Interfaçage d'une radiocommande de modélisme à un simulateur de vol.

J.-M Friedt, 31 juillet 2018

Je suis incroyablement mauvais pour faire voler des aéronefs radiocommandés. Mauvais serait même une insulte aux dieux de l'aéromodélisme : je n'ai jamais réussi à faire tenir un aéronef dans les airs pendant plus de temps que ne mettrait une pierre lancée depuis le sol à retomber. Par contre j'ai une vague idée sur la façon de programmer des microcontrôleurs : serais-je capable de m'exercer à moindres risques (pour moi, mon entourage et mes aéronefs) sur un ordinateur avant de sortir lancer ma prochaine maquette ?

Suite aux multiples mésaventures avec l'aéromodélisme, je possède pléthore de radiocommandes, dont certaines sont tellement anciennes qu'elles ne sont plus aux normes actuelles. Autant les sacrifier à l'autel du dieu de l'aéromodélisme et les exploiter pour mon entraînement (Fig. 1). Les manettes de ces radiocommandes sont de simples potentiomètres qui commandent le rapport cyclique du signal émis par la radiocommande en vue de commander l'angle du servo-moteur correspondant du côté du récepteur. Interfacer une telle radiocommande (analogique) sur notre ordinateur (numérique) se résume donc à la tâche d'acquérir la résistance du potentiomètre (représentative de la position de la manette), la retranscrire en un format numérique exploitable, la transmettre selon un protocole compréhensible du simulateur de vol sur lequel je désire m'entraîner, et ce à une vitesse suffisamment rapide pour rattraper rapidement par les commandes des manettes à mes déficiences de pilote. On notera que cette stratégie est aussi applicable pour une radiocommande numérique plus récente tel que décrit dans [1].

Un simulateur d'aéromodélisme libre est disponible sous GNU/Linux : CRRCSim [2]. De bonne réputation quant à son réalisme, ce simulateur supporte une multitude de protocoles de communication au delà de la souris qui est peu représentative du mode de commande d'un avion radiocommandé sur le terrain. Parmi les protocoles disponibles, FMSPIC [3] s'avère particulièrement simple à implémenter : une valeur d'entête égale à 0xFF (255 en décimal) initie la trame, suivie d'autant d'octets (valeurs comprises entre 0 et 0xFE, ou 0 et 254 en décimal) que de canaux de communication. Cette séquence se répète aussi vite que possible.

Figure 1: Montage final, avec la radiocommande pilotant le simulateur de vol `crrcsim`

Le principe de fonctionnement des manettes de la radiocommande est excessivement simple dans ce cas (Fig. 3) : chaque manette est connectée à une résistance variable (potentiomètre) de résistance totale R qui se divise, selon la position de la manette, en deux contributions $r1$ et $r2$ tel que $r1+r2=R$. Le principe du pont diviseur de tension est que si une différence de potentiel Vcc (tension de référence du convertisseur analogique-numérique – dans notre cas la tension d'alimentation) est appliquée aux bornes du potentiomètre R, alors la tension en son point milieu, qui porte l'information de position de la manette, est $r2/R \cdot Vcc$ sous hypothèse que le courant absorbé par le convertisseur analogique-numérique est négligeable devant le courant circulant dans R (noter que cette condition n'est pas toujours vérifiée puisque nombre de microcontrôleurs sont équipés de convertisseurs analogique-numérique d'impédance équivalente de l'ordre de 10 kΩ, nécessitant $R \ll 10$ kΩ). Si le potentiomètre parcourt toute son excursion, alors $r1$ va de 0 à R et la tension vue par le convertisseur est 0 à Vcc. Un convertisseur sur N bits convertit une tension d'entrée V en une valeur numérique b par $b = V/V_{ref} \cdot (2^N - 1)$: dans notre cas $N = 10$ et $V_{ref} = Vcc$. Nous verrons donc que les deux seules subtilités algorithmiques du programme tiennent d'une part dans le fait que les valeurs transmises par le protocole FMSPIC sont codées sur 8 bits alors que les valeurs lues sont sur $N = 10$ bits, et d'autre part dans le fait qu'en pratique $r1$ n'atteint pas les bornes 0 et R, faisant perdre de la résolution si nous n'y prenons pas garde (i.e. retrancher la valeur minimum et effectuer une homothétie de la plage analysée pour maximiser la plage des valeurs transmises).

Le rôle du microcontrôleur entre la radiocommande et l'ordinateur se résume donc à une boucle infinie chargée de

1. acquérir les $N = 3$ voies de mesure de la radiocommande. J'ai choisi une radiocommande 4-voies afin d'agir sur le lacet, le tangage et la puissance moteur, la 4ème voie n'étant pas utilisée pour le moment mais gardée en réserve pour contrôler le roulis le jour où la manipulation des ailerons sera maîtrisée,
2. ajuster les valeurs acquises pour exploiter la gamme des valeurs transmises, à savoir 0 à 0xFE : cela passera par une translation (valeur acquise en position minimum de la manette) et homothétie (multiplier la valeur résultante pour émettre 0xFE en position maximale de la manette),
3. émettre une trame au format `0xFE C1 C2 C3 C4` avec `Ci` les quatre valeurs des quatre canaux acquis, tel que décrit à [3].

