RIGA Lorenzo 13018
4EO

Électronique générale Eurobot 2017

Introduction

L'architecture électronique du robot 2017 est restée fort similaire à l'année précédente. Une Raspberry-pi fait office de contrôleur principal, ce dernier communiquant avec trois arduino uno par le bus I²C. Les codes pour toutes ces cartes son expliqué dans un autre document.

Une arduino Uno se charge de la régulation des moteurs. Elle envoi les commandes au contrôleur moteur et reçoit les informations des roues codeuses traitées par le FPGA. Une autre arduino se charge de gérer les 5 capteurs ultra-son et la dernière contrôle les servos et le(s) dynamixel. La carte de puissance à également été revu et est documentée dans un autre document.

La mise à la masse commune de tous les modules électronique du robot doit à tout moment être assurée ! Cela semble banale dit comme ça, mais une erreur de notre part à ce niveau là a détruit 2 Raspberry-pi… oups… Cela sera détaillé plus loin dans ce document.

Buffer roues codeuses

Toute la gestion par FPGA des roues codeuses a été reprise de l'année précédente. Les impulsions en quadrature des roues codeuses (0-5V) sont envoyées dans un FPGA cyclone IV fonctionnant en 0 et 3.3V. Le FPGA renvoi ensuite une information de direction et de distance à l'arduino uno fonctionnant en 0 et 5V. Les signaux doivent donc être convertis deux fois aux bons niveaux de tension. LA carte existait déjà, mais de nombreuses micro fissures dans les pistes ont provoqué une non-fiabilité de la carte, jusqu'à sa destruction. Une version nettoyée à donc été refaite.

Cette carte est basée sur deux 74HCT245 tristate buffer dont le schéma se trouve ci-dessous. Un fait la conversion 5V → 3.3V et l'autre reconverti en 5V. Ce qui à changé par rapport à la version précédente est une mise à disposition de tous les buffers des deux chips pour plus de flexibilité. Pour faire ça il à fallu prévoir une possibilité de mise à la masse facile de toutes les entrées non utilisées. (Laisser une entrée flottante peut en effet provoquer une flopé de disfonctionnements). Cela nous amène au deuxième grand changement, les connecteurs. Les "convis" ne sont absolument pas fiable, un fil est trop vite arraché. Tous les connecteurs ont donc été remplacés par des pin header. Les sorties sont sur des header à une rangée et les entrées sur des headers à eux rangées. La deuxième rangée est là pour fournir une connexion facile à la masse. Si une entrée est inutilisée, elle peut être mise à la masse avec un jumper. ATTENTION en mettant une connexion sur une entrée de buffer, il ne faut surtout pas se tromper de rangée ! Si par erreur le fil (venant du FPGA par exemple) est branché sur la rangée de masse au lieu d'une entrée de buffer, la sortie du FPGA est franchement court-circuitée et peut être détruite.

