensor is released, the voltage at the top plate drops back to zero. Since there are no moving elements, this type of slider can avoid the position "jumps" described earlier by a) sending its position only when touched and b) having the user take care to always touch the slider at the position of its on-screen analogue. These sliders were used in volume control to great effect.

Sending MIDI

The last step in the main routine is to send MIDI data to the application. All MIDI messages sent were three-byte messages, and were either Note On, Note Off or Controller Change messages. Interestingly, most buttons in the dJay interface are treated as momentary switches: Note On commands are used both to turn them on and to turn them off.

Additionally, at this stage in the code, state indicators such as LEDs in the play/pause buttons were set on or off through digital pins.

Results

This project was moderately successful. All of the general interface types needed by the project (buttons, sliders, rotary encoders) were implemented: however, only 4 out 9 controls that were initially proposed were successfully completed. Figure 6 shows the completed interface.

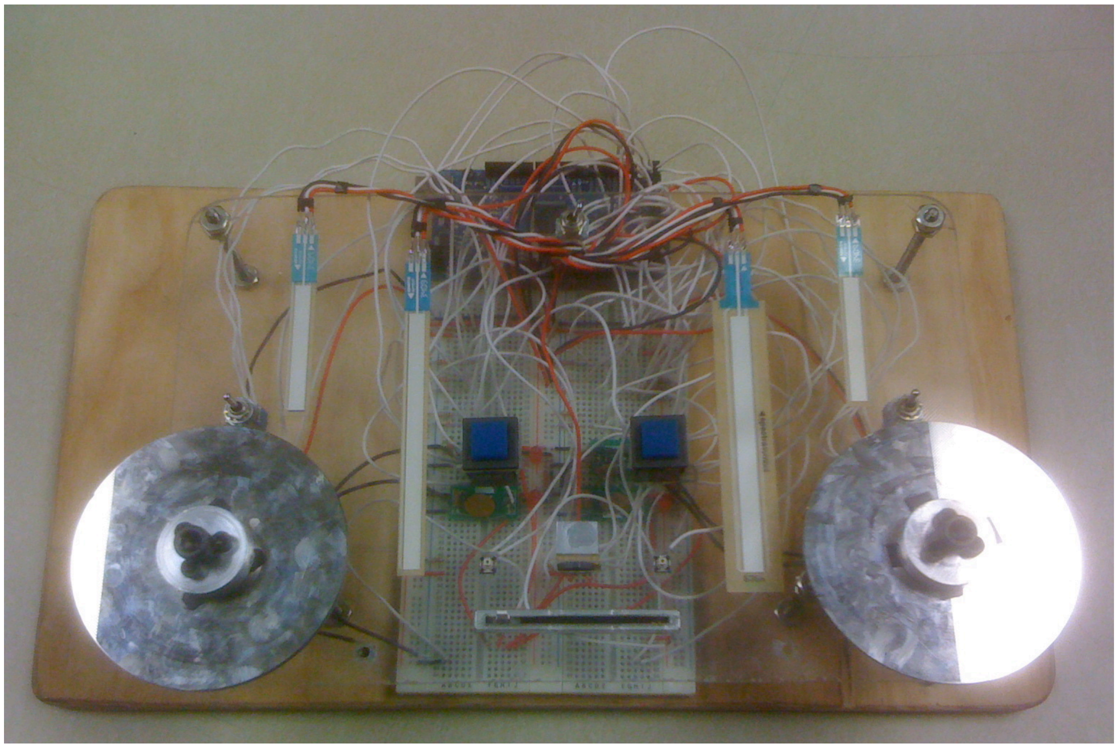


Figure 6 - Final Turntable Interface

Components shown are:

- Turntables: (mostly) successfully implemented as originally intended
- Volume Control: successfully implemented
- Crossfader: successfully implemented
- Play/Pause Buttons: successfully implemented. Additionally, feedback lights were also successfully implemented.

Components placed but not successfully implemented included speed/pitch sliders (top left and top right), speed/pitch selector switch (below speed/pitch sliders), autocrossfade selector (center, above crossfader) and autocrossfade left/right buttons (to left and right of autocrossfade selector). These components will be implemented as soon as sufficient time permits.

Overall, the performance of some elements of the system (especially the turntables) is uneven at best: significant work will be required before this interface can replace a mouse and keyboard.