Nous découpons le problème en quatre étapes qui forment la trame de cet article :

1. communiquer entre un microcontrôleur et un PC sur interface USB
2. acquérir la position de la manette et ajuster le biais et le gain pour convertir la mesure en pleine-échelle
3. former les phrases attendues par le protocole de communication
4. apprendre à voler.

1 Programmation du microcontrôleur

Le microcontrôleur sélectionné a besoin d'une interface de communication – idéalement USB pour facilement communiquer avec un ordinateur personnel – et de trois voies d'acquisition analogiques. Nombreuses sont les options disponibles : nous avons arbitrairement sélectionné l'Atmega32U4 disponible sur la plateforme Olimexion32U4 [4] pour une douzaine d'euros auprès d'Olimex. Ce microcontrôleur ne nécessite aucun périphérique de programmation autre qu'un câble USB A-miniB puisqu'il est fourni préprogrammé avec un *bootloader* chargé de réceptionner le programme transmis par l'ordinateur personnel et l'écrire en mémoire non-volatile. Ce microcontrôleur, malgré sa popularité avec les adeptes de la bibliothèque Arduino, se programme simplement en C au moyen des outils disponibles sous GNU/Linux dans les paquets `gcc-avr` et `avr-libc`, avec `avrdude` comme outil de transfert du binaire généré sur ordinateur personnel vers le microcontrôleur (cross-compilation pour générer une fichier vers une cible d'architecture distincte de celle de l'hôte sur lequel nous travaillons). On notera d'ailleurs que ce sont ces outils qui sont cachés derrière l'environnement intégrée de développement (IDE) Arduino [5], tel que nous le constatons dans la séquence de commandes affichées en bas de l'IDE lors de la compilation d'un projet.

L'environnement de travail étant posé, il nous reste à acquérir les trois voies du convertisseur analogique, les analyser pour observer leur valeur minimale et maximale, et communiquer sur USB. Ce dernier point est de loin le plus complexe, USB étant un protocole tellement versatile qu'il en devient difficile à maîtriser. Heureusement, une bibliothèque vient à notre aide – LUFA – opportunité de s'entraîner à la compilation séparée et l'édition de lien avec une bibliothèque externe.

1.1 Communication

Commençons par nous entraîner à communiquer sur bus USB, et pour atteindre ce résultat compiler en nous liant sur LUFA. Cette bibliothèque propose une multitude de fonctionnalités, digne de la complexité du protocole USB, et nous avons tronqué cette bibliothèque au strict minimum (`http://jmfriedt.free.fr/LUFA_light_EEA.tar.gz`) pour permettre l'émulation d'une communication asynchrone comme le ferait un port série compatible RS232 – implémenté comme le sous ensemble CDC (*Communication Device Class*) du protocole USB.

```
1  #include <avr/io.h>         // definition des @ des I/O
2  #define F_CPU 16000000UL    // vitess du processeur
3  #include <util/delay.h>     // _delay_ms
4  #include "VirtualSerial.h" // communication USB
5
6  extern USB_ClassInfo_CDC_Device_t VirtualSerial_CDC_Interface;
7  extern FILE USBSerialStream;
8
9  int main(void)
10 {SetupHardware();
11  CDC_Device_CreateStream(&VirtualSerial_CDC_Interface, &USBSerialStream);
12  GlobalInterruptEnable();
13  while (1)
14   {_delay_ms(50);
15    fwrite("Hello\r\n",sizeof(char),7, &USBSerialStream);
16 // les 3 lignes ci-dessous pour accepter les signaux venant du PC
17    CDC_Device_ReceiveByte(&VirtualSerial_CDC_Interface);
18    CDC_Device_USBTask(&VirtualSerial_CDC_Interface);
19    USB_USBTask();
```

```
  }
  return 0;
}
```

