Fonctionnement des Ordinateurs

TP2 - Développement embarqué sur plateforme MIPS (2ème partie)

B. Quoitin
Faculté des Sciences
Université de Mons

Résumé

Cette section a deux objectifs principaux. Premièrement, continuer à manipuler les ports GPIO et en renforcer la compréhension. En particulier, un nouveau mécanisme simplifiant la commande des GPIOs sera introduit : les *pseudo-registres*. Ensuite, cette séance se penchera sur la lecture des ports GPIO avec une application à la lecture de boutons et d'interrupteurs. Cette première partie sera réalisée en langage d'assemblage.

Deuxièmement, la programmation en langage d'assemblage étant relativement complexe et fastidieuse, la seconde partie de la séance sera consacrée à une initiation à la programmation de la plateforme ChipKit avec le langage C. Des éléments du langage C ainsi que les outils de compilations seront présentés. Le langage C permettra de progresser plus rapidement par la suite et de s'attaquer à des problèmes plus conséquents.

Table des matières

1 General Purpose Input Output (GPIO)

1.1 Pseudo-registres

La séance précédente traitait de la commande de LEDs au travers des ports GPIO. Changer l'état d'une broche du microcontrôleur nécessitait la modification d'un nombre limité de bits d'un registre SFR (p.ex. `LATG`). Un exemple d'une telle opération est rappelé à la Figure 1. Cette opération requiert 5 instructions dont 2 accès au bus mémoire (instructions `lw` et `sw`).

```
1 LATG=0xBF8861A0
2
3         li      $a2, LATG
4         lw      $a0, 0($a2)
5         li      $a1, 0x00000040
6         or      $a0, $a0, $a1
7         sw      $a0, 0($a2)
```

FIGURE 1 – Niveau logique haut (1) écrit dans `RG6`.

Ce type d'opérations est très fréquent. Afin de diminuer le nombre d'instructions et d'accès mémoires nécessaires à celles-ci, les ingénieurs de Microchip ont mis en place un mécanisme spécifique : les *pseudo-registres*. Ceux-ci permettent de changer ou inverser l'état d'un sous-ensemble de bits d'un registre SFR en un seul accès au bus (une seule instruction `sw`). A cette fin, à chaque registre SFR x d'adresse A sont associés 3 pseudo-registres x`CLR`, x`SET` et x`INV` situés aux adresses $A+4$, $A+8$ et $A+12$. Ces registres permettent respectivement de mettre à 0, de mettre à 1 ou d'inverser la valeur de bits dans le registre x.

La valeur y écrite dans un pseudo-registre associé au SFR x détermine quels bits doivent être modifiés. Les bits à 1 dans y sont modifiés, les autres restent inchangés. Le type de modification dépend du suffixe su pseudo-registre. Avec `CLR`, les bits sont mis à 0 ; avec `SET`, les bits sont mis à 1 ; et avec `INV`, les bits sont inversés.

Exemple. Au registre `TRISG` situé à l'adresse `0xBF886180` sont associés les pseudo-registres `TRISGCLR` (`0xBF886184`), `TRISGSET` (`0xBF886188`) et `TRISGINV` (`0xBF88618C`).

Supposons que l'on souhaite mettre les bits 2, 6 et 9 de `TRISG` à 1 et laisser les autres inchangés. Il suffit d'écrire dans le pseudo-registre `TRISGSET` une valeur dans laquelle les bits 2, 6 et 9 sont à 1 et tous les autres valent 0, c-à-d. la valeur `0x00000244`. La Figure 2 donne un extrait de programme en langage d'assemblage qui écrit cette valeur dans le pseudo-registre `TRISGSET`. Seules 3 instructions dont 1 seul accès mémoire sont nécessaires.

```
1 TRISG=0xBF886180
2 TRISGSET=TRISG+8
3
4         li      $a0, TRISGSET
5         li      $a1, 0x00000244
6         sw      $a1, 0($a0)
```

FIGURE 2 – Mise à 1 des bits 2, 6 et 9 de `TRISG` avec un pseudo-registre.