The most significant problems currently are with the capacitive touch sensors. Capacitive touch sensors are extremely temperamental, and especially so when the connection between the capacitive electrode and the main touch surface is a brush connection: currently, they frequently record false touches, and appear to become unstable over time. More design work will be required before the capacitive components function as intended.

Additionally, turntable control will also need to be refined. dJay's method of interpreting data from the rotary encoders is extremely unclear: there are five separate variables that govern its response to encoder data, and as of yet no satisfactory combination of values has been found. Currently, although turntable control has been implemented, it is not nearly as smooth as mouse control, and produces decidedly "quantized" scratching sounds. This is as much a problem of insufficient information as it is a problem of poor design, however: despite initial interest in the project, Algoriddim did not respond to requests for information about how dJay processes turntable rotation data. Hopefully they will be more forthcoming in the future.

Finally, the latency existing in the system will need to be reduced. The total system latency (measured as the time between when a control is activated physically and when it causes dJay to react) currently is around 126 ms. This is far too long for a DJ controller: professional systems have latencies as low as 5 ms, and even other amateur-designed systems maintain latencies well under 50 ms.

Conclusion

Despite the problems described above, however, valuable progress was made on this project. All of the basic interface types have been successfully implemented: the work remaining to be done consists of refining existing controls and expanding the total number of controls.

Future Work

Capacitive touch sensing is the highest priority for improvement: it will likely be improved by shielding of the turntable surfaces from the rest of the interface's electronics, and by improving the mechanical connection between the brushes and the turntable surfaces. Once capacitive sensing is reliable, turntable control can be refined: hopefully, Algoriddim will expedite this process by returning requests for information. The third priority is reducing latency: this should be investigated early on, however, as it is believed that the high latency is a factor in poor turntable control. One possible method of latency reduction is making the Arduino present as a MIDI USB device rather than as a generic USB device. This is difficult although not impossible to do with the Mega 1280, and apparently quite simple to do with the newer Mega 2560

With these three tasks successfully completed, attention can be turned to refining the other control methods (especially the SoftPots), adding new controls and investigating other control possibilities. Further exploration of dJay's MIDI Out capability is important, since successful employment of this capability would allow feedback to be implemented for sliders, either through non-tactile (LED light bars) or tactile (motorized slide potentiometers) means.

Additionally, in order to maximize the usability of the system, some sort of enclosure would need to be designed to protect the components. It would ideally be water- and shock-resistant, and would allow for easy transportation of the system.

Finally, at some point, learning how to DJ would be optimal.

Acknowledgments

The author would like to thank Professor Erik Cheever for his assistance throughout this project. He would also like to thank Ed Jaoudi for his assistance with parts procurement, and Frederik Seiffert at Algoriddim Software for assistance with MIDI communication with dJay.

Finally, the author would like to recognize Bourns, Inc. and ALPS for their generous donation of components to this project.

Bibliography

[1] SpikenzieLabs. (2010) SpikenzieLabs. [Online]. http://www.spikenzielabs.com/SpikenzieLabs/Serial_MIDI.html

[2] Unknown. The MIDI Specification. [Online]. http://www.oktopus.hu/imgs/MANAGED/Hangtechnikai_tudastar/The_MIDI_Specification.pdf

[3] Quantum Research Group Ltd. (2001) QProx QT113/QT113H Charge-Transfer Touch Sensor. [Online]. http://imagearts.ryerson.ca/sdaniels/physcomp/PDFs/qt113.pdf

