```
// ver 1.6 added bias and did final test on jig
// ver 1.5 4-5-2025 fix program flow bug.
// This version is ready for input test. Adjust pin count define to running pins.
// ver 1.0 3-20-2025

// Modified USB and MIDI code for project from Arduino.cc
//  project_MIDIUSB_write.ino
// A simple example based on open source USB Midi software from Arduino..
// Will be incomplete, very rough semi-functional allowing students to complete and refine.
// It may even have some bugs to fix.
// This simple code is using ANSI standard C. It is used in mission-critical software. Very reliable.
// https://en.wikipedia.org/wiki/ANSI_C
//
// The windows, mac and linux IDE ( integrated development enviromnent ) is found here:
// https://www.arduino.cc/en/software
//
// The embedded microcontroller is a Atmel SAM3X8E ARM Cortex-M3 CPU is on the Arduino DUE.
// It is the is the USB 'Device'. This is a good starting point for learning ARM basics.
// https://docs.arduino.cc/hardware/due/
// https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-11057-32-bit-Cortex-M3-Microcontroller-SAM3X-S
// note: Microchip inc has merged with Atmel
//
// An open source software based subtractive synthesizer 'SURGE XT' is the USB 'Host'.
// https://surge-synthesizer.github.io/
// note: the Surge synthesizer usb cable should be unplugged prior to DUE flash programming/debugging.
// Then plug back in, then In surge re-select the DUE midi port.
// I used a usb hub so the connectors on laptop or arduino do not wear out. Other OS its unnecessary.
// The final version will not have programming cable so all of this will be unnecessary.
// It seems to be a W11 thing. Prior to updates in W11 OS this were worked properly.
//
// Any midi synth may be used.
//
// See app notes on patches (sounds) compatable with percussion etc.
// Many keyboard style patches are simply not compatable
// due to excessive attack times in ADSR settings (Attack - Delay - Sustain - Release)
// A percussion patch should have fast Attack, everything else optional

#include <Arduino.h>  // default installed library
#include <MIDIUSB.h>  // add-on from arduino
#include <DueTimer.h> // add-on from arduino
// the add-on librarys are part of the IDE window on left bar. Just select 'add library' and install

// MIDI command structure used in example
// First parameter is the event type (0x09 = note on, 0x08 = note off).
// Second parameter is note-on/note-off, combined with the channel.
// Channel can be anything between 0-15. Typically reported to the user as 1-16.
// Third parameter is the note number (48 = middle C).
// Fourth parameter is the velocity (64 = normal, 127 = fastest).

int myLed = 13; // on board user debug led used by timer indicator
int Tick =0;  // Global tick, Dont care if it overflows, only a change is used.
int LastTick =0;

//global variables 0-69  (70 minus 1) is highest DIO assignment to be used. 70 is a pad int for safety.
// unused inputs should have a pull down resistor to mininize on-chip noise.
//
// internal pullups INPUT_PULLUPS could be used during init but the progran structure would change.
// and the piezeo element cannot drive the pullups directly.
//
//
// Debounce time 50ms
#define deBounce 75 // *** TBD
#define DEBUG // to run debug code blocks
#define OFFSET 48 // pin to midi offset. No boundry check are made in code!

#define Highest_Pin 28 // hi pin + 1, 2 pin testing
#define Lowest_Pin 22
// an array more useable than a 'structure'. Sort of a 'state machine'
int idle[Highest_Pin]; // input is idle, no action taken
int note_running[Highest_Pin]; //true false flag , is note running ?