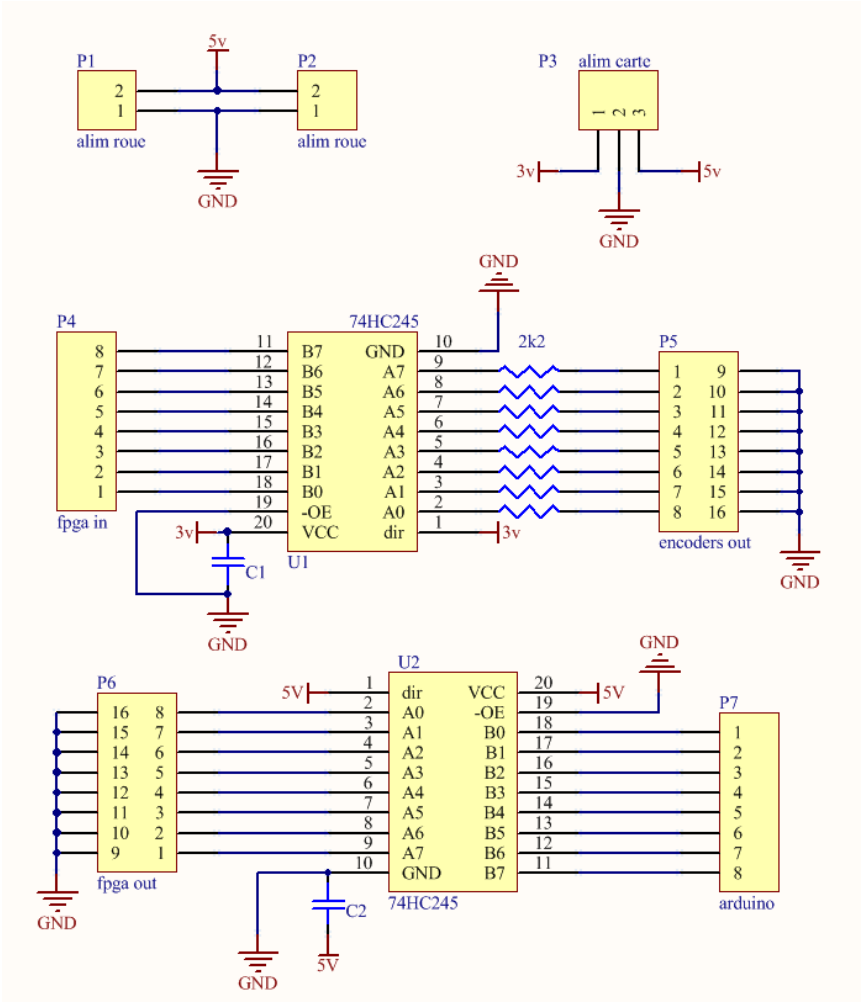


Figure 1 : Schéma carte buffers

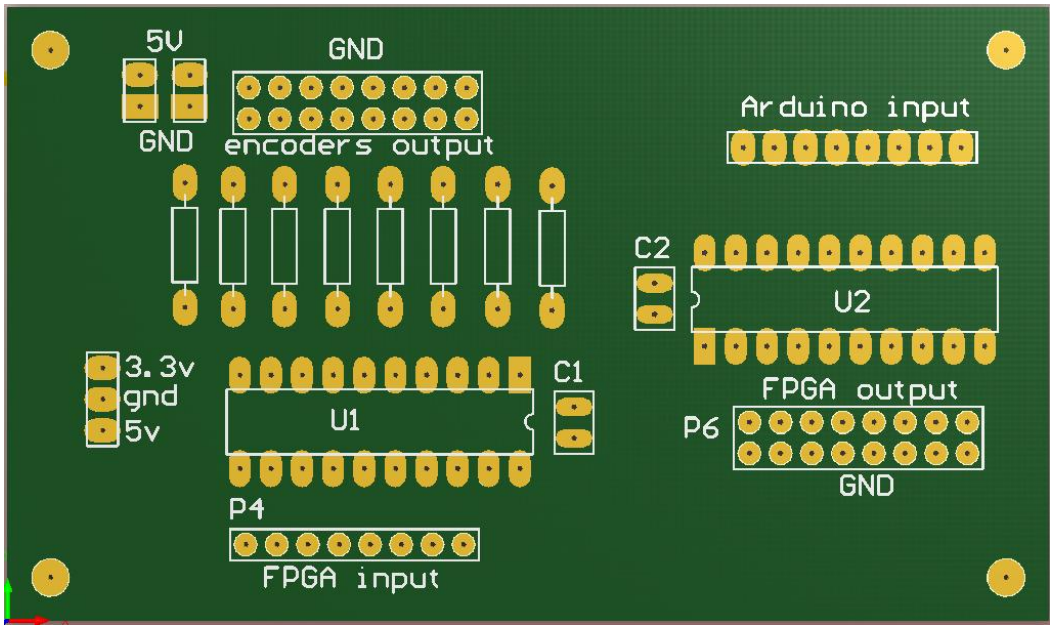


Figure 2 : carte buffers vue du dessus

Carte Arduinos

L'espace à bord du robot pour placé l'électronique était très très restreinte cette année. Or des arduinos uno prennent quand même pas mal de place et il faut encore en caser deux… Mais nous n'avions pas besoin de l'USB sur aucune des deux, ni de l'alimentation interne. Tout ce qui nous intéresse est le µC qui est un peu de style "flash and Forget" (cela n'a pas été tout à fait vrai).

Un petit PCB a donc été conçu sur lequel ont été placé deux ATMEGA328P-PU chacun dotés d'un quarts 16MHz. Ces deux chips sont reliés ensemble via un bus I²C, accessible de l'extérieur via un connecteur JST pour être relié à la Raspberry-Pi (Master).

Un des µC est connecté à un header 2 rangés pour les ultra-sons. Le pinout est repris dans les figures ci-dessous. Le deuxième µC à un peu plus de features autour. Ce dernier doit contrôler 3 servos donc 3 headers fournissent les connexions nécessaires. Ensuite il faut pouvoir contrôler des dynamixel via les pins Rx et Tx. Le petit circuit mentionné dans la datasheet des dynamixel avec des buffers tristate et un inverseur à donc été implémenté sur cette carte. Le connecteur dynamixel est muni d'un détrompeur, pas ceux des servos.

Les deux µC sont placés dans des sockets. Pour les programmer, on les retire pour les mettre dans le programmeur fait-maison puis on les replace dans leurs sockets respectifs.

