Projet Systèmes Embarqués S9
Station Météo Connectée

Rule The World - Cloud Based Sensor Network

LAKHAL Nada
LLORCA Vivien
LOUBENS Matthieu
GERVAIS Clément
Option SE

Table des matières

1 Introduction

Le but de ce projet est la réalisation d'une plateforme IoT appliquée à la remontée d'informations issues d'un capteur de température et à sa visualisation sur une interface web.
Le schéma ci-dessous présente les différents matériels/technologies/protocoles impliqués dans la mise en œuvre de la plateforme :

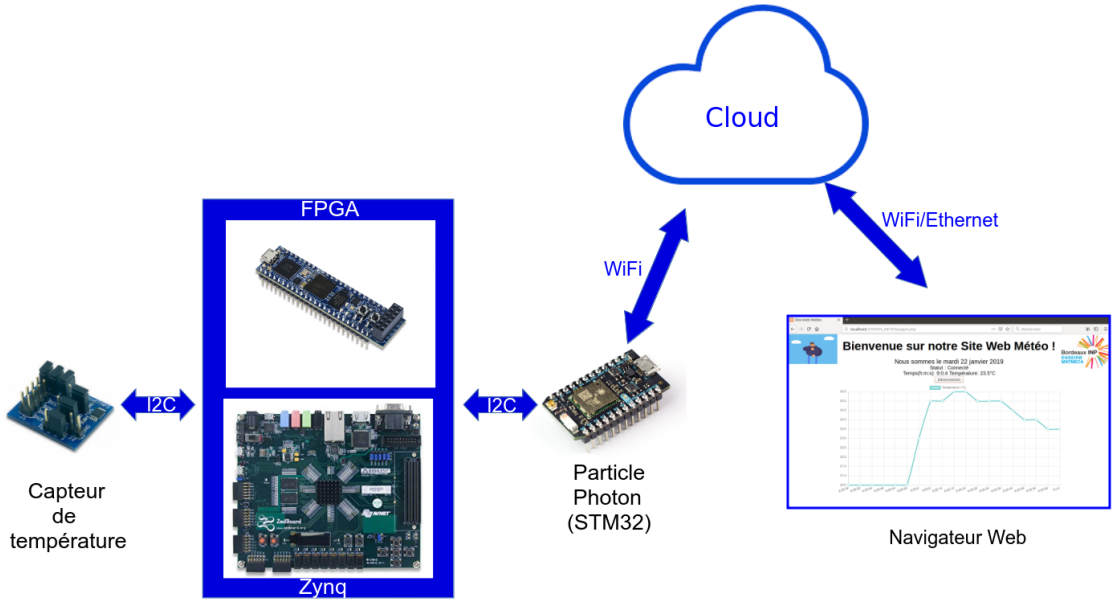


FIGURE 1

L'acquisition de données sera basée sur deux architectures différentes :
- Architecture FPGA
- Architecture Zynq

L'envoi de données vers une architecture de type Cloud sera réalisé à l'aide d'un module Particle Photon équipé notamment d'un contrôleur I2C, d'un micro-contrôleur et d'une puce WiFi.
Enfin la visualisation des données se fera via une interface Web.

Nous avons également cherché à minimiser la consommation d'énergie de notre plateforme et à la sécuriser en chiffrant les transmission de données.

Ce document fait état des développements effectués pour arriver à une plateforme fonctionnelle et aussi efficiente que possible.

2 Acquisition de données

2.1 Présentation du capteur de température

TCN75A (voir lien 1 en Annexe pour obtenir la datasheet du composant) sur FPGA (Cmod A7 Artix-7) via le protocole I2C :

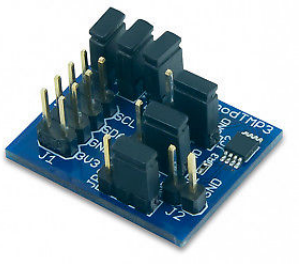


FIGURE 2

La donnée d'intérêt provient du capteur de température PMOD TMP3 de Digilent, conçu autour du chip TCN75A.

Ce composant possède les caractéristiques suivantes (liste non-exhaustive) :
- communication série I2C
- mesure configurable via un registre de configuration
- 2 registres de 8 bits pour coder la mesure

Nous allons commencer par présenter le protocole I2C permettant de communiquer avec ce capteur.

2.2 Protocole I2C

Le bus I2C permet de faire communiquer entre eux des composants électroniques très divers grâce à seulement trois fils. Le protocole du bus I2C définit la succession des états possibles sur les lignes SDA et SCL. En suivant est présenté la prise de contrôle du bus et la façon dont lire/écrire des données sur le bus I2C (les données sont transmises par octets) :

La figure ci-dessous représente une connexion entre un maître et un esclave suivant le protocole I2C :

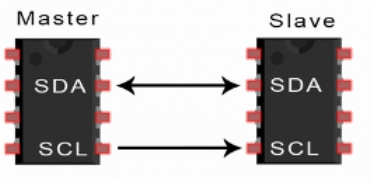


FIGURE 3: Connexion entre maître - esclave suivant le protocole I2C

C'est une connexion à deux fils pour transmettre les données :

— Signal de donnée - SDA (Serial Data) : Utilisé pour envoyer et recevoir les données
— Signal d'horloge – SCL (Serial Clock) : Porte le signal d'horloge

C'est une liaison en mode série permettant des échanges à la vitesse de 100 kbits par seconde. De plus c'est un processus synchrone dont l'horloge est controllée par le maître.

