



École Polytechnique de l'Université de Tours
 64, Avenue Jean Portalis
 37200 TOURS, FRANCE
 Tél. +33 (0)2 47 36 14 14
www.polytech.univ-tours.fr

Spécialité Informatique Industrielle

Version : 01 15/01/2020

CAHIER D'ANALYSE			
Projet :	Web Based Time Converter (WBTC)		
Emetteur :	J. LOCHE	Coordonnées : jeremy.loche@etu.univ-tours.fr	
Date d'émission :	13/01/20		
Validation			
Nom	Date	Valide (O/N)	Commentaires
Historique des modifications			
Version	Date	Description de la modification	
00	10/11/19	Rédaction initiale du CA	
01	15/01/2020	Ajout de l'analyse du bouton d'arrêt	

Table of Contents

1	Introduction	3
2	IHM Date et Heure « Time Circuit Display »	3
2.1	Apparence recherchée du « Time Circuit Display »	3
2.2	Affichage multiplexé	4
2.3	Solutions de pilotage multiplexé direct	6
2.4	Pilotage par drivers de LED sur registres à décalages :	9
2.5	Pilotage par composants drivers spécialisés	12
2.5.1	Drivers de LED : MAX6954 ou MAX6955	13
2.5.2	Driver de LED : HT16K33 (Retenue)	14
2.5.3	Librairie logicielle de pilotage TCD : HT16K33	17
3	Convecteur temporel « Flux Capacitor »	19
3.1	Pilotage des WS2812b	21
3.2	Driver de LEDs WS2812b HDL	21
3.3	Spécification de la fonction “Driver_WS2812b” :	22
3.3.1	Utilisation du driver HDL WS2812b version GPIO	26
3.3.2	Couche d’adaptation Avalon MM du driver WS2812b	27
4	Architecture matérielle globale	33
5	Gestionnaire système	34
5.1	Module de gestion du « Time Circuit Display »	35
5.1.1	Architecture générale du module	36
5.1.2	Processus Driver (HAL)	36
5.1.3	Serveur « timecircuitd »	36
5.1.4	Client CLI « timecircuitctl »	36
5.2	Module de gestion du « Flux Capacitor »	36
5.2.1	Flux capacitor Middleware (driver HAL)	36
5.2.2	Serveur « fluxcapacitord »	38
5.2.3	Client CLI « fluxcapacitorctl »	38
5.3	Module de gestion du bouton d’extinction	38
6	Serveur WEB de pilotage	40

1 Introduction

Ce cahier retrace les réflexions menées sur le projet WBTC avec notamment l'analyse et la conception du système d'affichage de la date et l'heure « Time Circuit Display » ainsi que le co-design de la partie convecteur temporel à base de LED WS2812b.

2 IHM Date et Heure « Time Circuit Display »

Dans cette partie, nous allons aborder l'analyse du système d'IHM « Time Circuit Display ». Dans le film *Retours vers le futur* il s'agit d'un élément phare du tableau de bord de la *De Lorean, fameuse machine à voyager dans le temps*. Nous allons donc détailler l'apparence attendue sur cet élément d'IHM. On abordera ainsi que les éléments fonctionnels nécessaires à son fonctionnement ainsi que les possibilités s'offrant à nous pour la réalisation.

2.1 Apparence recherchée du « Time Circuit Display »

Vous trouverez ci-dessous une image du « Time Circuit Display » extraite du film.



Figure : Apparence du « Time Circuit Display »

Source (valide au 21/11/2019):

<http://avidly.lareviewofbooks.org/2015/06/30/time-circuits-on-flux-capacitor-fluxing/>

Fonctionnellement, on remarque que cet affichage contient 7 fonctions distinctes d'affichage d'informations :

Fonction	Caractéristiques de l'affichage	Nombre de LED
Affichage Mois	3 x 14 ou 16 segments	42 ou 48
Affichage Jour	2 x 7 segments	14
Affichage Année	4 x 7 segments	28
Indication AM/PM	2 x LED	2
Affichage Heure	2 x 7 segments	14
Indication secondes	2 x LED	2
Affichage Minutes	2 x 7 segments	14
Toutes confondues	3 x 14 ou 16 segments + 10 x 7 segments + 4 x LED	116 ou 122

Cette première observation permet de dire que l'affichage du mois a été réalisé par des afficheurs alphanumériques de type 14 ou 16 segments. Pour le reste n'étant que des chiffres, l'affichage est

réalisé à l'aide d'afficheurs numérique 7 segments. On a en plus des LED individuelles utiles pour indiquer s'il s'agit d'heure du matin ou de l'après-midi et l'indication des secondes par les LED séparatrices de l'heure et des minutes.

Ceci amène donc le total à 116 (pour le choix d'afficheurs 14 segments) ou 122 LED (pour le choix d'afficheurs 16 segments) à piloter individuellement.

On souhaite afficher des caractères de l'alphabet (A à Z sans accents) sur la partie mois et des chiffres de 0 à 9 pour le jour, année, heure et minutes.

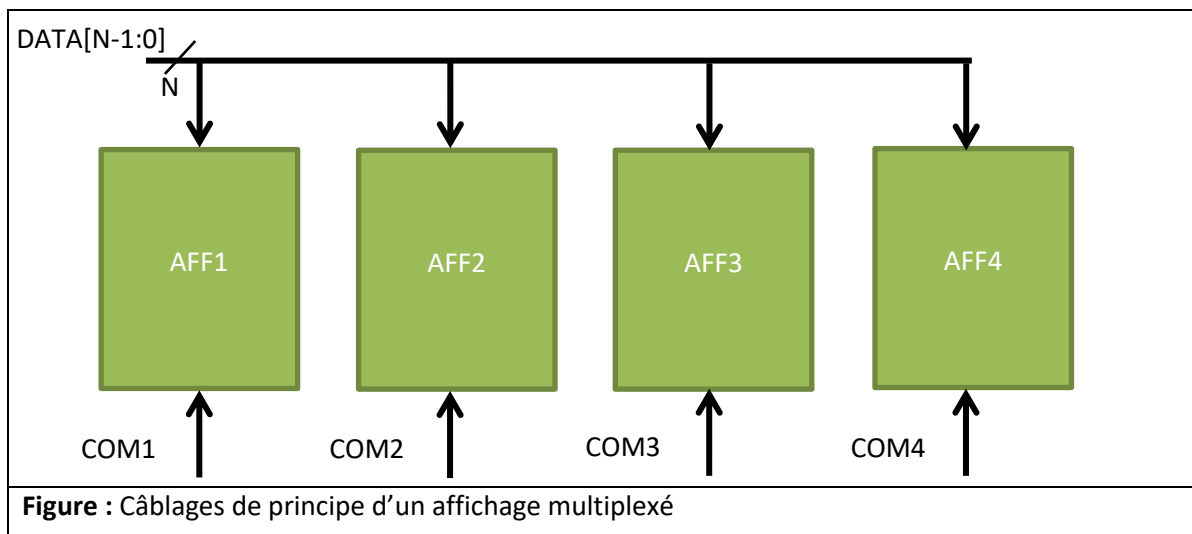
Il y a plusieurs stratégies de pilotages pour de tels types d'afficheurs. En effet, avoir 122 LED à piloter n'implique pas forcément de 122 lignes de commandes depuis un contrôleur.

Une technique couramment utilisée est un **multiplexage** à haute fréquence des afficheurs pour limiter le nombre de lignes de commande du contrôleur.

Cette technique ne fonctionne que parce qu'il existe l'effet de la **persistance rétinienne**. La persistance rétinienne permet d'allumer et d'éteindre rapidement des LED sans qu'un clignotement ne soit visible par l'œil humain. En clignotant assez vite (plusieurs centaines de Hz), on peut tromper l'œil et lui faire croire qu'en **moyenne** on est constamment allumé.

2.2 Affichage multiplexé

L'approche multiplexée correspond à mettre en commun la valeur des données à afficher sur les afficheurs à travers un bus parallèle.



Dans un affichage multiplexé, on dispose de **M afficheurs** à **N segments**. Le bus de donnée **DATA** de taille **N** est commun à tous les afficheurs. On utilise en suite **M** lignes de sélection **COM[i]** qui permettent de sélectionner l'afficheur à allumer. On a donc **N + M** lignes de pilotage pour **N x M** éléments à piloter. C'est bien plus efficace que de piloter les lignes une par une.

Le bus de données **DATA** est une représentation du symbole à afficher sur les afficheurs, chaque ligne de signal représente donc 1 segment sur l'afficheur.

L'idée du pilotage d'un afficheur est la suivante :

<p>Si COM[i]=1 Alors L'afficheur i allume les segments représentés par DATA</p> <p>Sinon Tous les segments de l'afficheur i sont éteints</p>
--

Algorithme : Logique d'affichage

On voit donc que si on alimente le signal COM de 2 segments, ils allumeront tous les deux les mêmes segments.

Pour pouvoir afficher des motifs différents sur chaque afficheurs, on doit allumer les COM un par un en changeant la valeur de DATA entre chaque changement.

Ceci donne l'algorithme suivant ou l'on dispose des valeurs de motifs à appliquer de **DATA** dans le tableau **PATTERN[M]** pour l'afficheur **i** et on alimente uniquement l'afficheur **i**. On laisse l'afficheur allumé pendant une durée **TON** puis on passe à l'afficheur suivant

On va balayer ainsi chaque afficheur modulo **M**. Si cela est fait suffisamment rapidement, on aura une impression que tous les afficheurs sont allumés en même temps.

<p>REPETER POUR i allant de 1 à M FAIRE :</p> <p> //Eteindre tous les afficheurs POUR j allant de 1 à M FAIRE : COM[j]=0 FINPOUR</p> <p> //Placer la donnée de l'afficheur sur DATA DATA=PATTERN[i]</p> <p> //Allumer le segment sélectionné COM[i]=1 ATTENDRE(TON) FINPOUR FINREPETER</p>
--

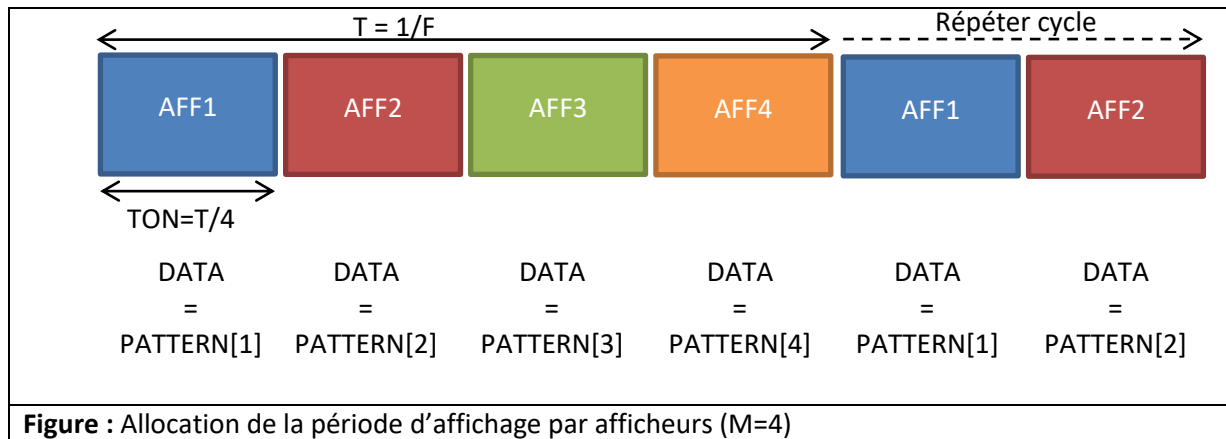
Algorithme : Processus d'affichage multiplexé
--

Avec cette technique, on a uniquement un afficheur allumé en tous temps pour une durée de **TON**. Pour balayer tous les afficheurs, on voit qu'on a une période de rafraichissement **T=M x TON**.

Le rapport cyclique **TON/T** de chaque afficheur est de **1/M**. Le jeu est ensuite de choisir **T=1/F** suffisamment cours pour avoir l'effet de la persistance rétinienne.

En général, on prendra **F>=100Hz** mais pas trop élevé pour que les afficheurs ait le temps de répondre et de s'allumer. La fréquence minimale pour l'œil est de 30 Hz, mais il vaut mieux aller plus vite pour éviter le scintillement à 50Hz et 60Hz (fréquence du réseau électrique en Europe et USA).

Le fait que les afficheurs soient allumés pendant **1/M** du temps implique que la luminosité perçue par l'œil sera de **1/M** de la luminosité totale que peut produire l'afficheur s'il était allumé 100% du temps.



On remarque qu’avec cette technique, plus on va multiplexer d’afficheurs et moins la luminosité perçue par l’œil humain ne sera importante (du fait d’un rapport cyclique faible).

C’est presque systématiquement que le pilotage multiplexé est utilisé pour des matrices de LED ou des afficheurs à segments.

Nous allons voir les différentes façons de réaliser ce multiplexage avec leurs avantages et leurs inconvénients.

2.3 Solutions de pilotage multiplexé direct

Cette solution correspond à la solution proposée par Florian Rémy, précédent étudiant sur ce projet. Son idée était de réaliser le multiplexage directement au niveau du FPGA en tirant notamment 14 lignes de données parallèles **DATA** et 13 lignes de sélection d’afficheur.

Son idée consiste à utiliser les GPIO de la carte DE0-Nano-SOC pour réaliser le multiplexage. On monopoliserait donc 27 sorties pour réaliser le pilotage.

On utilise ensuite directement les sorties PUSH/PULL 3.3V de la carte pour alimenter chaque le bus **DATA** et des transistors de puissance MOSFET canal N pour piloter les broches COM des afficheurs 14 segments à cathode commune.

Le principe de pilotage est le suivant :

1. Placer un niveau logique 1 (i.e. 3.3V) sur les bits du BUS **DATA** pour tous les segments à allumer.
2. Placer un niveau logique 1 (i.e. 3.3V) sur le **COM_i** de l’afficheur à allumer

La structure de base proposée par Florian est dessinée dans la figure suivante. On dispose de N résistances de limitation de courant sur le bus de données. Les MOSFET N agissent comme des interrupteurs commandés permettant d’ouvrir ou fermer le circuit de l’afficheur. Si le MOSFET est piloté (donc passant), l’afficheur allumera les segments pilotés par le bus **DATA**. Si le MOSFET est bloqué, l’afficheur sera éteint.

L’exemple est appliqué à un ensemble de 2 afficheurs 7 segments mais peut être appliqué à des afficheurs 14 ou 16 segments. Il est nécessaire d’ajouter des résistances de limitation de courant sur chaque ligne du bus DATA en amont des afficheurs pour les protéger.