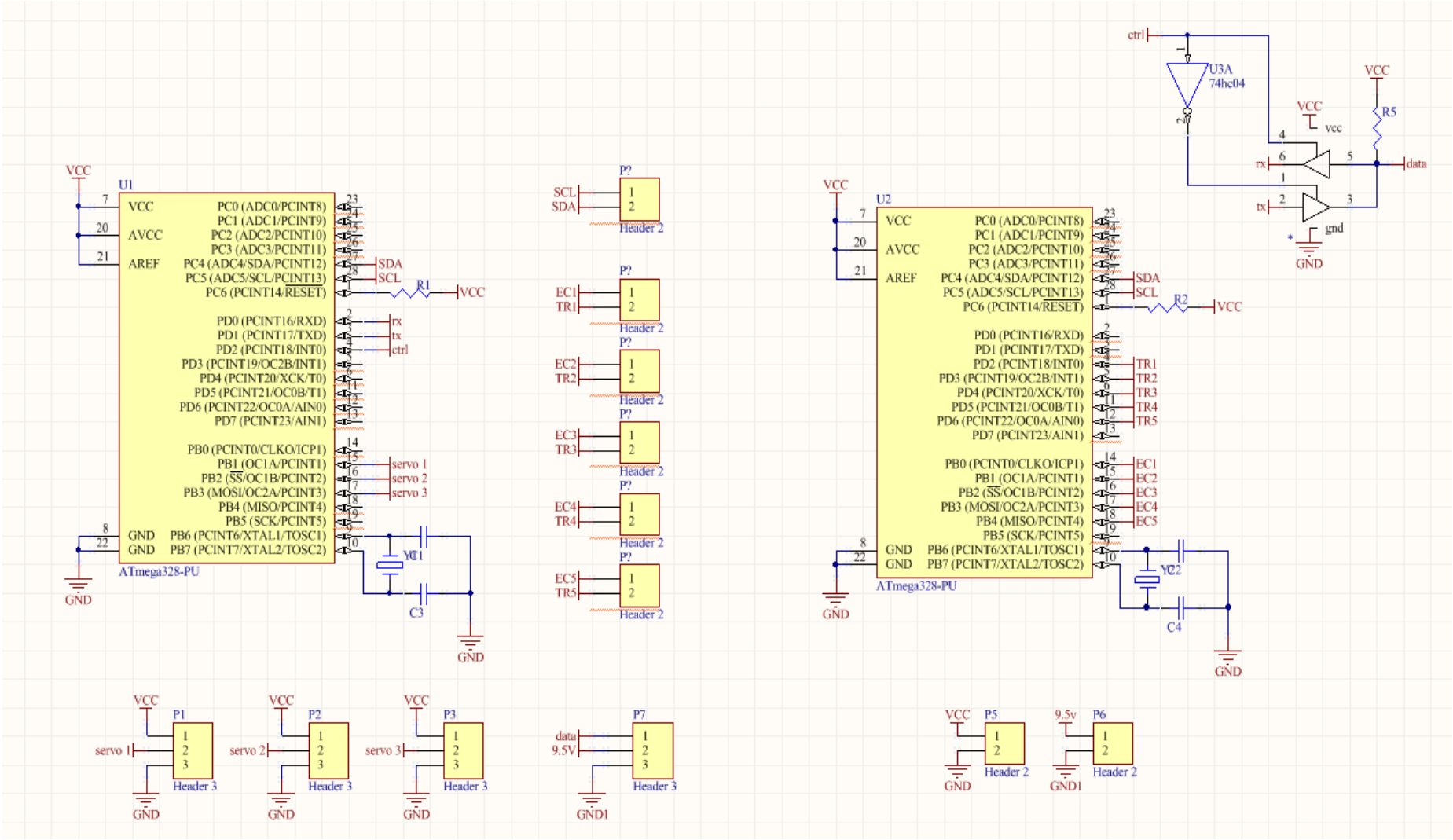
La carte est enfin munie d'une entrée 5V pour tous les circuits logiques et une entrée 9.5V pour les Dynamixel.