La figure ci-dessous présente le modèle des trames envoyées lors d'une communication entre le maître et l'esclave :

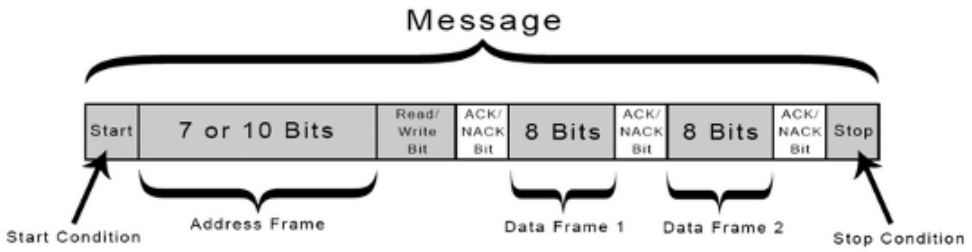


FIGURE 4: Trame d'un message du protocole I2C

Prise de contrôle du bus :

Pour transmettre les données il faut surveiller :

— la condition de départ : SDA passe à 0 et SCL reste à 1
— la condition d'arrêt : SDA passe à 1 et SCL reste à 0

Après avoir vérifié que le bus est libre, le circuit prend le contrôle du bus et génère le signal d'horloge.

Transmission d'un octet

Le maître transmet le bit de poids fort sur SDA et valide la donnée en appliquant le niveau haut sur SCL. Lorsque SCL retombe à '0', il poursuit avec le bit suivant jusqu'à ce que l'octet complet soit transmis.
Le mâitre envoie le bit ACK à '1' et l'esclave impose un niveau '0' à ce bit pour signaler que la transmission est finie et s'est déroulée correctement.

Transmission d'une adresse :

Chaque composant impliqué dans une liaison I2C possède une adresse unique qui le caractérise. Cette adresse est codée sur 7 bits et est transmise sous la forme d'un octet : les 7 bits d'adresses suivis du bit R/W.
Ce bit spécifie si le maître veut effectuer une lecture ou une écriture sur le bus I2C :

— R/W =à '0' : Ecriture
— R/W =à '1' : Lecture

Lecture d'une donnée :

Le maître envoie l'adresse puis attend le bit ACK positionné à '1' suivi des données émises sur SDA par l'esclave. La maître lit ces données et positionne ACK à '0' s'il souhaite continuer la lecture ou à '1' pour stopper la transmission.

2.3 Acquisition FPGA

Cette partie présente la mise en œuvre de la récupération des données du capteur de température TCN75A (voir lien 1 en Annexe pour obtenir la datasheet du composant) sur FPGA (Cmod A7 Artix-7) via le protocole I2C :

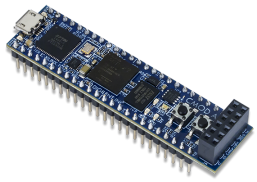


FIGURE 5: FPGA Cmod A7 Artix-7

Nous avons donc réalisé l'implémentation d'un module I2C en VHDL contrôlé par une FSM. L'ensemble de ces deux modules étant ensuite inclus dans un Top_Module. Nous allons maintenant décrire les modules I2C_MASTER et FSM permettant de récupérer la température du capteur TCN75a :

I2C_master :

Comme son nom l'indique ce module fonctionne donc comme « maître », c'est donc lui qui initie une communication avec un esclave ; ici notre capteur de température. Nous ne rentrerons pas ici dans le détail de fonctionnement du protocole I2C puisque celui-ci a été présenté dans la partie 2.2 et que ce module fonctionne donc logiquement de la même façon.

Le module se présente de la façon suivante :

FIGURE 6: Schéma du module I2C Master

On retrouve donc les signaux SDA (inout) et SCL (out) typiques de la communication I2C.

Voici la signification des autres signaux :

clock_in : représente le signal d'horloge cadencé à 12MHz du FPGA Cmod A7.

reset : permet de positionner SCL et SDA à 1 (en attente d'une condition de START) afin d'initier un dialogue avec un nouvel esclave.

mode : permet de choisir la vitesse de transmission du bus I2C. Le mode « 0 » permet d'utiliser la vitesse de transmission « Standard » (100 kHz) tandis que le mode « 1 » permet d'utiliser la vitesse de transmission « Rapide » (400 kHz). En fonction du mode choisi, un compteur permet d'obtenir cette fréquence en fonction de la fréquence du FPGA Cmod A7 (12 MHz). A rappeler que le bus étant synchrone, le maître impose donc l'horloge via la ligne SCL.

chip_enable : permet d'activer le compteur en charge du calcul de la fréquence d'utilisation du bus (100kHz ou 400kHz). Actif niveau bas.

R_or_W : permet de spécifier si une écriture ou une lecture doit être réalisée sur le bus :
- « 0 » : mode écriture
- « 1 » : mode lecture

data_in : correspond à la donnée qui doit être écrite sur le bus (SDA).

STOP : signal permettant de signaler que la communication entre le maître et l'esclave doit être interrompue.

data_out : correspond à la donnée lue sur le bus I2C (SDA).

ACK : signale la bonne émission ou la bonne réception d'une donnée (qui est codée sur 1 octet).

NACK : permet de signaler un défaut dans l'envoi (écriture) ou la réception (lecture) de la donnée (qui est codée sur 1 octet). Quand le maître est le récepteur, NACK est positionné à 1 pour interrompre le dialogue, avant d'envoyer la condition d'arrêt de la communication.

FSM :

Afin de contrôler le module I2C_MASTER, nous avons donc mis en œuvre la FSM suivante. La FSM dans sa version terminée possède davantage de signaux. Pour cette partie, nous nous contentons simplement de présenter les signaux utiles à la communication I2C. Au fur et à mesure du rapport, nous rajouterons les autres signaux que nous avons utilisés.

