# Décodage de trames RC5 sous Arduino

François Pessaux

December 22, 2014

## 1 RC5

Le protocole RC5 (`http://fr.wikipedia.org/wiki/RC5_(protocole)`) est un mécanisme de transmission d'information binaire utilisé par les télécommandes infrarouge en particulier pour les appareils audiovisuels. Il s'appuie sur un codage Manchester (`http://fr.wikipedia.org/wiki/Codage_Manchester`) généralement transmis à une fréquence de 36KHz (ce que nous considérerons dans tout ce qui suit).

Une trame se compose de 14 bits:

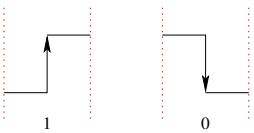
- 2 bits à 1 de synchronisation.
- 1 bit de répétition qui change à chaque nouvel appui consécutif sur une même touche.
- 5 bits d'adressage (définit à quel équipement est destiné la trame).
- 6 bits de commande (décrit la touche pressée).

Le détail des valeurs d'adresse et de commande se trouvent un peu partout sur Internet et n'est pas l'objet de notre document.

Chaque bit a une longueur de 1778 $\mu s$, ce qui donne une durée totale de trame de $14 \times 1778\ \mu s = 24.892\ ms$. Après chaque trame un silence de 88.886 $ms$ est observé. Donc, la période totale entre 2 trames est de $24.892\ ms + 88.886\ ms = 113.778\ ms$.

## 2 Manchester

Ce codage représente les bits de la manière suivante:



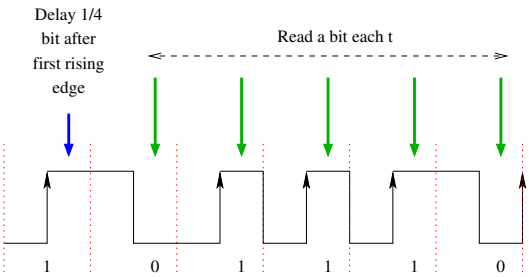

Attention, nous faisons explicitement la différence entre niveau haut/bas *logique* et *physique* (et idem pour front montant/descendant). Par niveau *logique* haut, nous considérons état haut "sur le papier" (et inversement pour bas). Ainsi un niveau *logique* bas pendant $\frac{1}{2}$ bit suivi d'un niveau *logique* haut pendant l'autre $\frac{1}{2}$ bit représente un 1.

Similairement, un front montant *logique* représente un 1 et un front descendant *logique* représente un 0.

Cette différence explicite entre *logique* et *physique* est due au fait que les récepteurs infrarouge sont à un niveau *physique* haut pour signaler l'absence de signal IR et bas pour signaler la réception d'un signal IR. Autrement dit, *logique* et *physique* sont inversés. Nous raisonnerons uniquement en terme *logique* (le lien avec *physique* se faisant simplement en inversant le sens du front sur lequel une interruption sera générée – ce qui sera vu plus tard).

# 3 Décodage: solution non retenue

Puisque une trame commence toujours par un bit à 1, il faut attendre le premier front montant qui signale un début de transmission (donc de bit à 1). Un petit schéma valant mieux qu'un long discours, on se rend compte que si on attend $\frac{1}{4}$ de bit après le premier front montant et que l'on lit ensuite le niveau du signal à une période de 1 bit (que nous appèlerons $t$ dans toute la suite), alors on récupère directement le bit émis.




Cette solution a 2 inconvénients. D'une part elle n'est pas très robuste aux perturbations du signal, d'autre part elle nécessite une attente active qui monopolise le CPU pendant toute la durée du traitement.

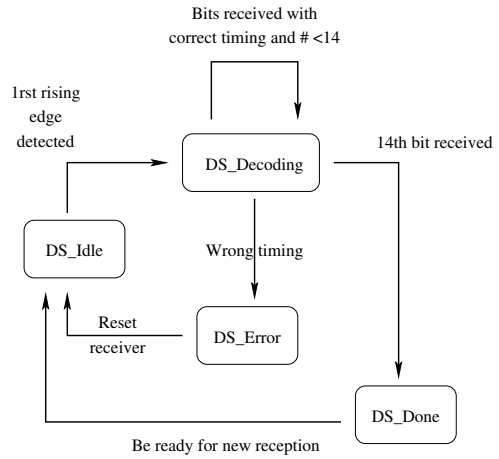
# 4 Décodage: la solution retenue

## 4.1 Principe

Le décodage des trames va être implémenté par interruption en détectant le temps écoulé entre 2 fronts montants *logiques* (donc *physiques* descendants). Notre décodeur peut être dans 4 états:

- En attente de réception d'un premier bit de trame (`DS_Idle`).
- En cours de réception d'une trame (`DS_Decoding`).
- Avoir terminé la réception d'une trame (`DS_Done`).
- En erreur (`DS_Error`).