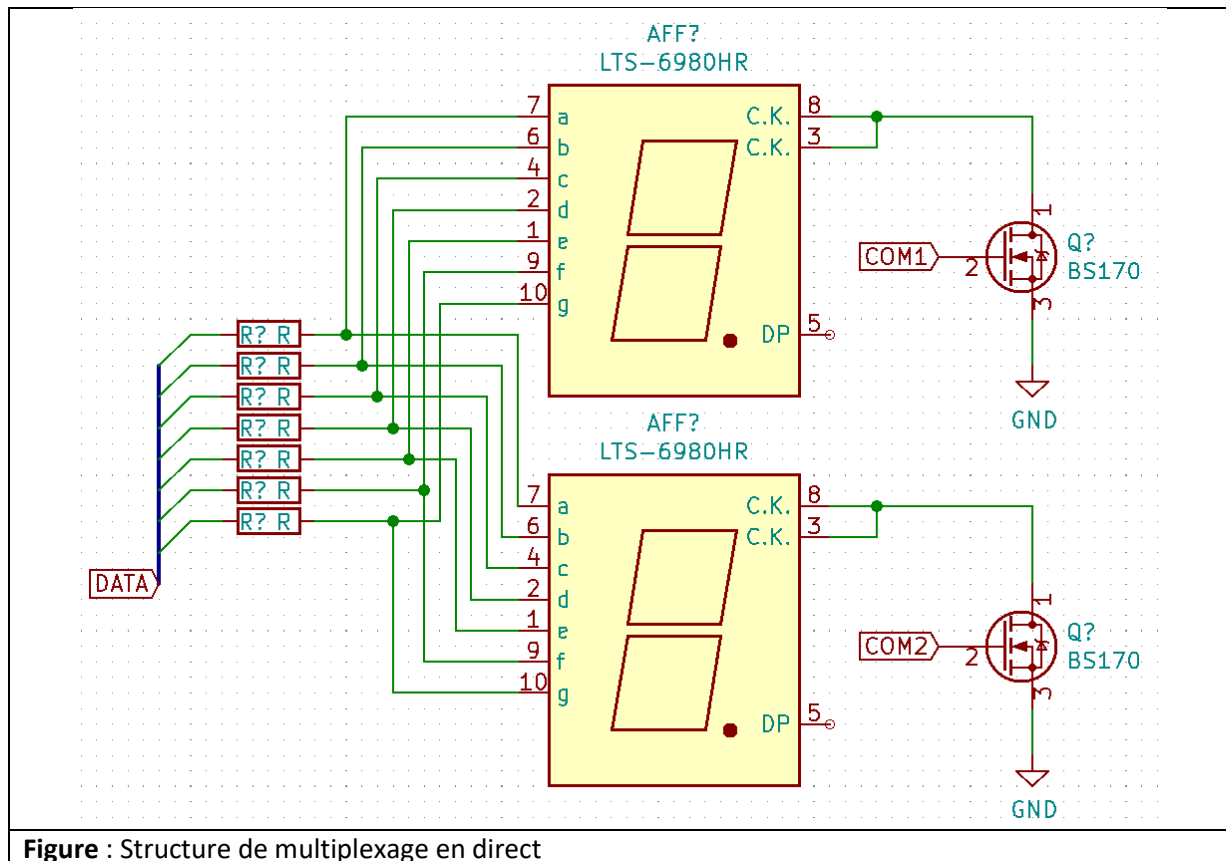


Figure : Structure de multiplexage en direct

La valeur des résistances R de limitation de courant est calculée à partir :

1. $U = 3.3V$ = la tension d'alimentation que peut fournir la carte DE0-Nano sur ces GPIO
2. $I_d = 20mA$ = courant maximum souhaité dans les LED/segments.
3. $V_f = 1.63V$ = tension de seuil des LED

L'équation de R :

$$R = (U - V_f) / I_d = (3.3 - 1.63) / 0.02 = 83.5 \text{ Ohms}$$

Les transistors MOSFET N doivent être choisis avec les caractéristiques suivantes :

1. **$I_{drain} (max)$** : courant maximum dans le DRAIN du transistor doit être supérieur à la somme des courants consommés dans le cas où tous les segments de l'afficheur seraient allumés.
 - a. $I_{drain}(max) \geq N \times I_d$
 - b. Avec I_d , courant d'alimentation des segments défini précédemment
 - c. Avec N , nombre de segments par afficheur
2. **V_{sgth}** : tension de seuil du transistor MOSFET, si $V_{gs} > V_{sgth}$ alors le transistor est **passant**, sinon il est **bloqué**.
 - a. Il est nécessaire de valider l'interfaçage entre la carte DE0-Nano-Soc , qui a des niveaux de signaux compatible 3.3V CMOS, et le transistor MOSFET.
 - b. **$V_{sgth} > V_{ol}(max)$ et $V_{sgth} < V_{oh}(min)$**
 - i. $V_{ol}(max)$: tension maximale pour un niveau logique '0'
 - ii. $V_{oh}(min)$: tension minimale pour un niveau logique '1'
 - iii. $V_{ol}(max) \text{ CMOS} = 1/3 * V_{DD} = 1/3 * 3.3V = 1.1V$
 - iv. $V_{oh}(min) \text{ CMOS} = 2/3 * V_{DD} = 2/3 * 3.3V = 2.2V$

Tension CMOS	0V à Vol(max)=1.1V '0' assuré	Vol(max)=1.1V à Voh(min)=2.2V 'X' indéterminé	Voh(min)=2.2V A Voh(max)=VDD=3.3V '1' assuré
Etat transistor MOSFET N	Etat bloqué	Etat quelconque	Etat passant

Il faut s'assurer que le transistor est bien **bloqué** quand la sortie de la carte DE0-Nano-Soc applique un **niveau logique '0' sur COMi**. Pour cela il faudra de satisfaire la condition **Vgsth > Vol(max)** qui assurera que le transistor sera bloqué pour un niveau logique '0'. De plus il faudra que pour un **niveau logique '1' sur COMi**, le transistor soit **passant**, impliquant que **Vgsth < Voh(min)**. En choisissant un transistor avec Vgsth =1.5V ou 2V, ces conditions seront validés.

Avec tel multiplexage le courant maximal consommé par le circuit devrait être celui consommé par 1 seul afficheur allumé au complet car à chaque instant il n'y a qu'un seul afficheur alimenté.

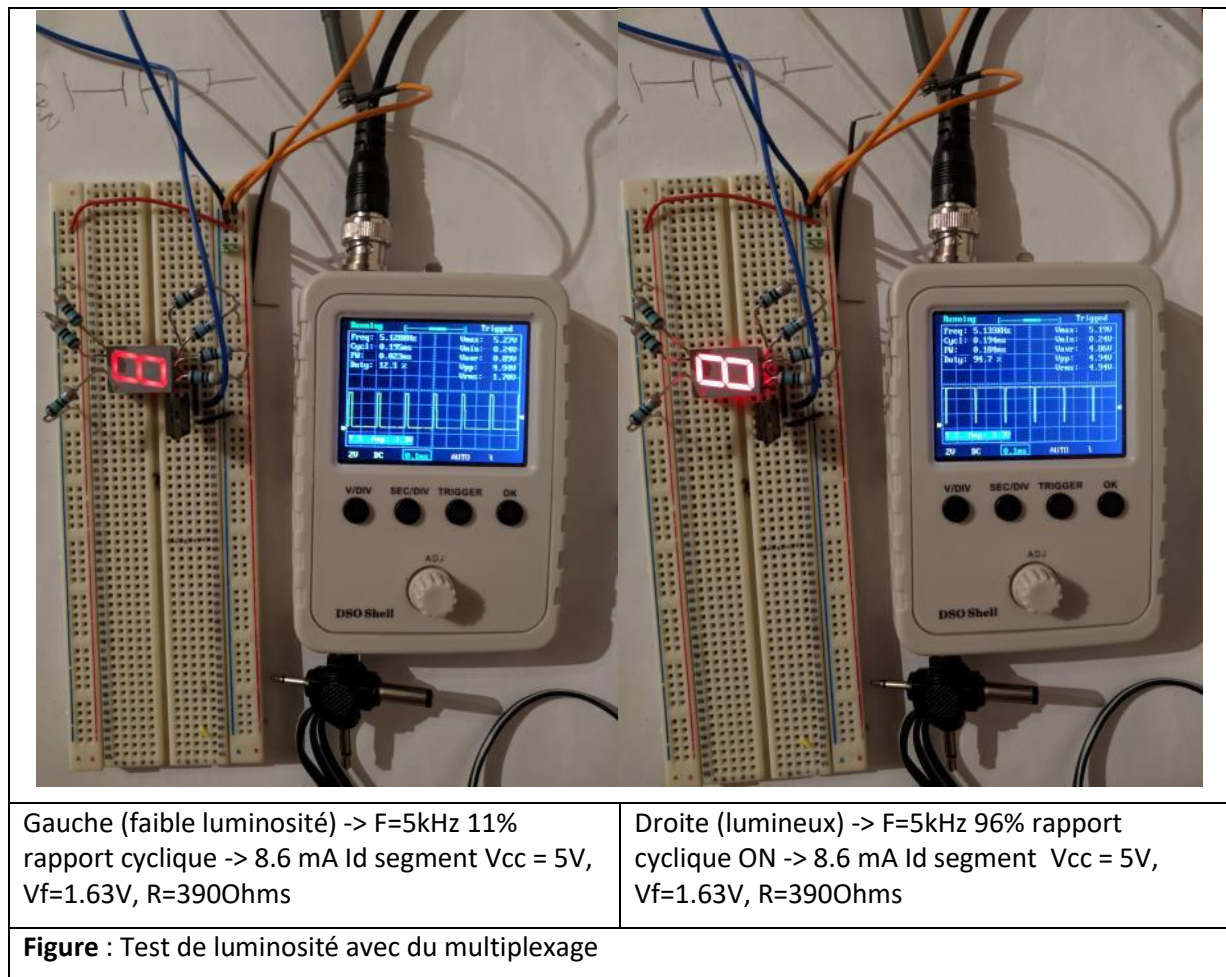
$$I(\max) = I_{\text{segment}(\max)} * N (\text{nb segments/afficheur})$$

Cette équation est valide dans le cas où l'on ne fait pas d'erreur de pilotage et où l'on ne pilote qu'un afficheur à la fois.

Cette solution implique les informations suivantes sur l'architecture:

- Un bus de données parallèle 14 bits + 13 bits sélection afficheur.
- GPIO utilisés sur la carte DE0-Nano-Soc=27 par « Time Circuit Display »
- 1/13 = 7.7% de rapport cyclique allumé
- Pas de circuit d'adaptation de puissance pour les LED -> Alimentation en direct via GPIO de la DE0-Nano, attention à ne pas piloter
- Etude menée sans tests réels et sans réalisation par Florian Rémy.

Test du 03/10/19 : Eclairage via multiplexage



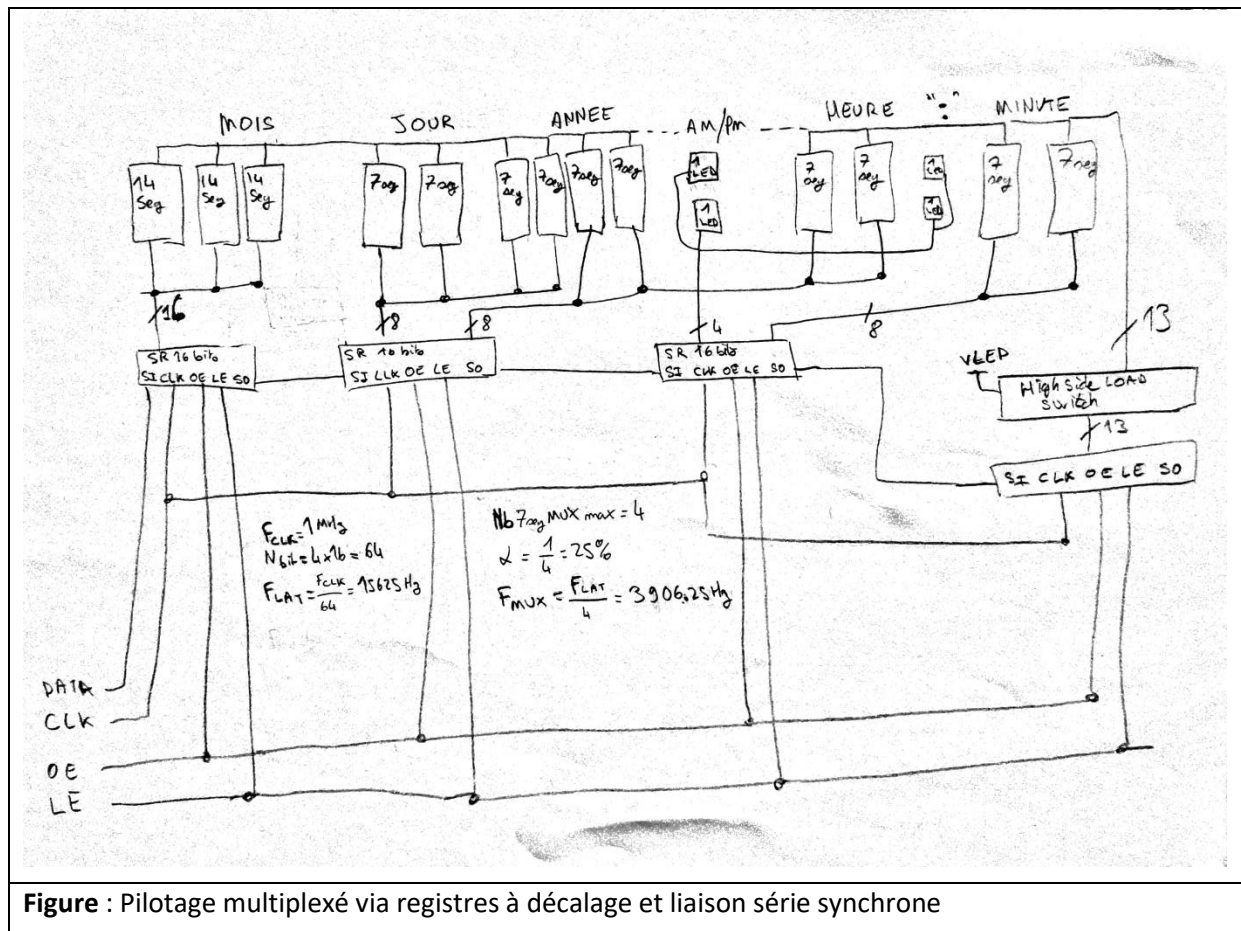
Bilan du test : Une configuration multiplexé en suivant les recommandations indiquées par l'étude de l'étudiant précédent montre une trop faible luminosité même à 11% de rapport cyclique. Il faut donc une nouvelle idée, soit booster les courant d'appels dans les segments, soit prévoir de ne pas multiplexer et piloter la puissance directement avec le FPGA mais avec des **composants spécialisés**.

2.4 Pilotage par drivers de LED sur registres à décalages :

Une seconde idée de pilotage basée sur des composants spécialisés de type registres à décalage pour LED est envisageable pour pallier aux problèmes potentiels d'alimentation posés par le pilotage direct. L'idée est de remplacer le bus parallèle direct fournis par les broches de la carte par une liaison série haute vitesse.

Globalement, on peut réaliser un multiplexage du même principe que celui présenté précédemment, mais en le divisant sur plusieurs registres à décalages.

Il faut voir les registres à décalage comme des bus parallèles DATA sur lesquels sont branchés au plus 4 afficheurs. Le multiplexage peut donc se faire avec un rapport cyclique de 25% et garantir une bonne luminosité.