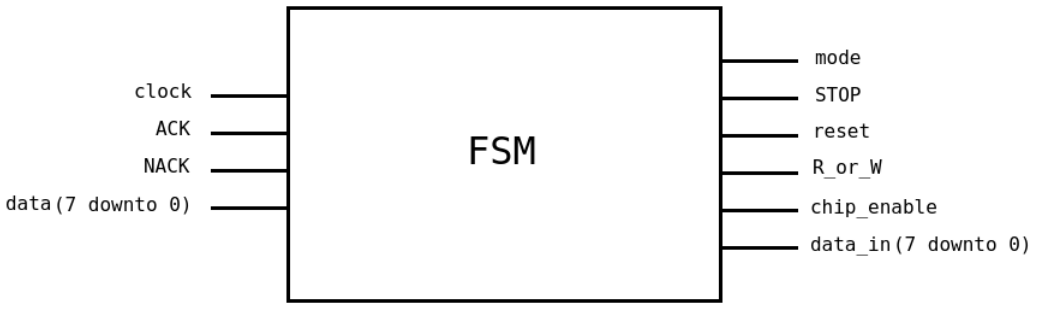


FIGURE 7: Schéma de la FSM

Les signaux utilisés sont donc tous liés au module I2C_MASTER présenté précédemment (hormis clock et clock_in qui représentent l'horloge du FPGA Cmod A7). Nous n'allons donc pas les présenter de nouveau. A noter que le signal data correspond au signal data_out de l'I2C_MASTER.

Au final, on peut représenter l'interaction entre les deux modules par l'intermédiaire de la figure suivante :

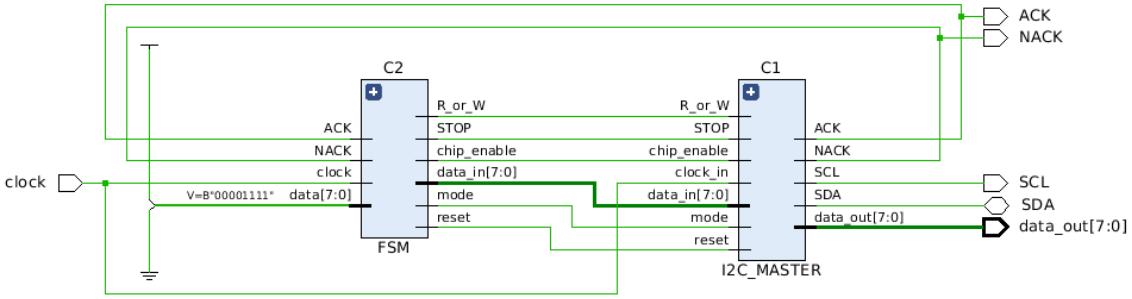


FIGURE 8: Interconnexion des blocs I2C Master et FSM

Nous avons donc contrôlé le module I2C_MASTER en fonction des signaux ACK et NACK par l'intermédiaire des signaux mode, STOP, reset, R_or_W, chip_enable et data_in de la FSM. En effet, les signaux ACK et NACK permettant de vérifier la bonne lecture ou la bonne écriture d'un octet sur SDA ; ils nous ont donc permis d'orchestrer la récupération de la donnée de température issue du capteur TCN75a. En effet, d'après la figure suivante issue de la datasheet du TCN75a, la récupération de la donnée de température se réalise en 5 étapes :

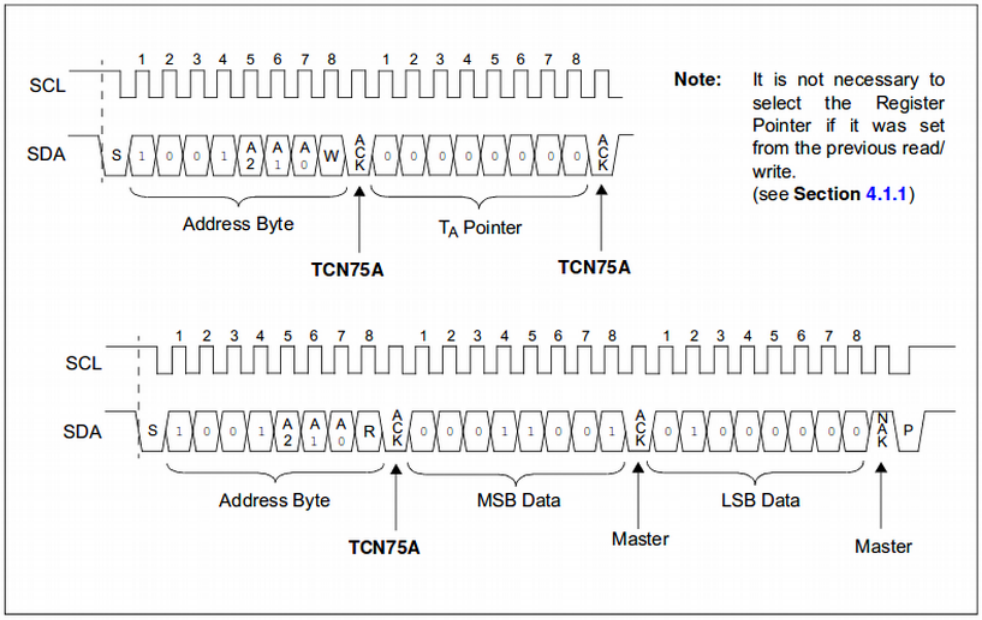


FIGURE 9: Diagramme présentant les écritures et/ou lectures à réaliser pour lire la température stockée dans le registre TA (issu de la datasheet du TCN75a)

Nous avons donc mis en œuvre ces 5 étapes dans notre FSM :

1ère étape : Envoi de l'adresse 0x48 sur SDA. Cette adresse permet de communiquer avec le capteur de température. Dans le premier état de notre FSM nous avons donc :

STOP <= '0' ;
mode <= '0' ;
reset <= '0' ;
R_or_W <= '0' ;
chip_enable <= '0' ;

Nous avons ainsi choisi de communiquer avec le capteur de température à une vitesse de 100kHz (d'où mode = '0') et réaliser un reset. Nous avons ensuite spécifié que nous voulions écrire sur le bus I2C d'où R_or_W = '0'.
A noter que sous forme binaire, l'octet à envoyer au capteur de température est :

data_in <= "10010000"

Les 7 premiers bits correspondant bien à l'adresse 0x48 du TCN75a ; le dernier bit étant positionné à 0 pour stipuler une écriture.