Ainsi, communiquer se résume à initialiser l'interface de communication USB `VirtualSerial_CDC_Interface` et le descripteur de fichier correspondant `USBSerialStream`. Une fois ces périphériques initialisés (`CDC_Device_CreateStream()` suivi des interruption matérielles `GlobalInterruptEnable()`, il ne reste qu'à exploiter les fonctions de `stdio` pour remplir le tampon du descripteur de fichier. Périodiquement, ce tampon est transmis par `CDC_Device_USBTask()`. Il est judicieux, mais non nécessaire en l'absence de communication de l'ordinateur personnel vers le microcontrôleur, de sonder l'état du tampon de réception et de le vider périodiquement, même si nous n'utilisons pas les informations acquises : c'est le rôle de `CDC_Device_ReceiveByte()`.

```
EXEC=main
# repertoire contenant LUFA/VirtualSerial : a adapter a son arborescence
REP=$(HOME)/Atmega32
CC=avr-gcc
CFLAGS=-mmcu=atmega32u4 -Os -Wall -I$(REP)/VirtualSerial/ -I$(REP)/lufa-LUFA-140928 -DF_USB=16000000UL -std=gnu99

all: $(EXEC).out

$(EXEC).out: $(EXEC).o
   $(CC) $(CFLAGS) -L$(REP)/VirtualSerial -o $(EXEC).out $(EXEC).o -lVirtualSerial

$(EXEC).o: $(EXEC).c
   $(CC) -O2 $(CFLAGS) -c $(EXEC).c

flash: $(EXEC).out
   avrdude -c avr109 -b57600 -D -p atmega32u4 -P /dev/ttyACM0 -e -U flash:w:$(EXEC).out

clean:
   rm *.o $(EXEC).out
```

La seule subtilité de ce programme tient en sa compilation : nous devons informer le compilateur – `avr-gcc` – de l'emplacement des fichiers d'entête de la bibliothèque implémentant les fonctions requises pour la communication USB. Afin de systématiser la méthode de compilation et ne pas devoir retapper à chaque compilation la longue séquence des options, nous exploitons le `Makefile` ci-dessus – appelé par la commande `make` – pour compiler. `make` suppose avoir une règle (label avant " :") définissant la soluition à atteindre, ici générer l'exécutable `main.out`. Pour rendre le fichier de configuration `Makefile` facilement exploitable pour compiler un autre programme, nous passons par la variable `EXEC` pour n'avoir qu'un endroit à modifier pour compiler un nouveau programme. Cette règle `all` présente une dépendance (la règle après le " :") : cette dépendance doit être résolue avant d'exécuter la commande chargé de résoudre la règle en cours (la commande sous la règle et sa dépendance, préfixée par convention par une tabulation). Ainsi, ici nous voulons réaliser `${EXEC}.out` qui dépend de `${EXEC}.o`. À la première compilation, l'objet n'existe pas donc nous faisons appel à la règle `${EXEC}.c`. Ce fichier existe, donc `Makefile` compare la date de l'objet avec la date de modification du fichier source : si ce dernier est plus récent, la compilation est effectuée, générant l'objet, qui permet alors de résoudre la condition sur `all` et de générer l'exécutable par édition de lien (ici qui se contente de lier l'objet avec la bibliothèque `libVirtualSerial.a` pour résoudre les fonctions appelées mais non-définies dans `main.c`).

La génération de `libVirtualSerial.a` est un peu longue à détailler ici et le lecteur est renvoyé vers `jmfriedt.free.fr/TP_Atmega32U4_GPIO.pdf` pour une description détaillée de la procédure à suivre. À l'issue de cette compilation, le microcontrôleur émet en continu un message sur son port série configuré en port série virtuel : sous GNU/Linux, l'interface `/dev/ttyACM0` est créée, dont nous affichons les transmissions par `minicom -D /dev/ttyACM0` ou `screen /dev/ttyACM0`. Ce programme affichera en continu le même message, ligne après ligne.

Le problème de la communication entre le microcontrôleur et le PC est donc résolu : il reste à acquérir la position des manettes, et former des phrases respectant le protocole de communication.

1.2 Acquisition et communication

Maintenant que nous savons comment communiquer sur bus USB, nous désirons communiquer les valeurs acquises par les convertisseurs analogique-numérique, représentatives des positions des manettes. Rares sont les microcontrôleurs équipés de plusieurs convertisseurs analogique-numérique physiques : dans la plupart des cas, un unique convertisseur voit son entrée multiplexée vers une multitude de broches associées à divers canaux. Nous pouvons alors *séquentiellement* sonder la mesure de chacune de ces entrées. Plutôt que tout de suite émettre des trames binaires – simples à interpréter pour un ordinateur mais illisibles pour un humain – nous allons transmettre des messages au format ASCII formé de caractères affichables sur un terminal tel que `minicom` ou `screen`.

L'identification des broches associées au convertisseur analogique-numérique serait normalement simple, à la lecture de la documentation technique du microcontrôleur, si Arduino ne venait pas perturber la sérigraphie sur le circuit imprimé de façon incohérente avec l'assignation des broches du microcontrôleur (Fig. 3). En effet, le schéma du circuit Olimex nous informe que les broches nommées A0 à A5 sur le circuit imprimé sont en fait associées aux fonctions ADC7, 6, 5, 4, 1, 0 (non seulement les broches ne sont pas contigü es, mais en plus le décompte se fait en sens inverse). Nous pallions donc cette déficience en testant quelle voie de la radiocommande est lue, et en appelant la position correspondante du multiplexeur de l'ADC dans la fonction `adc_read()`.

