

□ Reverse- engineering et hacking

- [Reverse-ingeneering de la graveuse laser Jinsoku LE1620](#)

Reverse-ingeneering de la graveuse laser Jinsoku LE1620

Informations

- Christian Simon
- FabLabSU
- Date de fin : 29 avril 2022

Contexte - Motivation

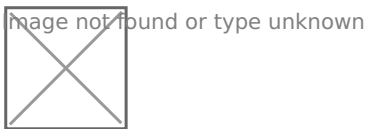
Il s'agit de détourner une CNC peu chère de son usage initial. C'est un point de départ intéressant pour construire des machines 2D+ rapidement : l'électronique et le bâti sont déjà montés, ça permet de faire vite plein de choses.

C'est initialement dans le cadre du projet [Pillink](#) que j'y ai pensé. Mais on pourrait également transformer cette découpeuse laser en découpeuse à plasma (ou autre !) avec un effort minimal.

L'étape indispensable est de pouvoir modifier la tête pour contrôler *autre chose* que le faisceau laser de gravure, sans changer le reste.

Présentation de la machine

C'est graveuse laser à bas-coût que l'on peut trouver sur de nombreux site, dont [Amazon](#), pour 245€ environ, produite par Genmitsu et [SainSmart](#).



Elle est basée sur un contrôleur dont le firmware est basé sur l'open-source [grbl](#).

On peut donc la contrôler avec la plupart des logiciels usuels : [LaserGRBL](#) (Windows seulement), [Lightburn](#) (non-libre). On pense aussi à [UGS](#) (Universal Gcode Sender, libre et multiplateforme).

Références : [Le blog Ben Maker](#)

On trouve également quelques “unboxing” sur Youtube, mais ils sont sans intérêt pour la plupart.

Démarche

La puissance du laser est fixée dans le Gcode, qui est transmis au contrôleur, qui envoie un PWM à une carte fille montée sous le bras, qui elle-même contrôle et alimente le laser.

On va donc chercher à exploiter ce PWM pour déclencher d'autres actions : démarrer l'aspiration (ouvrir une électrovanne), actionner un servo-moteur, ouvrir un relais. Les valeurs du PWM étant autant de codes d'actions possible... à la précision et au bruit près !

Décodage d'un PWM

La première étape est d'arriver à décoder un PWM avec un Arduino. La meilleure lecture est le blog de [Ben Ripley](#), qui présente 3 méthodes :

-

Simple, basique avec la fonction `pulseIn()`.

-

Avec des interruptions externes à coups de `attachInterrupt()`.

-

Avec des bibliothèques qui implémentent des fonctions autour de ces interruptions.

Il utilise `[PinChangeInt]`(<https://playground.arduino.cc/Main/PinChangeInt/> "https://playground.arduino.cc/Main/PinChangeInt/") mais ont trouvé très vite ses évolutions dont `[PinChangeInterrupt]`(<https://github.com/NicoHood/PinChangeInterrupt> "https://github.com/NicoHood/PinChangeInterrupt"), incluse dans la base de bibliothèques de l'IDE Arduino.

On trouve facilement des exemples de gens l'utilisant, par exemple [le blog QuadMeUp de @pspychalski](#). L'exemple permet de décoder des signaux RC transmis par une télécommande avec un PWM à 50 Hz... La première inconnue est donc la capacité à travailler à plus haute fréquence.

Pour prendre en main cela, on programme un premier Arduino pour générer des PWM de 0 (`dutycycle` 0%) à 255 (`dutycycle` 100%), selon une entrée. Le PWM généré par l'Arduino UNO est à 490 Hz, déjà de fréquence plus élevée. Ce PWM est envoyé vers l'Arduino qui décode, et allume des LED selon la valeur décodée.

Lorsque cet ensemble fonctionne, on a le code suivant.

```
#include <PinChangeInterrupt.h>
```

```
/*
```