1.2 Lire l'état d'un interrupteur ou d'un bouton

Cette section s'intéresse à la lecture de l'état d'un interrupteur ou d'un bouton-poussoir. Nous utilisons à cet effet la carte d'extension *Basic I/O Shield* qui comporte notamment 4 interrupteurs à glissière `SW1` à `SW4` et 4 boutons-poussoirs `BTN1` à `BTN4`.

☞ **ATTENTION ! Cet exercice nécessite de connecter la carte d'extension *Basic I/O Shield* sur la carte *ChipKit Uno32*. Il est IMPERATIF de couper l'alimentation de la carte ChipKit préalablement, en débranchant le câble USB.** Demandez éventuellement à un assistant de vous montrer comment effectuer cette manipulation.

L'interaction avec les boutons et interrupteurs et relativement similaire à la commande de LEDs. Il faudra effectuer les étapes suivantes

Bouton / interrupteur	Broche
BTN1	RF1
BTN2	RD5
BTN3	RD6
BTN4	RD7
SW1	RD8
SW2	RD9
SW3	RD10
SW4	RD11

TABLE 1 – Connexion des boutons et interrupteurs.

— **Connexion entre processeur et bouton.** Il est nécessaire de déterminer à quel port chaque bouton/interrupteur est connecté. La Table 1 donne la broche du microcontrôleur à laquelle chaque bouton (`BTN`x) et interrupteur (`SW`x) est connecté. Cette table est établie sur base du schéma électrique de la carte dont un extrait est donné en annexe à la Figure 11.

— **Configurer en entrée**. Il est nécessaire de configurer les broches liées aux boutons/interrupteurs comme des entrées. Ceci est effectué au travers des registres `TRIS`x. Pour rappel, si le bit n du registre `TRIS`x vaut 1, le port n est configuré comme une entrée (valeur par défaut), sinon il est configuré comme une sortie.

— **Lecture de l'état**. L'état de chaque broche en entrée d'un port x peut être lu via le registre `PORT`x. Chaque bit correspond à une broche. Si le bit n vaut 1, c'est qu'un niveau logique haut est présent sur la broche. Sinon, c'est qu'un niveau logique bas est présent.

1.3 Auto-évaluation

Questions.

— Utilisez les pseudo-registres dans le programme faisant clignoter une LED développé lors de la séance précédente. Utilisez pour cela uniquement les registres `TRIS`x`CLR` et `LAT`x`INV`.

— Ecrivez un programme en langage d'assemblage MIPS qui reporte l'état des interrupteurs `SW1` et `SW2` sur les LEDs `LD4` et `LD5` de la carte ChipKit Uno32 respectivement.

Le programme doit lire en permanence (dans une boucle sans fin) l'état des interrupteurs (lecture de `PORT`x) et écrire ce dernier sur la broche de la LED correspondante (écriture de `LAT`y).

2 Programmation en C

La programmation en langage d'assemblage s'avère vite difficile lorsque la complexité des programmes croît. A partir de cet exercice, les programmes seront développés en langage C. Il est utile de consulter l'introduction à la programmation en C fournie en annexe de ce document, sur la plateforme Moodle.

2.1 Introduction

Le langage C a une syntaxe très proche du langage Java avec lequel vous êtes déjà familiarisé. En guise d'introduction, prenons l'exemple du programme en C illustré à la Figure 5. Celui-ci fait clignoter la LED `LD4` (broche `RG6`), de la même façon que ce qui a été réalisé auparavant en langage d'assemblage MIPS.

```
1  #include <p32xxxx.h>
2
3  void delay()
4  {
5    unsigned int i= 10000000;
6    while (i-- > 0);
7  }
8
9  int main()
10 {
11   TRISGCLR= 0x00000040;
12   LATGCLR= 0x00000040;
13   while (1) {
14     delay();
15     LATGINV= 0x00000040;
16   }
17 }
```

FIGURE 5 – Programme en C qui fait clignoter `LD4`.