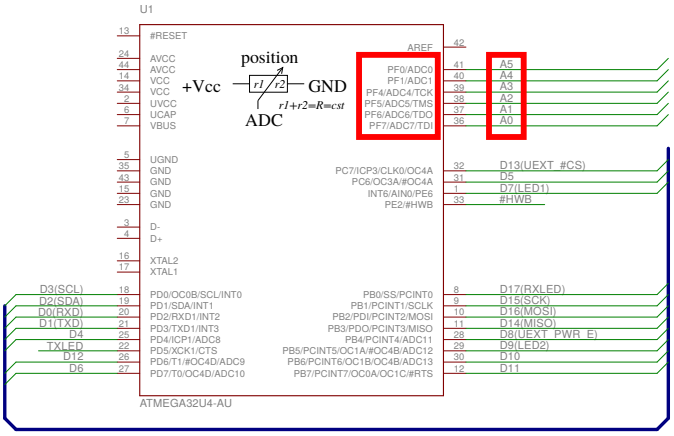

Figure 2: Schéma de la carte Olimexino32U4, mettant en évidence l'incohérence entre les broches du microcontrôleur et la sérigraphie du circuit imprimé. En insert dans le schéma : câblage d'un potentiomètre tel qu'utilisé pour lire la position de chaque manette. La résistance variable (point milieu) est polarisée entre la masse et la tension de référence du convertisseur analogique-numérique, ici la tension d'alimentation du micrcontrôleur.

```
#include <avr/io.h> //E/S ex PORTB
#define F_CPU 16000000UL
#include <util/delay.h>  // _delay_ms
#include "VirtualSerial.h"

[... initialisation variables USB ...]

void adc_init()
{DIDR0=0xff; // http://www.openmusiclabs.com/learning/digital/atmega-adc
 ADMUX = (1<<REFS0); // AREF = AVcc, 2.5 V si (1<<REFS0|1<<REFS1)
 // ADC Enable and prescaler of 128 : 16 MHz/128=125 kHz
 ADCSRA = (1<<ADEN)|(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0);
}

unsigned short adc_read(uint8_t ch)
{ch &= 0x07;               // ch\in[0..7]
 ADMUX=(ADMUX & 0xF8)|ch; // clears the bottom 3 bits before ORing

 ADCSRA |= (1<<ADSC);       // start single convertion
 while(ADCSRA & (1<<ADSC)); // poll jusqu'a fin de conversion
 return (ADC);
}

void affiche(unsigned short v,char *s)
{char c; // decoupe v en quartets et remplit la chaine de chars s avec le code ASCII
 c=(v>>12)&0x000f;if (c<10) s[0]=c+'0'; else s[0]=c-10+'A';
 c=(v>> 8)&0x000f;if (c<10) s[1]=c+'0'; else s[1]=c-10+'A';
 c=(v>> 4)&0x000f;if (c<10) s[2]=c+'0'; else s[2]=c-10+'A';
 c=(v    )&0x000f;if (c<10) s[3]=c+'0'; else s[3]=c-10+'A';
}

int main(void){
 short res=0;
 int adcnum;
 char s[25];

 [... initialisation USB tel que vu auparavant ...]
 adc_init();

 while (1)
 {_delay_ms(50);
  for (adcnum=0;adcnum<4;adcnum++)
   {if (adcnum<2) res=adc_read(adcnum);
      else res=adc_read(adcnum+2);
    affiche(res,&s[5*(adcnum)]);
    s[5*(adcnum)+4]='␣';
```

```
   }
  s[20]='\r';
  s[21]='\n';
  fwrite(s,1,22, &USBSerialStream);
//  les 3 lignes ci-dessous pour accepter les signaux venant du PC
  CDC_Device_ReceiveByte(&VirtualSerial_CDC_Interface);
  CDC_Device_USBTask(&VirtualSerial_CDC_Interface);
  USB_USBTask();
 }
 return 0;
}
```

Nous évitons en général d'utiliser les fonctions de `stdio` telles que `sprintf()`, gourmandes en ressources et généralement bien trop riches en fonctionnalités pour nos besoins restreints sur microcontrôleur. Ici, la fonction affiche découpe un mot codé sur 16 bits (`short`) et remplit un tampon des codes ASCII correspondant à la représentation en hexadécimal de chaque quartet. Ainsi, lors de l'exécution de ce programme, nous obtenons un affichage de la forme

```
0204 01BE 00FE 01DC
0204 01BE 00FE 01DB
0204 01BE 00FE 01DC
0204 01BE 00FE 01DC
0204 01BE 00FE 01DB
0205 01BC 00FE 01DB
0204 01A8 00FE 01DC
0203 018D 00FE 01DB
0203 016D 00FE 01DC
0203 0148 00FE 01DC
0203 0120 00FE 01DB
0203 00FE 00FE 01DC
0203 00E0 00FE 01DB
0203 00C2 00FE 01DB
0203 00A1 00FE 01DC
0203 007A 00FE 01DB
0204 0070 00FE 01DB
0203 0070 00FE 01DB
```

alors que nous manipulons la manette associée au second canal d'acquisition. Le contenu d'un tel fichier se lit par exemple dans GNU/Octave par