Define pins used to provide RC PWM signal to Arduino

Pins 8, 9 and 10 are used since they work on both ATmega328 and

ATmega32u4 board. So this code will work on Uno/Mini/Nano/Micro/Leonardo

See PinChangeInterrupt documentation for usable pins on other boards

```
*/
```

```
const byte channel_pin[] = {8, 9, 10};
```

```
volatile unsigned long rising_start[] = {0, 0, 0};
```

```
volatile long channel_length[] = {0, 0, 0};
```

```
#define led_r 7
```

```
#define led_j 6
```

```
#define led_v 5
```

```
#define led_b 4
```

```
int etat;
```

```
void setup() {
```

```
    Serial.begin(57600);
```

```
    pinMode(channel_pin[0], INPUT); pinMode(channel_pin[1], INPUT); pinMode(channel_pin[2], INPUT);
```

```
    pinMode(led_r, OUTPUT);
```

```
    pinMode(led_j, OUTPUT); pinMode(led_v, OUTPUT); pinMode(led_b, OUTPUT);
```

```
    attachPinChangeInterrupt(digitalPinToPinChangeInterrupt(channel_pin[0]), onRising0, CHANGE);
```

```
    attachPinChangeInterrupt(digitalPinToPinChangeInterrupt(channel_pin[1]), onRising1, CHANGE);
```

```
    attachPinChangeInterrupt(digitalPinToPinChangeInterrupt(channel_pin[2]), onRising2, CHANGE);
```

```
    digitalWrite(led_r, HIGH);
```

```
    digitalWrite(led_j, HIGH);
```

```
    digitalWrite(led_v, HIGH);
```

```
    digitalWrite(led_b, HIGH);
```

```
    delay(1000);
```

```
    digitalWrite(led_r, LOW);
```

```
    digitalWrite(led_j, LOW);
```

```
    digitalWrite(led_v, LOW);
```

```
    digitalWrite(led_b, LOW);
```

```
    delay(1000);
```

```
    digitalWrite(led_r, HIGH);
```

```
    digitalWrite(led_j, HIGH);
```

```
    digitalWrite(led_v, HIGH);
```

```

digitalWrite(led_b, HIGH);
delay(1000);
digitalWrite(led_r, LOW);
digitalWrite(led_j, LOW);
digitalWrite(led_v, LOW);
digitalWrite(led_b, LOW);
}

void processPin(byte pin) {
    uint8_t trigger = getPinChangeInterruptTrigger(digitalPinToPCINT(channel_pin[pin]));
    if(trigger == RISING) { rising_start[pin] = micros(); }
    else if(trigger == FALLING) { channel_length[pin] = micros() - rising_start[pin]; } }

void onRising0(void) { processPin(0); }
void onRising1(void) { processPin(1); }
void onRising2(void) { processPin(2); }

void loop() {
    Serial.print(channel_length[0]);
    etat=map(channel_length[0], 0, 1000, 1, 5);
    Serial.print(" | ");
    Serial.print(etat);
    Serial.print(" | ");
    Serial.print(channel_length[1]);
    Serial.print(" | ");
    Serial.print(channel_length[2]);
    Serial.println("");

    int etat;

    void setup() {
        Serial.begin(57600);
        pinMode(channel_pin[0], INPUT);
        pinMode(channel_pin[1], INPUT);
        pinMode(channel_pin[2], INPUT);
        pinMode(led_r, OUTPUT);
        pinMode(led_j, OUTPUT);
        pinMode(led_v, OUTPUT);
        pinMode(led_b, OUTPUT);
    }
}

```

```
attachPinChangeInterrupt(digitalPinToPinChangeInterrupt(channel_pin[0]), onRising0, CHANGE);
attachPinChangeInterrupt(digitalPinToPinChangeInterrupt(channel_pin[1]), onRising1, CHANGE);
attachPinChangeInterrupt(digitalPinToPinChangeInterrupt(channel_pin[2]), onRising2, CHANGE);
```

```
/* test */
```

```
digitalWrite(led_r, HIGH);
digitalWrite(led_j, HIGH);
digitalWrite(led_v, HIGH);
digitalWrite(led_b, HIGH);
delay(1000);
digitalWrite(led_r, LOW);
digitalWrite(led_j, LOW);
digitalWrite(led_v, LOW);
digitalWrite(led_b, LOW);
delay(1000);
digitalWrite(led_r, HIGH);
digitalWrite(led_j, HIGH);
digitalWrite(led_v, HIGH);
digitalWrite(led_b, HIGH);
delay(1000);
digitalWrite(led_r, LOW);
digitalWrite(led_j, LOW);
digitalWrite(led_v, LOW);
digitalWrite(led_b, LOW);
}
```

```
void processPin(byte pin) {
  uint8_t trigger = getPinChangeInterruptTrigger(digitalPinToPCINT(channel_pin[pin]));
  if(trigger == RISING) {
    rising_start[pin] = micros();
  }
  else if(trigger == FALLING) {
    channel_length[pin] = micros() - rising_start[pin];
  }
}
```

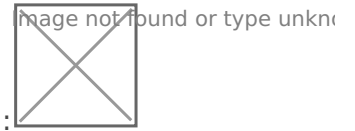
```
void onRising0(void) { processPin(0); }
void onRising1(void) { processPin(1); }
void onRising2(void) { processPin(2); }
```

```
void loop() {  
  Serial.print(channel_length[0]);  
  etat=map(channel_length[0], 0, 1000, 1, 5);  
  Serial.print(" | ");  
  Serial.print(etat);  
  Serial.print(" | ");  
  Serial.print(channel_length[1]);  
  Serial.print(" | ");  
  Serial.print(channel_length[2]);  
  Serial.println("");  
  
  switch (etat) {  
    case 1:  
      digitalWrite(led_r, HIGH);  
      digitalWrite(led_j, LOW);  
      digitalWrite(led_v, LOW);  
      digitalWrite(led_b, LOW);  
      break;  
    case 2:  
      digitalWrite(led_r, LOW);  
      digitalWrite(led_j, HIGH);  
      digitalWrite(led_v, LOW);  
      digitalWrite(led_b, LOW);  
      break;  
    case 3:  
      digitalWrite(led_r, LOW);  
      digitalWrite(led_j, LOW);  
      digitalWrite(led_v, HIGH);  
      digitalWrite(led_b, LOW);  
      break;  
    case 4:  
      digitalWrite(led_r, LOW);  
      digitalWrite(led_j, LOW);  
      digitalWrite(led_v, LOW);  
      digitalWrite(led_b, HIGH);  
      break;  
  }  
}
```

Du Gcode au PWM

Avant de faire interpréter le PWM à l'Arduino désormais programmé, on va vérifier les caractéristiques du PWM qui sort du contrôleur, en fonction des Gcode envoyés.