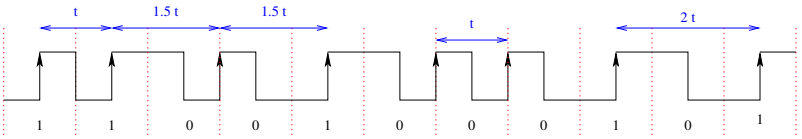
On implémente donc ce comportement par un automate à 4 états:




Tant qu'aucun front montant *logique* initial n'a été détecté l'automate reste dans l'état `DS_Idle`.
Sur la réception du premier front montant *logique*, il passe dans l'état `DS_Decoding`, mémorise le premier bit (forcément à 1) et incrémente le nombre de bits lus.
Dans l'état `DS_Decoding`, sur chaque front montant *logique* il calcule le temps écoulé depuis le précédent front montant. S'il est compatible avec les équation ci-dessous, alors un ou deux bits de plus sont reçus. On mémorise la date du front montant courant, on mémorise les bits reçus et on incrémente le nombre de bits reçus en accord. Si le nombre de bits reçus est égal à 14 alors on passe dans l'état `DS_Done`, sinon on reste en `DS_Decoding`. Si le temps écoulé est incompatible avec les équations, alors il y a eu une erreur de transmission/réception et l'automate passe en état `DS_Error`.
Dans l'état `DS_Done` exploite les bits reçus, puis on remet l'automate dans l'état `DS_Idle` pour attendre la trame suivante.
Dans l'état `DS_Error`, on peut générer du debug, mais surtout on attend la durée d'une trame sans rien faire avant de remettre l'automate dans l'état `DS_Idle` pour attendre la trame suivante. Cette attente nous permet d'être certain que la lecture ne reprendra pas au milieu de la trame que l'on vient de rejeter.

## 4.2 Les équations

Pour déterminer la suite de bits, on s'aide du jeu d'équations qui va suivre que l'on peut illustrer sur un schéma. Pour rester robuste aux variations de timing dans le signal, on mesurera le temps écoulé entre 2 fronts "à une tolérance près" ($30\mu s$ dans notre programme).



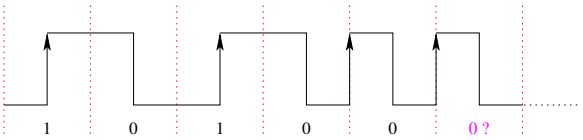

$$\delta = t \Rightarrow \left\{ bit_{i+1} = bit_i \right.$$

$$\delta = 1.5t \Rightarrow \left\{ \begin{array}{l} bit_i = 0 \Rightarrow bit_{i+1} = 0, bit_{i+2} = 1 \\ bit_i = 1 \Rightarrow bit_{i+1} = 0 \end{array} \right.$$

$$\delta = 2t \Rightarrow \left\{ \begin{array}{l} bit_i = 0 \Rightarrow \text{error} \\ bit_i = 1 \Rightarrow bit_{i+1} = 0, bit_{i+2} = 1 \end{array} \right.$$

### 4.3 La subtilité

Travailler sur la détection de fronts *logiques* montants est très pratique car on se synchronise très simplement dès la réception d'un tel front. Le gros inconvénient est le fait que si le dernier bit d'une trame est un zéro, l'automate va rester à attendre indéfiniment un front montant qui ne viendra pas. Le schéma ci-dessous décrit un tel scénario. Les avant-derniers bits à 0 ne gênent pas puisqu'un front montant vient les signaler grâce au bit qui suit. Par contre, ce n'est pas le cas pour le dernier.




Pour contourner le problème il faut détecter que le signal est à l'état *logique* bas $\frac{1}{2}$ bit après le dernier front *logique* montant.

On peut décider de faire ce traitement dans la fonction `loop` par attente active. Mais ce n'est pas souhaitable car notre `loop` peut nous servir à faire autre chose pendant l'attente de réception.

L'autre solution est d'utiliser une autre interruption, un timer que l'on armera lorsque l'on aura reçu 13 bits. On réglera le timeout de ce timer sur la durée de $\frac{1}{2}$ bit. La routine de traitement de cette interruption consistera alors simplement à désactiver le timer (pour éviter qu'il ne se re-déclenche), mémoriser le dernier bit qui est un zéro et passer l'automate dans l'état `DS_Done`.