- 4 registres à **décalage chaînés spécialisés pour le pilotage de LED** -> 64 sorties parallèles pour 1 entrée série synchrone haute vitesse.
- Multiplexage des afficheurs sur maxi 4 éléments -> Rapport cyclique d'allumage minimal = 25% -> Bonne luminosité atteignable
- Multiplexage par **Anode Commune**
- Mise en commun des bus de données sur 1 driver à décalage pour calibrer la puissance lumineuse à 1 référence d'afficheur -> Homogénéisation des intensités lumineuses
- **Interface FPGA = 4 fils** : DATA, CLK, OE, LE -> Série synchrone haute vitesse (~1 à 10MHz) + Synchronisation des étages de sorties LE + Modulation de la puissance lumineuse en sortie OE (PWM 1kHz à 100kHz)
- Possible de connecter des packs d'afficheurs 7 segments et 14/16 segments contenant 1, 2, 3 ou 4 digits avec bus de données commun et anodes communes. (Difficile de trouver sur 14/16 segments des simples digits, les multi-digits sont très communs et peu cher)
- Modulation de la puissance lumineuse en sortie possible
- Rajouter 1 ligne d'afficheur = Mobiliser seulement 4 broches supplémentaires du FPGA
- **Fonctionnalités**
 - Calibration du courant consommé par groupe d'afficheurs -> Homogénéité des intensités lumineuses
 - Réglage commun de l'intensité lumineuse via 1 PWM -> Réduction de la consommation où adaptation à la luminosité ambiante possible
 - Avec des drivers spécialisés, on économise les résistances de limitation de courant -> Réduction des composants

- Interface série 100% numérique et haute vitesse
 - Idéal pour un FPGA -> On peut synthétiser la logique combinatoire de pilotage des segments et sérialiser l'ensemble pour piloter les drivers de LED.
 - Peu gourmand en GPIO seulement 4
 - Pour $F_{CLK} = 1\text{MHz}$ on peut rafraichir les **64 broches** des drivers à $F_{LAT} = F_{CLK}/64 = 15625$. Si on veut multiplexer jusqu'à 4 afficheurs, il faudra rafraichir l'état des GPIO 4 fois pour que chaque afficheur ait pu être allumé $F_{REFRESH} = F_{LAT} / 4 = F_{CLK} / 256 = 3.9 \text{ kHz}$. On peut donc facilement jouer avec la persistance rétinienne car la fréquence du multiplexage pourra **atteindre 3.9 kHz bien au-dessus des 85Hz** nécessaires pour la persistance de vision.
- Interface de puissance garantie par des drivers spécialisés, pas de courant consommé depuis la carte FPGA elle-même (juste les entrées des registres à décalages soit quelques uA pendant les commutations).
- Possibilité de brancher des LED classiques et pas juste des 7 segments grâce au GPIO des drivers de LED en RAB.
- Courant maximal théorique -> Tous segments + LED ON = 116 LED x 10 mA = 1.16 A
- Références utilisables :
 - STP16CPC26(T-M-P)TR (0.712€ pc 05/10/19): <https://fr.rs-online.com/web/p/drivers-dafficheur-a-led/8805421/>
 - STP16CP05(T-M-P)TR (1.39 € pc 05/10/19): <https://fr.rs-online.com/web/p/drivers-dafficheur-a-led/7147692/>
 - P-MOS (As high side load switch) :
 - $V_{ds} \text{ max} = -20\text{V}$ $I_d = -3.7\text{A}$ $V_{gsth} = -0.55\text{V}$: <https://fr.rs-online.com/web/p/transistors-mosfet/0301322/>
 - $V_{ds} \text{ max} = -20\text{V}$ $I_d = -2\text{A}$ $V_{gsth} = -0.8\text{V}$: <https://fr.rs-online.com/web/p/transistors-mosfet/6710435/>
 - Afficheurs :
 - 7 segments : <https://www.mouser.fr/ProductDetail/Kingbright/SA08-11SRWA?qs=sGAEpiMZZMvkC18yXH9iljoyVwNe42b3vWJw%252BNzfHew%3D>
 - 16 segments : <https://www.mouser.fr/ProductDetail/Kingbright/PSA08-12SRWA?qs=sGAEpiMZZMvkC18yXH9iljoyVwNe42b3Eszh7tFCL4%3D>
 - Coût estimé de la solution globale de carte fille :
 - $3 \times 16\text{seg} + 10 \times 7\text{seg (anode com)} + 4 \times \text{STP16CPC} + 13 \times \text{MOS P} + 20 \times 0805 \text{ R} + 4 \times \text{POT } 5\text{k} + 10 \times \text{C } 0805 \text{ } 100\text{nF}$
 - $8,52 + 11,70 + 2,65 + 3,48 + 0,64 + 0,96 + 0,56 = 28,51 \text{ €}$
 - Prix partie pilotage : 8,29€
 - Prix afficheurs : 20,22€
 - / !\ EN REAPPRO DEPUIS 08/10/19 / !\
 - 35 à 45€ de composants
 - 70 à 80€ de PCB
 - Total : 105 à 125€
 - Disponibilité fournisseur au 07/10/19 : Tous OK.

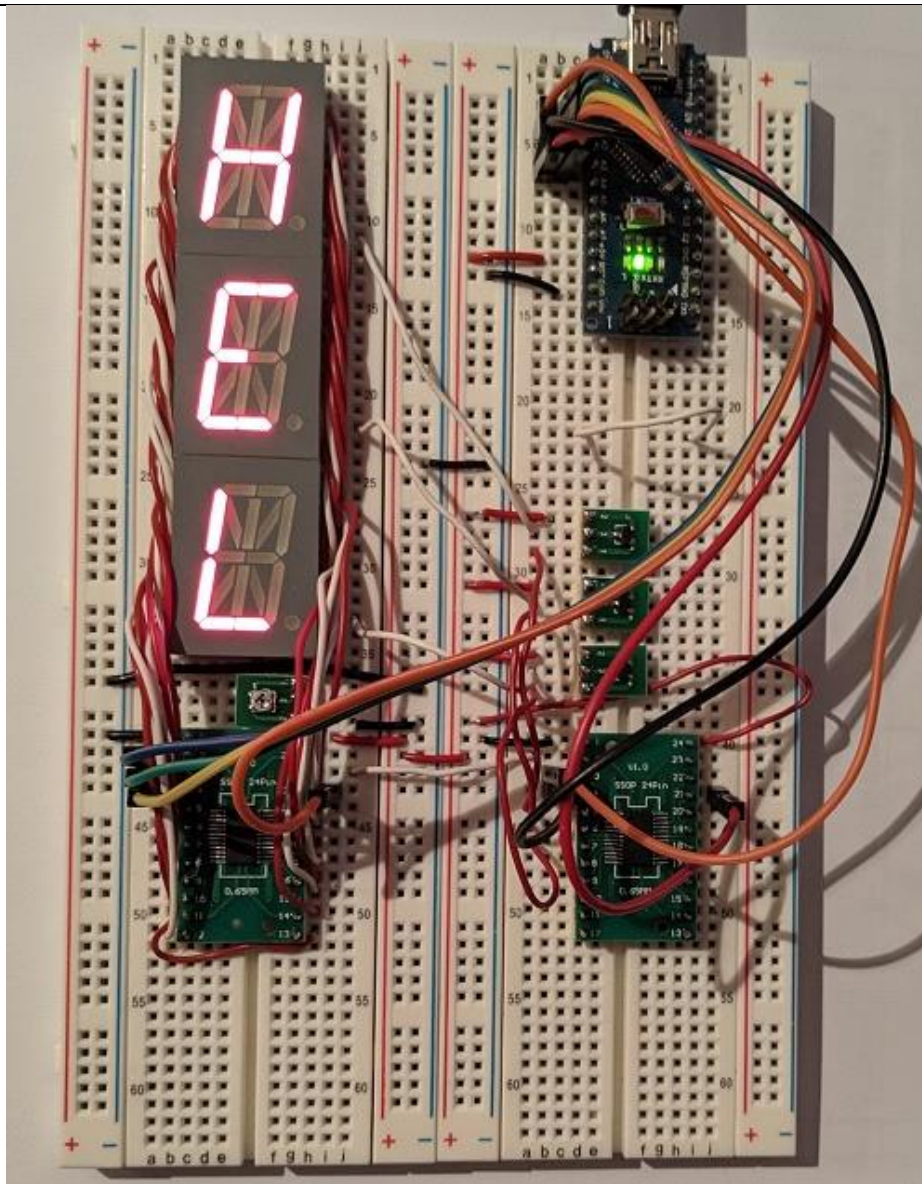


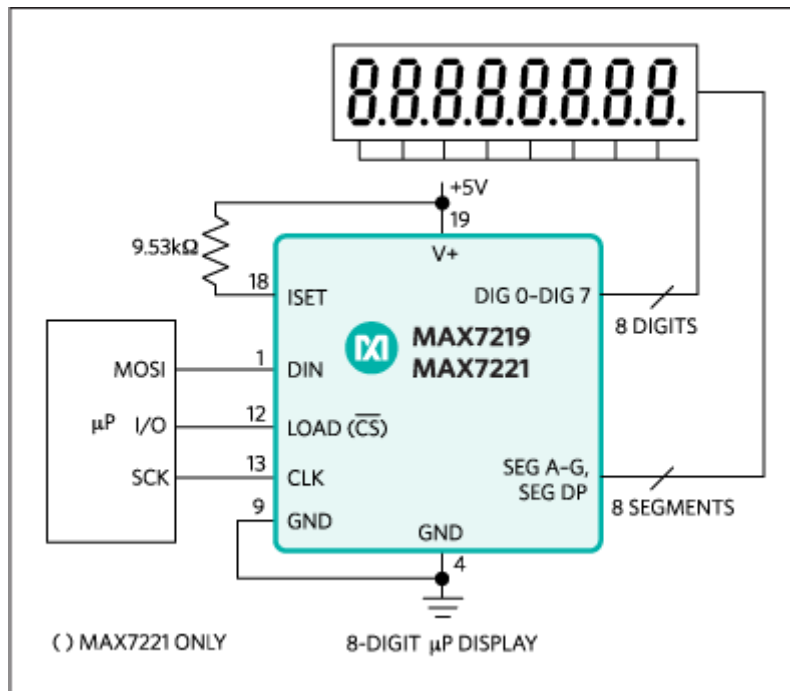
Figure : Preuve de concept câblée sur une platine breadboard et pilotage réalisé par une carte Arduino Nano.

2.5 Pilotage par composants drivers spécialisés

Il existe des drivers spécialisés pour piloter des 7 segments ou des matrices de LED. Ces composants ont l'avantage de réaliser le multiplexage des afficheurs en interne. Ce sont donc des composants fortement optimisés pour cette application. L'idée est de communiquer sur un **bus inter-IC** de type **SPI** ou **I2C** pour piloter le driver. Il ne reste donc qu'à indiquer au composant quel segment de la matrice allumer et le driver s'occupera du pilotage.

Exemple :

- **MAX7219** jusqu'à 8 afficheurs.
- **MAX69xx** drivers de LED dont 7 segments et 16 segments



MAX7219EWG+T (9.14€ pc) : <https://www.mouser.fr/ProductDetail/Maxim-Integrated/MAX7219EWG%2bT?qs=sGAEpiMZZMvbyeSUH4qH%2FPktRagWmJSX>

Caractéristiques :

- 10MHz Serial Interface
- **Individual LED Segment Control**
- Decode/No-Decode Digit Selection
- 150μA Low-Power Shutdown (Data Retained)
- Digital and Analog Brightness Control
- Display Blanked on Power-Up
- **Drive Common-Cathode LED Display**
- Slew-Rate Limited Segment Drivers for Lower EMI (MAX7221)
- **SPI, QSPI, MICROWIRE** Serial Interface (MAX7221)
- 24-Pin DIP and SO Packages

Globalement, on peut trouver des drivers spécialisés sur bus SPI. Ces drivers sont relativement chers.

2.5.1 Drivers de LED : MAX6954 ou MAX6955

Drivers de LED, 7-14-16 segments sur bus SPI. Ces drivers sont spécialisés pour le pilotage de LED au format afficheurs 16, 14 ou 7 segments.

Caractéristiques :

- High-Speed 26MHz SPI/QSPI/MICROWIRE®-Compatible Serial Interface
- 2.7V to 5.5V Operation
- Drives **Up to 16 Digits 7-Segment, 8 Digits 14-Segment, 8 Digits 16-Segment, 128 Discrete LEDs, or a Combination of Digit Types**
- **Drives Common-Cathode Monocolor and Bicolor LED Displays**
- **Built-In ASCII 104-Character Font** for 14-Segment and 16-Segment Digits and **Hexadecimal Font for 7-Segment Digits**
- Automatic Blinking Control for each Segment

- 10µA (typ) Low-Power Shutdown (Data Retained)
- 16-Step Digit-by-Digit Digital Brightness Control
- Display Blanked on Power-Up
- Slew-Rate Limited Segment Drivers for Lower EMI
- Five GPIO Port Pins Can Be Configured as Key-Switch Reader to Scan and Debounce Up to 32 Switches with n-Key Rollover
- IRQ Output when a Key Input Is Debounced
- 36-Pin SSOP and 40-Pin DIP and TQFN Packages
- Automotive Temperature Range Standard

C'est un **composant hautement intégré** qui réalise beaucoup des fonctions présentés dans la solution à drivers de LED sur registres à décalages.

Implications pour le projet :

- **Fonctionnalités :**
 - Intensité lumineuse bonne et paramétrable
 - Table de caractères ASCII pour le **mois** est incluse
 - Table de caractères HEX pour les autres 7 segments est incluse
 - Courant ajustable en externe
 - GPIO disponibles pour compléter l'IHM : LED ou BP ?
- **Limitations :**
 - Possible de mixer 14 segments et 7 segments mais driver l'ensemble de la carte avec 1 seul MAX 69xx est impossibles sans passer en mode adressage individuel de segments -> perte de l'intérêt des tables ASCII et HEX.
- **Nombre de composants réduits :**
 - 1 MAX6954 pour les 3 afficheurs 14 segments
 - 1 MAX6954 pour les 10 afficheurs 7-segments
- HDL revient à faire du protocole de commande du MAX6954 : intéressant ?

MAX6954 (16.88€ pcs) : <https://www.mouser.fr/ProductDetail/Maxim-Integrated/MAX6954AAX%2b?qs=sGAEpiMZZMsPdFgpQMKDR%2FOo0Y%252B3%2FmKR6MQiLgauxnU%3D>

Ces composants **sont cher** parce qu'ils sont sophistiqués au niveau de leurs fonctionnalités. Au vue de nos spécifications, un simple composant de pilotage de matrice de LED sera suffisant et assez bon marché.

2.5.2 Driver de LED : HT16K33 (Retenue)

Une solution à base de HT16K33 existe déjà et a été analysée et réalisée par un internaute. La faisabilité est donc garantie ! (Cf . Sources plus bas)

Ce driver intelligent est dédié au pilotage de matrices de LED 16x8=128 LED. On dispose donc d'un bus de donnée 16 bits qui pilote les **anodes** des LED et d'un bus de **commun qui pilote les cathodes** de la matrice de LED. On peut donc envisager de connecter des afficheurs 16-14-7 segments à **cathode commune** sur ce type de driver. Grâce au driver, on peut piloter individuellement chaque segment des afficheurs. C'est aussi le driver **le moins cher et le plus populaire** pour cette application. C'est donc très facile de s'en procurer.

Il dispose ensuite d'une mémoire RAM 16x8 pour contenir l'état de chaque LED de la matrice. L'ensemble est accessible est paramétrable individuellement depuis **une liaison I2C**.

Dans notre cas cela implique :

Sur le bus de données :

- 16 Anodes des 3 afficheurs 16 segments
- 8 + 8 Anodes de 2 afficheurs 7 segments

Sur le pilotage des **cathodes communes** :

- 1 COM par afficheur 16 segments -> 3 COM monopolisés
- 1 COM pour chaque couple d'afficheurs 7 segments (14 lignes) -> 5 COM monopolisés
- Le driver a 8 COM donc c'est adapté.

Pilotage :

- Niveau Co-design -> Déployer un **soft-core NIOS 2 avec un bus I2C** ou utiliser le **HPS et un de ses périphériques I2C**
- Implémenter un **driver logiciel bas niveau HT16K33**
- Implémenter un **driver logiciel haut niveau Time Circuit Display**
- Testable avec une *carte microcontrôleur classique type Nucleo ou Arduino*
- C.I. dédié au pilotage des LED, nombre de composants réduits et standard au format break out, facile à remplacer en cas de panne.

Coût estimé de la solution :

- 3*16 seg + 10*7 seg (cathode com) + HT16K33 breakout board
- 7,80 + 16,60 + 5,36 = 29,76€
- Partie circuits pilotage : 5,36€
- Partie afficheurs : 24,40€
- PCB : 80-90€

Cette solution a été retenue pour plusieurs raisons :

- Fonctionnement garantie par la preuve de concept
- Portable sur une plateforme microcontrôleur (et pas que le FPGA DE0-Nano-Soc)
 - Réutilisabilité du design
- Nombre de composant extrêmement réduit, essentiellement un driver HT16K33 et les afficheurs à segments.
- Faible coût du circuit intégré spécialisé par rapport aux solutions concurrentes.

