Création d'un émetteur RTTY à base d'arduino (1ère partie)

Le RTTY c'est quoi ? :

Le RTTY (Radiotélétype en français) est apparu dans le monde des télécommunications radio-amateur dans le milieu des années 1940 aux Etats-Unis quand les téléscripteurs (ou télétypes, ou encore teleprinter en anglais) devinrent disponibles.

Mais la naissance du **TTY** commence bien avant cela. Tout d'abord le principe de téléscripteur est apparu en 1849 où un circuit fut mis en service entre Philadelphie et New-York (Attention il n'est pas encore question de radio-tty ici). Ensuite en 1874 Emile Baudot mit au point un système de codage utilisant 5-bits pour représenter un caractère ce qui permit la transmission de messages contenant des lettres de l'alphabet bien sûr mais en plus des signes de ponctuation.

Le premier système de communication RTTY commercial fut crée en 1932 entre Honolulu et San Fransisco et entre San Fransisco et New-york en 1934.

Ce moyen de communication fut grandement amélioré durant la seconde guerre mondial et c'est à ce moment que la modulation FSK fut utilisée pour transmettre les signaux radio. Nous le verrons plus tard dans cet article, c'est cette modulation qui est encore principalement utilisée de nos jours.

Mais qu'est ce qu'un téléscripteur ? Et bien c'est une machine permettant de générer et de recevoir des messages sous forme de signaux électriques. En gros des grosses machines à écrire où les caractères ne s'impriment pas sur du papier, mais sont convertis en signaux électriques. C'est le TTY de RTTY, le R étant pour Radio.

Le RTTY est donc un moyen de communication par radio entre deux téléscripteurs.

Le RTTY comment ?

Il existe beaucoup de protocoles différents pour faire du RTTY. Que ce soit au niveau de l'encodage des données, qu'au niveau de la modulation.
Dans cet article nous allons utiliser le Murray code (C'est le code Baudot avec l'ajout des caractères spéciaux CR et LF) pour ce qui concerne le codage des données et en terme de modulation, la plus présente pour ce mode de communication dans le milieu radio amateur à savoir FSK/170 à une vitesse de 45.45 bauds.

Le codage :

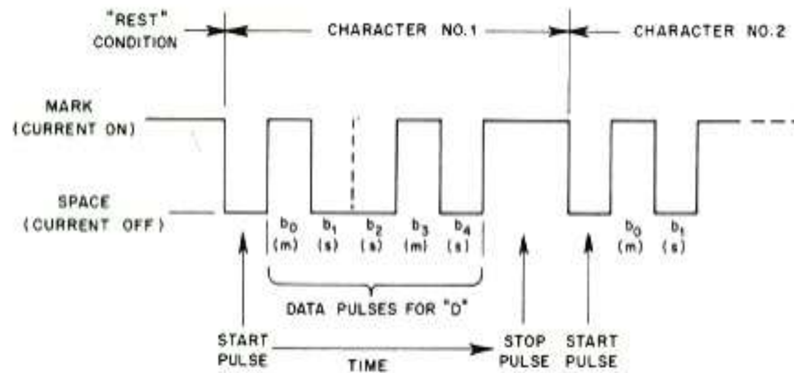
Le codage des données n'est pas très compliqué. Les caractères sont codé sur 5 bits suivant ce tableau :