Le point d'entrée du programme est la fonction `main`. Cette fonction effectue les actions suivantes :

1. Elle configure la broche `RG6` en sortie (via le registre `TRISGCLR`).
2. Elle place ensuite cette broche dans un état logique bas (via le registre `LATGCLR`).
3. Finalement, elle appelle la fonction `delay` puis change l'état de `RG6` (via le registre `LATGINV`). Ces deux opérations sont effectuées dans une boucle `while` infinie. La condition de continuation de la boucle est toujours vraie [1].

Une première observation importante est que les registres `TRISGCLR`, `LATGCLR` et `LATGINV` sont manipulés comme des variables. Ces « variables » sont déclarées dans le fichier en-tête `p32xxxx.h` qui est inclus avec la directive `#include` au début du programme.

La fonction `delay` fonctionne de la même manière que celle réalisée en langage d'assemblage. Le délai d'attente est obtenu en effectuant 10 millions d'itérations de la boucle vide `while`. Le nombre d'itérations est spécifié par la valeur initiale de la variable `i`. Cette dernière est décrémentée à chaque itération (`i--`). La boucle continue tant que `i` est non nulle (`i > 0`).

2.2 Compilation

Ce programme peut être compilé à l'aide des commandes montrées à la Figure 3. La première ligne effectue la compilation d'un programme en langage C vers un code objet. L'option `-mprocessor` indique au compilateur la version du microcontrôleur qui est utilisée (ici PIC32MX320F128H). L'option `-c` indique que `gcc` ne doit faire que la compilation (sans cette option, l'édition des liens est également effectuée). L'option `-O0` (lettre `O` suivie du chiffre `0`) désactive les optimisations. La seconde ligne effectue l'édition des liens, de la même façon qu'avec les programmes écrits en langage d'assemblage.

Le fichier objet obtenu (`main.bin`) doit ensuite être converti au format Intel Hex avec `xc32-objcopy`) et chargé dans la mémoire Flash du microcontrôleur avec `avrdude`. Les commandes à utiliser sont rappelées à la Figure 4. Référez-vous à l'énoncé de la séance précédente pour les détails.

2.3 Auto-évaluation

Questions simples

— Assurez-vous d'avoir bien compris le programme en C et d'être capable de le compiler, de le linker et de le télécharger dans le microcontrôleur. **Vérifiez que le résultat attendu est celui obtenu (clignotement de la LED) ! ! !**
Mesurez la fréquence de clignotement obtenue. Correspond-t-elle à celle du même programme écrit en langage d'assemblage (1,33Hz) ? Si non, pourquoi cette différence ? Regardez au code objet généré avec `objdump` et comparez le au

1. Le langage C n'a pas de type booléen : la valeur 0 est fausse et une valeur différente de 0 est vraie

```
bqu@:~$ xc32-gcc -mprocessor=32MX320F128H -Wall -Werror -O0 -c -omain.o main.c
bqu@:~$ xc32-gcc -T ldscript.ld -o main.bin main.o
bqu@:~$
```

FIGURE 3 – Compilation d'un programme écrit en C pour la plateforme *ChipKit Uno32*.

```
bqu@:~$ xc32-objcopy -O ihex main.bin main.hex
bqu@:~$ avrdude -C avrdude.conf -p pic32-360 -c stk500 -P /dev/ttyUSB0-U
flash:w:main.hex
bqu@:~$
```

FIGURE 4 – Chargement du programme compilé dans la mémoire flash du microcontrôleur.

programme que vous aviez écrit en langage d'assemblage lors de la séance précédente. Quelles sont les différences que vous remarquez ?