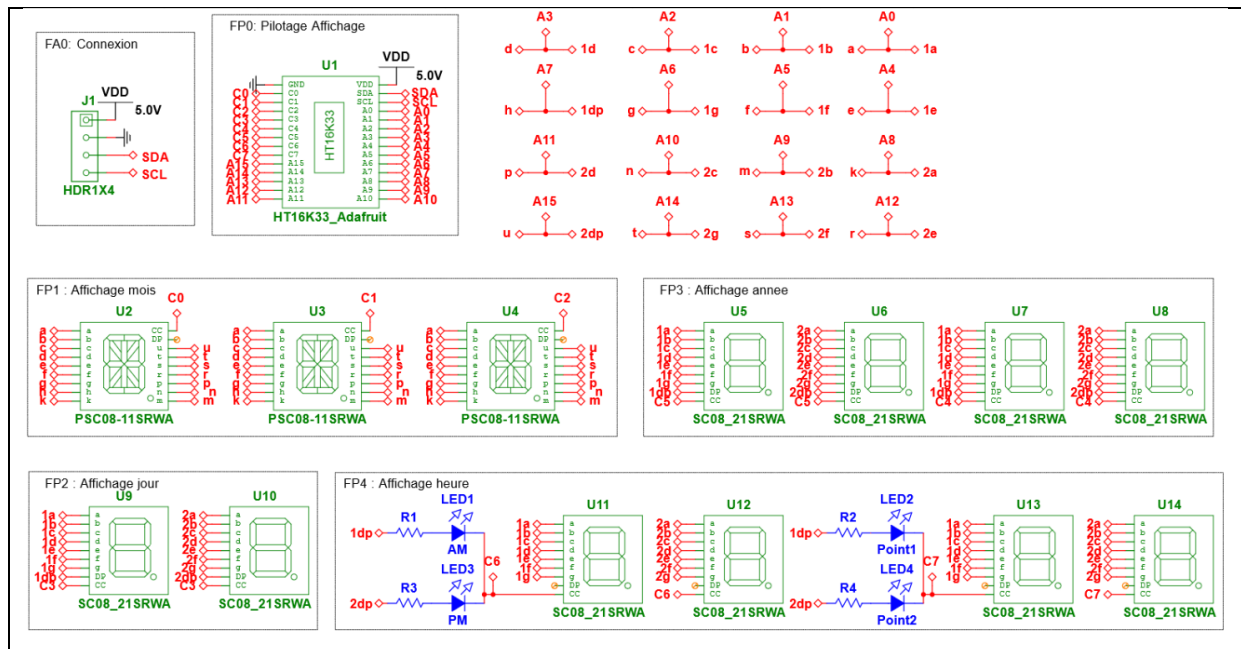


Figure : Schéma fonctionnel et structurel Time Circuit Display

Sources : Repo. Github LOCHE Jérémy

<https://github.com/lochej/TimeCircuitDisplay>

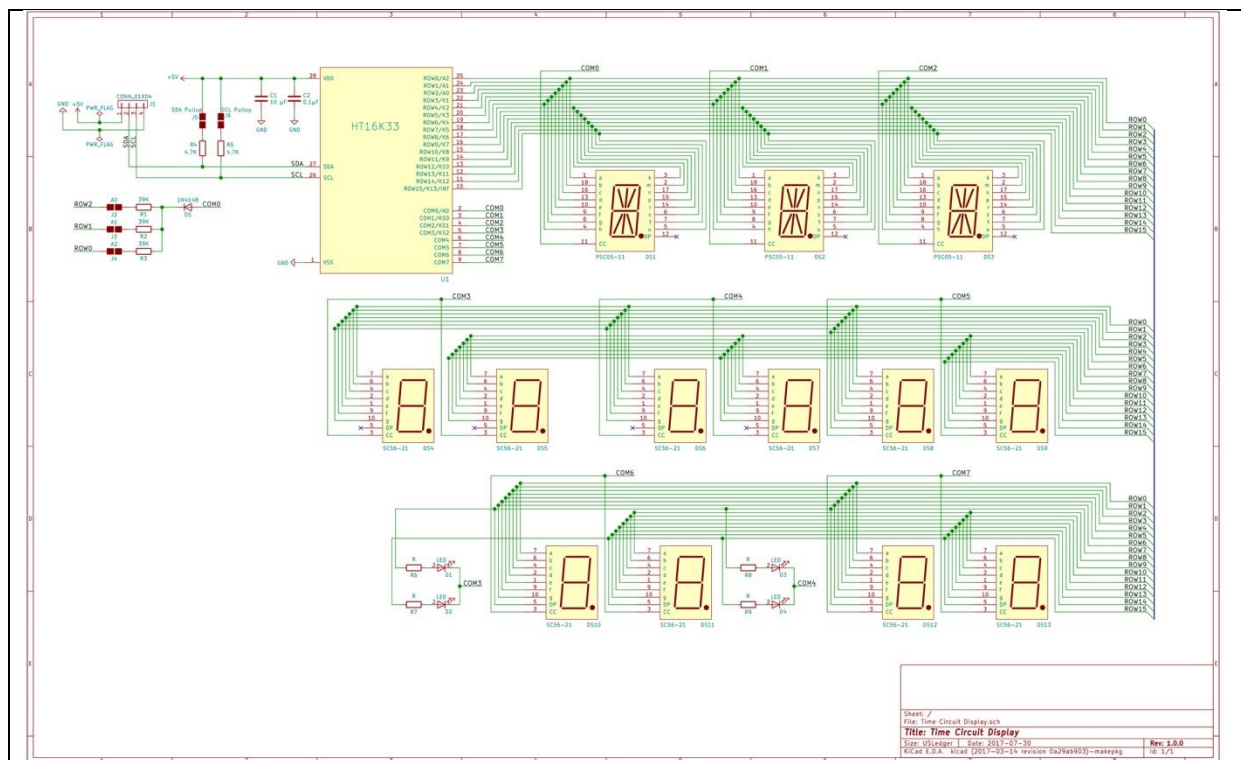


Figure : Schéma du Time Circuit Display (Parts Not Included) (inspiration uniquement)

Sources (valides au 03/12/19):

<https://www.partsnotincluded.com/bttf/driving-the-time-circuit-display-matrix/>

<https://www.partsnotincluded.com/bttf/time-circuit-display-schematic/>

<https://www.partsnotincluded.com/bttf/assembling-time-circuit-display-pcbs/>

<https://www.partsnotincluded.com/bttf/designing-the-time-circuit-display-pcb/>

2.5.3 Librairie logicielle de pilotage TCD : HT16K33

Pour piloter le TCD dans la version HT16K33, on a besoin d'un contrôleur I2C sur un microcontrôleur par exemple. Pour le test, la librairie logicielle sera implémentée en s'éloignant le plus possible des spécificités bas niveau de la cible MCU qui pilotera l'affichage.

Les objectifs sont les suivants :

- Fournir une API haut niveau permettant de piloter le TCD :
 - Piloter les heures, minutes, jour, mois, année à travers des fonctions haut niveau
 - Piloter l'intensité lumineuse du TCD
- Fournir une API bas niveau de pilotage du HT16K33.
 - Initialiser et configurer le HT16K33
 - Piloter l'intensité lumineuse
 - Allumer ou éteindre les LED de la matrice
- Réaliser uniquement l'adaptation bas niveau du pilotage I2C pour le pilotage du HT16K33, la librairie haut niveau ne doit pas changer ni le reste du programme.

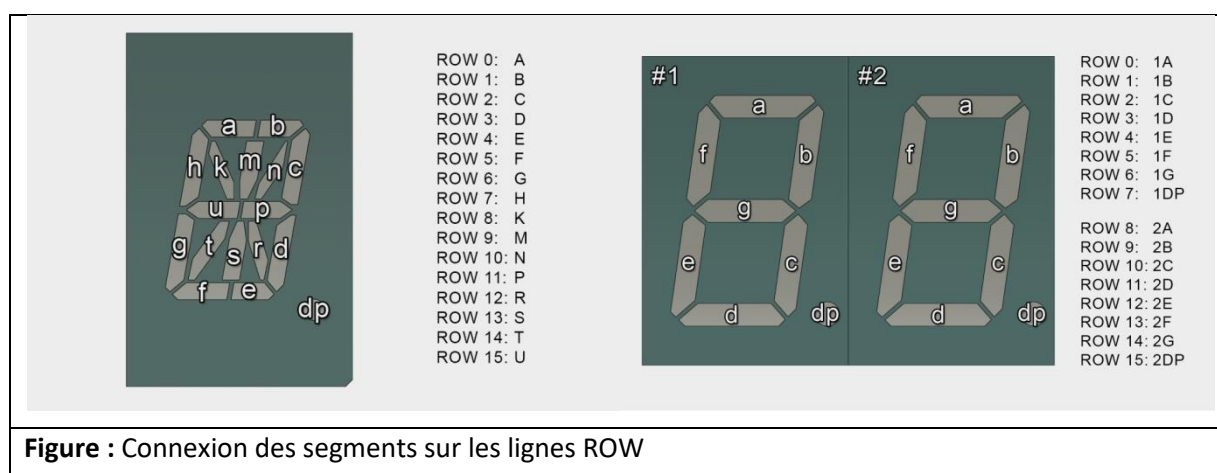
Il y a plusieurs jeux de registres dans le HT16K33, on peut envoyer des commandes pour configurer le composant et écrire le contenu de la RAM qui correspond à l'état des segments.

Documentation : <https://www.holtek.com/documents/10179/116711/HT16K33v120.pdf>

Les registres du composant HT16K33 offrent une correspondance en mémoire 1 pour 1 des segments. C'est-à-dire que pour piloter l'état des LED connectés au COM0 et sur les ROW[15:0] on va écrire dans les registres 0 et 1.

	D7	D6	D5	D4	D3	D2	D1	D0
RAM : 0x00	7	6	5	4	3	2	1	0
RAM : 0x01	15	14	13	12	11	10	9	8

Donc pour les 16 bits de la ligne, on a le msb qui pilote la LED 15 jusqu'au lsb qui pilote la LED 0 de la ligne. En ce qui concerne le câblage, les lignes ROW[15:0] sont connectés comme dans la figure ci-dessous.



On peut résumer cela dans le tableau suivant :

Table : Association des segments sur chaque ligne du HT16K33																
ROW[15:0]	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
16 seg.	U	T	S	R	P	N	M	K	H	G	F	E	D	C	B	A
ROW[15:0]	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
2x7 seg.	2DP	2G	2F	2E	2D	2C	2B	2A	1DP	1G	1F	1E	1D	1C	1B	1A

Concernant la génération de la police d'écriture sur les afficheurs 16 et 7 segments, on peut utiliser des polices de base disponible sur mon Github. La police d'écriture correspond à un ensemble de définition d'entier non signés où chaque bit d'un mot représente l'état d'un segment pour un caractère donné.

Motifs 16 et 7 segments en C : <https://github.com/lochej/LED-Segment-ASCII>

La police des afficheurs 16 segments est organisé sur un entier non signé 16 bits avec l'assignation des bits suivante :

Font_16_seg	b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
	U	T	S	R	P	N	M	K	H	G	F	E	D	C	B	A

La police des afficheurs 7 segments est organisée sur un entier non signé 8 bits avec l'assignation de bits suivante :

Font_7_seg	b7	b6	b5	b4	b3	b2	b1	b0
	DP	G	F	E	D	C	B	A

Comme on peut le constater avec le câblage suivi plus haut et l'agencement de la RAM du HT16K33, nous pourrions directement utiliser les valeurs données par la police d'écriture pour les écrire dans la RAM du composant. Pour un **caractère** donné, on a juste à récupérer l'entier depuis la police d'écriture pour ce caractère et le charger dans les registres de la ligne correspondant à notre afficheur.

Table : Association des segments sur chaque ligne du HT16K33																
ROW[15:0]	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
16 seg.	Font_16_seg[caractère]															
ROW[15:0]	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
2x7 seg.	Font 7 seg[caractère 1]								Font 7 seg[caractère 2]							

La RAM du composant HT16K33 est organisée en 16 octets. Un groupe de 2 octets contient l'état d'une ligne de la matrice de LED. Le tableau ci-dessous résume les adresses de chaque octet pour chaque ligne et colonnes.

@RAM HT16K33	ROW[15:8]	ROW[7:0]
COM0	0x01	0x00
COM1	0x03	0x02
COM2	0x05	0x04
COM3	0x07	0x06
COM4	0x09	0x08
COM5	0x0B	0x0A
COM6	0x0D	0x0C
COM7	0x0F	0x0E

Maintenant que l'on a le schéma de la mémoire RAM du composant, on peut y placer les valeurs pour chacun des afficheurs du système. Conformément au câblage (Cf schéma structurel), on peut remplir la RAM du HT16K33 de la façon suivante pour afficher la date et l'heure voulue :

Contenu RAM	ROW[15:8]		ROW[7:0]	
COM0	Font_16_seg[mois_lettre_1]			
COM1	Font_16_seg[mois_lettre_2]			
COM2	Font_16_seg[mois_lettre_3]			
COM3	Font_7_seg[jour_unités]		Font_7_seg[jour_dizaines]	
COM4	Font_7_seg[annee_unités]		Font_7_seg[annee_dizaines]	
COM5	Font_7_seg[annee_centaines]		Font_7_seg[annee_milliers]	
COM6	LED3 « PM »	Font_7_seg[heure_unités]	LED1 « AM »	Font_7_seg[heure_dizaines]
COM7	LED4 « Point2 »	Font_7_seg[minutes_unités]	LED2 « Point1 »	Font_7_seg[minutes_dizaines]

Les LED 1,2,3 et 4 sont pilotés grâce aux bits de poids fort des registres 0x0C 0x0E 0x0D et 0x0F respectivement. Ces LED permettent notamment de réaliser un affichage de l'heure du matin ou de l'après-midi (AM/PM anglo-saxon) et de faire clignoter les deux points « : » entre les heures et les minutes pour indiquer les secondes par exemple.

Maintenant qu'on sait comment générer les segments à afficher à l'aide de la police d'écriture et qu'on sait où placer les données d'affichage dans la mémoire RAM du HT16K33, il n'y aura plus qu'à coder tout cela dans un microcontrôleur ou processeur en C.

3 Convecteur temporel « Flux Capacitor »

Le convecteur temporel du projet sera basé sur des LED adressables WS2812b RGB. La réalisation matérielle a été réalisée par Alexis Rolland. Par conséquent il ne reste que la partie commande et animations à réaliser.

Le dispositif est basé sur 3 PCB de 8 LED WS2812b **duinoPeak**. Le plan de chainage est indiqué dans la figure ci-après.

On peut diviser le convecteur temporel en 3 branches **0,1,2**.

Si on indice les LED (id_branche) de chaque branches de 0 à l'intérieur jusqu'à 7 vers l'extérieur du convecteur, on obtient les indices de LED dans la chaîne (id_chaine) correspondants :

1. Branche 0 : id_chaine = 7 - id_branche
2. Branche 1 : id_chaine = 8 + id_branche

3. Branche 2 : $id_chaîne = 16 + id_branche$

On a désormais une manière simple d'identifier l'indice d'une LED dans la chaîne basée sur son emplacement dans une des branches.