NUMBER OF SIGNAL	START ELEMENT	CODE ELEMENTS 1	2	3	4	5	STOP ELEMENT	AMERICAN TELETYPE COMMERCIAL KEYBOARD	
1		●	●				●	A	—
2		●			●	●	●	B	?
3			●	●	●		●	C	:
4		●			●		●	D	$
5		●					●	E	3
6		●		●	●		●	F	!
7			●		●	●	●	G	&
8				●		●	●	H	£
9			●	●			●	I	8
10		●	●		●		●	J	'
11		●	●	●	●		●	K	(
12			●			●	●	L	)
13				●	●	●	●	M	.
14				●	●		●	N	,
15					●	●	●	O	9
16			●	●		●	●	P	0
17		●	●	●		●	●	Q	1
18			●		●		●	R	4
19		●		●			●	S	BELL
20						●	●	T	5
21		●	●	●			●	U	7
22			●	●	●	●	●	V	;
23		●	●			●	●	W	2
24		●		●	●	●	●	X	/
25		●		●		●	●	Y	6
26		●				●	●	Z	"
27					●		●	CARRIAGE RETURN	
28			●				●	LINE FEED	
29		●	●	●	●	●	●	LETTERS	
30		●	●		●	●	●	FIGURES	
31				●			●	SPACE	
32							●	BLANK	

1. ● = MARKING ELEMENT

2. AN AUTOMATIC MOTOR STOP FACILITY IS OPTIONAL IN PLACE OF THE SECONDARY OF LETTER 'H' ON THE TELETYPEWRITER

par exemple le A est égale à 11000.

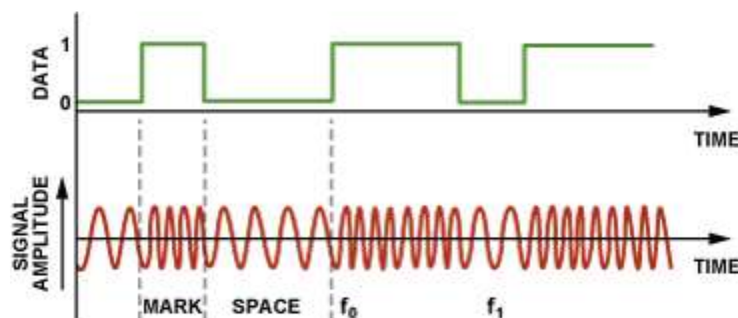


Par contre remarquez qu'il y a deux caractères pour 11000, 'A' et '-'. C'est là qu'entre en jeu la notions de LETTERS et FIGURES. Il n'y a pas assez de combinaison pour coder toutes les lettres et les ponctuations, les chiffres,... sur 5 bits, c'est pour cela qu'il y a deux codes spéciaux qui vont permettre de passer d'un tableau à l'autre (le tableau des lettres et le tableau des figures). C'est 11111 pour dire que ce qui arrive doit être décodé comme étant un caractère alphabétique et 11011 pour décoder les données en tant que caractères de ponctuations, chiffres, ...

Il y a aussi dans ce tableau une colone “stop element”, cela correspond au “stop bit”, ou aux bits qui définissent la fin de la transmission d'un message (ici un message étant un caractère ou une figure). Dans cet article nous n'allons pas utiliser 1 stop bit mais 2 comme utilisé de nos jours.

De la même manière que le stop bit, il y a le start bit (valeur 0) qui va servir à annoncer l'arrivée de données (ici une lettre ou une figure).

Voilà pour ce qui est de l'**encodage**. Pour la modulation nous allons utiliser le Frequency shift keying (FSK) avec un shift (distance entre les deux fréquences utilisées pour la modulation) de 170 Hertz. Une fréquence qui représentera les 0 (SPACE) et l'autre pour les 1 (MARK). Nous verrons aussi plus tard la modulation AFSK.



Le matériel :

- Arduino IDE : https://www.arduino.cc/en/Main/Software
- Un arduino (J'ai utilisé un nano, mais n'importe quel autre type fera l'affaire) : https://store.arduino.cc/

- Un émetteur NTX2B : http://www.radiometrix.com/content/ntx2b
- Des résistances de 100k, 2x22k, 2x4.7k
- Une breadboard
- des fils
- Visual studio code ou Vscodium (optionnel) : https://github.com/VSCodium/vscodium ou https://code.visualstudio.com/download

Pour l'écoute :

- Gqrx : http://gqrx.dk/
- Fldigi : https://sourceforge.net/projects/fldigi/
- Un dongle SDR (j'ai utilisé un Funcube 2 pro) : http://www.funcubedongle.com/?page_id=286

C'est parti ! :

Nous allons commencer par coder l'encodage des données. Pour les lecteurs n'étant pas très familliers avec le langage C++, je recommande la lecture de quelques tutoriaux. Je vais bien sûr expliquer le code, mais pas revenir sur des notions de bases du language (par exemple, la notion de fonctions, conditions, itérations, ...).