— Ecrivez un programme qui affiche un entier sur 8 bits en binaire à l'aide des LEDs `LD1` à `LD8` de la carte Basic I/O Shield. La Table 2 donne les broches du microcontrôleur connectées aux LEDs sur cette carte.
Remarquez que pour réaliser ce programme, **il n'est pas nécessaire** d'écrire un algorithme qui convertit un entier en binaire ! ! !

— Ecrivez un programme « chenillard » qui fait défiler les LEDs de la carte *Basic I/O Shield* d'un côté à l'autre puis dans l'autre sens, et ceci en boucle.

— Ecrivez un programme qui compte le nombre de pressions sur le bouton `BTN1` et qui affiche ce nombre en binaire sur les LEDs de la carte *Basic I/O Shield*.

LED	Broche
LD1	RE0
LD2	RE1
LD3	RE2
LD4	RE3
LD5	RE4
LD6	RE5
LD7	RE6
LD8	RE7

TABLE 2 – Broches connectées aux LEDs de la carte Basic I/O Shield.

Questions avancées.

— La manipulation des registres en C s'effectue comme si ces registres étaient des variables. Comment ce « miracle » est-il possible ? Quel est le code objet généré pour l'accès à la « variable » `TRISGCLR` ? Allez voir comment ces « variables » sont déclarées dans le fichier `p32xxxx.h`.

— Que se passe-t-il si vous compilez le programme en activant les optimisations en utilisant l'option `-O1` ? Pourquoi le programme ne fonctionne-t-il plus ?
Utilisez `xc32-objdump` pour découvrir le code qui a été généré.

A Annexes

Cette annexe donne des détails supplémentaires sur les TPs. **La lecture de cette annexe est optionnelle.**

A.1 Automatiser la compilation avec un script

Afin d'éviter de ré-introduire les commandes permettant l'assemblage, l'édition des liens, la conversion au format Intel Hex et le téléchargement vers la flash du microcontrôleur, écrivez un script tel qu'illustré à la Figure 7. Placez ce script dans un fichier `compile.sh` et rendez le exécutable avec la commance `chmod +x compile.sh`. Vous pouvez ensuite lancer la compilation et le téléchargement simplement avec `./compile.sh`.

```
1  #!/bin/sh
2
3  NAME="main1a"
4  PORT=/dev/ttyUSB0
5
6  rm -f $NAME.o $NAME.bin $NAME.hex
7
8  xc32-as -O0 -o $NAME.o $NAME.s
9  xc32-gcc -T ldscript.ld -o $NAME.bin \
       $NAME.o
10 xc32-objcopy -O ihex $NAME.bin $NAME.hex
11 avrdude -C avrdude.conf -p pic32-360 -c \
       stk500 -P $PORT -U flash:w:$NAME.hex
```

FIGURE 7 – Script effectuant la compilation d'un programme.

A.2 Automatiser la compilation avec `make`

Un autre moyen d'automatiser la compilation est d'utiliser l'utilitaire `make`. Les règles de la compilation sont alors décrites dans un fichier appelé `Makefile`.

Un fichier `Makefile` fournit à l'utilitaire `make` deux types d'informations :

1. la façon de créer chaque cible : p.ex. comment créer un fichier objet à partir d'un fichier source en langage C
2. les dépendances entre cibles : p.ex. le fichier final `.hex` dépend d'un fichier object qui lui même dépend d'un fichier source en langage C.

Sur base des dépendances, `make` ne recrée que ce qui est nécessaire. Ainsi, si un seul fichier source est modifié dans un grand projet, tout le projet ne doit pas être recompilé, mais seul le fichier modifié et ses dépendances seront recompilés. `make` se base sur la date de dernière modification des fichiers pour déterminer ce qui doit être reconstruit.

Une règle dans un `Makefile` prend la forme suivante. La première ligne de la règle indique le nom de la cible (ce qui doit être construit) suivi du caractère : suivi des éventuelles dépendances, i.e. les noms d'autres fichiers ou cibles qui doivent exister ou avoir été construits avant l'exécution de cette règle. Les lignes suivantes de la règle sont toutes précédées d'une tabulation et contiennent les commandes *shell* nécessaires à la construction de la cible.