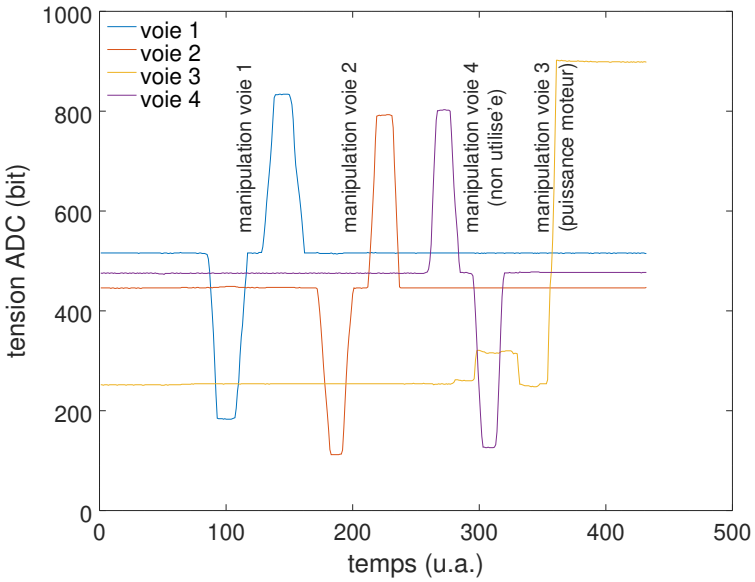

Figure 3: Tracé des mesures des diverses voies de la radiocommande, alors que chaque manette est séquentiellement manipulée entre ses valeurs extrêmes

```
f=fopen('fichier.txt');d=fscanf(f,"%x");fclose(f);d=reshape(d,4,length(d)/4);
```

qui nous permet ensuite d'en tracer le contenu par `for k=1:4;plot(d(k,:));hold on;end`. Le même résultat s'obtiendrait facilement avec `gnuplot` si nous avions pensé à préfixer chaque valeur hexadécimale des symboles "0x" lors de l'affichage. Nous constatons sur la Fig. 3 que la valeur moyenne de chaque voie n'est pas la même, et après avoir manipulé jusqu'aux valeurs extrêmes chaque manette séquentiellement, que seule une fraction de la plage de mesure est exploitée (par exemple la voie 2 ne descend que jusqu'à 0x70, ou 112 en décimal, et non 0). La voie 3, associée à la puissance moteur, ne possède pas de ressort de rappel et ne revient pas à sa valeur moyenne après manipulation. Nous avons donc validé, par ce tracé, notre capacité à lire la position de chaque manette et la transmettre dans un format lisible par un humain (code ASCII) à un ordinateur. Il nous faut maintenant adapter ce code pour respecter le protocole FMSPIC.

1.3 Acquisition, traitement et communication

Lorsque nous lisons la valeur des divers potentiomètres liés aux diverses manettes (Fig. 4), nous constatons que la gamme complète 0 à $2^{10}-1$ n'est pas atteinte. Nous devrons donc dans un premier temps éliminer le biais de mesure pour ramener la valeur minimum à 0, puis effectuer une homothétie sur les données pour exploiter au mieux les 8 bits mis à notre disposition à le protocole FMSPIC, sachant que les convertisseurs de l'Atmega32U4 sont codés sur 10 bits. Dans le cas particulier de notre radiocommande, nous constatons que la valeur minimale disponible au point milieu de chaque potentimètre est de l'ordre de 0xA0=160, et que l'excursion des mesures est d'environ 570 (sur les 10 bits disponibles). Si après avoir retranché le biais de 0xA0 nous désirons exploiter 10 bits, il nous faut multiplier par 1,5. Un microcontrôleur gère de façon très inefficace les calculs sur nombres à virgule, et nous exploitons le fait que $x \times 1,5 = x + x/2$ et que la division par 2 d'un nombre entier est un dácalage à droite d'un bit (de même que la division entière par 10 d'un nombre représenté en décimal s'obtient en éliminant le dernier chiffre) pour implémenter