Quelques tutoriaux :

http://www.cplusplus.com/doc/tutorial/
https://openclassrooms.com/fr/courses/1894236-programmez-avec-le-langage-c
https://www.learncpp.com/

Par ailleurs Albert (ON5AM) a écrit dans cette même section un tutoriel sur Arduino dont je recommande vivement la lecture.

Pour notre émetteur, je vais lui demander d'envoyer le message “CQ CQ ON3DEX CQ CQ ON3DEX “ en continu.

Tout cela étant dit, commençons.

On va d'abord déclarer les constantes qui vont représenter les caractères et les codes de ceux-ci. Pour cela nous avons besoin de 3 tableaux :

```
const char FIGURES[] = " \r\n1234567890-\a@!&#'()\"/:;?,.";
const char LETTERS[] = " \r\nQWERTYUIOPASDFGHJKLZXCVBNM";
const byte baudot[] = { 0x4, 0x8, 0x2, 0x1d, 0x19, 0x10, 0xa, 0x1, 0x15, 0x1c, 0xc, 0x3, 0xd, 0x18, 0x14, 0x12, 0x16, 0xb, 0x5, 0x1a, 0x1e, 0x9, 0x11, 0x17, 0xe, 0xf, 0x13, 0x6, 0x7 };
```

Le premier tableau (FIGURES) contient les caractères spéciaux et les nombres. Le second tableau (LETTERS) contient l'alphabet et les caractères de retour à la ligne.
Le tableau baudot quant à lui contient les codes hexadécimaux des figures et lettres. L'orde ici à son importance, ainsi par exemple à la position 3 (index 2, car en C++ les positions commencent à 0) du tableau baudot (baudot[2]) est contenu le code de 1 (figure[2]) et Q (letters[2]).

```
#define DTX 6

#define MARK 105 // 170Hz 3
#define SPACE 100 // 0
#define SPEED 22 // 45.45 bauds => 1/45.45 = 22 ms

#define SHIFT_LTRS 0x1f
#define SHIFT_FIGS 0x1b
```

Ensuite passons aux "defines", ce sont en fait des variables constantes (dont la valeur ne changera pas). On peut très bien les déclarer comme des variables standards, mais c'est une question de bonne pratique et de clarté de code. Passons à l'explication de celles-ci :

- DTX 6 : DTX va représenter ici la pin de l'arduino qui est utilisée (6).
- MARK 105 et SPACE 100 : MARK définit la valeur (105) que l'on va envoyer sur la pin 6 de l'arduino pour transmettre une MARK. SPACE (100) la valeur envoyée sur la pin 6 pour transmettre un SPACE. Pourquoi 100 et 105 ?

Le NTX2B à +- un shift de 6000 Hertz entre les deux fréquences maximales sur lesquelles il peut transmettre. Il attend sur sa pin TXD un voltage entre 0 et 3 V . Sachant cela, on peut en déduire que un Shift de 1 Hertz équivaut à 3/6000, donc un pas de 0.0005 V. Nous, nous voulons un shift de 170 Hz, il nous faut donc un pas de 0.085 V (170 x 0,0005). Les pins de l'arduino permettant d'envoyer une tension analogique (PWM) entre 0 et 5v attendent une valeur comprise entre 0 (0 V) et 256 (5 V) , on ne peut envoyer une tension maximal que de 3 V vers le NTX2B, donc on doit choisir une valeur entre 0 et 153 (3 / 5 x 256).