```

```
int last_capture[Highest_Pin]; // keep a count of interrupt 1ms
int mton_count[Highest_Pin];  // min time on before turning off after triggered, a constant set during s
int PinTmp=0;


bool ledOn = false; // debug global variable

void noteOn(byte channel, byte pitch, byte velocity) {
  midiEventPacket_t noteOn = {0x09, 0x90 | channel, pitch, velocity};
  MidiUSB.sendMIDI(noteOn);
}

void noteOff(byte channel, byte pitch, byte velocity) {
  midiEventPacket_t noteOff = {0x08, 0x80 | channel, pitch, velocity};
  MidiUSB.sendMIDI(noteOff);
}

void setup() {
    delay(100); //startup safety
    int i;
  for ( i=Lowest_Pin; i<Highest_Pin; i++ ) //using DIO pin numbers for clarity D22 - D69 , do not use th
  {
     pinMode(i, INPUT); delay(1); // note: delay is a blocking function.
     } // All these now inputs, slight delay between writes as the control sect are clocked.

  // and initalize data used
for ( i=Lowest_Pin; i<Highest_Pin; i++) //
{
idle[i]= 1; // inital state is idle
note_running[i] = 0; //true false , is note running ?
last_capture[i] = 0;;  // min time on before turning off, this in a input debounce scheme
}


// set up timer
    pinMode(myLed, OUTPUT);
    Timer3.attachInterrupt(myHandler);
    Timer3.start(1000); // interrupts every 1 millisecond ( 1,000 microseconds )

// set up debug port
  Serial.begin(115200);
    delay(100); //  Delay for uart clock

}

//interrupt timer 1ms

void myHandler(){
  Tick++; // Global variable
}

// Not used, maybe later
// First parameter is the event type (0x0B = control change).
// Second parameter is the event type, combined with the channel.
// Third parameter is the control number number (0-119).
// Fourth parameter is the control value (0-127).
void controlChange(byte channel, byte control, byte value) {
  midiEventPacket_t event = {0x0B, 0xB0 | channel, control, value};
  MidiUSB.sendMIDI(event);
}


// Never ending main loop
void loop() {
  int Statetmp; // local variable in loop
  int i;

  while(1)
  {  // never exits, if it does the software is broken
    ledOn = !ledOn; // debug led , measure frequency of loop on LED
```

```
    digitalWrite(myLed, ledOn); // Led on, off, on, off...

  for ( i=Lowest_Pin; i<Highest_Pin; i++ ) //using DIO pin numbers for clarity
  {
    PinTmp = i; // Global tmp
    Statetmp = digitalRead(i);  // chech if hi
    if(  (Statetmp == 1) && (note_running[i]==0)  ){ //note went on, send to midi
      idle[i]=0; // note is engaged, not idle
      note_running[i]= 1; //
      last_capture[i] = 0; //reset
      noteOn(0, ((byte)i + OFFSET), 64);   // MIDI Channel 0, C, normal velocity. manual test, range che
      MidiUSB.flush(); //this may not be necessary. have to look at flush code.

      #ifdef DEBUG
      Serial.print("Sending note on ");
      Serial.println(((byte)i + OFFSET),DEC);
      #endif
    }
  }
 // Check for pin Low and note time out
   for ( i=Lowest_Pin; i<Highest_Pin; i++ ) //using DIO pin numbers for clarity
  {
    Statetmp = digitalRead(i);
    if ((Statetmp == 0 ) && (last_capture[i] >= deBounce) ){ // debounce time expired and pin is low.
      idle[i]=1; // input is now idle
      last_capture[i] = 0; // reset
      note_running[i]= 0; // reset too
      noteOff(0, ((byte)i + OFFSET), 64);  // Channel 0 (ch one in people speak), C, normal velocit
      MidiUSB.flush(); //this may not be necessary. have to look at flush code.

      #ifdef DEBUG
      Serial.print("Sending note off ");
      Serial.println(((byte)i + OFFSET),DEC);
      #endif
    }
  }


    if(Tick != LastTick) // one ms background timer
    { // 1ms timer capture update
      LastTick = Tick; // update tick
          for ( i=Lowest_Pin; i<Highest_Pin; i++ ) //using DIO pin numbers for clarity
      {
      if ( idle[i] != 1) last_capture[i]++; // only increment if not idle
      }
    }

} //inner forever loop

  Serial.println("Program failed out of main loop");

  }// main
```