2ème étape : La seconde étape consiste donc à écrire sur le bus I2C l'adresse du registre TA afin d'obtenir la valeur de la température. Mais avant de réaliser cela, nous réalisons un test sur le signal ACK afin de savoir si celui-ci est à 1 ; autrement dit pour savoir si l'étape 1 a bien été réalisée et que le TCN75a a acquitté la réception de l'octet 0x90 (10010000). Lorsque ACK est positionné, on écrit alors l'adresse du registre TA sur le bus soit 0x00. Sous forme binaire, l'octet à envoyer au capteur de température est :

data_in <="00000000"

3ème étape : La troisième étape consiste cette fois-ci à l'envoi sur le bus I2C de l'adresse 0x48 afin de lire la valeur de la température. Mais avant de réaliser cela, nous réalisons toujours un test sur le signal ACK afin de savoir si celui-ci est à 1 ; autrement dit pour savoir si l'étape 2 a bien été réalisée et que le TCN75a a acquitté la réception de l'octet 0x00 (00000000). Lorsque ACK est positionné, on envoie alors l'adresse 0x48. A noter que sous forme binaire, l'octet à envoyer au capteur de température est :

data_in <="10010001"

Les 7 premiers bits correspondant bien à l'adresse 0x48 du TCN75a ; le dernier bit étant positionné à 1 pour stipuler une lecture.
Étant donné que nous voulons réaliser une lecture via le bus I2C, nous avons donc spécifié via la FSM :

R_or_W <= '1' ;

4ème et 5ème étape : Les quatrièmes et cinquièmes étapes sont identiques. En effet, nous réalisons d'abord la lecture de l'octet de poids fort (MSB Data) qui correspond à la partie entière de la température puis à la lecture de l'octet de poids faible (LSB Data) qui correspond à la partie décimale de la température (codée sur 8 bits et qui doit être divisée par 256 pour obtenir la partie décimale réelle).

Mais comme auparavant, avant de passer à l'étape 4, nous réalisons toujours un test sur le signal ACK afin de savoir si celui-ci est à 1 ; autrement dit pour savoir si l'étape 3 a bien été réalisée et que le TCN75a a acquitté la réception de l'octet 0x91 (10010001). Par la suite, lorsque l'ACK est positionné, on lit alors l'octet « MSB data » et c'est notre I2C_MASTER qui acquitte cette fois la bonne lecture (ACK='1'). Nous continuons et procédons ensuite à la lecture de l'octet « LSB data ». Pendant ces deux étapes, nous avons donc toujours :

R_or_W <= '1' ;

Afin de spécifier au capteur de température la fin de la transmission, le signal STOP est passé à 1 :

STOP <= '1' ;

Cela déclenche ainsi l'envoi d'un NACK sur le SDA, ce qui implique le passage du signal NACK à 1 pour l'I2C_MASTER :

NACK <= '1' ;

Nous passons ensuite les signaux suivants à 0 ; en attente d'un nouveau dialogue avec un esclave :

```
STOP <= '0';
mode <= '0';
reset <= '0';
R_or_W <= '0';
```

Ainsi, la valeur de la température est récupérée sous forme de deux octets (MSB data et LSB data). Dans notre code VHDL, ces deux valeurs sont stockées dans deux variables data_MSB et data_LSB en attente du cryptage que nous évoquerons dans la suite. A noter que l'implémentation VHDL de notre module a d'abord été simulée avant d'être validée expérimentalement.

2.4 Acquisition Zynq

Pour la deuxième méthode d'acquisition nous avons choisi d'utiliser une carte de développement Zedboard développée par Digilent.

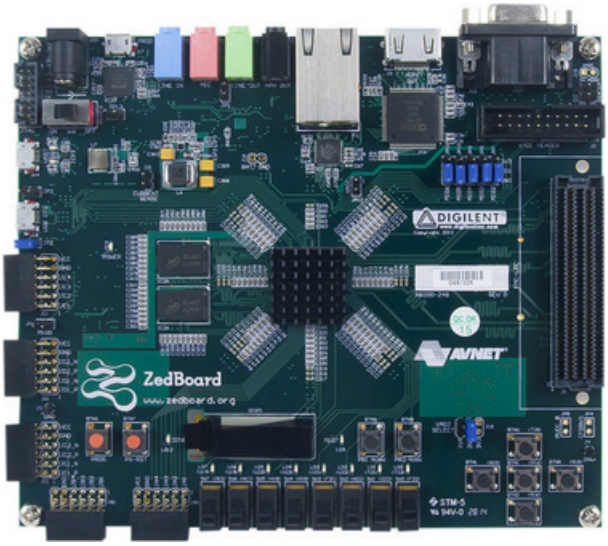


FIGURE 10: Carte de développement Zedboard

La Zedboard est équipée d'un SoC programmable Zynq-7000 constitué d'un processur dual-core ARM Cortex A9 et d'une matrice FPGA Artix-7 ainsi que de multiples périphériques et contrôleurs. Nous allons utiliser l'intelligence offerte par le processeur afin de faciliter notre développement (ou tout du moins d'exporter la complexité matérielle vers une complexité logicielle) tout en essayant de limiter la consommation d'énergie. Nous utiliserons également les capacités de la matrice FPGA afin d'intégrer une IP de chiffrement (voir partie 3.2.3).

2.4.1 Création du design sous Vivado

Vivado est un logicel développé par Xilinx qui permet notamment la configuration d'un système complexe comme le Zynq. Nous pouvons en particulier configurer le Processing System (PS) et la Programmable Logic (PL) à l'aide d'un block design dans lequel nous ajoutons des IPs et les interconnectons.