Maintenant comment connaitre le nombre équivalent à un shift de 170 Hz entre nos deux valeurs à transmettres à la pin PWM ? Tout d'abord, il faut connaitre le nombre de V par pas entre 0 et 5 V , pour cela nous avons la formule : 5 / 256 = 0,0195 V . Maintenant nous pouvons calculer le nombre à avoir entre nos deux valeur pour la pin PWM : 0.085 x 0,0195 = 4,35 que l'on peut arrondir à 5.

- SPEED 22 : SPEED définit la vitesse de transmission. Nous le verrons, la méthode qui va nous permettre de contrôler la vitesse attend une valeur en milliseconde. Pour connaitre cette valeur nous permettant de transmettre à 45.45 bauds nous faisons le calcul : 1 s / 45.45 = 22 ms.

- SHIFT_LTRS 0x1f : Correspond à la valeur binaire du changement vers le décodage de lettres en hexadécimale (cf le tableau ci-dessus).

- SHIFT_FIGS 0x1b : Correspond à la valeur binaire du changement vers le décodage de caractère spéciaux et de ponctuation (cf le tableau ci-dessus).

```
void setup() {
    pinMode(DTX, OUTPUT);
}

void loop() {
    rtty_send("CQ CQ ON3DEX CQ CQ ON3DEX ");
}
```

Ici dans la méthode Setup du framework d'arduino qui nous permet d'initialiser ce que l'on souhaite au démarrage du programme, nous configurons la pin DTX (6) en mode OUTPUT via l'utilisation

de la méthode pinMode d'arduino. Pour rappel la méthode setup n'est appelée qu'une fois à la mise sous tension de l'arduino.

La méthode loop est le point d'entré de notre programme et est éxécutée à l'infini tant que l'arduino est sous tension. Dedans nous appelons la méthode rtty_send avec en paramètre le texte que nous voulons transmettre (CQ CQ ON3DEX CQ CQ ON3DEX).

Nous allons voir ci-dessous en détail ce que fait la méthode rtty_send.

```
void rtty_send(char * szMessage) {
    char * p = szMessage;
    char * find = NULL;
    char c = *p;
    byte index = 0;
    short shift = 0;

    while(*p) {
        find = strchr(FIGURES, c);
        if(find != NULL) {
            index = find - FIGURES;
            if(index > 2 && shift != 1) {
                shift = 1;
                rtty_txByte(SHIFT_FIGS);
            }

            rtty_txByte(baudot[index]);
        }
        else {
            c = toupper(c);
            find = strchr(LETTERS, c);
            if(find != NULL) {
                index = find - LETTERS;
                if(index > 2 && shift != 2) {
                    shift = 2;
                    rtty_txByte(SHIFT_LTRS);
                }

                rtty_txByte(baudot[index]);
            }
        }

        p++;
        c = *p;
    }
}
```

C'est ici que les choses sérieuses commencent :-) . Les 5 premières lignes déclarent les variables dont nous allons avoir besoin dans cette méthode.

- p (char * p) qui va être un pointeur vers notre message (char * szMessage) passé en paramètre de la méthode.
- Find qui est aussi un pointeur de caractère que nous utiliserons plus tard.
- C (char) qui est un caractère est initilisé avec le premier de p (*p qui lui même pointe vers notre message :-) vous suivez ? Donc c contient la valeur 'C' qui est le premier caractère de notre message cq cq on3dex ...).
- index qui est de type byte initialisé à 0. On verra pourquoi.
- Shift qui est un entier (short donc max 255) qui lui aussi est initialisé à 0.

► A la ligne 7 nous entrons dans une boucle, boucle qui va continuer temps que la valeur pointée

par p est différente de 0. Donc nous allons boucler sur chaque caractère de notre message et une fois arrivé à la fin de celui-ci nous sortirons de la boucle.