Il faudra également désactiver le timer dans tous les cas où le dernier ($14^{\text{ème}}$) bit reçu était en fait un 1 afin que le handler ne se déclenche pas sur timeout alors que le dernier bit a déjà été obtenu. De manière similaire, il faudra le désactiver dans le cas où l'automate passe en erreur (au cas où l'erreur serait survenue lors de l'attente du dernier bit).

## 5 Le code

Afin d'utiliser l'interruption 0, on branche le récepteur IR sur la broche 2. En effet ce couple est lié matériellement.

La variable `prev_change_date` mémorise la date en $\mu s$ à laquelle le dernier front montant a été détecté.

La constante `TIMING_TOLERANCE` définit le nombre de $\mu s$ autour des timings théoriques que l'on considère comme valides. Autrement dit, les vérifications de délais écoulés entre 2 fronts montants *logiques* seront faits $\pm$`TIMING_TOLERANCE`.

La variable `prev_bit` mémorise la valeur du bit précédemment lu. Lors de la réception du premier bit de trame cette valeur n'a pas d'impact et est initialisée à 1 puisque le premier bit est forcément un 1.

La variable `nb_read_bits` mémorise le nombre de bits lus au cours du décodage.

La variable `change_delta` sert à calculer le temps écoulé entre la réception du précédent front montant et le courant. Cette variable devrait être locale au handler d'interruptions de front montant. Mais pour pouvoir debugger et

émettre un message d'erreur mentionnant le délai calculé en cas d'erreur, cette variable doit être accessible à la fonction `loop`. Donc... être globale.

La macro `STOP_TIMER` sert juste à inliner lisiblement le code de désactivation du timer sans devoir passer par un appel de fonction.

La fonction `start_timer1_half_bit` sert à initialiser le timer 1 afin qu'il se déclenche au bout du temps correspondant à $\frac{1}{2}$ bit plus la tolérance que l'on s'autorise (`TIMING_TOLERANCE`). La formule de cuisine servant à calculer la valeur à mettre dans `OCR1A` est issue des informations données dans le datasheet du ATmega368 (c.f. `http://www.atmel.com/devices/atmega328.aspx`). On remarquera que le prescale utilisé est 64 car sa plage de 524.288 $ms$ pour une précision de $4\mu s$ est suffisante pour attendre $\frac{1}{2}$ bit (c.f. `http://playground.arduino.cc/Code/Timer1`). On aurait très bien pu prendre également un des prescales inférieurs.

Le handler d'interruption timer 1 est déclaré par `ISR (TIMER1_COMPA_vect)`. Son travail consiste à stopper le timer 1 (celui-là même qui l'a déclenché). Si ce timer est déclenché c'est qu'il s'est écoulé $\frac{1}{2}$ bit sans qu'un front montant n'ait été détecté pour le dernier bit de la trame (puisque le handler d'interruption de front montant désactive le timer si un front montant est détecté pour le dernier bit). Ainsi, le dernier bit de la trame est forcément un 0, ce qui clôt la trame et fait passer l'automate dans l'état `DS_Done`. Pour mémoriser ce dernier bit, on fait juste un décalage à gauche 1 fois de la valeur `read_bits`.

La fonction `pin_2_logical_rising_occurred` est le handler d'interruption attaché à la détection de fronts montants *logiques* ... donc à la détection de fronts *physiques* descendants (donc lié avec le flag `FALLING` lors de l'appel à la fonction `attachInterrupt`). Ce handler commence par récupérer la date courante en $\mu s$ puis discrimine sur l'état courant de l'automate (technique habituelle d'implémentation des automates par un `switch (state) case ...: state = ... ; break ;` ).

- Dans l'état `DS_Idle`, s'il est invoqué c'est qu'un front montant *logique* a été détecté. Donc c'est le premier bit de la trame qui est arrivé. Logiquement, on mémorise la date courante pour s'en souvenir comme date précédente. L'automate passe alors en état `DS_Decoding`, le bit précèdent (pour le coup prochain) est donc à 1 (`prev_bit = 1`), on a lu 1 bit (`nb_read_bits = 1`) et c'est un 1 (`read_bits = 1`).
- Dans l'état `DS_Done`, la précédente trame a été reçue et n'a *a priori* pas encore été traitée puis que l'on n'est pas prêt à lire une trame suivante. Donc on reste dans le même état en ne faisant rien. Notez le "fall through" qui embraye directement sur le cas de l'état suivant (`DS_Error` décrit ci-après).
- Dans l'état `DS_Error`, on était précédemment dans un état d'erreur qui n'a pas encore été réglé puisque l'on n'est pas encore prêt à lire une trame suivante. Donc on reste dans le même état en ne faisant rien.
- Dans l'état `DS_Decoding`, on est en cours de décodage d'une trame. On a déjà reçu le premier bit de départ (à 1) et jusqu'à présent on n'est pas en erreur. Donc on calcule de temps écoulé depuis le dernier front montant. En accord avec les équations précédentes, on détermine le ou les bits reçus (on les mémorise en décalant le mot contenant les précédents et