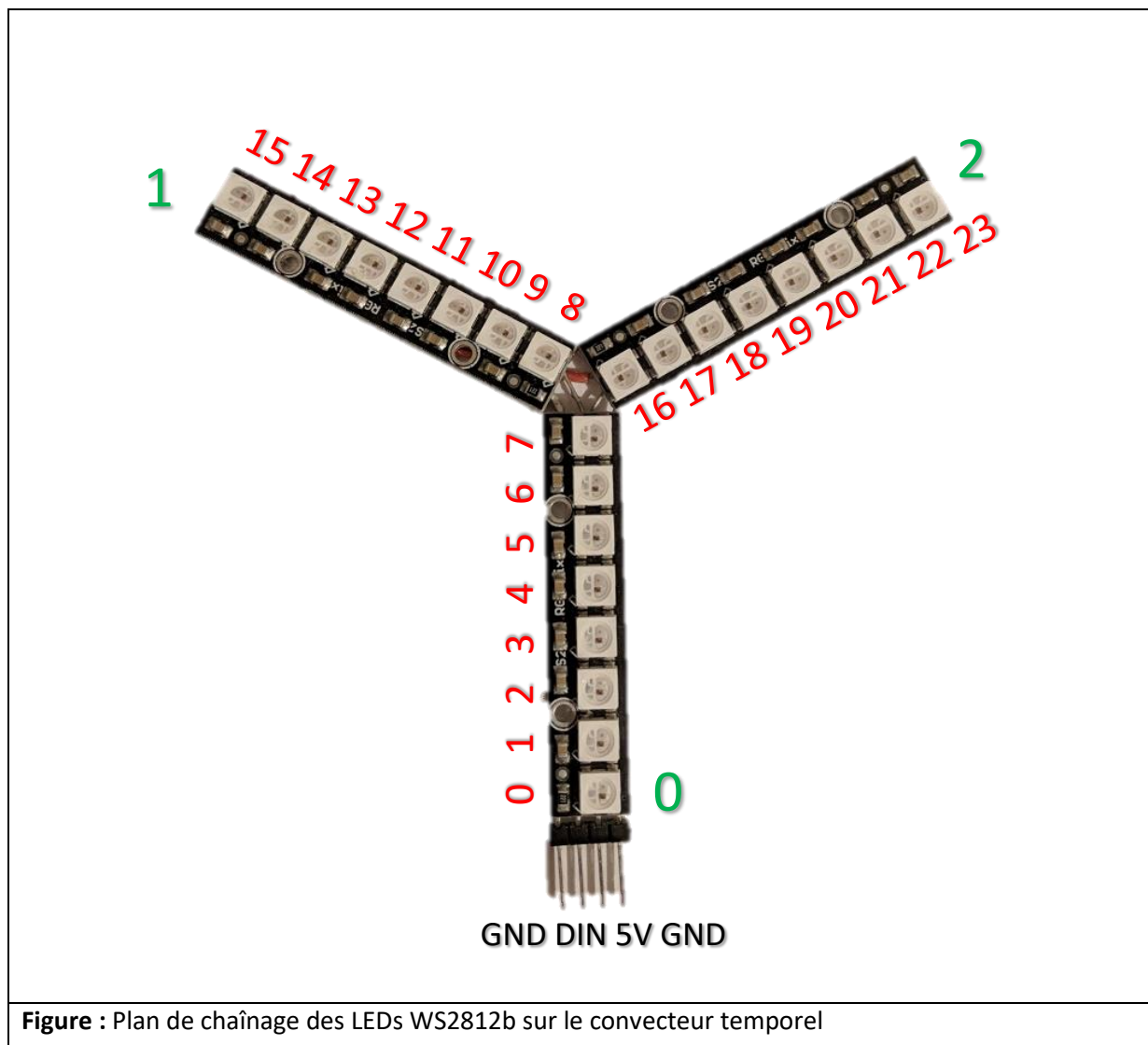


Figure : Plan de chaînage des LEDs WS2812b sur le convecteur temporel

Les LED sont alimentés en **5V continu** et piloté par une seule ligne de données compatible TTL et 3.3V. Les détails de pilotage seront énoncés dans la partie suivante.

La consommation maximale de ce module de LEDs est déterminée par l'équation suivante :

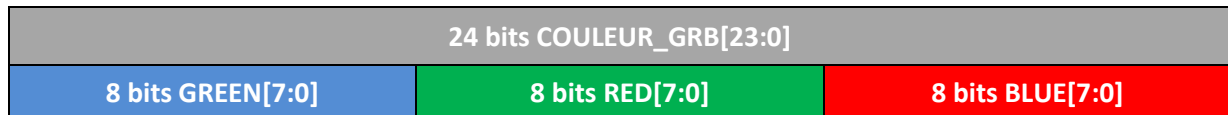
$$I_{max}(LED) \times NB_LEDs = 60mA \times 24 = 1440 \text{ mA}$$

Les LEDs ne consomment 1.4A que si elles sont toutes allumées au blanc fixe. Cette configuration ne sera pas utilisée en pratique du fait du trop fort éblouissement engendré par cette configuration. De plus, les LED ne seront pas allumées au blanc fixe mais plutôt avec des couleurs réduisant la consommation. Globalement, on espère n'utiliser que jusqu'à 25% de l'intensité lumineuse maximale. En moyenne, on aura au pire environ 360 mA de consommation.

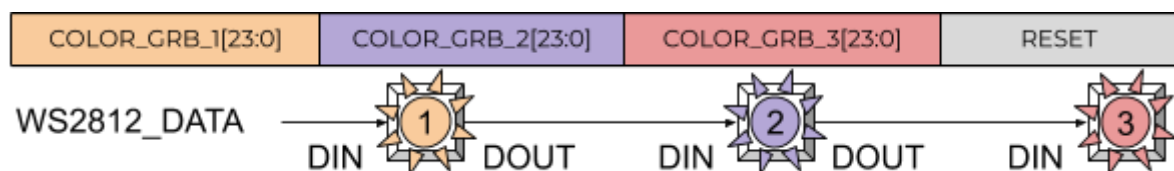
3.1 Pilotage des WS2812b

Les LED WS2812b sont des LED chaînables ([Datasheet WS2812b](#) et [Datasheet SK6812](#)) pilotable via un seul GPIO au travers d'une liaison série asynchrone de type MLI (modulation de largeur d'impulsion)/PWM.

La couleur de chaque LED est définie par un flux de 24 bits sur la liaison série.



Les informations de couleurs transmises sur la liaison sont chaînées les unes après les autres. Les 24 premiers bits sont pour la première LED dans la chaîne, puis les seconds 24 bits pour la deuxième LED et ainsi de suite. Une fois les 24 bits des LED envoyés, un signal de RESET ≥ 50 us est envoyé sur la ligne.



Les signaux 24 bits, sont mis en forme de bits '1' et '0' de la façon suivante :

Timings de référence (DOUT)		
T1H	0.6us \pm 150ns	
T1L	0.6us \pm 150ns	
T0H	0.3us \pm 150ns	
T0L	0.9us \pm 150ns	
RESET	≥ 50 us	

Table : Timings de référence pour le pilotage de LED WS2812b/SK6812

On sait désormais comment les signaux de pilotage des LEDs sont générés. On peut tout à fait implémenter une commande à partir d'un périphérique PWM cadencé à **800kHz** pour lequel le rapport cyclique est de **50%** pour envoyer un '1' et **25%** pour envoyer '0'. Il est aussi possible de générer ces signaux en utilisant du bit-banging pour générer les timings à l'aide de code logiciel. Il faut cependant pouvoir produire un code qui s'exécute de manière très prédictive et fiable. C'est le cas sur un coeur temps réel, un microcontrôleur ou un FPGA. Dans notre cas, la commande sera implémentée sur un FPGA à l'aide de synthèse matérielle Verilog.

3.2 Driver de LEDs WS2812b HDL

Pour piloter les LED WS2812b, on utilisera un de mes travaux réalisés en DII 4A. J'avais déjà implémenté un driver de LED WS2812b sur une carte FPGA Cyclone 4 DE2-115 mais il comportait

quelques erreurs de timings. Malgré un comportement intégralement validé en simulation, le résultat réel sur une carte FPGA était un pilotage incohérent des LED WS2812b. Ces problèmes étaient majoritairement dus à une synthèse trop peu optimisée pour fonctionner sur une cible réelle. Une réécriture a donc été nécessaire.

L'analyse ayant déjà été réalisée sur ce module, je vais reprendre succinctement les éléments analysés dans ce travail.

Pour générer les signaux nécessaires au pilotage des LED WS2812b, l'idée est de mettre en place une mega fonction HDL programmée en Verilog.

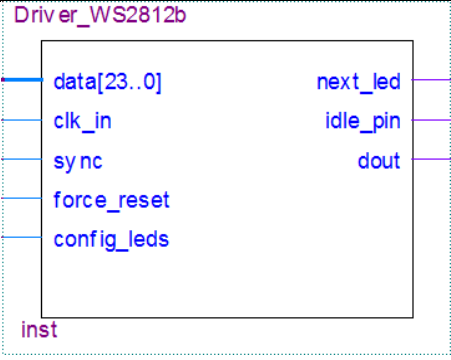
On commence par choisir le mode de transfert des données au driver.

On se base sur un bus de donnée parallèle 24 bits synchronisés par un signal permettant de demander la mise en place de la prochaine LED.

Le nombre de LED devant être paramétrable, on rajoute une entrée de configuration permettant de sauvegarder le nombre de LED à piloter dans le driver. On pourra donc changer le nombre de LED à piloter, sans avoir à retoucher au matériel.

3.3 Spécification de la fonction "Driver_WS2812b":

Au final, l'interface du driver est composée de 31 bits (Cf. schéma ci-dessus):

Table : Driver de LED WS2812b Bloc fonctionnel	
	
Entrées	Description
SYNC	Signal de démarrage de l'envoi des LEDs. Actif à l'état haut pendant au moins 300ns
FORCE_RESET	Signal de reset du driver. Arrête la séquence en cours et place le driver en mode <i>IDLE</i> . Actif à l'état bas pendant au moins 300ns.
CONFIG_LEDS	Signal de sauvegarde du nombre de LEDs du driver. Actif à l'état haut et en état <i>IDLE</i> uniquement (IDLE_PIN à '1') pendant 300ns au moins. Pour configurer, appliquer le nombre de LEDs sur DATA et effectuer une pulse à l'état haut pendant 300ns au moins. Ne pas oublier de remettre config à 0.
DATA[23:0]	Bus de données. Si IDLE_PIN est à '1', correspond au nombre de LEDs du driver, sinon (IDLE_PIN à '0') correspond à la couleur de la LED courante.
CLK_IN	Entrée d'horloge du driver rapport cyclique 50%. Attention, cette clock est divisée pour générer les timings du signal DOUT , assurez-vous de fournir une clock suffisamment rapide. ex: CLK_IN=50 000 000 Hz
Sorties	Description
NEXT_LED	Signal de pilotage des LED WS2812b, 800kHz, rapport cyclique 50% = '1' 25%='0'. A appliquer sur la pin DIN de la chaîne de LED WS2812b.
DOUT	Signal d'état du Driver de LED. IDLE_PIN = '0' signifie que le driver est en train d'envoyer les données aux LED. IDLE_PIN = '1' signifie que le driver est prêt à

	recevoir un coup de SYNC ou de configurer le nombre de LED avec CONFIG .
IDLE_PIN	Signal d'état du Driver de LED. IDLE_PIN = '0' signifie que le driver est en train d'envoyer les données aux LED. IDLE_PIN = '1' signifie que le driver est prêt à recevoir un coup de SYNC ou de configurer le nombre de LED avec CONFIG .

Le comportement interne du driver de LED est représenté par une machine d'états. Ce design est plus simple et plus fiable que le précédent développé en 4A. L'idée est de faire évoluer la machine d'état à chaque front montant sur CLK. Ensuite, les analyses comportementales (behavioral modeling) et d'équations logiques (gate level modeling) sont utilisées pour générer les signaux.

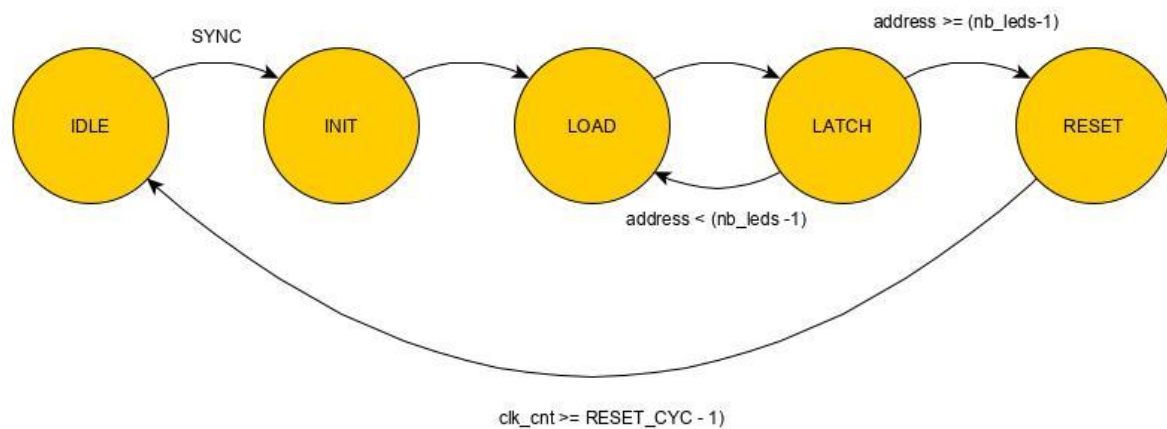
Pour générer les timings, on se base sur une horloge de référence **CLK_IN** de fréquence connue et fixe. L'objectif est d'utiliser un compteur interne **clk_cnt** pour diviser cette horloge et générer des événements à des timings voulus.

On définit donc les valeurs suivantes (reprise du tableau des timings vu précédemment) :

1. **H0_CYC** : nombre de périodes de CLK_IN pour créer le timing **T0H (~0.3us)**
2. **L0_CYC** : nombre de périodes de CLK_IN pour créer le timing **T0L (~0.9us)**
3. **H1_CYC** : nombre de périodes de CLK_IN pour créer le timing **T1H (~0.6us)**
4. **L1_CYC** : nombre de périodes de CLK_IN pour créer le timing **T1L (~0.6us)**
5. **RESET_CYC** : nombre de périodes de CLK_IN pour créer le timing **RESET (50us)**

Par exemple, pour générer le rapport cyclique de sortie sur DOUT, on va devoir compter suffisamment de période de CLK_IN pour pouvoir générer les 25% et 50% de rapport cyclique. Donc à minima, **Freq(CLK_IN) >= 4*Freq(DOUT)**. Plus la fréquence est élevée et meilleure sera la qualité du signal DOUT en sortie. D'après les tests, une fréquence de 50MHz fournis les bon timings.

Figure : Machine d'état du driver de LED WS2812b



Etat	Description
IDLE	Etat de base du driver, il ne fait rien. Dans cet état, si un front montant sur CLK est appliqué alors que CONFIG_LEDS est à 1, alors le registre interne nb_leds sera chargé avec la valeur placée sur DATA . Si le signal SYNC est à 1 lors du front sur CLK , alors on passe à l'état INIT .
INIT	Dans l'état INIT , le registre address interne correspondant à l'identifiant de la LED actuellement transmise est mis à 0 et on passe à l'état LOAD .
LOAD	Dans l'état LOAD , le registre led_data interne est chargé avec la valeur présente sur DATA (échantillonnage du bus DATA). Un compteur interne de cycle horloge clk_cnt est remis à zéro (pour générer les timings de DOUT), le compteur de bits bit_address permettant d'adresser le bit du registre led_data à envoyer est mis à 0. On passe ensuite à l'état LATCH .
LATCH	Dans l'état LATCH , à chaque front sur CLK , le compteur clk_cnt est incrémenté. Le compteur est utilisé pour la génération des timings de DOUT . Le pilotage de DOUT est décrit plus bas par un algorithme.
RESET	Dans l'état RESET , la sortie DOUT est maintenue 0 zéro pendant RESET_CYC cycles de CLK_IN et ensuite on repasse en état IDLE .