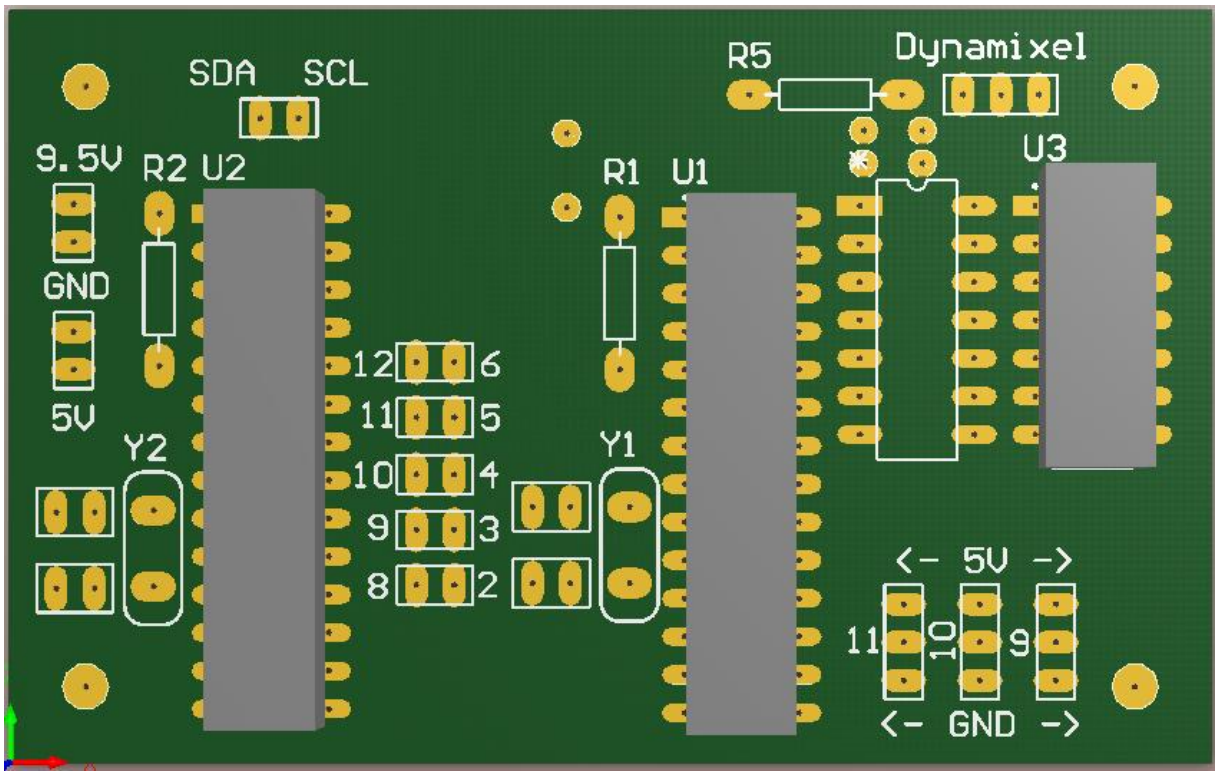
ERRATA

Deux erreurs ont été faites lors de la conception de cette carte. Elles ont été réparées ou contournées mais le lecteur doit être prévenu.

- **Connecteurs servos** : Par mégarde je me suis trompé dans le pinout pour les connecteurs servos. Normalement ce devrait être VCC (5V) au milieu et GND et PWM aux extrémités. C'est d'ailleurs une sécurité au cas où on le branche à l'envers, il n'y à pas moyen d'inverser les polarités. Cette sécurité est également implémentée sur les dynamixel. Sur la carte, j'ai mis PWM au milieu et VCC et GND aux extrémités… Au lieu de refaire toute la carte j'ai juste défait les connecteurs des servos et inversé les fils. Il ne faut pas oublier cette étape si on veut remplacer un servo !

- **Communication Dynamixel** : Le dynamixel communique via une communication série bidirectionnelle half-duplexe. Le fil data fait donc transiter des bits dans les deux sens. Pour cela l'arduino est doté de deux buffers tristate en antiparallèle. L'arduino génère ensuite une commande "ctrl" qui est niée par un 74hc04. Un buffer reçoit "ctrl" et l'autre reçoit "ctrl nié" ce qui assure qu'à aucun moment les deux ne soit actifs. Lors de la conception je me suis trompé en reliant ctrl au buffer Rx et ctrl nié au buffer Tx au lieu de faire l'inverse. Donc quand l'arduino essayait d'envoyer une commande Tx était bloqué et cette commande ne parvenait pas au dynamixel. Cette fois une réparation de fortune était possible en grattant des pistes et en soudant des petits fils. Ces réparations ne sont jamais élégantes et devrait être évitées le plus possible !! Voici cependant deux conseils : vérifiez que les pistes sont bien coupées avec le Mantis et/ou au testeur de continuité. Collez les fils sur la carte à la cyanolite. Ces fils sont souvent les premiers points de défaillances.


U1
ATmega328-PU
VCC
GND
AVCC
AREF
PC0 (ADC0/PCINT8)
PC1 (ADC1/PCINT9)
PC2 (ADC2/PCINT10)
PC3 (ADC3/PCINT11)
PC4 (ADC4/SDA/PCINT12)
PC5 (ADC5/SCL/PCINT13)
PC6 (PCINT14/RESET)
PD0 (PCINT16/RXD)
PD1 (PCINT17/TXD)
PD2 (PCINT18/INT0)
PD3 (PCINT19/OC2B/INT1)
PD4 (PCINT20/XCK/T0)
PD5 (PCINT21/OC0B/T1)
PD6 (PCINT22/OC0A/AIN0)
PD7 (PCINT23/AIN1)
PB0 (PCINT0/CLKO/ICP1)
PB1 (OC1A/PCINT1)
PB2 (SS/OC1B/PCINT2)
PB3 (MOSI/OC2A/PCINT3)
PB4 (MISO/PCINT4)
PB5 (SCK/PCINT5)
PB6 (PCINT6/XTAL1/TOSC1)
PB7 (PCINT7/XTAL2/TOSC2)
SDA
SCL
R1
rx
tx
ctrl
servo 1
servo 2
servo 3
C3
P?
Header 2
EC1
TR1
EC2
TR2
EC3
TR3
EC4
TR4
EC5
TR5
P1
P2
P3
Header 3
P7
data
9.5V
GND1
U2
U3A
74hc04
vcc
gnd
R5
data
R2
TR1
TR2
TR3
TR4
TR5
EC1
EC2
EC3
EC4
EC5
C4
P5
P6
9.5v



N.B. Les chiffres à coté des headers correspondent aux numéros de pin Arduino. Les masses pour les connecteurs 9.5v et 5v sont au même potentiel évidement mais ne sont pas connectée ensemble sur la carte pour éviter une boucle de masse et toute autre joyeuseté. Les deux masses sont connectées ensemble au niveau de la carte de puissance. Si on désire alimenter la carte avec une autre alim, il ne faudra pas oublier de prendre ça en compte.