Création d'un block design :

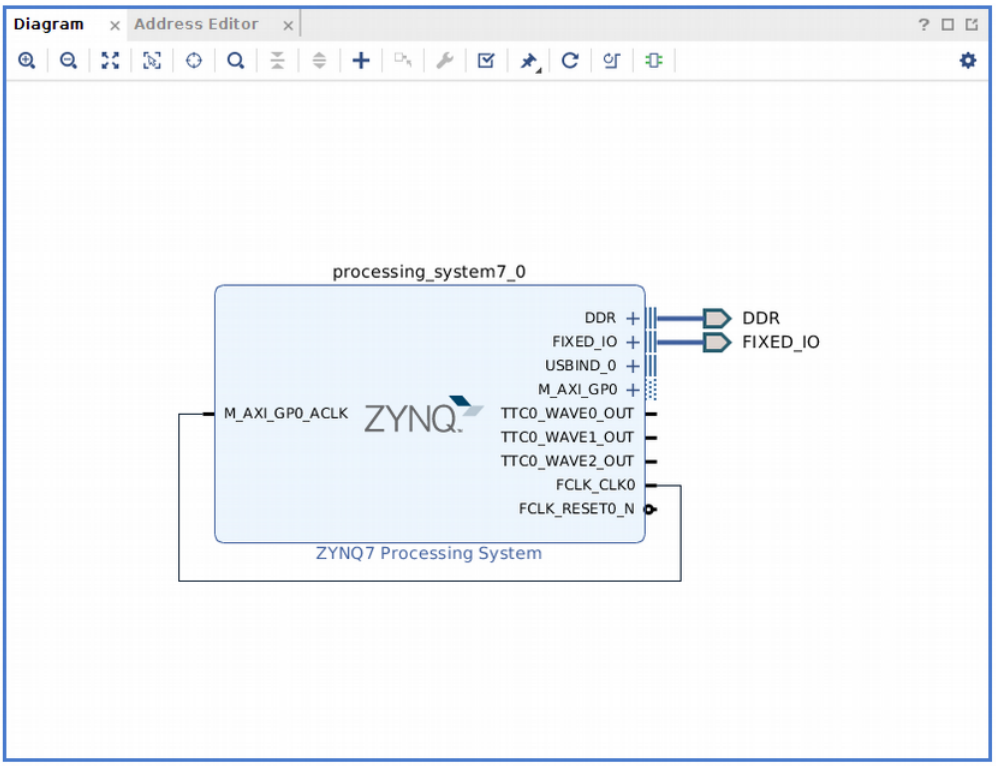


FIGURE 11: Block design composé de l'IP Zynq

Nous utilisons dans un premier temps uniquement la partie PS du SoC Zynq. Le SoC Zynq-7000 possède un contrôleur I2C directement accessible par le PS et par le périphérique d'entrée/sortie multiplexé (MIO) comme nous pouvons le voir sur la figure suivante :

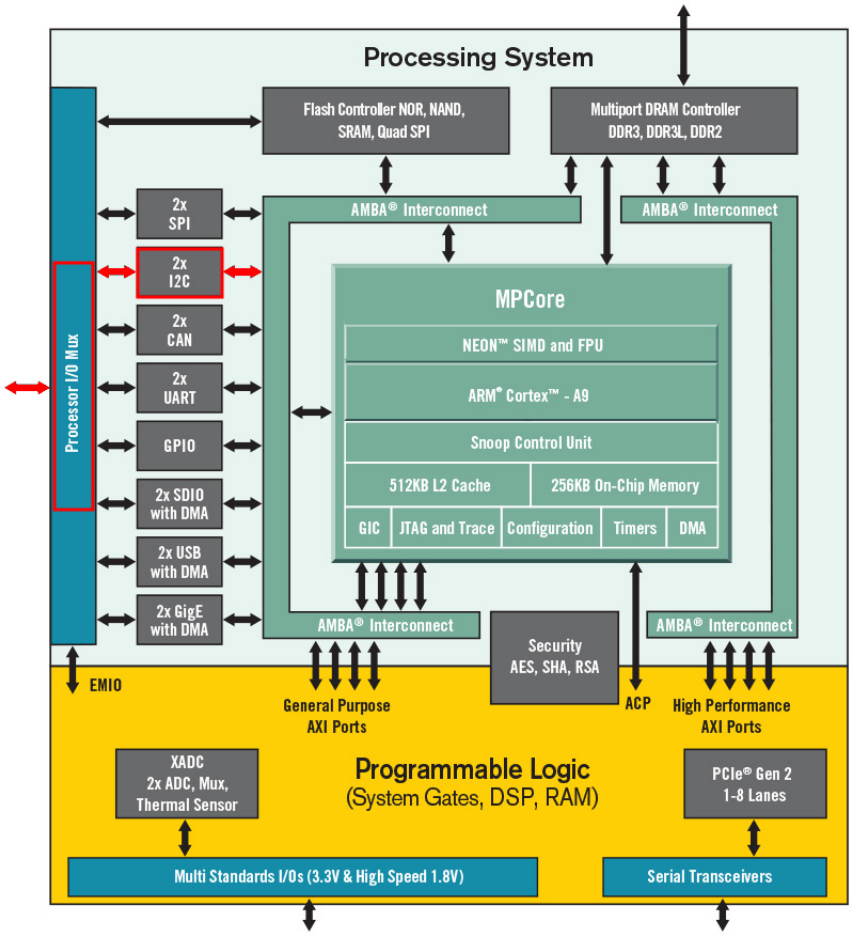


FIGURE 12: Architecture du SoC Zynq-7000 ARM/FPGA

Nous devons l'activer et préciser à quels ports d'entrée/sortie le connecter (fonctionnalité de multiplexage) :