l'opération sous forme de `x+x>>2`. Finalement, ayant étendu la mesure sur toute la gamme des 10 bits du convertisseur, nous nous ramenenons à la donnée sur 8 bits transmise par FMSPIC par un nouveau décallage vers la droite de 2 bits du résultat précédent, en prenant soin de tester les seuils compatibles avec le protocole, à savoir transmettre une valeur entre 0 et 254 (=0xFE).

```
#include <avr/io.h> //E/S ex PORTB
#define F_CPU 16000000UL
#include <util/delay.h>  // _delay_ms
#include "VirtualSerial.h"

// [... initialisation USB ...]
// [... fonctions d'init. et lecture ADC ...]

int main(void){
  short res=0;
  int minval[4],adcnum;
  char s[25],cmd;

  [... initialisation USB tel que vu auparavant ...]

  minval[0]=0xCC;
  minval[1]=0xA3;
  minval[2]=0xB2;
  minval[3]=0x90;
  adc_init();
  cmd=0x50;
  s[0]=0xff;s[1]=cmd;s[2]=cmd;s[3]=cmd;s[4]=cmd;

  while (1)
  {_delay_ms(50);
   for (adcnum=0;adcnum<4;adcnum++)
    {if (adcnum<2) res=adc_read(adcnum)-minval[adcnum];
        else res=adc_read(adcnum+2)-minval[adcnum];
     res=res+(res>>1);     // * 1.5
     res=res>>2;           // /4
     if (res>255) res=254; // threshold
     if (res<0) res=0;
     s[adcnum+1]=(res&0xff);
    }

    fwrite(s,1,5, &USBSerialStream);
//  les 3 lignes ci-dessous pour accepter les signaux venant du PC
    CDC_Device_ReceiveByte(&VirtualSerial_CDC_Interface);
    CDC_Device_USBTask(&VirtualSerial_CDC_Interface);
    USB_USBTask();
  }
  return 0;
}
```

Nous avons choisi ici d'attendre 50 ms entre deux transmissions. Au débit nominal de 19200 bauds, et un protocole de communication asychrone de N81 (pas de bit de parité, 8 bits/donnée et 1 bit de stop, soient 10 bits par octet transmis), le bus RS232 est capable de transmettre 1920 octets/seconde ou, compte tenu des 5 octets/trame (un octet de début de trame et 4 voies), 384 trames/s. Sans prétention de réagir à des évènements aussi brefs que 3 ms, nous avons choisi d'abaisser le débit de données en attendant un peu plus longtemps entre deux messages transmis.

Ce programme est plus difficile à déverminer car la sortie binaire n'est pas lisible par un humain. Néanmoins, en demandant à `minicom` de sauver dans un fichier les messages transmis ("CTRL-a" puis "l" comme *log*), nous pouvons afficher le contenu du fichier binaire par `xxd` qui fournit une sortie de la forme

```
00000310: ff75 691c acff 746a 1c7d ff75 691c 7cff  .ui...tj.}.ui.|.
00000320: 746a 1c7c ff75 691c 7cff 746a 1c7d ff74  tj.|.ui.|.tj.}.t
```

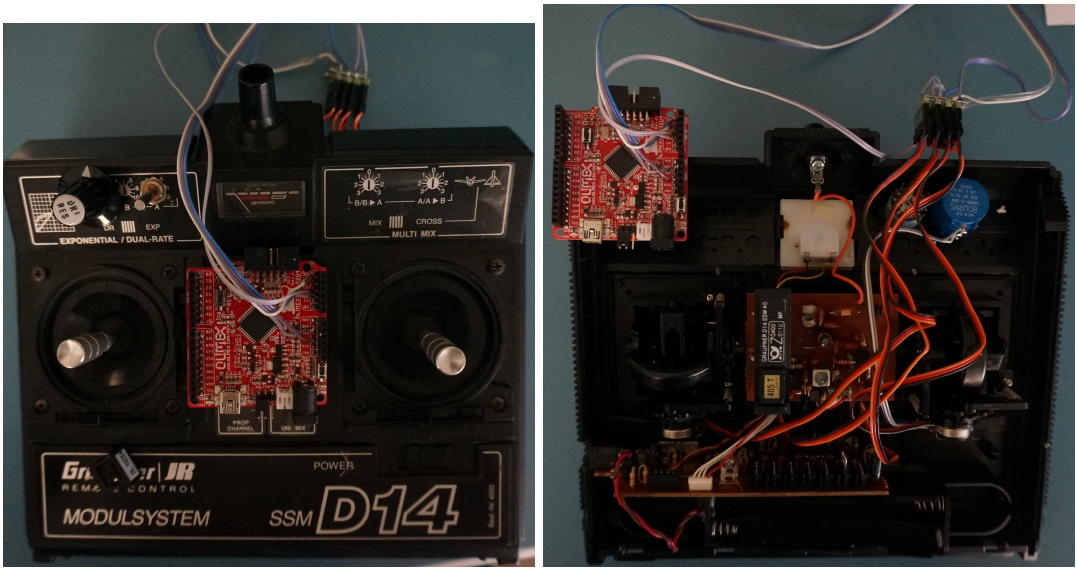

FIGURE 4 – Connexion du microcontrôleur à la radiocommande : l'électronique interne est déconnectée, seuls les potentiomètres sont connectés aux extrémités à la masse et tension d'alimentation, et le point milieu sur chaque entrée de convertisseur analogique-numérique. Aucun dommage irréversible à la radiocommande n'est nécessairem nous n'avons fait que déconnecter les nappes venant des manettes pour les connecteur au microcontrôleur.

dans lequel nous trouvons bien le débit de trame 0xff suivi des mesures des 4 voies de la radiocommande – entre 0x00 et 0xfe – avant de commencer une nouvelle trame. Ici aussi, le contenu de ce fichier binaire se tracerait sous GNU/Octave, cette fois par `f=fopen('fichier.b` avec éventuellement un test sur la voie contenant l'entête de trame pour vérifier qu'il n'y a pas un octet perdu de temps en temps (indice : la valeur 0x00 n'est pas enregistrée par `minicom` comme une valeur transmise). Le résultat d'une telle analyse est affichée sur la Fig. 5.