Exemple simplifié. Un exemple d'un fichier `Makefile` est donné à la Figure 8.

```
1  main.bin: main.o
2      xc32-gcc -T ../ldscript.ld \
           -mprocessor=32MX320F128H -o \
           main.bin main.o
3
4  main.o: main.c
5      xc32-gcc -c -Wall -Werror -O0 -o \
           main.o main.c
6
7  main.hex: main.bin
8      xc32-objcopy -O ihex main.bin main.hex
9
10 upload: main.hex
11     avrdude -C avrdude.conf -p pic32-360 \
           -c stk500 -P /dev/ttyUSB0 -U \
           flash:w:main.hex
```

FIGURE 8 – Exemple de `Makefile` simple pour compiler un programme C pour PIC32.

Ce fichier contient 4 règles. La première règle se situe aux lignes 1 et 2. La ligne 1 définit la cible, le fichier `main1.bin`, et une unique dépendance, le fichier `main.o`. Cette dépendance indique que le fichier `main.o` doit être créé avant que cette règle ne puisse être exécutée. La ligne 2 indique comment la cible `main.bin` est construite : ici, en utilisant `xc32-gcc` comme *linker*.

Les lignes 4 et 5 définissent une autre règle visant à produire la cible `main.o` à partir de la dépendance `main.c`. La cible est construite en utilisant `xc32-gcc` pour compiler `main.c` en fichier objet `main.o`.

Pour exécuter ce `Makefile`, il suffit d'invoquer l'utilitaire `make`. Ce dernier recherche automatiquement un fichier nommé `Makefile` dans le répertoire courant. Lorsque `make` est invoqué sans argument (voir Figure 6), il construit la première cible rencontrée (précédée par toutes ses dépendances). Dans l'exemple, la première règle du `Makefile` est celle qui construit la cible `main.bin`.

Il est possible d'invoquer d'autres règles en indiquant le nom de la cible comme argument de `make`. L'exemple suivant indique comment « construire » la règle `upload` (voir Figure 9). Lorsque cette règle est invoquée, le programme est chargé dans le micro-

```
bqu$ make
xc32-gcc -c -Wall -Werror -O0 -o main.o main.c
xc32-gcc -T ../ldscript.ld -mprocessor=32MX320F128H -o main.bin main.o
bqu$
```

FIGURE 6 – Invocation d'un `Makefile` avec la cible par défaut.

contrôleur. Notez que `make` effectuera automatiquement la compilation ou la re-compilation du programme pour satisfaire les dépendances de la règle `upload` (création de `main.hex`).

Exemple plus complet. Un exemple de `Makefile` plus complet est présenté à la Figure 10. Celui-ci permet d'effectuer la compilation du même projet que celui donné à la Figure 8. Les différences par rapport à l'exemple précédent sont entre autres l'utilisation de nombreuses variables, de règles de création génériques et d'une règle de nettoyage (`clean`).

— **Variables.** Des variables peuvent être définies et leur valeur utilisée à différents endroits dans le `Makefile`. Par exemple la variable `TARGET` donne le nom du fichier cible (`main`) sans extension. La variable `SRCS` contient la liste des fichiers source à compiler. Il est ainsi possible d'ajouter un fichier source supplémentaire sans devoir changer le reste du `Makefile`.

— **Règle générique.** Une règle générique est utilisée pour générer un fichier objet quelconque (extension `.o`) à partir d'un fichier source C de même nom (extension `.c`). Cette règle apparaît aux lignes 24 et 25. L'en-tête de la règle contient juste `.c.o:` et le corps de la règle indique comment créer un fichier objet en compilant un fichier C.

— **Règle de nettoyage**. La règle `clean` a pour objectif de supprimer tous les fichiers produits lors de la compilation : fichiers objects (`.o`), binaire (`.bin`) et Intel Hex (`.hex`).