Hélas, l'interface UGS est incapable (*a priori*) d'allumer/éteindre/moduler le laser de la machine. Pour trouver les Gcode à envoyer, on a donc recours à Lightburn. La configuration du logiciel pour la Jinsoku-LE1620 est détaillée sur [le blog Ben Maker](#).



On met en place 4 tracés, en définissant 4 lignes à 4 puissances différentes :

Le Gcode est sauvegardé

```
; LightBurn 1.1.03
; GRBL device profile, absolute coords
; Bounds: X17.22 Y16.41 to X37.78 Y65.59
G00 G17 G40 G21 G54
G90
M4
; Cut @ 100 mm/sec, 20% power
M9
G0 X17.219Y16.408
M3
; Layer C00
G1 Y65.592S200F6000
G1 X20.781
G1 Y16.408
G1 X17.219
; Cut @ 100 mm/sec, 40% power
M9
G0 X23.219Y16.408
; Layer C01
G1 Y65.592S400
G1 X26.781
G1 Y16.408
G1 X23.219
; Cut @ 100 mm/sec, 80% power
M9
G0 X29.219Y16.408
```



```

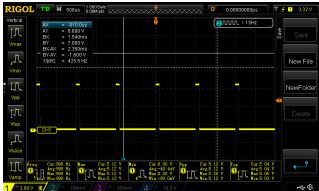
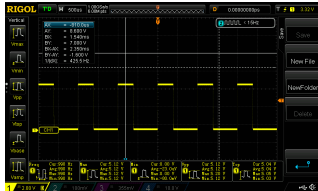
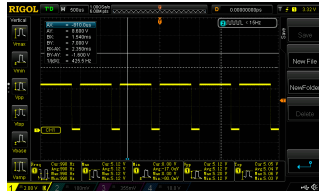
; Layer C03
G1 Y65.592S800
G1 X32.781
G1 Y16.408
G1 X29.219
; Cut @ 100 mm/sec, 100% power
M9
G0 X34.219Y16.408
; Layer C02
G1 Y65.592S1000
G1 X37.781
G1 Y16.408
G1 X34.219
M9
G1 S0
M5
G90
; return to user-defined finish pos
G0 X0 Y0
M2

```

Pour comprendre ce code, on se reporte à la documentation de [grbl](#), page "Laser Mode".

En examinant le Gcode, on repère des lignes G1 qui sont suivies de SXXX et FXXX. On a en particulier choisi le mode d'opération M3 "puissance constante" (et non M4 modulé en fonction de la vitesse de déplacement). Le code du réglage de la puissance est SXXXX (de 0 à 1000), et F est le "feed-rate".

Avec Lightburn, on envoie divers séquences, et on observe alors à l'oscilloscope :

séquence envoyée	M3 G1S100F100	M3 G1S400F100	M3 G1S800F100
PWM constaté	10%	40%	80%
observation			

Accessoirement, on peut mesurer la fréquence du PWM, qui est 1kHz, conformément d'ailleurs à ce qui est annoncé dans la documentation de grbl.

Décodage par l'Arduino et remontage global

Un point important est d'assurer la continuité des masses sur l'ensemble du montage. Même ainsi les moteurs pas-à-pas génèrent un bruit important sur le PWM que l'on cherche à exploiter.

Je dessine alors un carré avec 4 côtés de couleurs différentes dans Lightburn, j'exporte le Gcode, que j'ouvre avec UGS :

```
; LightBurn 1.1.03
; GRBL device profile, absolute coords
; Bounds: X20 Y30 to X50 Y60
G00 G17 G40 G21 G54
G90
M4
; Cut @ 100 mm/sec, 20% power
M9
G0 X20Y30
M3
; Layer C00
G1 X50S200F6000
; Cut @ 100 mm/sec, 40% power
M9
G0 X50Y30
; Layer C01
G1 Y60S400
; Cut @ 100 mm/sec, 60% power
M9
G0 X50Y60
; Layer C03
G1 X20S600
; Cut @ 100 mm/sec, 100% power
M9
G0 X20Y60
; Layer C02
G1 Y30S1000
M9
G1 S0
M5
G90
; return to user-defined finish pos
G0 X0 Y0
M2
```

Voici le résultat (intégration du fichier vidéo uploadé sur Bookstack)

Et encore (fichier vidéo accessible depuis Peertube) :