Masses communes

Un petit mot sur ce sujet très important. Toutes les cartes électroniques du robot doivent impérativement rester à la même masse pour éviter tout problème de sur tension et garantir des niveaux logiques correctes. On garantissait cela au travers de la carte de puissance. Ce qu'il s'est passé en revanche c'est que pour programmer l'arduino pour les moteurs, on l'a débranchée de l'alimentation du robot pour la brancher à un pc. Mais celle-ci était toujours raccordée aux autres cartes via le bus I²C. Au moment ou l'arduino était débranchée de l'alim alors que tout le reste était sous tension, sa masse était flottante. Lorsqu'on la touche avec nos mains pleine de doigts chargés électriquement on à soudainement induit un choque électrostatique détruisant les transistors sur toutes les autres cartes reliées au bus. Cela est arriver deux fois…

Solution : pour éviter que cela se reproduise, on a rajouté une connexion de masse entre l'arduino et la Raspberry-pi pour être certain de garantir l'équipotentielle. Au prix de former une boucle de masse probablement mais bon, nous n'avions pas le choix.

I2C Raspberry-Pi/Arduino

La Raspberry-Pi travail avec des niveaux logiques de 3.3v et l'arduino 5v, pourtant ils peuvent communiquer en I2C "sans problème" à quelques conditions. Celles-ci sont souvent indiquées sur le net, mais il est bon de les rappeler et de les expliquer pour éviter de couteuses erreurs.

L'I2C est un bus à collecteur ouvert. Ce qui signifie qu'il faut le tirer à vcc via des résistances de pull-up et la transmission se fait en tirant le bus à la masse. Le lecteur attentif verra que sur aucun circuit ne se trouve de résistance de pull-up. C'est parce que la Raspberry-pi à déjà des résistances pull-up de 1.8K en interne qui tire le bus à 3.3V. Niveau logique que l'arduino sait interpréter. Si par mégarde quelqu'un vient rajouter des résistances de pull-up vers 5v deux choses peuvent arriver. Soit les résistances sont grande (souvent en numérique on met 10K à toutes les sauces) et alors les 1.8K l'emporte et le bus reste à 3.3v. Soit la résistance est faible (1k est moins courant mais régulièrement utilisé aussi) et le diviseur de tension fait que le bus est tiré à 4.4V pouvant potentiellement détruire la Raspberry-pi.

Enfin il est de notoriété publique que le bus I2C est très sensible aux perturbations et que le bus doit être le plus court possible. Pas vraiment le cas sur le robot… Le bus I2C est formé de deux fils qui se baladent un peu proche de l'alim à découpage et du contrôleur moteur… L'idéal aurait vraiment été de faire un shield Raspberry-pi muni des 3 ATMEGA328P-PU pour minimiser l'utilisation de place et réduire la taille du bus au minimum. Pour cela il faudrait un moyen de programmer facilement les trois µC directement sur le shield sans devoir les retirer de leurs sockets. Je suis actuellement en train de travailler sur un tel programmeur basé sur celui qui est expliqué plus bas. Si il aboutit je tacherai de le faire savoir.

Programmeur Arduino DIY

Le tuto en annexe explique très bien comment utiliser une arduino uno pour programmer un ATMEGA328P-PU mais ce document à quelques années donc voici quelques précisions.

Tout d'abord un shield arduino uno est présent au labo pour ne pas devoir utiliser une breadboard et des petits fils.

Dans l'étape 1 ils mentionnent d'utiliser Arduino 1.0.1 mais si vous utilisez ma dernière version de l'ide ça marchera tout aussi bien. Ensuite ils compliquent les choses pour dire qu'il faut juste uploader sur l'arduino le code "ArduinoISP" présent dans les exemples.

A l'étape 3 ils demandent de copier le chemin vers le Hex avec l'ide en mode verbos. Quand vous compilez il y a beaucoup de lignes, ce que vous chercher ce trouve à la fin de la troisième ligne en partant du bas.

A l'étape 4 : quand vous téléchargez WinAVR, AVRdude se trouve dans le dossier bin.

Ensuite dans l'exemple vous pouvez suivre aveuglement les explications en changeant juste le COM port pour qu'il convienne et en vous assurant dans le gestionnaire de périphérique que votre arduino uno est bien avec un baud rate de 19200. (De base il est de 9600).

Avant de pouvoir uploader un code sur un ATMEGA328P-PU **neuf**, il faut d'abord faire une manip en plus pour configurer le µC pour fonctionner avec un oscillateur externe de 16Mhz.