Algorithme : Algorithme de l'état LATCH

```
//Compter les cycles avec clk_cnt
clk_cnt ← clk_cnt +1

//A-t-on envoyé tous les bits du mot led_data
Si bit_address < 24 alors :

    //Les couleurs sont transmises MSB first
    //Envoi d'un 1
    Si led_data[23-bit_address] = 1 alors :

        //Réaliser le pilotage de DOUT en fonction des cycles
        Si clk_cnt < H1_CYC alors :
            DOUT ← 1
        Sinon :
            DOUT ← 0

        Si clk_cnt >= (H1_CYC + L1_CYC) alors :
            //Passer au bit suivant
            bit_address ← bit_address + 1
            //Remise à zéro du compteur de cycles
            clk_cnt ← 0

    //Envoi d'un 0
    Sinon :
        //Réaliser le pilotage de DOUT en fonction des cycles
        Si clk_cnt < H0_CYC alors :
            DOUT ← 1
        Sinon :
            DOUT ← 0

        Si clk_cnt >= (H0_CYC + L0_CYC) alors :
            //Passer au bit suivant
            bit_address ← bit_address + 1
            //Remise à zéro du compteur de cycles
            clk_cnt ← 0

Sinon :
    //A la fin de l'envoi d'une LED
    //Soit on a encore des LED à envoyer
    Si address < (nb_leds-1) alors :
        state ← LOAD
        address ← address + 1
    //Soit on a envoyé toutes les LED et on envoi le signal RESET
    Sinon :
        state ← RESET
        clk_cnt ← 0
```

Cette vue de la synthèse matérielle du driver de WS2812b est programmée un peu à la manière d'un logiciel. Il est vrai qu'on peut repenser la structure sous forme de blocs fonctionnels plutôt que par un algorithme. Ceci permet notamment de découper d'avantage l'algorithme en blocs plus simple.

Pour ma part, il a été rapide de travailler sur une méthode algorithmique séquentielle que de penser la chose sous forme blocs fonctionnels souvent adaptés à des processus asynchrones ou combinatoires.

3.3.1 Utilisation du driver HDL WS2812b version GPIO

On a vu précédemment comment construire le driver WS2812b au niveau matériel du FPGA.

Pour utiliser le driver, il suffit maintenant de piloter les entrées à l'aide un micro-processeur ou d'un microcontrôleur à travers des broches GPIO.

L'algorithme d'utilisation est le suivant :

Phase d'initialisation :

1. Appliquer un FORCE_RESET
 - a. Placer 0 sur FORCE_RESET pendant 300ns ou plus
 - b. Placer 1 sur FORCE_RESET
2. Configurer le nombre N de LED du driver
 - a. Placer le nombre de LED en binaire naturel sur DATA[23 :0]
 - b. Placer 1 sur CONFIG_LEDS pendant 300ns ou plus
 - c. Placer 0 sur CONFIG_LEDS

Phase d'utilisation :

1. Placer la couleur de la LED[0] sur DATA
2. Appliquer un SYNC
 - a. Placer 1 sur SYNC pendant 300ns environ
 - b. Placer 0 sur SYNC
3. Attendre un front montant sur NEXT_LED
4. Placer la couleur de la LED[1] sur DATA
5. Attendre un front montant sur NEXT_LED
6. ... Répéter 4 et 5 pour chaque LED jusqu'à la Nième LED

Dans mon cas, j'ai implémenté ce fonctionnement en C sur un microcontrôleur synthétisé dans le FPGA avec un NIOS2/e.

L'objectif est d'utiliser un ensemble de GPIO pour chaque signal du driver WS2812.

Lorsqu'on applique un SYNC, on active une interruption sur **front montant** au niveau du signal NEXT_LED. Lorsqu'un front est détecté, une routine d'interruption est appelée. On va donc placer la couleur de la LED suivante sur le bus DATA, et incrémenter un compteur pour passer à la LED suivante. Lorsque ce compteur atteint la valeur de la dernière LED ou que le signal IDLE est à 1, on sait qu'on a tout envoyé et que le driver a terminé la transmission aux LED.

Cette méthode fonctionne bien et est portable, malheureusement, elle nécessite de disposer d'un gestionnaire d'interruptions rapide pour placer à temps la donnée de la LED suivante sur DATA.

Sur une cible FPGA sur laquelle on fait de la co-conception matériel/logiciel, on va pouvoir utiliser un bus de communication mappé en mémoire et non plus des GPIO pour piloter le matériel.

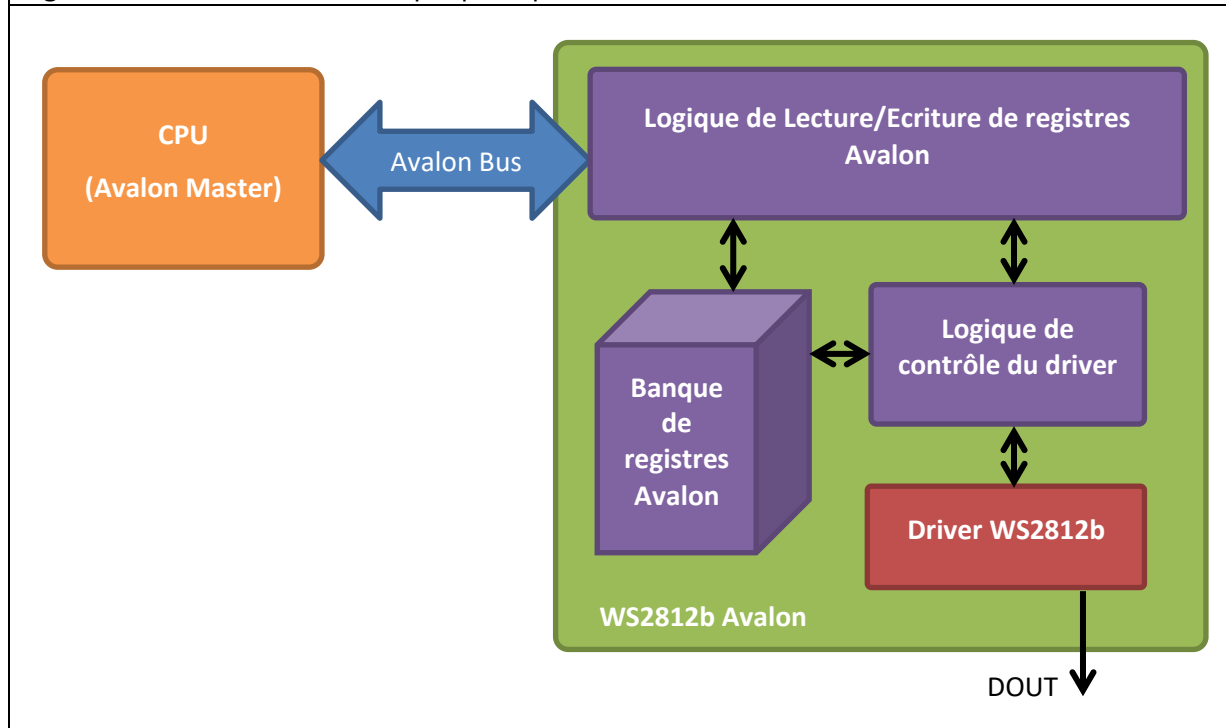
Pour un FPGA Intel (Altera) ce bus est appelé **Avalon**.

3.3.2 Couche d'adaptation Avalon MM du driver WS2812b

On a vu précédemment que la version présentée du driver WS2812b est purement basée sur des GPIO actuellement. Pour rendre ce périphérique plus facile d'utilisation et plus intégré au niveau matériel, on peut créer un périphérique Avalon à partir du driver WS2812b.

Un module Verilog Avalon synthétisé dans le FPGA Intel exposera des entrées/sorties compatibles avec le bus Avalon, d'autres entrées/sorties non spécifiques Avalon, ainsi qu'un ensemble de registres adressable depuis le master Avalon (CPU, MCU). Agir sur ces registres permettra d'agir sur le driver matériel en dessous.

Figure : Illustration d'un module périphérique Avalon



Le principe de création d'un module périphérique Avalon est de spécifier les registres que l'on va utiliser dans le périphérique et de créer les logiques d'adaptation au module matériel de base qu'on voulait adapter au départ.

Avec les signaux du bus Avalon, on peut savoir quand un maître va s'adresser à notre esclave. Pour un maître, accéder à un esclave Avalon va revenir à lire ou écrire à une adresse mémoire connue. Or un CPU peut très simplement adresser une zone de son espace mémoire pour de la RAM, et il pourra très bien le faire pour un périphérique.

Point important d'architecture :

Ce type de composant doit être adapté à l'architecture en question.

Le module basé sur des GPIO est indépendant de la plateforme FPGA sur lequel on le synthétise et pourrait très bien être fabriqué dans un circuit à part entière.

Lorsque l'on crée un module venant s'interfacer sur le bus mémoire interne d'un processeur (dans notre cas, le bus Intel FPGA Avalon), il faut réaliser une interface spécifique à ce bus.

Si demain nous décidons de créer un module sur un bus mémoire pour FPGA Xilinx, nous allons devoir réécrire la partie **logique de lecture/écriture des registres** à minima pour piloter les registres interne du module.

« Si la conception de la banque de registres et de la logique de contrôle du driver est couplée faiblement à la logique de lecture/écriture des registres, alors le portage devrait être relativement aisé. »

L'avantage de travailler sur un module mappé en mémoire permet au processeur applicatif exécutant du code logiciel d'accéder très facilement et en un minimum de cycles d'horloge au périphérique matériel par de simples lecture et écritures à des adresses mémoires connues.

C'est fini pour le point important.

Dans ma phase de conception de ce module Avalon, je me suis rendu compte qu'il est intéressant de simplifier le problème de la logique de lecture/écriture Avalon en la découpant dans 2 process distincts. On aura donc un **process d'écriture** déclenché par les signaux Avalon correspondant à une procédure d'écriture et un **process de lecture** déclenché par les signaux Avalon associés à une procédure de lecture.

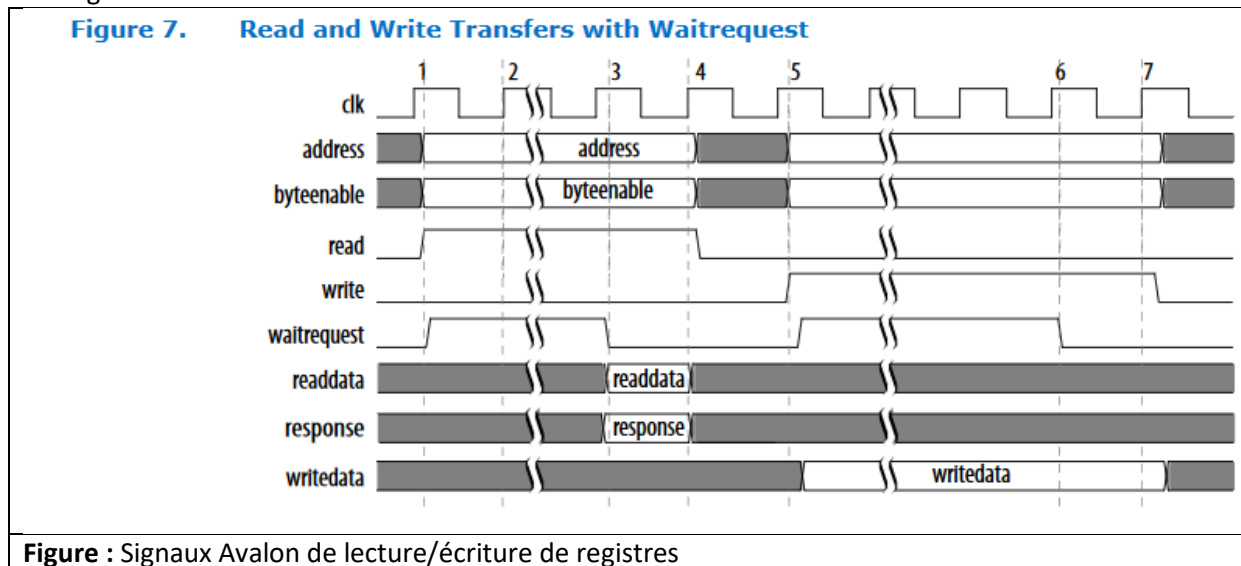
Ceci permet d'améliorer la lisibilité du code Verilog et de découper notre problème sous problèmes moins complexes.

Pour créer la logique de lecture et écriture, on se base sur les spécifications du bus Avalon.

Spécification Avalon (valide au 20/12/2019) :

https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl_avalon_spec.pdf?GSA_pos=2&WT.oss_r=1&WT.oss=avalon

Les signaux de commande d'écriture et lectures sont les suivants :



Le détail des signaux est disponible dans la spécification mais nous ne nous intéresserons qu'à une partie d'entre eux pour réaliser nos fonctions.

Les signaux intéressants pour nous sont :

Signal	Direction	Description
Synchronisation du matériel		
clk	Master -> Slave	Horloge du système du module esclave
reset	Master -> Slave	Entrée de reset pour la réinitialisation du module esclave
Logique de lecture de registres		
read	Master -> Slave	Entrée de signalisation d'une lecture de registre par le master sur l'esclave
readdata	Slave-> Master	Bus de sortie de la valeur du registre lue sur l'esclave
waitrequest	Slave -> Master	Sortie de mise en attente du master pendant la lecture du registre sur l'esclave (pendant que la valeur du registre est positionnée sur readdata)
address	Master -> Slave	Bus d'entrée signalant l'adresse du registre concerné par la procédure de lecture
Logique d'écriture de registres		
write	Master -> Slave	Entrée de signalisation d'une écriture de registre par le master sur l'esclave
writedata	Master -> Slave	Bus d'entrée de la valeur à écrire sur le registre de l'esclave
address	Master -> Slave	Bus d'entrée signalant l'adresse du registre concerné par la procédure d'écritures

A travers ce tableau récapitulatif, j'ai exposé les signaux associés aux actions concernées de lecture et d'écriture.

Je vais détailler les processus d'écriture et de lecture.

3.3.2.1 Procédure d'écriture d'un registre Avalon

L'écriture dans un registre Avalon se décrit de la façon suivante :
Nous supposons que le **reset** n'est pas appliqué pour que la procédure fonctionne.

A chaque front montant de clk :

Si reset = 1 Alors :

Sinon :

Si write = 1 Alors :

Décoder l'adresse du registre grâce à la valeur **address**

Mettre à jour le registre concerné en évaluant la valeur de **writedata**

C'est tout pour la procédure d'écriture. C'est la plus simple à gérer. Le principe sera toujours le même que ce soit pour lancer un sous process à l'écriture dans un registre ou juste mettre à jours une donnée dans l'esclave Avalon, on utilisera cette procédure.

3.3.2.2 Procédure de lecture d'un registre Avalon

La lecture d'un registre Avalon est plus compliquée que l'écriture. En effet, il va falloir gérer le signal **waitrequest**. Mon étude a montré que ce signal doit être générer de manière asynchrone pour coller avec la spécification du bus Avalon. Le process est donc découpé en deux morceaux :

1. Génération du signal **waitrequest**
2. Réponse à une lecture par le pilotage de **readdata**

Génération du signal waitrequest :

D'après la documentation des signaux Avalon, il est nécessaire que **waitrequest** passe à **1** de façon asynchrone juste après l'assertion du signal **read**. De plus il doit retomber à **0** dès que la donnée est placée sur **readdata**.

Pour cela, nous avons besoin d'un registre qui sera piloté par le process et synchronisé de la même façon que **readdata** pour satisfaire la seconde contrainte. Pour satisfaire la première contrainte, il faut que **waitrequest** soit piloté par **read** dès que celui-ci est placé à 1.

Au final, cela revient à forcer à 0 **waitrequest** lorsque la donnée est disponible sur **readdata** et la placer à 1 lorsqu'une transaction de **read** est demandée. On peut résumer cela en une condition booléenne **and**.

reg _waitrequest //création d'un registre _waitrequest assignable dans le process

assign waitrequest = read & _waitrequest //Assigner à waitrequest read et _waitrequest

Réponse à une lecture par le pilotage de readdata :

Lorsque vient le moment de lire les données, on va passer par un process synchronisé sur l'horloge **clk**.