► Ensuite ligne 8 nous regardons si la valeur de c (rappel, dans le premier passage de boucle celui-ci équivaut au premier caractère de notre message) est présente dans le tableau des figures, si c'est le cas, find pointera vers cette valeur dans le tableau des figures.

► C'est ce que nous testons à la ligne 9 et si c'est le cas nous entrons dans le if ligne 10. A cette ligne nous calculons l'index (la position) dans le tableau des figures de notre caractère.

► Ligne 11, si l'index est plus grand que 2 et que la valeur de shift est différente de 1 nous entrons dans la condition. Pourquoi cette condition ? Tout d'abord, pourquoi plus grand que 2 ? car dans les tableaux FIGURES et LETTERS les deux premiers caractères sont identiques, donc nous n'avons pas besoin d'envoyer un signal de shift au décodeur. Maintenant pour la condition qui implique que shift doit être différent de 1 et bien si c'est le cas, cela veut dire que du côté récepteur on s'attend à recevoir une lettre (ou indéterminé si rien n'a encore été envoyé) et que donc il faut envoyer le signal de shift vers le décodage de figures.

► Donc si on est dans ce cas là, on rentre dans le if (ligne 12 et 13), on met la variable shift à 1 et on appelle la méthode rtty_txByte en passant en paramètre la valeur hexadécimal qui représente le shift vers les figures (cf tableau du code baudot).

► Ensuite ligne 16 nous appelons encore la méthode rtty_txByte mais cette fois avec le code correspondant à la figure que nous voulons transmettre (la position dans le tableau baudot définie par la variable index).

► Sinon (ligne 18) c'est que c'est une lettre que nous voulons transmettre et donc on fait la même logique sauf que l'on va utiliser le tableau LETTERS pour retrouver l'index à aller chercher dans le tableau baudot et que si l'on doit faire un shift vers l'envoi de lettre, c'est le code correspondant à ce dernier qui sera transmis dans la deuxième condition.

► Aux lignes 30 et 31, on incrémente le pointeur de la zone mémoire pointée par p pour passer au caractère suivant et ensuite nous affectons ce caractère à la variable c et la boucle continue ainsi jusque-là fin du message (szMessage).

```
void rtty_txByte(byte b) {
    rtty_txBit(0); // Start bit

    // Send byte, bit by bit starting from left (MSB)
    for(byte i = 5; i > 0 ; i--) {
        if(b & (1 << (i-1)))
        {
            rtty_txBit(1);
        }
        else {
            rtty_txBit(0);
        }
    }

    rtty_txBit(1); // Stop bits
    rtty_txBit(1);
}
```

Nous voici dans la méthode rtty_txByte, voyons ce qu'elle fait. En fait cette méthode va se charger du séquencement de l'envoi des lettres et figures.

A la première ligne nous appellons la méthode rtty_txBit avec comme paramètre 0. Cela va envoyer le start bit (cf le tableau du code ci-dessus).

A la ligne 4 nous entrons dans une boucle for qui va itérer 5 fois (i = 5, i = 4, ..., i = 1). Cette boucle va nous permettre de décomposer notre byte et d'envoyer les données bit par bit en commencant par la gauche (comme dans le tableau, sinon l'ordre inverse n'enverrai pas le bon caractère ou la bonne figure). Voici un tableau qui explique la condition du if ligne 5 :

Avant de regarder ce tableau il faut d'abord savoir que l'instruction a << x représente un décalage de x bits vers la gauche de la valeur a et a & b représente un "ET" logique entre a et b (1 & 1 = 1, 1 & 0 = 0, 0 & 1 = 0, 0 & 0 = 0).
Ceci étant dit, continuons.

La première colonne donne la valeur résultante de l'expression qui va nous permetre de créer le masque à appliquer pour notre & logique.

La seconde donne le résultat de l'opération et logique entre notre masque et la valeur c (dans notre exemple 11010, la lettre J ou la figure ').