Il faut juste envoyé la commande suivante (en changeant le numéro de port évidement) :

```
avrdude -P COM6 -b 19200 -c avrisp -p m328p -U lfuse:w:0xFF:m -U hfuse:w:0xDE:m -U efuse:w:0x05:m
```

Votre chip est prêt à être programmé !

Programming a Target AVR using Arduino Uno board loaded with ArduinoISP

(April 21, 2015, author: B. Lazar)

Arduino Uno + **ArduinoISP** form an ordinary **ISP programmer** that is used by **avrdude** to program various 8 bit AVRs with any kind of hex file.
*No bootloader is needed, no hardware removal from **Arduino Uno** is necessary*

It may happen that you have an **Arduino Uno** that you bought, played a bit with it and now you want to move further, to build your own board, which can be just a simple AVR microcontroller + an oscillator + a LED, but you do not have an ISP programmer.

In reality, **Arduino Uno** is everything you need to program a large variety of AVRs. You just have to follow the four steps described below:

1) Open **Arduino 1.0.1** and upload **ArduinoISP** to **Arduino Uno**.
2) Connect the **ISP pins** of **Arduino Uno** to the **Target AVR** as indicated in the Fig.1.
3) Write your source code, compile it and generate the hex file.
4) Use **avrdude** command lines to upload the hex file into the **Target AVR**.

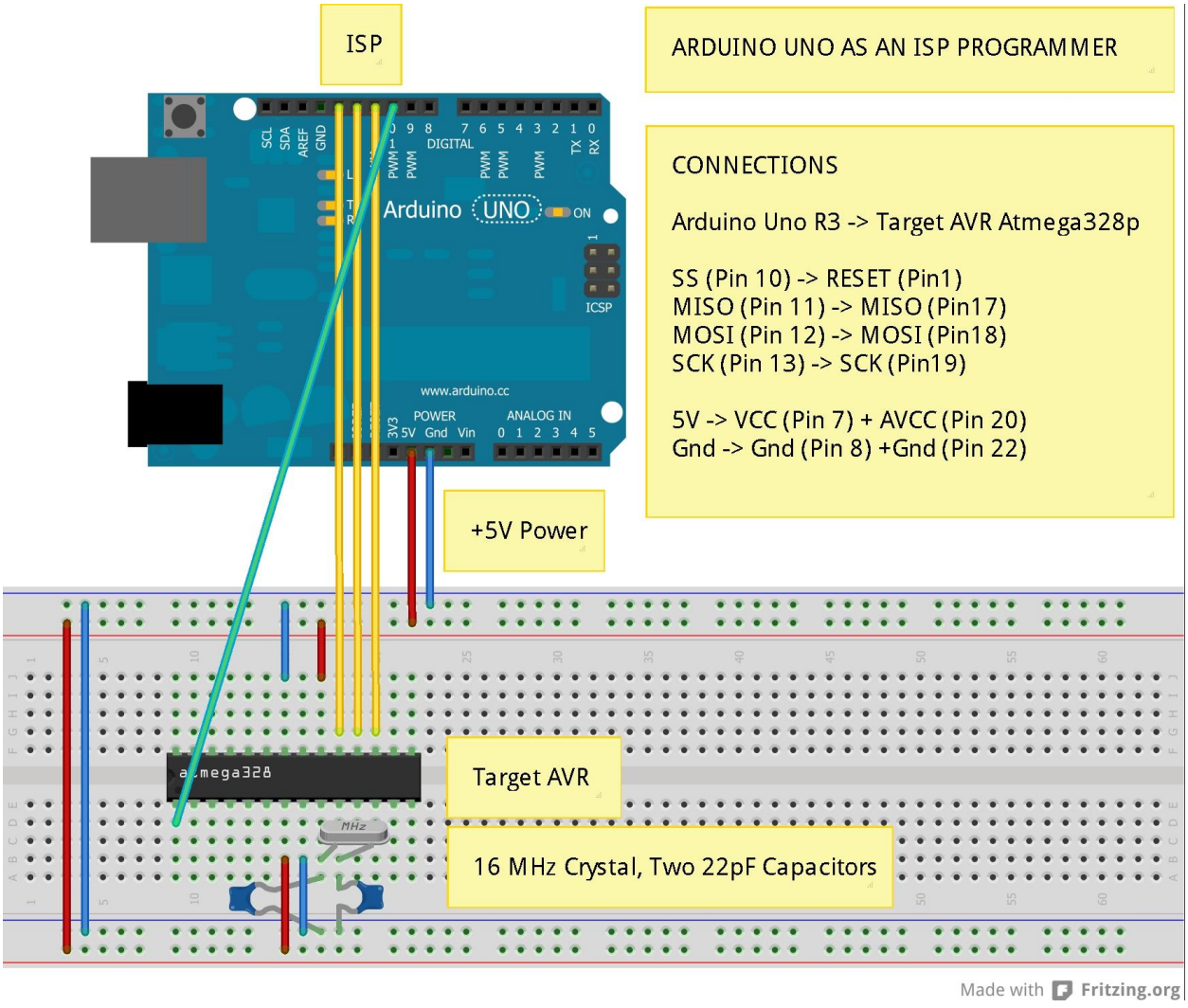


Fig. 1

Each of the four steps explained in detail:

(1) Open Arduino 1.0.1 and upload ArduinoISP to Arduino Uno

Start **Arduino 1.0.1**, go to **Files->Examples** and open **ArduinoISP**. Then from **Tools->Board** select **Arduino Uno**, from **Tools->Programmer** select **AVR ISP** or **Arduino as ISP**. Access **Tools->Serial Port** and write down the name of the port (ex. **COM3**) which is needed for **avrdude**.