A chaque front montant de clk :

Si reset = 1 Alors :

Placer **_waitrequest** à **1**

Placer **readdata** à **0**

Sinon :

Si read = 1 Alors :

Décoder l'adresse du registre à lire avec **address**

Placer la valeur du registre sur **readdata**

Placer **_waitrequest** à **0**

Sinon :

Si read = 0 Alors :

Placer **_waitrequest** à **1**

Si la procédure de lecture du registre est plus longue qu'un seul cycle, il ne faudra placer **_waitrequest** à 0 que le **readdata** sera mise à jours. On peut donc prendre plusieurs cycles pour lire une donnée dans la procédure **read** et ne placer **_waitrequest** à 0 qu'à la fin du traitement.

Cette procédure de lecture a été assez difficile à mettre en place notamment à cause de ce signal **waitrequest**. Cependant en analysant bien la documentation et en m'inspirant des exemples de périphériques produits par d'autres utilisateurs, j'ai réussi à créer un composant fonctionnel.

Détails et implémentation Verilog sur mon Github (valide au 20/12/2019):

https://github.com/lochej/WS2812_Avalon_Driver

3.3.2.3 Banque de registres et logique de contrôle

On a vu comment étaient constitués la couche de lecture et écriture du module Avalon. Cependant, il reste deux briques nécessaires, la banque de registres et la logique de contrôle.

Globalement, on a vu que pour piloter le module Driver de LED WS2812b, il est nécessaire de placer les données de couleur des LED sur le bus DATA au bon moment. Pour cela, on a besoin de stocker dans une banque de registre la couleur de chaque LED (i.e. 24 bits par LED).

Concernant la banque de registres, nous allons devoir aussi stocker le nombre de LED dans la banque de registres. Lorsque le composant est dans l'état IDLE alors on place le nombre de LED au format binaire naturel sur le bus DATA, sinon, c'est la couleur de la LED courante à afficher.

De plus à chaque front montant sur NEXT_LED, il faut placer la couleur de la prochaine LED à afficher sur le bus DATA.

On a donc besoin d'un compteur modulo NB_LED qui est dynamique avec entrée RESET asynchrone pour adresser la couleur de LED correspondante dans la banque de registres.

On peut résumer notre système avec les blocs suivants. Attention, sur ce schéma n'apparaissent pas les horloges de commandes, mais tous les modules sont synchronisés sur une unique horloge. L'ensemble est donc full-synchrone.

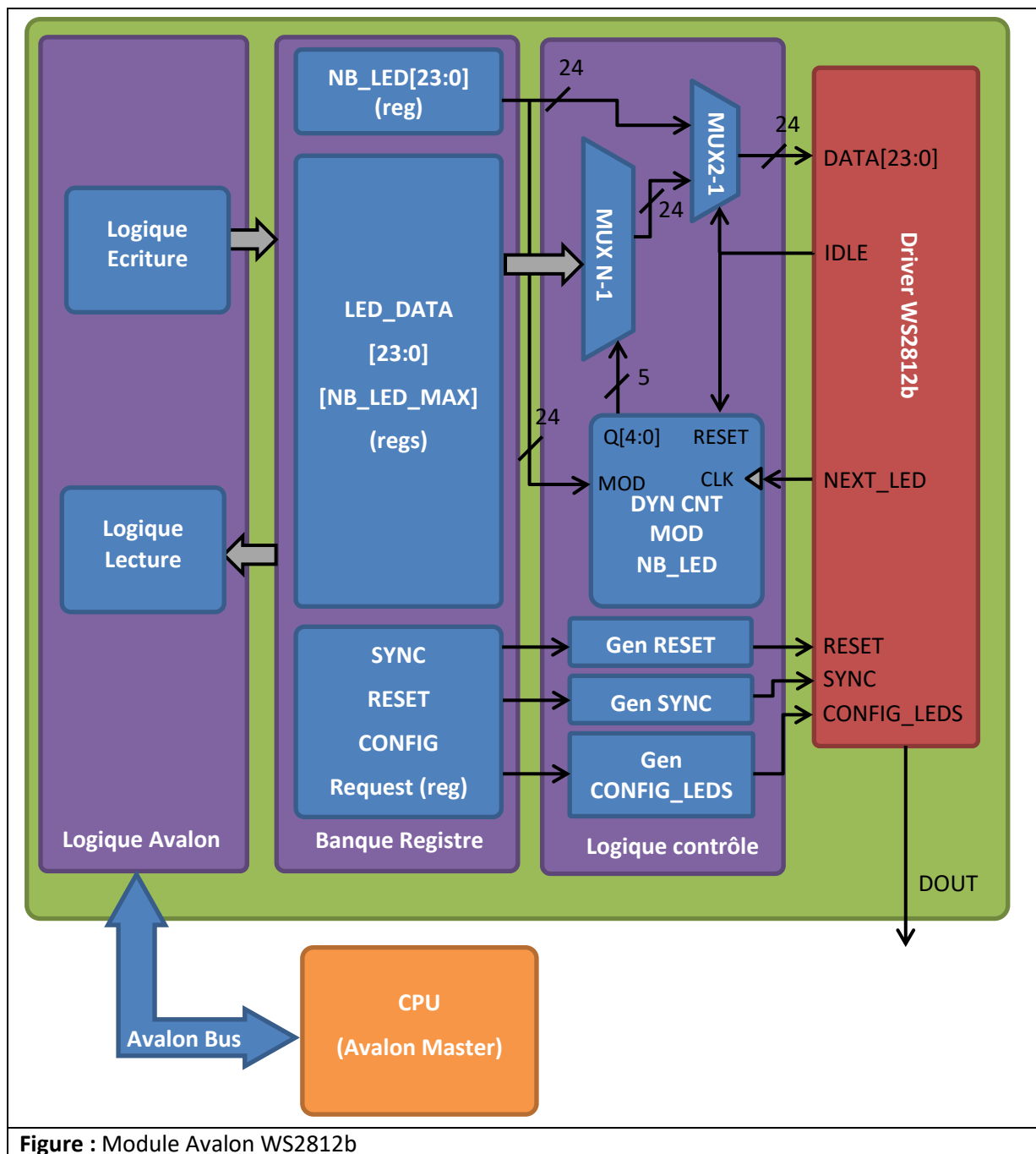


Figure : Module Avalon WS2812b

4 Architecture matérielle globale

On a vu comment j'ai choisi d'interfacer mes composants dans le système. Les LED WS2812b seront donc commandé par le FPGA directement à l'aide d'un périphérique matériel Avalon de ma conception. Le Time Circuit Display sera quant à lui piloté par un périphérique I2C standard. Sur notre carte DE0-Nano-Soc, il y a 2 périphériques I2C exposé du côté HPS (processeur Cortex A) qui sont directement utilisables. S'ils n'avaient pas été disponibles, nous aurions dû synthétiser un contrôleur I2C dans le FPGA pour y accéder depuis la partie Linux.

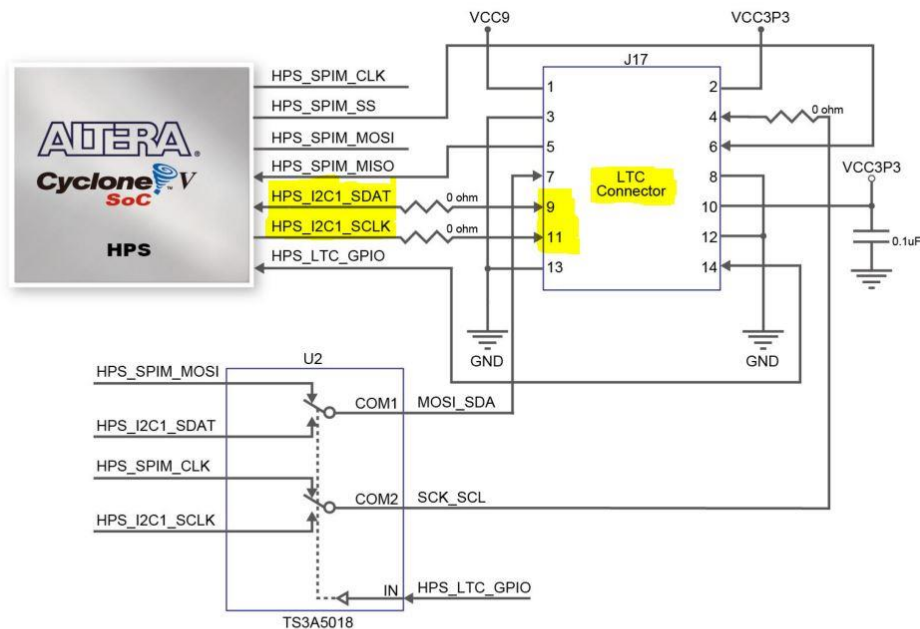


Figure 3-27 Connections between the HPS and LTC connector

Connecteur LTC :

1. GND (pin 13,3)
2. I2C1-SDA (pin 9)
3. I2C1-SCL (pin 11)

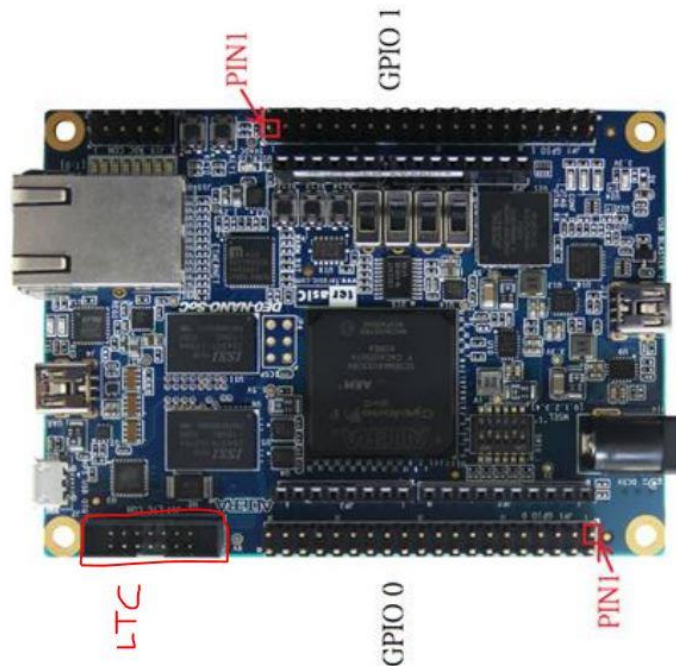


Figure : Port I2C1 sur le connecteur LTC DE0-Nano-Soc

La sortie DOUT du driver WS2812b peut quant à elle être routée sur n'importe laquelle des broches du connecteur GPIO0 ou GPIO1. J'ai choisis la broche GPIO0[0] pour DOUT.

Concernant l'alimentation du système, on dispose de points d'alimentation 5V sur les connecteurs GPIO0 et GPIO1.

D'après la documentation, l'énergie consommable sur ces ports ne dépend que de la capacité de l'alimentation connectée sur la carte.

1. Courant maximum consommé par « Time Circuit Display » = 250 mA
2. Courant maximum consommé par « Flux Capacitor » = 1440mA

La consommation totale ne devrait pas excéder les **2A** bien qu'en réalité, le système lumineux ne sera pas utilisé à 100% de sa luminosité. On devrait plutôt être en moyenne sur les lignes 5V de la carte aux alentours de **300 à 400 mA**.

Si on prévoit un adaptateur secteur pouvant fournir 5V et 2A continus, alors ce sera suffisant.

5 Gestionnaire système

Après avoir beaucoup parlé de matériel, nous allons maintenant parler de la partie logicielle du système. Le gestionnaire système est une des couches que j'ai spécifiées permettant de gérer la configuration du système à travers une interface en ligne de commande et une liaison IPC (Inter-Process Communication).

Pour rappel, l'objectif du gestionnaire système est de sauvegarder la configuration et gérer l'état du système.

Pour cela, on doit gérer les éléments suivants :

1. Time circuit display
2. Flux capacitor
3. Date et heure

Pour que tout cela soit évolutif, j'ai choisi de ne pas coupler les éléments dans un seul gros processus mais plutôt en modules. Le gestionnaire système est donc un ensemble de module, il n'y a pas à proprement parlé de logiciel « gestionnaire système ».

Chaque module gèrera et sauvegardera l'état de son élément.

Mon idée de base est la suivante :

1. Organiser les modules en 3 éléments :
 - a. 1 ou plusieurs processus de HAL pilotant concrètement l'IHM
 - b. 1 démon écoutant sur une liaison IPC (type serveur)
 - c. 1 interface en ligne de commande (type client)
2. Chaque module gère son fichier de configuration
3. Chaque module est chargé au démarrage par le système

Pour les choix d'implémentation, je recherche plusieurs qualités :

1. Format de stockage de configuration :

- a. Human readable/editable : pour réaliser des modifications à la main facilement
 - b. Machine readable/editable : le principe de base, le logiciel doit pouvoir lire et éditer sa configuration.
 - c. Evolutif en compatibilité ascendante : La montée en fonctionnalités doit être possible
- 2. Communication IPC :
 - a. Type d'IPC :
 - i. Multi-langage et interopérable : pour connecter n'importe quel client au démon
 - ii. Cloisonné à la machine
 - b. Format de données IPC :
 - i. Multi-langage et interopérable
 - ii. Evolutif en compatibilité ascendante
 - iii. Peu verbeux : pour la surcharge protocolaire
- 3. Programmation haut-niveau
 - a. Code source maintenable et lisible

Pour le choix du système de stockage et du format de données IPC, j'ai choisi le **JSON** qui dispose de toutes les qualités énoncées précédemment.

Pour le type de communication IPC, on peut utiliser les sockets UNIX ou les tubes nommés.

L'avantage des deux technologies est qu'elles sont disponibles dans tous les langages. Les bibliothèques pour gérer les sockets UNIX ou standards étant très répandues, ce sera donc mon choix pour l'implémentation.

Un socket UNIX ne passe pas par les couches réseau et par conséquent reste cloisonné à la machine. Il peut fonctionner en mode DATAGRAM ou STREAM, indiquant qu'on peut envoyer des paquets de données sans connexions (DATAGRAM/UDP) ou d'établir un flux binaire full duplex avec connexion entre les deux pairs (STREAM/TCP).

Concernant le langage de programmation, un langage interprété comme le Python 3 que je maîtrise est particulièrement efficace pour réaliser ce genre de programmes. La gestion native du JSON et des sockets UNIX est un plus non négligeable. Étant un langage interprété, la livraison de l'exécutable correspondra aussi à la livraison des sources.

L'idée générale est la suivante :

- 1. Le démon du module va gérer le processus driver HAL du périphérique piloté en le démarrant en tant que processus fils.
 - a. On peut donc connaître l'état du processus driver en l'interrogeant depuis le démon
 - b. On peut faire passer des informations au processus driver avec les entrées/sorties standard
 - c. On peut démarrer, tuer ou stopper ce processus fils depuis le démon
 - d. Le démon peut être un superviseur du processus driver
- 2. L'interface en ligne de commande ou tout autre service pourra venir se connecter au serveur démon pour piloter le périphérique. Au final, l'interface en ligne de commande ne sera qu'un client comme un autre utilisable dans des scripts, depuis un Shell ou en tant que processus fils appelé par un autre programme.

5.1 Module de gestion du « Time Circuit Display »

Pour gérer le « Time Circuit Display », on dispose d'un processus driver qui va piloter le time circuit display via le bus I2C1, d'un serveur qui va superviser ce processus et lui passer des paramètres (luminosité, on/off) et d'un client en ligne de commande.