Appendix 1: Code

The code used in this project is attached below. Unfortunately, the code used during presentation was overwritten inadvertently: consequently, this code does not implement the full functionality shown during the presentation of this project. However, it is working code, and all general functions used in the final project presentation (buttons, faders, encoders) are present here.

```
/*
TurntableTest3
Julian Leland, E91B
Swarthmore College, 2010

This code reads quadrature input and outputs MIDI data formatted to control a turntable in Algoriddim
Software's dJay. The encoder is modeled after the Vestax Spin controller, and uses the same MIDI
messages.

This code operates both turntables and the fader as of 11/17/10.

Unfortunately, TurntableTest4 was lost: consequently, this code does not implement the full functionality
shown during the presentation of this project. However, it is working code, and all general functions used
in the final project presentation (buttons, faders, encoders) are present here.

To-do:
  - Incorporate any changes made in TurntableTest2 post 11/11/10
  - Fix scratching logic so it doesn't make that horrible sound

*/

// Initialize variables for Channel A (left)
int chanA = 0x00;                       // Channel number
volatile int encAPos = 0;               // Encoder position
int curSpeedA = 0;                      // Current speed
const int touchChanA = 12;              // Channel for touch sensor
int touchA = 0, chgTouchA = 0;          // Variables for touch sensor
const int jogChanA = 22;                // Channel for jog sensor
int jogA = 0, chgJogA = 0;              // Variables for jog sensor

// Initialize variables for Channel B (right)
int chanB = 0x01;        // These variables work identically to the variables for Channel A
volatile int encBPos = 0;
int curSpeedB = 0;
const int touchChanB = 14;
int touchB = 0, chgTouchB = 0;
const int jogChanB = 23;
int jogB = 0, chgJogB = 0;

// Initialize general variables
int firstPos = 0, secondPos = 0;      // Encoder Position variables - used to calculate speed
int voltage = 0;                      // Voltage measured at crossfader
int curFader = 0, prevFader = 0;      // Variables used for crossfader smoothing
```

```
// Setup routine
void setup() {
  Serial.begin(57600);                         // Start serial, 57600 baud
  attachInterrupt(5, ChanA1, CHANGE);          // Attach interrupts on pins 18-21. This means that these pins
  attachInterrupt(4, ChanA2, CHANGE);          // will detect rising OR falling edges
  attachInterrupt(3, ChanB1, CHANGE);
  attachInterrupt(2, ChanB2, CHANGE);
}

// Main routine
// Pointers are used so that the same code can be used with both channels.
void loop() {
  checkSpeed(&encAPos, &curSpeedA);                                // Check speed on Chan A
  checkTouch(touchChanA, &touchA, &chgTouchA);                     // Check whether Platter A is touched
  checkJog(jogChanA, &jogA, &chgJogA);                             // Check if jog wheel is active
  // Send appropriate MIDI commands
  sendMIDINote(chanA, touchA, chgTouchA, jogA, chgJogA, curSpeedA);
  checkSpeed(&encBPos, &curSpeedB);                                // Repeat for Channel B
  checkTouch(touchChanB, &touchB, &chgTouchB);
  checkJog(jogChanB, &jogB, &chgJogB);
  sendMIDINote(chanB, touchB, chgTouchB, jogB, chgJogB, curSpeedB);
  faderPos();                                                      // Check fader position

}


void checkSpeed(volatile int *encPos, int *curSpeed) { // Check current speed. Input is encoder position of
                                                       // channel being checked: output is current speed.
  int rawSpeed = 0;
  firstPos = *encPos;                                  // Read encoder position
  delay(25);                                           // Wait 25 ms
  secondPos = *encPos;                                 // Read encoder position again
  rawSpeed = (secondPos - firstPos);                   // Speed is difference between positions

  if (rawSpeed > 0) {
    *curSpeed = map(rawSpeed, 1, 200, 65, 127);        // If turning CW, map speed into range from 65-127
  }                              // These speed values correspond to the permissible range
                                 // of MIDI note values
  else if (rawSpeed < 0) {
    *curSpeed = map(rawSpeed, -1, -200, 63, 0);        // If turning CCW, map speed into range from 63 to 0
  }
  else *curSpeed = 64;                                 // Otherwise, speed = 64.
/*  if (analogRead(0) <= 200) {                        // This is all test code - preserved for later use
    Serial.print(secondPos);
    Serial.print(",");
    Serial.print(firstPos);
    Serial.print(",");
    Serial.print(secondTime);
    Serial.print(",");
    Serial.print(firstTime);
    Serial.print(",");
    Serial.print(dir);
    Serial.print(",");
    Serial.println(curSpeed);
```

```
 }*/
}