La dernière colonne donne le résultat final de toute la condition.

```
c = 11010
iteration 1
        (1 << (5-1)     (c & (1 << (i-1))     ((c & (1 << (i-1))) != 0)
         10000          11010
                        10000                 16 différent de 0
                        -----
                        10000                 vrai

iteration 2
        (1 << (4-1)     (c & (1 << (i-1))     ((c & (1 << (i-1))) != 0)
         01000          11010
                        01000                 8 différent de 0
                        -----
                        01000                 vrai

iteration 3
        (1 << (3-1)     (c & (1 << (i-1))     ((c & (1 << (i-1))) != 0)
         00100          11010
                        00100                 0 différent de 0
                        -----
                        00000                 faux

iteration 4
        (1 << (2-1)     (c & (1 << (i-1))     ((c & (1 << (i-1))) != 0)
         00010          11010
                        00010                 2 différent de 0
                        -----
                        00010                 vrai

iteration 5
```

(1 << (1-1)	(c & (1 << (i-1))	((c & (1 << (i-1))) != 0)
00001	11010	
	00001	0 différent de 0

	00000	faux

Suivant que le résultat soit vrai ou faux, on enverra respectivement 1 ou 0. Ensuite nous terminons cette méthode avec l'envoi de 2 stop bits.

```
void rtty_txBit(bool bit) {
    if(bit) {
        digitalWrite(DTX, HIGH);
    }
    else {
        digitalWrite(DTX, LOW);
    }

    delay(SPEED);
}
```

La méthode rtty_txBit est très simple. Celle-ci prend en paramètre la valeur du bit à transmettre et si celui-ci est égal à 1 (ligne 1) on appelle la méthode digitalWrite d'arduino pour envoyer une tension de 5 V sur la pin DTX (la pin 6 sur l'arduino) et si la valeur du bit est égal à 0 (ligne 4) alors nous faisons la même chose mais nous mettons le voltage de la pin à 0 V.

Pour finir nous faisons une pause de 22 milli secondes (pour avoir une vitesse de 45.45 bauds).

Petite précision ! Ici nous utilisons digitalWrite, cette partie du code peut changer (et nous le verrons dans la seconde partie de l'article) suivant la méthode utilisée pour transmettre les données à notre émetteur (en FSK) ou à nos enceintes (en AFSK). Nous verrons trois méthodes dans cet article :

- Envoyer les données via l'utilisation d'une pin I/O permettant l'utilisation de la fonction PWM des microcontrolleurs.
- Envoyer les données via une pin I/O "numérique", le switch entre les fréquences se faisant à l'aide d'un diviseur de tension.
- Envoyer les données sous forme de sons sur deux fréquences AF différentes. Nous ferrons ici de l'AFSK (audio FSK).