Finally, press **Files->Upload** to load **ArduinoISP** to **Arduino Uno**. This operation transforms **Arduino Uno** into an **ISP programmer**.

(2) Connect the ISP pins of Arduino Uno to the Target AVR as indicated in Fig.1

It is extremely important to make sure that the ISP pins of **Arduino Uno** are connected to the **Target AVR** like this:

Arduino Uno	**Target AVR** (ex. Atmega328p)
SS	**RESET**
MOSI	**MOSI**
MISO	**MISO**
SCK	**SCK**

Frequent wiring mistakes are Arduino Uno(SS) -> Target AVR(SS) and Arduino Uno(RESET) -> Target AVR(RESET).

As a note: The crystal and its two capacitors have to be as close to XTAL1, XTAL2 pins of the **Target AVR** as possible (like in Fig. 1). If you position them in a different configuration, far from the processor, running wires between capacitors, crystal and the AVR, the newly formed oscillator can refuse to work and **avrdude** will throw the error "Invalid device signature" when you try to program the microcontroller.

(3) Write your source code, compile it and generate the hex file

In **Arduino 1.0.1->Files->Preferences** check **Show verbose output during compilation**. This will display a lot of information, when the code is compiled, including the location of the hex file.

Go to **File->Examples->01.Basics** and load **Blink** example. This will be considered as your source code.

Go to **Sketch** menu and press **Verify/Compile**.

When the compilation completes search in the lower side of the screen the path of the generated Blink hex file. It looks like this one:

```
C:\Users\Victor\AppData\Local\Temp\build1325897986179133197.tmp\Blink.cpp.hex
```

Select it with the mouse and press Ctrl-C followed by a Paste in Notepad, for instance. You will need the path for **avrdude**.

REMARK: The hex file can be generated with any IDE you like, not necessarily using **Arduino 1.0.1**. You can write your code in: **BASCOM**, **CodeVision**, **AVR Studio**, **IAR** which, in most cases, are far better choices than **Arduino 1.0.1**.

ATTENTION! If you use **Arduino 1.0.1** to generate a hex file, you have to select the right board/processor from **Tools->Board**. For example, you simply can not compile the **Blink** Led code for **Arduino Uno**

(**Atmega328p**) and expect it to work on a **target Atmega32** microcontroller. Also, even if, your target chip on the breadboard is an **Atmega328p DIL28**, you have to take into account that **pin 13** of **Arduino Uno** (the one that drives the led) corresponds to **pin 19 (SCK)** on your target **Atmega328p** microcontroller.
AS A CONCLUSION: Arduino 1.0.1 works well for compiling code written for the boards in its list.
For anything else, different boards or stand alone AVRs, it is highly indicated to use a general purpose IDE, not Arduino 1.0.1.

(4) Use avrdude command lines to upload the hex file into the Target AVR

A**vrdude** comes bundled with **WinAVR**, a software that is free and can be found on the internet.
Locate **avrdude** inside **WinAVR**, start the Windows standard application **Command Prompt** and launch **avrdude** from within **Command Prompt**.
A better choice would be to use a software, similar to **Command Prompt** but much easy to use, called **PowerCmd** because it allows you to save and copy-paste the long **avrdude** command lines, with many options and complicated paths to hex files, necessary to program an AVR (see the example).

At this moment, it is supposed that **Arduino Uno** is already loaded with **ArduinoISP**, connected to the **PC** with an **USB** cable and all wires, between the **Target AVR** and **Arduino Uno** acting as an **ISP** programmer, are in their place as in Fig. 1.

For the example that follows, the **Target AVR** is chosen to be an **Atmega328p** already set to run at **16 MHz**.

EXAMPLE:

Launch **PowerCmd** (or Command Prompt**)** and copy this line:

```
c:\>avrdude -P COM3 -b 19200 -c avrisp -p m328p -n
```

in one of its command windows.

If your AVR is different then change the option **-p m328p** to **-p …** corresponding to your microcontroller. Most likely you do not have to modify the other options. (Note: The command ***c:\>avrdude -?*** will display a list with explanations regarding the meaning of the switches: ***–P, -b, -c, -p, -n*** and many others.)

Press **Enter**.