en rajoutant ou non la valeur du ou des nouveaux bits par un ou bit-à-bit (`read_bits = (read_bits << ...) | ...`). Puis on incrémente le nombre de bits lus (`nb_read_bits`) en fonction. De même, on mémorise le dernier bit lu (`prev_bit = ...`).
Et là, attention, on regarde (dans tous les cas où les timings sont respectés) si l'on vient de lire l'avant dernier ($13^{\text{ème}}$) bit. Si oui alors on lance le timer pour détecter l'éventuel timeout dû à un bit final à 0 (appel de `start_timer1_half_bit`). Et si non, on regarde si l'on vient de lire le dernier ($14^{\text{ème}}$) bit. Si c'est le cas alors on a fini la trame et l'automate peut passer en état `DS_Done`.
Et si jamais aucun timing n'a été respecté, alors l'automate passe en état d'erreur (en stoppant le timer au cas où il aurait été lancé et que l'erreur soit survenue en attendant le dernier bit).

La fonction `setup` ne fait pas grand chose de terrible. Elle configure les ports reliés aux LEDs afin qu'ils soient en sortie. Le plus important est qu'elle attache notre handler d'interruptions (qui détecte les fronts montants *logiques*) à la survenue de changement de fronts *physiques* descendants (`attachInterrupt (IT_NUM, pin_2_logical_rising_occurred, FALLING) ;`.

La fonction `loop` ne fait qu'afficher des informations en cas d'erreur ou les bits reçus en cas de succès.

**Remarque sur l'acquisition de la date** : nous utilisons la fonction `micros` pour récupérer la date courante. Hors cette fonction fait un débordement au bout d'environ 70 minutes pour revenir à 0 (c.f. `http://arduino.cc/en/reference/micros`). Il en découle qu'au bout de 70 minutes le calcul du temps écoulé entre 2 fronts peut être faux. Il y a deux solutions à cela.

La première consiste à ignorer en se disant que si l'on perd une trame, on récupèrera bien la suivante: l'utilisateur au bout de la télécommande appuiera de nouveau en ne voyant rien se passer, ou bien le dispositif enverra une prochaine trame et celle perdue sera ignorée. Ceci est possible dans le cas où la réception des trames n'est pas critique.

Dans le cas contraire, il faut prendre en compte le débordement par un modulo, ce qui implique de s'autoriser une division à chaque calcul de temps écoulé.

Dans notre cas, nous avons simplement ignoré ce problème de débordement en nous autorisant à perdre une trame dont la réception s'étalerait sur les dernière $\mu s$ de chaque intervale de 70 minutes.

# 6 Erreurs faciles à commettre et pénibles à trouver

Le principe algorithmique d'un tel décodage est très simple. Cependant, un certain nombre d'erreurs peuvent vous pourrir la vie. De manière non exhaustive suit une petite liste, un "best of".

- Mettre une variable locale à la fonction `loop` en comptant qu'elle soit rémanente d'un "tour" de `loop` à l'autre. Non, non, non, la fonction

`loop` est bien **appelée** de manière infinie et non inlinée. Donc comme toute fonction ses variables locales disparaissent à la fin de son "tour" d'exécution.

- Se tromper de télécommande pour tester: j'ai longuement tenté de comprendre pourquoi mon code ne fonctionnait pas jusqu'à troquer ma télécommande de lecteur CD contre celle ... de la TV. Et là, ça fonctionnait nettement mieux.
- Choisir une tolérance de timing trop petite. Initialement j'avais opté pour 20 $\mu s$ ce qui s'est avéré trop juste. Les signaux émis par ma télécommande de test n'étaient pas suffisamment justes par rapport aux timings théoriques.
- Oublier d'interrompre le timer une fois le dernier bit reçu.
- Oublier d'interrompre le timer en cas de passage en état d'erreur.
- Faire `Serial.print` dans le handler d'interruption pour débugger. Ça ne marche tout simplement pas.
- Faire des tonnes de `Serial.print` (dans `loop` par exemple) dans l'espoir de surveiller le déroulement de l'automate. La fonction `Serial.print` prend énormément de temps à s'exécuter et durant ce temps un paquet d'interruptions de fronts montants a le temps d'arriver. Autrement dit, vos messages de debug sont dépassés par le flot d'interruptions et ne sont plus pertinents.
- Oublier de définir comme `volatile` les variables partagées entre les fonctions et particulièrement modifées par les handlers d'interruptions.