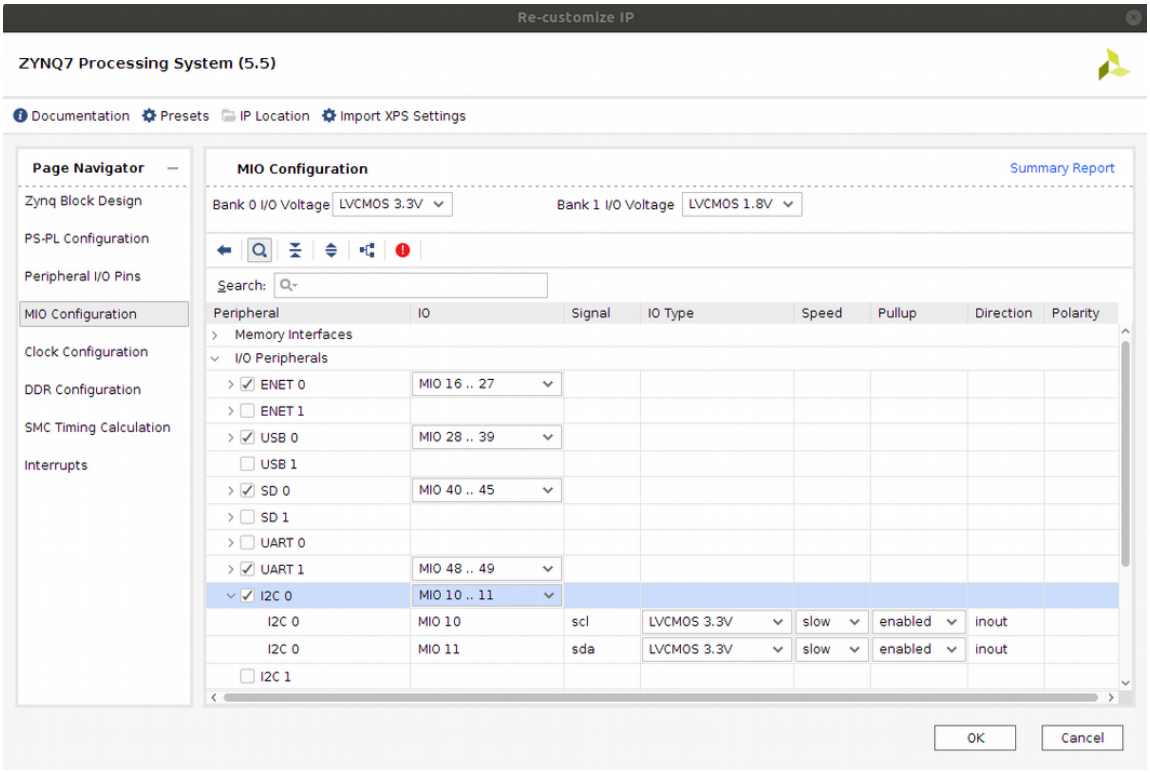


FIGURE 13: Page de configuration du PS

Nous choisissons les entrées MIO 10-11 qui sont respectivement connectées aux pins JE2 et JE3 du connecteur Pmod JE sur la Zedboard :

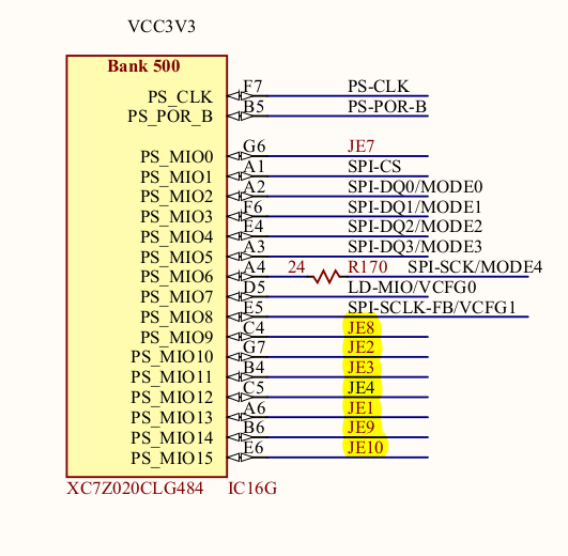


FIGURE 14: Schéma d'interconnexion des MIO avec les périphériques de la Zedboard

Nous activons également les résistances de PULL-UP sur ces pins, configuration nécessaire au bon fonctionnement de l'I2C. Nous synthétisons et implémentons le design et générons le bitstream. Nous exportons ensuite la configuration matérielle et lançons le Software Development Kit de Xilinx (XSDK) permettant la programmation du PS du Zynq.

2.4.2 Programmation du Processing System (PS) sous XSDK

Nous créons une application C en standalone visant le CPU 0 et générons un Board Support Package (BSP) associé à la configuration matérielle. Le BSP est une couche logicielle contenant des drivers et fonctions propres à la configuration matérielle pour laquelle il a été généré. En particulier, les fichiers xiicps_master.c/xiicps.h contiennent des fonctions permettant de piloter le contrôleur I2C (liste non-exhaustive) :

```
1 XiicPs_LookupConfig()      /* récupère la configuration par défaut. */
2 XIicPs_CfgInitialize()     /* initialise l'instance I2C avec la configuration précé
      demment récupérée */
3 XIicPs_SetSClk()           /* sélection de la vitesse de transmission du bus I2C */
4 XIicPs_SelfTest()          /* test si le contrôleur I2C a été correctement configuré
      */
5 XIicPs_BusIsBusy()         /* vérifie si le bus I2C est utilisé */
6 XIicPs_MasterSendPolled() /* envoie des données dans la FIFO du contrôleur I2C et
      attend que le périphérique esclave les récupère */
7 XIicPs_MasterRecvPolled() /* se met en attente de données à lire dans le registre de
       données. */
```

Nous avons créé une librairie i2c_lib.h/c permettant d'utiliser plus aisément ces fonctions :

```
1 /* configuration du contrôleur I2C avec une vitesse de transmission de 100 kbit/s.
      */
2 i2c_config( ... );
3 /* lecture de nbBytes octets à  l adresse  regAddr du périphérique I2C  d adresse
      deviceAddr. Le résultat est stocké dans bufOut. */
4 i2c_read( ..., u8 deviceAddr, u8 regAddr, u8* bufOut, u32 nbBytes );
5 /* écriture des nbBytes octets du buffer bufIn dans le registre regAddr du périphé
      rique I2C  d adresse  deviceAddr. */
6 i2c_write( ..., u8 deviceAddr, u8 regAddr, u8* bufIn, u32 nbBytes );
```