5.1.1 Architecture générale du module

L'architecture générale est la suivante pour ce module :

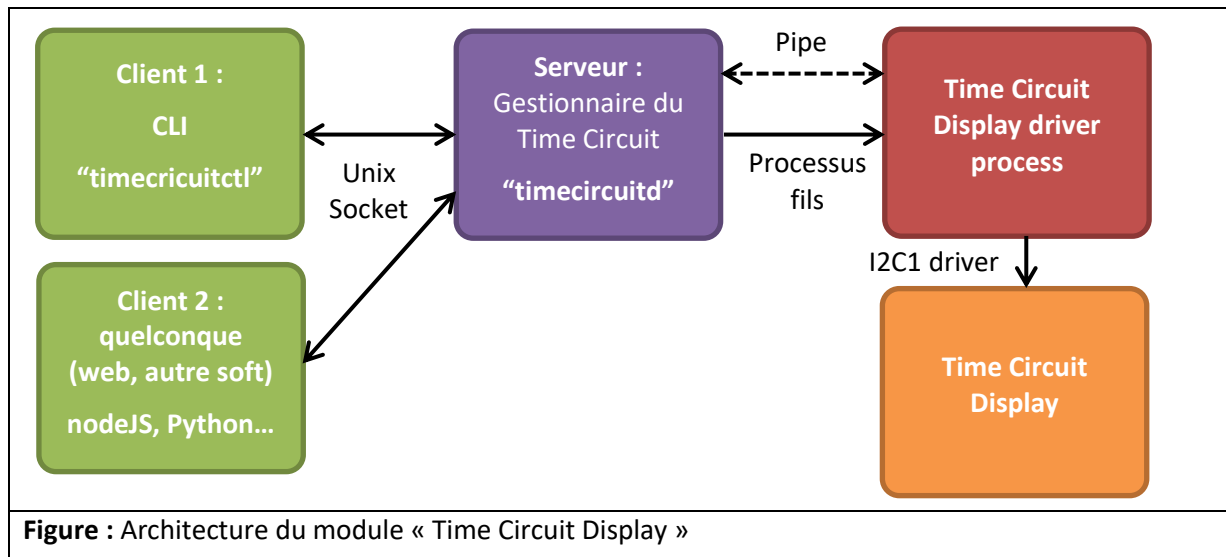


Figure : Architecture du module « Time Circuit Display »

5.1.2 Processus Driver (HAL)

Le processus driver qui réalise le pilotage du TCD est un programme assez simple qui récupère la date et l'heure du système courante pour l'afficher sur la carte électronique. Il est codé en C et prends en paramètre de lancement la luminosité à appliquer sur le TCD.

Il est instancié par le serveur qui lui passe les paramètres de la configuration à son démarrage. Ce processus fait partie de la HAL.

5.1.3 Serveur « timecircuitd »

Le serveur « timecircuitd » est chargé de démarrer et passer en arguments les paramètres du TCD au processus driver. Si un nouveau paramètre lui est passé par un client (ex : CLI ou autre) alors le serveur va redémarrer le processus driver en lui passant les nouveaux paramètres. Il sera démarré automatiquement par le système au démarrage. A chaque réception d'un nouveau paramètre, le serveur met à jour sa configuration dans son fichier de configuration.

5.1.4 Client CLI « timecircuitctl »

A l'image des services systemd sous linux, une interface en ligne de commande est disponible pour gérer le service (serveur).

5.2 Module de gestion du « Flux Capacitor »

La gestion du convecteur temporel (Flux Capacitor) passe par l'utilisation de la couche HAL.

5.2.1 Flux capacitor Middleware (driver HAL)

Pour piloter les LED WS2812b, on se rappelle que l'on doit utiliser le module Avalon mappé en mémoire. Or pour cela il faut accéder à l'espace mémoire physique du système et il vaut mieux

éviter que ce soit un programme utilisateur qui y accède. Pour résoudre cette question, j'ai repris l'analyse réalisée en 4A sur le LED WALL (sur lequel j'ai aussi travaillé).

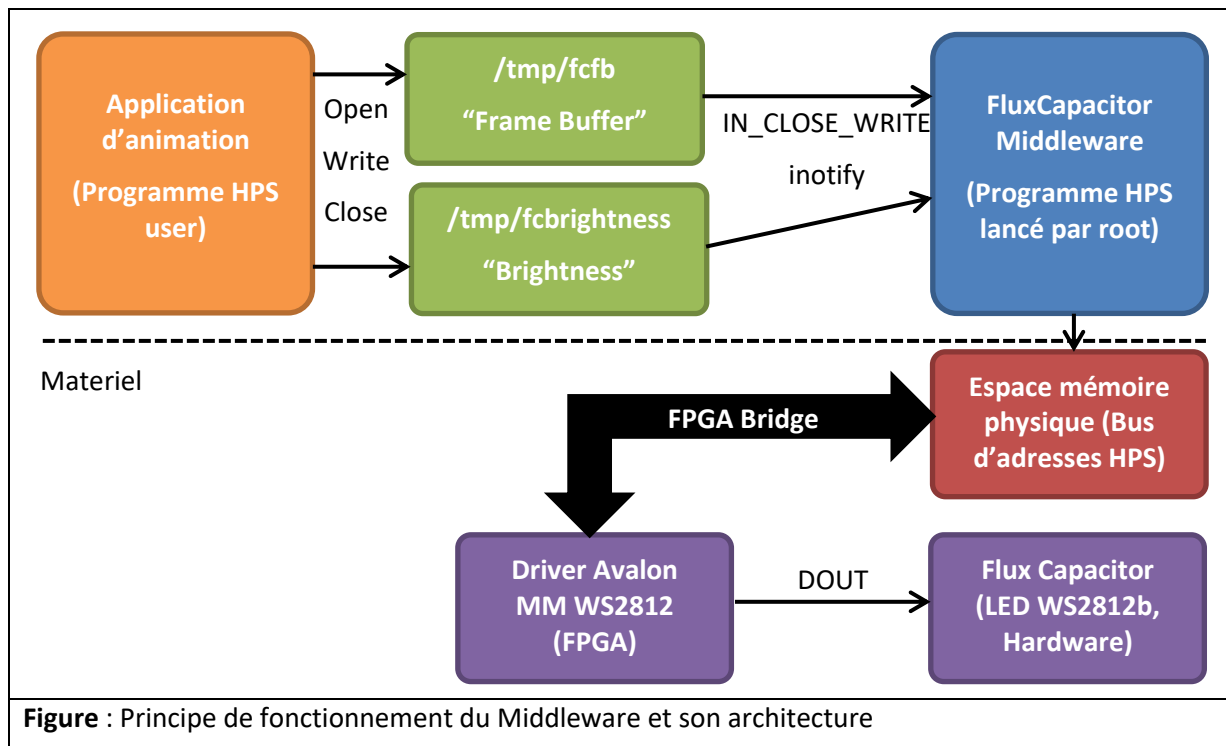
L'idée est la suivante :

1. Un programme appelé **Middleware** est exécuté par l'utilisateur **root** et est le seul à avoir accès à l'espace mémoire physique (kernel) de la machine. Ainsi ce programme testé et simple ne réalisera pas de fuite mémoire ni débordements pouvant causer des plantages ou des brèches de sécurité.
2. Il expose un ou plusieurs fichiers dans l'espace mémoire temporaire **/tmp** dans lequel peuvent écrire les applications pour mettre en œuvre le matériel.
3. Le processus Middleware est réveillé par évènement de type **inotify** sur les fichiers exposés dans **/tmp** lorsqu'une application veut utiliser le matériel

En d'autres mots, c'est une sorte de driver linux qui est ici créé mais sans les difficultés de travailler en mode kernel.

Ce module HAL permet donc d'agir sur les LED WS2812 du convecteur temporel.

5.2.1.1 L'architecture du middleware



Globalement, avec le middleware, on crée une abstraction de l'accès physique au matériel. Un programme applicatif réalisant des animations sur le convecteur temporel n'aura qu'à savoir ouvrir, écrire dans un fichier et le fermer pour piloter les LEDs. Nous avons utilisé ce système dans le mur de LED de Polytech Tours qui est aussi équipé de LED WS2812 et cela s'est avéré très efficace, performant et économe en ressources.

Autre point intéressant est qu'on dispose de fichiers dans le système qui contiennent l'état des LED, on peut donc s'en servir pour faire des prévisualisations de ce que le matériel affiche en lisant aussi ces fichiers.

Le matériel est abstrait à travers des fichiers ce qui est très simple à utiliser. Enfin, **inotify** est un mécanisme d'attente passive sur des fichiers ou répertoires. Ainsi, un programme peut attendre un ou plusieurs évènements sans consommer de CPU et il sera débloqué par le noyau lorsque la dite action sera effectuée sur le fichier. C'est aussi un moyen de réaliser de la communication interprocessus autre que par un pipe, un socket ou un tube nommé.

Attention : Il faut s'assurer d'une chose cependant, les fichiers dans **/tmp** seront potentiellement écrits plusieurs dizaines de fois par secondes ce qui peut endommager de la mémoire flash ou la carte SD si le répertoire **/tmp** est monté sur un support de stockage et non en RAM.

Il faut donc monter /tmp en RAM !

Donc le Middleware sera autonome et sera lancé par le système au démarrage lorsque le FPGA sera programmé avec le module WS2812.

5.2.1.2 Format de données Middleware

Couleurs des LED (framebuffer) :

Pour garder un format de données efficace, j'ai repris la même analyse que menée sur le LED Wall. Le fichier **/tmp/fcb** contient une **frame** affichée sur les LED WS2812b. A chaque frame générée par une application, le programme Middleware va afficher la couleur sur les LED.

Le format est le suivant : **r0g0b0r1g1b1r2g2b2....rigibi**

Ou :

- r0 l'octet de la couleur rouge pour la led 0 dans la chaîne (0 à 0xFF)
- g0 l'octet de la couleur verte pour la led 0 dans la chaîne (0 to 0xFF)
- b0 l'octet de la couleur bleue pour la led 0 dans la chaîne (0 to 0xFF)

Suivi de la couleur des autres LED de la chaîne. Les données sont écrites en binaires dans le fichier. On a donc 3 octets RGB par LED dans le fichier.

Luminosité des LED :

Pour gérer la luminosité des LED, une application d'animation peut soit agir sur le **framebuffer** directement avec son algorithme de gestion de la luminosité ou utiliser le fichier **/tmp/fcbrightness** dans lequel il suffit d'écrire au format ASCII l'intensité lumineuse de 0 à 255.

Le Middleware applique alors une mise à l'échelle des couleurs en fonction de la valeur de la luminosité. L'intensité de chaque composantes RGB sera multipliée par le facteur **brightness/255**.

5.2.2 Serveur « fluxcapacitord »

Le serveur « fluxcapacitord » va gérer un ensemble de processus d'animations disponibles. Il pourra de la même façon que « timecircuitd » démarrer ou tuer le processus d'animation.

5.2.3 Client CLI « fluxcapacitorctl »

C'est le même principe que pour « timecircuitctl ».

5.3 Module de gestion du bouton d'extinction

Lors des spécifications, il a été demandé à ce que le système soit équipé d'indicateurs de fonctionnement ainsi que d'un bouton d'extinction permettant d'éteindre proprement le système d'exploitation avant de couper l'alimentation.

Pour cela, on dispose d'une LED embarquée et d'un bouton poussoir tous deux connectée au processeur HPS et donc directement accessible depuis l'OS Linux.

A l'aide de la librairie **gpiod** linux et de l'utilitaire **gpiomon** on peut détecter des fronts montant ou descendant par interruption.

Ceci est très utile pour réaliser une attente passive sur une entrée du système.

Le bouton de la carte et la LED sont branchés respectivement sur le port 1 pin 25 et 24.

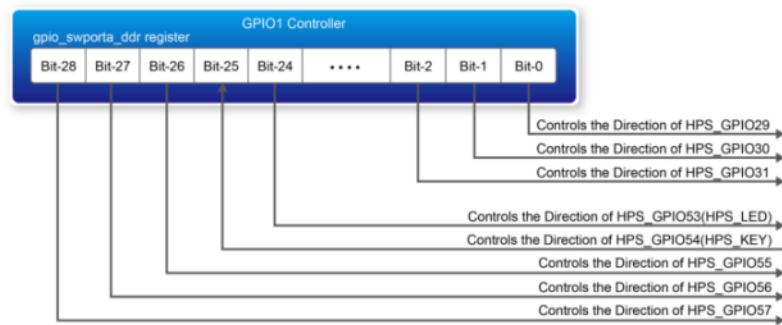


Figure 6-5 gpio_swporta_ddr register in the GPIO1 controller

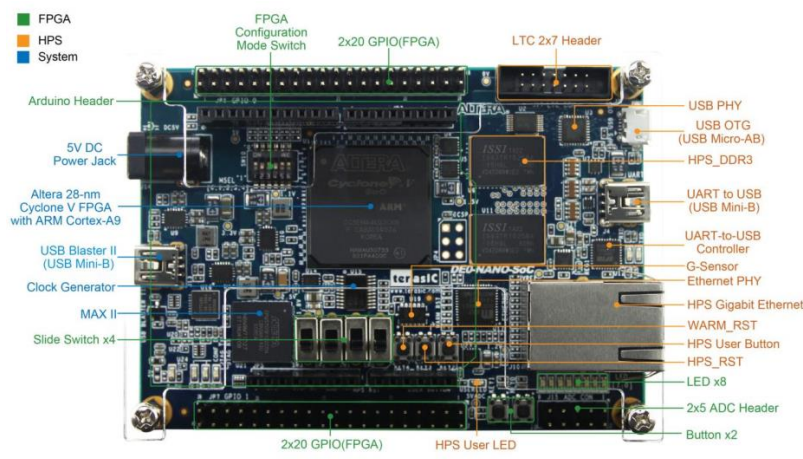


Figure 2-1 DE0-Nano-SoC development board (top view)

Figure : Schéma de câblage des boutons et LED HPS

Enfin, la LED intégrée à la carte est pilotable via l'interface `/sys/class/leds/hps_led0` de manière standard à linux.

On aura alors 3 état de la LED **HPS_User_LED** :

1. **Clignotement de type heart-beat** : système allumé ne pas couper l'alimentation.
2. **Allumé au fixe** : système en cours d'extinction, ne pas couper l'alimentation.
3. **Eteinte** : système Linux éteint, on peut couper l'alimentation.

Pour lancer la procédure d'extinction du système, il suffira alors de monitorer l'état du bouton **HPS_User_Button** actif à l'état bas sur la broche GPIO(1,25) avec **gpiomon**.

Lorsque l'appui est détecté, on allume la LED au fixe et on lance la procédure d'extinction du système avec la commande **poweroff** de Linux. Lorsque le système est éteint, la LED s'éteint.

6 Serveur WEB de pilotage

Le serveur WEB correspond uniquement à une interface de configuration graphique qui est compatible PC et mobile. Sur cette interface, on pourra modifier les paramètres des périphériques du système. Globalement on ne disposera que d'un ensemble de boutons, sliders, listes et champs de textes pour piloter les éléments.

Etant novice en site WEB, j'ai proposé de partir sur une solution compatible avec les systèmes embarqués pour l'IOT et l'automatisme.

Cette solution s'appelle **NodeRED**. On dispose d'un dashboard sur lequel on va pouvoir créer des interfaces graphiques et d'une interface d'administration pour la création de chaînes d'évènements appelés « flows ».

La programmation du site WEB est alors graphique et comprends front-end et back-end dans un seul package et un seul serveur. C'est donc pour moi qui suis complètement étranger au WEB un excellent moyen de réaliser les objectifs en un minimum de temps. La solution est alors robuste et simple à maintenir et à faire évoluer.

Avec **NodeRED**, on peut aisément agir sur des processus et fichiers du système. On peut donc appeler les utilitaires **timecircuitctl** et **fluxcapacitorctl** pour contrôler les périphériques de l'IHM.