void checkTouch(int readChan, int *touch, int *chgTouch) { // Check touch sensor status. Input is channel
                                                // being read, output is state and chgState variables
 if (analogRead(readChan) <= 200 && *touch == 0) {      // The capacitive sensor outputs roughly 2.5 V, so
                                                  // it is read with an analog pin
   *touch = 1;     // If we weren't touched before, but we are now, change touch = 1 and chgTouch = 1
   *chgTouch = 1;
 }
 else if(analogRead(readChan) <= 200 && *touch == 1) {
   *chgTouch = 0;  // If we were touched before, and are still touched, change chgTouch = 0, touch still = 1
 }
 else if(analogRead(readChan) > 200 && *touch == 1) {
   *touch = 0;     // If we were touched before but aren't now, change touch = 0, chgTouch = 1
   *chgTouch = 1;
 }
 else {
   *chgTouch = 0;  // If we were not touched before, and still aren't touched, both variables = 0
 }
}

void checkJog(int jogChan, int *jog, int *chgJog) {        // This code works identically to checkTouch,
 if (digitalRead(jogChan) == HIGH && *jog == 0) {        // with the exception that it reads digital
   *jog = 1;                                              // pins instead of analog pins
   *chgJog = 1;
 }
 else if(digitalRead(jogChan) == HIGH && *jog == 1) {
   *chgJog = 0;
 }
 else if(digitalRead(jogChan) == LOW && *jog == 1) {
   *jog = 0;
   *chgJog = 1;
 }
 else {
   *chgJog = 0;
 }
}

void sendMIDINote(int chan, int touch, int chgTouch, int jog, int chgJog, int curSpeed) {
 if (touch == 1 && chgTouch == 1) {
 // 'Touching' the turntable is accomplished by sending a Note On command
   Serial.print((0x90 | chan), BYTE);             // Note on, MIDI channel selected by chan
   Serial.print(0x2E, BYTE);                      // Play Note 0x2E
   Serial.print(127, BYTE);                       // Velocity (unused)
 }
 if (touch == 0 && chgTouch == 1) {
// 'Releasing' the turntable is accomplished by sending a Note Off command
   Serial.print((0x80 | chan), BYTE);              // Note off, MIDI channel selected by chan
   Serial.print(0x2E, BYTE);                      // Play Note 0x2E
   Serial.print(127, BYTE);                       // Velocity (unused)
 }
  if (jog == 1 && chgJog == 1) {
// 'Touching' the jog wheel is accomplished by sending a Note On command
```

```
    Serial.print((0x90 | chan), BYTE);              // Note on, MIDI channel selected by chan
    Serial.print(0x2D, BYTE);                       // Play Note 0x24
    Serial.print(127, BYTE);                        // Velocity (unused)
  }

  if (jog == 0 && chgJog == 1) {
// 'Releasing' the jog wheel is also accomplished by sending a Note On command
    Serial.print((0x90 | chan), BYTE);              // Note off, MIDI channel selected by chan
    Serial.print(0x2D, BYTE);                       // Play Note 0x24
    Serial.print(127, BYTE);                        // Velocity (unused)
  }

  if (curSpeed != 64) {                       // Only send speed data if the turntables are spinning
    Serial.print((0xB0 | chan), BYTE); // Controller change, MIDI channel selected by chan
    Serial.print(0x10, BYTE);             // Address turntable scratch (coarse pan control in MIDI protocol)
    Serial.print(curSpeed, BYTE);       // Spin speed
  }
}

void faderPos() {
voltage = analogRead(4);                            // Read voltage on Pin 4
  curFader = map(voltage, 5, 550, 0, 127);     // Map voltage from permissible range onto 0 - 127 range
  Serial.print(0xB2, BYTE);                         // Controller Change, Channel 3
  Serial.print(0x08, BYTE);                         // Controller 0x08
  Serial.print(curFader, BYTE);                     // Controller position
}

void ChanA1(){
  if (digitalRead(18) == digitalRead(19)) { // When an interrupt is triggered, check the current state of the
                                                     // two channels
     ++encAPos;                                      // If they're equal, then rotation is CW
  }
  else {
     --encAPos;                                     // Otherwise, rotation is CCW
  }
}

void ChanA2(){
  if (digitalRead(18) == digitalRead(19)) {       // If an interrupt is triggered on the other channel, check the
                                                        // current state of the two channels
     --encAPos;                                         // Since we're considering the other channel, if they're equal,
  }                                                     // then rotation is CCW
  else {
     ++encAPos;                                       // Otherwise, rotation is CW
  }
}

void ChanB1(){                                      // Code for Channel B is identical to Channel A code.
  if (digitalRead(20) == digitalRead(21)) {
     ++encBPos;
  }
  else {
     --encBPos;
  }
```

```
}

void ChanB2(){
  if (digitalRead(20) == digitalRead(21)) {
      --encBPos;
  }
  else {
      ++encBPos;
  }
}
```