A.3 Connexion des boutons-poussoir et interrupteurs

La Figure 11 présente le circuit électrique qui relie les boutons-poussoir et les interrupteurs de la carte Basic I/O Shield au microcontrôleur PIC32.

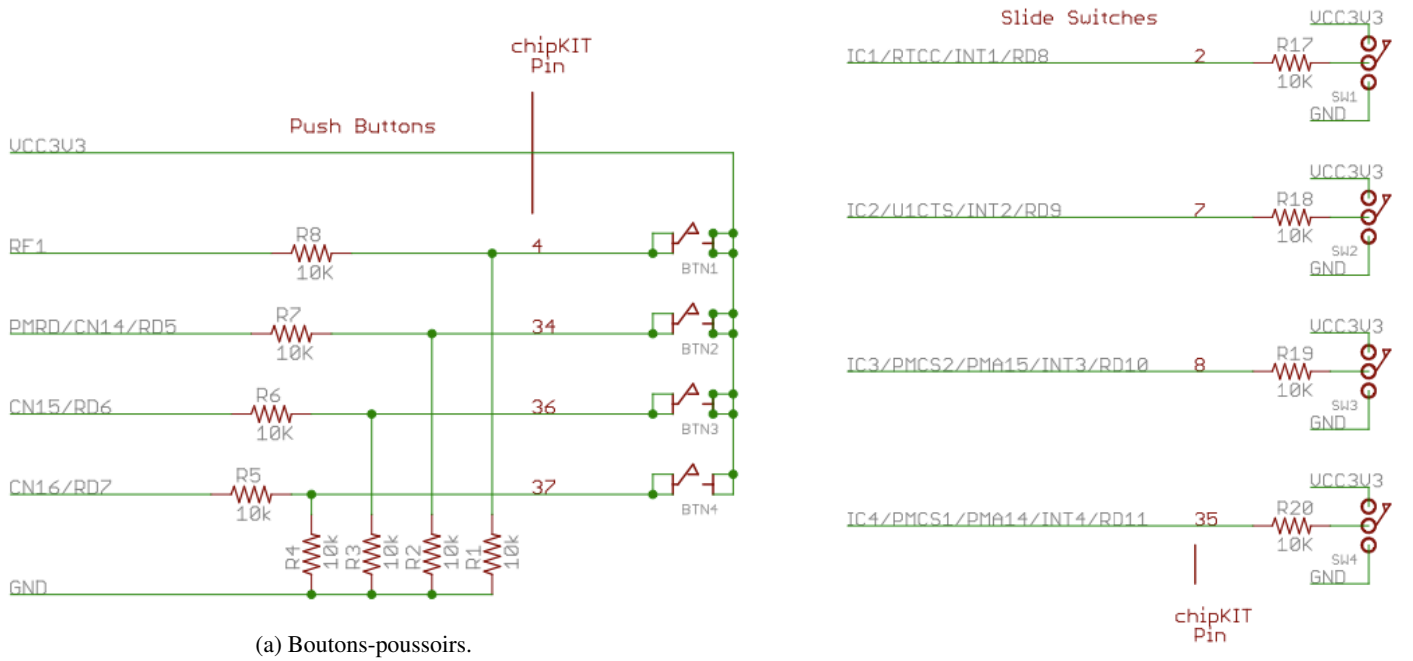
Lorsqu'un bouton-poussoir est enfoncé, la tension électrique

```
bqu@:~$ make upload
xc32-objcopy -O ihex main.bin main.hex
avrdude -C avrdude.conf -p pic32-360 -c stk500 -P /dev/ttyUSB0 -U
flash:w:main.hex
(...)
bqu@:~$
```