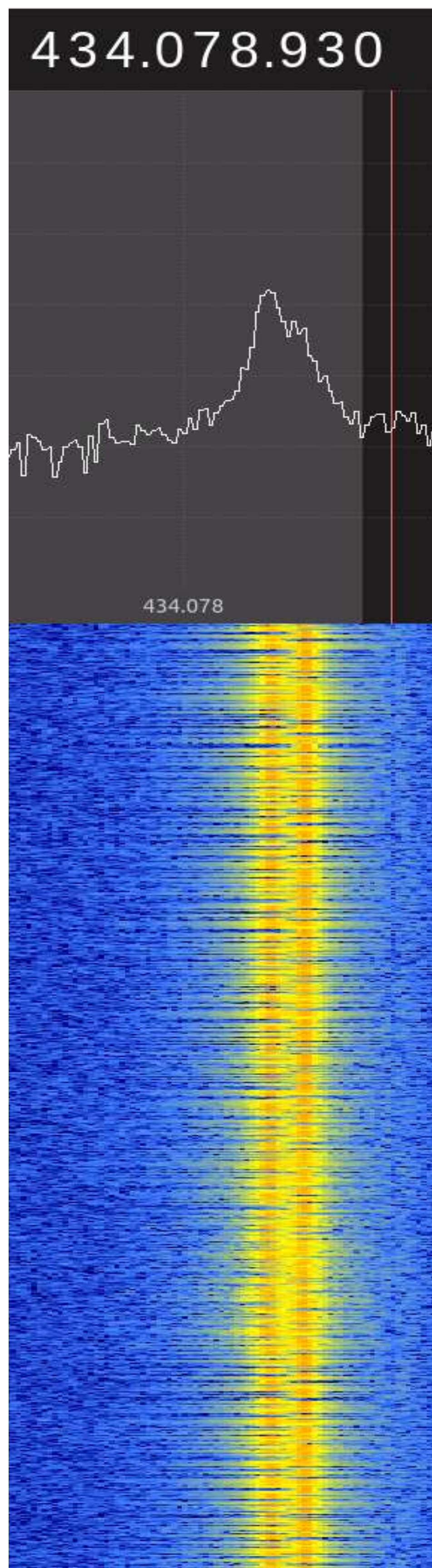
Maintenant que nous avons coder notre encodeur de données, il ne nous reste plus qu'a compiler la solution et l'uploader sur notre arduino :

```
Done uploading.

                                  Block Poll               Page                       Polled
           Memory Type Mode Delay Size  Indx Paged  Size   Size #Pages MinW  MaxW   ReadBack
           ----------- ---- ----- ----- ---- ------ ------ ---- ------ ----- ----- ---------
           eeprom        65    20     4    0 no       1024    4      0  3600  3600 0xff 0xff
           flash         65     6   128    0 yes     32768  128    256  4500  4500 0xff 0xff
           lfuse          0     0     0    0 no          1    0      0  4500  4500 0x00 0x00
           hfuse          0     0     0    0 no          1    0      0  4500  4500 0x00 0x00
           efuse          0     0     0    0 no          1    0      0  4500  4500 0x00 0x00
           lock           0     0     0    0 no          1    0      0  4500  4500 0x00 0x00
           calibration    0     0     0    0 no          1    0      0     0     0 0x00 0x00
           signature      0     0     0    0 no          3    0      0     0     0 0x00 0x00

         Programmer Type : Arduino
         Description     : Arduino
         Hardware Version: 2
         Firmware Version: 1.16
         Vtarget         : 0.0 V
         Varef           : 0.0 V
         Oscillator      : Off
         SCK period      : 0.1 us

avrdude: AVR device initialized and ready to accept instructions

Reading | ################################################## | 100% 0.00s

avrdude: Device signature = 0x1e950f (probably m328p)
avrdude: reading input file "/tmp/arduino_build_864193/rtty-divider.ino.hex"
avrdude: writing flash (1318 bytes):

Writing | ################################################## | 100% 0.42s

avrdude: 1318 bytes of flash written
avrdude: verifying flash memory against /tmp/arduino_build_864193/rtty-divider.ino.hex:
avrdude: load data flash data from input file /tmp/arduino_build_864193/rtty-divider.ino.hex:
avrdude: input file /tmp/arduino_build_864193/rtty-divider.ino.hex contains 1318 bytes
avrdude: reading on-chip flash data:

Reading | ################################################## | 100% 0.31s

avrdude: verifying ...
avrdude: 1318 bytes of flash verified

avrdude done.  Thank you.
```

Ceci cloture la **première partie de l'article** concernant le RRTY. J'espère que cela vous aura été utile.
Le code complet se trouve sur Github à cette adresse : https://github.com/onxdex/rtty-divider

En guise de consolation, voici déjà quelques images du résultat final de notre émetteur (FSK avec diviseur de tension) :

Dans gqrx :



Dans fldigi :

Par Julien ON3DXE

Cet article est la propriété du site ON5VL