Avrdude should display the message:

```
c:\>avrdude -P COM3 -b 19200 -c avrisp -p m328p -n
avrdude: AVR device initialized and ready to accept instructions
Reading | ################################################## | 100% 0.14s
avrdude: Device signature = 0x1e950f
avrdude: safemode: Fuses OK
avrdude done.  Thank you.
```

telling you that the **Target AVR** is accessible and can be programmed. (**-n** option instructs **avrdude** to make no modification inside the **Target AVR**).

You can program now the **Target AVR** with the **Blink.cpp.hex** file, generated at point (3), by copying the **avrdude** command, highlighted in red (see below), in **PowerCmd** and pressing **Enter**.
Do not forget to change the path to the hex file (the string of characters between “”) according to the settings in your computer. The rest of the command line, including the final “**:i**” will likely remain unchanged.

```
c:\>avrdude -P COM3 -b 19200 -c avrisp -p m328p -u -U
flash:w:"C:\Users\Victor\AppData\Local\Temp\build1325897986179133197.tmp\Blink.
cpp.hex":i

avrdude: AVR device initialized and ready to accept instructions

Reading | ################################################## | 100% 0.16s

avrdude: Device signature = 0x1e950f
avrdude: NOTE: FLASH memory has been specified, an erase cycle will be performed
         To disable this feature, specify the -D option.
avrdude: erasing chip
avrdude: reading input file
"C:\Users\Victor\AppData\Local\Temp\build1325897986179133197.tmp\Blink.cpp.hex"
avrdude: writing flash (1084 bytes):

Writing | ################################################## | 100% 1.76s

avrdude: 1084 bytes of flash written
avrdude: verifying flash memory against
C:\Users\Victor\AppData\Local\Temp\build1325897986179133197.tmp\Blink.cpp.hex:
avrdude: load data flash data from input file
C:\Users\Victor\AppData\Local\Temp\build1325897986179133197.tmp\Blink.cpp.hex:
avrdude: input file
C:\Users\Victor\AppData\Local\Temp\build1325897986179133197.tmp\Blink.cpp.hex contains
1084 bytes
avrdude: reading on-chip flash data:

Reading | ################################################## | 100% 1.31s

avrdude: verifying ...
avrdude: 1084 bytes of flash verified

avrdude done.  Thank you.
```

Success! At this moment your **Target AVR** is programmed and ready to execute the code.

Most common errors you may get from avrdude:

1)
```
c:\>avrdude -P COM3 -b 19200 -c avrisp -p m328p -n
avrdude: ser_open(): can't open device "\\.\COM3": The system cannot find the file
specified.
```

Reason: The **USB** connection between the **PC** and **Arduino Uno** is not good. If the connection is good but the error still keeps coming press the **Reset** button of **Arduino Uno**.

2)
```
c:\>avrdude -P COM3 -b 19200 -c avrisp -p m328p -n
avrdude: ser_open(): can't open device "\\.\COM3": Access is denied.
```

Reason: Another application that uses **COM3** port, likely a **Serial Monitor**, is running. Close it.

3)
```
c:\>c:\>avrdude -P COM3 -b 19200 -c avrisp -p m328p -n
Access is denied.
```

Reason: **c** drive appears twice: "*c:*\>*c:*\>".

```
4) c:\>avrdude -P COM3 -b 19200 -c avrisp -p m328p -n
avrdude: stk500_program_enable(): protocol error, expect=0x14, resp=0x50
avrdude: initialization failed, rc=-1
         Double check connections and try again, or use -F to override
         this check.
avrdude: stk500_disable(): protocol error, expect=0x14, resp=0x51
avrdude done.  Thank you.
```

Reason: The **Target AVR** is not powered or does not exist.
This message is also useful for testing, before connecting any **Target AVR**, that **Arduino Uno** + **ArduinoISP** work as an **ISP programmer** and **the connection to the PC is proper**.

```
5) c:\>avrdude -P COM3 -b 19200 -c avrisp -p m328p -n
avrdude: stk500_getsync(): not in sync: resp=0xf0
avrdude done.  Thank you.
```

Reason: **Arduino Uno** needs a **Reset**.

```
6) c:\>avrdude -P COM3 -b 19200 -c avrisp -p m328p -n
avrdude: AVR device initialized and ready to accept instructions
Reading | ################################################## | 100% 0.14s
avrdude: Device signature = 0xffffff (Device signature = 0x000000)
avrdude: Yikes!  Invalid device signature.
         Double check connections and try again, or use -F to override
         this check.
```

Reasons:

a) ISP connections **(SS, MOSI, MISO, SCK)** -> **(RESET, MOSI, MISO, SCK),** between **Arduino Uno** and the **Target AVR**, are not good, wrong wiring.
b) The AVR oscillator (external) does not work.

List of commonly used AVRs and how they should be connected to an Arduino Uno, for programming

If you do not use an Atmega328p target then you should adapt the wiring in Fig. 1 according to your specific target AVR (ex. Attiny13, 84, 85, Atmega8, etc.).

Attiny13

Uno -> pin Attiny13

SS -> 1
MOSI -> 5
MISO -> 6
SCK -> 7