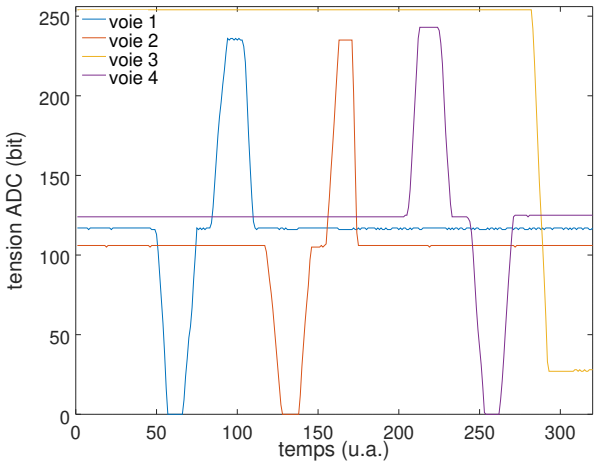

Figure 5: Conversion des valeurs brutes au format FMSPIC : les données sont cette fois codées sur 8 bits (valeurs comprises entre 0 et 255 – 254 en pratique pour ne pas confondre avec l'octet de synchronisation de trame) et exploitent toute la gamme de mesure.

2 Connexion avec le simulateur de vol

Ayant testé notre capacité à acquérir les positions des manettes et les communiquer, il nous reste à nous interfacer à CRRCSim. Mieux vaut commencer avec un motoplaneur lent pour tester notre capacité à voler : nous choisissons (Options → Airplane → Allegro, puis dans Select Config : Allegro E-Lite) un motoplaneur peu vif qui nous laissera amplement le temps de corriger nos maladresses de débutant. Pour modifier l'interface par défaut qu'est la souris, nous sélectionnons le menu Options → Controls → Input Method : la sélection de FMSPIC ajoute deux nouveaux onglets que sont le débit de communication et l'interface de communication. Le débit n'a pas d'imprtance dans le contexte de l'émulation du port asynchrone par une liaison USB, mais l'interface de communication pose problème : CRRCSim n'est prévu (occasion d'encore médire sur les interfaces graphiques – une interface en ligne de commande ne serait pas limitée par une telle décision du développeur) que pour communiquer sur un des ports série natifs `/dev/ttyS0` à `/dev/ttyS3`. Ici, l'interface CDC se nomme `/dev/ttyACM0` : nous pallions au manquement de l'interface graphique en effectuant un lien symbolique entre l'interface USB `/dev/ttyACM0` et un des ports série natifs, encore rarement disponibles physiquement sur les ordinateurs modernes. Ainsi, `ln -s /dev/ttyACM0 /dev/ttyS3` relie virtuellement `/dev/ttyS3` à `/dev/ttyACM0`, résolvant notre problème (Fig. 6).

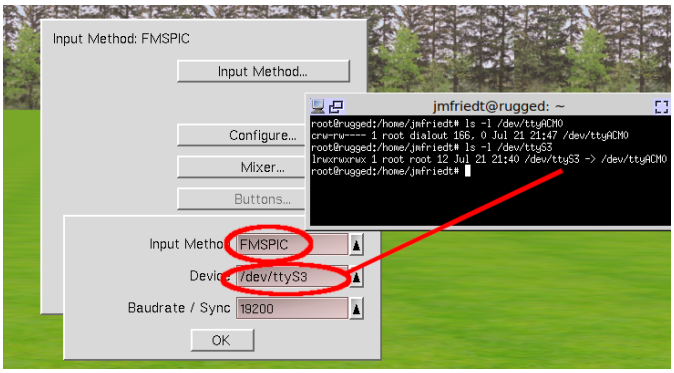

FIGURE 6 – Lien symbolique entre l'interface émulant le port de communication asynchrone sur USB `/dev/ttyACM0` avec une des ports série prévus dans CRRCSim, ici arbitrairement `/dev/ttyS3`.

3 Apprendre à voler

Ayant configuré le mode de communication de CRRCSim pour communiquer par protocole FMSPIC sur le bon port, il nous reste à valider que le simulateur de vol comprend nos ordres et que chaque manette transmet une information pleine échelle. Pour ce faire, nous utilisons l'option de Configuration de CRRCSim qui applique immédiatement les commandes transmises par le microcontrôleur sur le modèle d'avion visible en arrière plan (Fig. 7).