FIGURE 9 – Invocation d'un `Makefile` avec la cible `upload`.

```
1  CC=xc32-gcc
2  CFLAGS=-Wall -Werror -O1
3  LDFLAGS=-mprocessor=32MX320F128H
4
5  TARGET=main
6  SRCS=$(TARGET).c
7  OBJS=$(SRCS:.c=.o)
8
9  AVRDUDE=avrdude
10 AVRDUDE_PORT=/dev/ttyUSB0
11 AVRDUDE_CONF=avrdude.conf
12 AVRDUDE_FLAGS=-C $(AVRDUDE_CONF) -p pic32-360 -c stk500 -P $(AVRDUDE_PORT)
13
14 $(TARGET).bin: $(OBJS)
15     $(CC) -T ../ldscript.ld $(LDFLAGS) -o $(TARGET).bin $(OBJS)
16
17 $(TARGET).hex: $(TARGET).bin
18     xc32-objcopy -O ihex $(TARGET).bin $(TARGET).hex
19
20 upload: $(TARGET).hex
21     $(AVRDUDE) $(AVRDUDE_FLAGS) -U flash:w:$(TARGET).hex
22
23 SUFFIXES: .c .o
24
25 .c.o:
26     $(CC) -c $(CFLAGS) -o $@ $<
27
28 PHONY: clean
29
30 clean:
31     @rm -f $(OBJS) $(TARGET).bin $(TARGET).hex
```

FIGURE 10 – Exemple de `Makefile` générique pour compiler un programme C pour PIC32.



(a) Boutons-poussoirs.

(b) Interrupteurs à glissière.

FIGURE 11 – Extrait du schéma électrique de la carte *Basic I/O Shield* (boutons et interrupteurs).

présentée sur la broche correspondante du microcontrôleur est proche de la tension d'alimentation (3,3V), ce qui correspond à un état logique haut [2]. Lorsqu'un bouton-poussoir est relâché, cette tension est proche de 0V, ce qui correspond à un état logique bas [3].

A.4 Connexion des LEDs

La Figure 12 présente le circuit électrique qui relie les LEDs de la carte Basic I/O Shield au microcontrôleur PIC32. ATTENTION, les LEDs sont connectées aux broches `RE0` à `RE7`, ce qui n'apparaît pas clairement sur la Figure.

Remerciements

Merci à A. BUYS et D. HAUWEELE pour leur relecture attentive d'une version antérieure de ce document.

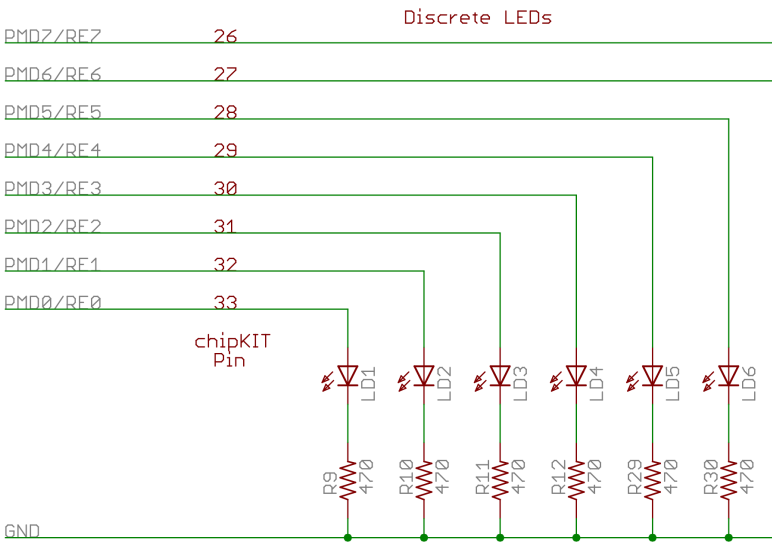


FIGURE 12 – Extrait du schéma électrique de la carte *Basic I/O Shield* (LEDs).

2. Un état logique haut (1) correspond à une tension électrique $\geq$ V_{IH}=2,64V.
3. Un état logique bas (0) correspond à une tension électrique $\leq$ V_{IL}=0,66V.