Le but est de récupérer la valeur de la température mesurée par le capteur PMOD TMP3 de Digilent et basé sur le chip TCN75A de Microchip Technology.
Le TCN75A comporte des registres utilisateurs programmables permettant entre autres de régler la résolution de la mesure (de 0,5°C à 0.0625°C) ainsi que le mode de mesure (continu, « one-shot », etc.).
Dans le but de limiter la consommation d'énergie, nous avons choisi le mode de mesure « one-shot » qui consiste à endormir le composant et à le réveiller à chaque mesure. Ceci permet de diviser par 100 la consommation de courant (200A en mode mesure continu contre 2A lorsque le composant est endormi).

L'étude de la datasheet du TCN75A nous a permis de déterminer les adresses des registres dans lesquels lire/écrire afin de réaliser la mesure ou la configuration de la température :

REGISTER 5-1: REGISTER POINTER

U-0	U-0	U-0	U-0	U-0	U-0	R/W-0	R/W-0
0	0	0	0	0	0	P1	P0
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 7-2 **Unimplemented:** Read as '0'

bit 1-0 **Pointer** bits

00 = Temperature register (T_A)
01 = Configuration register (CONFIG)
10 = Temperature Hysteresis register (T_{HYST})
11 = Temperature Limit-set register (T_{SET})

FIGURE 15: Adresses des registres du TCN75A

Nous utiliserons ainsi 2 registres du TCN75A pour notre application :

Le registre de configuration d'adresse 0b00000001 :

REGISTER 5-3: CONFIGURATION REGISTER (CONFIG) — ADDRESS <0000 0001>b

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
One-Shot	Resolution		Fault Queue		ALERT Polarity	COMP/INT	Shutdown
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 7 **ONE-SHOT** bit
1 = Enabled
0 = Disabled (Power-up default)

bit 6-5 ΣΔ **ADC RESOLUTION** bits
00 = 9 bit or 0.5°C (Power-up default)
01 = 10 bit or 0.25°C
10 = 11 bit or 0.125°C
11 = 12 bit or 0.0625°C

bit 4-3 **FAULT QUEUE** bits
00 = 1 (Power-up default)
01 = 2
10 = 4
11 = 6

bit 2 **ALERT POLARITY** bit
1 = Active-high
0 = Active-low (Power-up default)

bit 1 **COMP/INT** bit
1 = Interrupt mode
0 = Comparator mode (Power-up default)

bit 0 **SHUTDOWN** bit
1 = Enable
0 = Disable (Power-up default)

FIGURE 16: Registre de configuration du TCN75A

Afin de configurer la mesure en mode « one-shot », la datasheet nous précise qu'il faut dans un premier temps que le capteur soit en mode « shutdown ». Nous écrirons donc les 2 octets successifs suivants dans le registre de configuration :

— 0b00000001 : configuration en mode « shutdown »
— 0b11100000 : configuration en mode « one-shot » et résolution de la mesure sur 12 bits

Une fois notre capteur configuré, nous pouvons lire la valeur de la température.

Le registre de température ambiante d'adresse 0b00000000 :

REGISTER 5-2: AMBIENT TEMPERATURE REGISTER (T_A) — ADDRESS <0000 0000>b

Upper Half:							
R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0
Sign	2^6 °C	2^5 °C	2^4 °C	2^3 °C	2^2 °C	2^1 °C	2^0 °C
bit 15							bit 8

Lower Half:							
R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0
2^{-1} °C/bit	2^{-2} °C	2^{-3} °C	2^{-4} °C	0	0	0	0
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

FIGURE 17: Resgitres de mesure du TCN75A

Ce registre est composé de 2 octets, le premier codant la partie entière de la mesure de température, le deuxième la partie décimale.

Nous avons développé une deuxième librairie tempSensor_lib.h/c, spécifique au capteur de température, et définissant les fonctions suivantes :

```
1 temp_configMeasure( ..., u8 resolution ) /* utilise la fonction i2c_write() de la
      librairie i2c_lib.h/c afin d écrire dans le registre de configuration. */
2 temp_measure( ..., u8 *buffer ) /* simple wrapper de la fonction i2c_read() de la
      librairie i2c_lib.h/c permettant la lecture des 2 octets du registre de tempé
      rature ambiante et la stocke dans buffer. */
```

Nous avons ensuite écrit notre premier programme, réalisant 10 mesures et affichant le résultat sur l'UART de debug. En voici une version simplifiée :

```
int main(void)
{
    int count = 0;
    float temp;
    int whole, thousandths;
    u8 RecvBuffer[2];       /* Reception buffer */

    /********************** I2C0 configuration ***************************/

     i2c_config(IIC0_DEVICE_ID, &Iic0);

    /********************** TMP3 measure configuration *******************/

    temp_configMeasure( &Iic0, TEMP_RESOLUTION_12BITS );

    /********************** 10 measures then exit ************************/
    while( count++ < 10 ) {
        temp_measure( &Iic0, RecvBuffer );

        temp = (float)RecvBuffer[0] + (float)(RecvBuffer[1]/256.0);
        whole = temp;
        thousandths = (temp - whole) * 1000;
        xil_printf( "%d %d\r\n", RecvBuffer[0], RecvBuffer[1] );
        xil_printf("Temperature : %d.%03d *C\r\n", whole, thousandths);

        usleep(2000000);
    }

    return XST_SUCCESS;
}
```