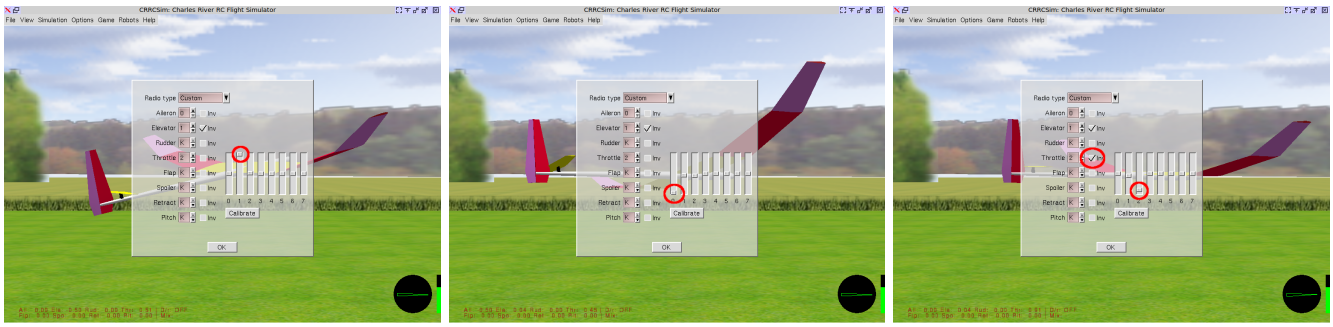

FIGURE 7 – Test des diverses commandes de l'avions par le menu Options → Controls → Configure : de gauche à droite, le lacet (canal 1), le tangage (canal 2) et la puissance moteur (canal 3) après avoir inversé cette commande pour tenir compte du câblage de l'alimentation et masse aux bornes de ce potentiomètre.

Ces étapes complétées, et étant certains que nos commandes sont fidèlement retranscrites au simulateur de vol, nous sommes prêts à apprendre à voler (Fig. 1). Une demi journée d'entraînement nous a permis de passer du stade du crash au premier demi-tour au stade de l'atterrissage contrôlé après quelques tours de terrain. Reste à voir comment une telle performance se traduit avec un "vrai" avion.

4 Conclusion

Par ce petit projet simple et ludique, nous nous sommes efforcés de découper en étapes élémentaire un objectif *a priori* ambitieux d'interfacer une radiocommande de modélisme à un ordinateur personnel pour commander le simulateur de vol CRRCSim. Cette exploration nous a amené à communiquer entre le microcontrôleur et l'ordinateur par interface USB, puis acquérir les valeurs analogiques représentatives de la position de chaque manette, avant de former les trames attendues par le protocole de communication. L'ensemble de ces développements est effectuée avec le compilateur `gcc` configuré pour générer des binaires pour la cible AVR, avec compilation au travers de `Makefile`, afin de proposer une flux de traitement généraliste appliquable à à peu près tout microcontroleur, et ce dans un objectif d'efficacité du code résultant sans s'encombrer d'une bibliothèque bridant la gamme des cibles accessibles tel que Arduino.

Le lecteur le plus curieux pour étendre cette étude pour exploiter CRRCSim non plus comme simple simulateur de vol pour entraîner un pilote humain mais pour s'interfacer avec un pilote automatique [6] : les fonctionnalités de CRRCSim ont été étendues pour s'interfacer avec le contrôleur de centrale inertielle Ardupilot.

L'Atmega32U4 est le sujet du cours d'électronique numérique de licence3 de la formation d'électronique programmable à l'Université de Franche Comté à Besançon : les supports de cours détaillant la séquence de développement sont disponibles à `http://jmfriedt.free.fr/index.html#l3ep`. Une archive des programmes se trouve à `github.com/jmfriedt/l3ep`.

Références

[1] D. Bodor, *Télécommandez vos montages Arduino*, Hackable **06** (2015)

[2] Les sources de CRRCSim sont à `sourceforge.net/projects/crrcsim/` mais aucune documentation plus complète sur son fonctionnement interne ne semble disponible.

[3] Une description informelle du protocole FMSPIC est à `www.min.at/prinz/?x=entry:entry130320-204119`.

[4] 12,95 euros à `www.olimex.com/Products/Duino/AVR/OLIMEXINO-32U4/open-source-hardware`.

[5] J.M Friedt & É. Carry, *L'environnement Arduino est-il approprié pour enseigner l'électronique embarquée ?*, Proc. CETSIS 2017, disponible à `jmfriedt.free.fr/cetsis2017_arduino.pdf`

[6] I. Lugo-Cárdenas, G. Flores & R. Lozano, *The MAV3DSim : A simulation platform for research, education and validation of UAV controllers*, Proc. 19th World Congress The International Federation of Automatic Control (IFAC) pp. 713–717 (2014) et `http://ardupilot.org/dev/docs/simulation-2sitl-simulator-software-in-the-loopusing-using-the-crrcsim-simulator.html`