En se connectant à l'UART nous constatons que la mesure est réalisée avec succès et avec la résolution maximale :

```
I2C 0 Configuration
Temperature register: Upper Half: 1C Lower Half: 00
Temperature : 28.000 *C
Temperature register: Upper Half: 1B Lower Half: 90
Temperature : 27.562 *C
Temperature register: Upper Half: 1B Lower Half: 40
Temperature : 27.250 *C
Temperature register: Upper Half: 1A Lower Half: A0
Temperature : 26.625 *C
Temperature register: Upper Half: 1A Lower Half: 10
Temperature : 26.062 *C
Temperature register: Upper Half: 19 Lower Half: B0
Temperature : 25.687 *C
Temperature register: Upper Half: 19 Lower Half: 90
Temperature : 25.562 *C
Temperature register: Upper Half: 19 Lower Half: 90
Temperature : 25.562 *C
Temperature register: Upper Half: 19 Lower Half: 80
Temperature : 25.500 *C
Temperature register: Upper Half: 19 Lower Half: 80
Temperature : 25.500 *C
Data acquisition ends
```

FIGURE 18: Résultats de la mesure affichés sur la liaison UART

3 Transmission des données et chiffrement

Maintenant que nous sommes capables de récupérer notre donnée de température sur architectures FPGA et Zynq, nous allons voir dans un premier temps comment transmettre cette donnée à la carte Particle Photon.
Ensuite nous mettrons en oeuvre le chiffrement de cette donnée avant de pouvoir enfin la transmettre au Cloud.

3.1 Transmission des données au Photon

3.1.1 Présentation du Particle Photon

L'essentiel du Particle Photon

Le module Photon de Particle est une carte de développement IoT WiFi miniature basée sur un microcontrôleur ARM Cortex M3 à 120 MHz et sur un module WiFi BCM43362.

Le Photon est idéal pour des projets connectés et se programme facilement en utilisant le même langage que les cartes Arduino. Il comporte 18 E/S digitales dont 9 PWM, 8 entrées analogiques, 2 sorties analogiques et embarque des interfaces SPI et I2C comme en témoigne la figure suivante :

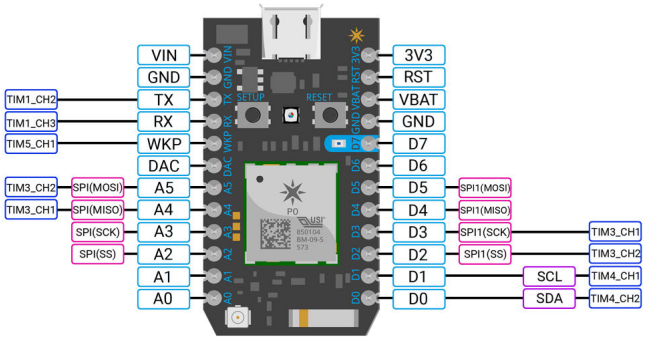


FIGURE 19: Présentation de la carte Particle Photon

Le module Wi-Fi du Photon permet de le connecter sur un réseau local de la même façon qu'un téléphone ou un ordinateur. Le Photon est programmé pour rester connecté sur Internet (pour autant qu'il puisse se connecter sur le réseau sans fil ciblé). Lorsqu'il est connecté sur Internet, il établit une connexion avec le Cloud Particle de particle.io. En se connectant sur le Cloud, le Photon devient accessible depuis n'importe quel endroit grâce à sa simple Particle Device Cloud API.

L'accès au Cloud Particle et à ses services est gratuit. Il dispose de nombreuses fonctionnalités pour réaliser des objets/projets connectés, permettant notamment :

- La création d'applications à l'aide de l'environnement de développement en ligne (Web IDE).
- La mise-à-jour/programmation du Firmware via la connexion WiFi.

Dans le cadre de notre projet, nous avons donc utilisé l'interface Web IDE proposée pour réaliser nos applications bien que d'autres options soient disponibles (Particle IDE, Particle CLI notamment). Nous avons pour cela créé un compte utilisateur en ligne afin de pouvoir programmer notre Photon et bien évidemment y avoir accès partout dans le monde.

Boutons et Leds du Particle Photon :

Il est important de spécifier le rôle et la signification des boutons et des leds du Particle Photon. En effet, ils ont un rôle important et permettent notamment de connecter ou de vérifier la bonne connexion à Internet.

Les boutons :

Le Photon possède deux boutons :
- Un bouton **RESET** de réinitialisation.
- Un bouton **SETUP** de configuration.

Voici les rôles et principales manipulations pouvant être réalisés à l'aide de ces boutons :

Le bouton RESET permet de faire une réinitialisation matérielle du Photon (« hard reset »), ce qui reboot le microcontrôleur. Cela peut permettre le redémarrage d'une application ayant précédemment été téléversée sur le Photon.

Le bouton SETUP est utilisé pour deux fonctions différentes après le reboot du Photon :
- Maintenir le bouton SETUP enfoncé pendant 3 secondes place le photon en mode Soft-AP (point d'accès logiciel) pour le connecter sur un réseau Wi-Fi. La led RGB doit alors commencer à clignoter en bleu.
- Maintenir le bouton SETUP enfoncé pendant 10 secondes efface les réseaux WiFi enregistrés dans la mémoire du Photon.

Le bouton SETUP a également d'autres fonctions avant le reboot du Photon parmi lesquelles :
- Maintenir le bouton SETUP enfoncé, presser brièvement le bouton RESET et attendre moins de 3 secondes permet d'entrer dans le mode User Safe Mode (mode où la led RGB clignote en magenta). C'est le mode le plus fréquent, parce qu'il n'exécute pas d'application. Ce mode peut être utile lorsqu'un bug d'une application empêche de re-flasher le Photon via la connexion WiFi.
- Maintenir le bouton SETUP enfoncé, presser brièvement le bouton RESET et attendre de 3 à 6 secondes permet d'entrer en mode Bootloader (mode où la led RGB clignote en jaune). Dans ce mode, il est possible de reprogrammer le Photon via USB ou JTAG.

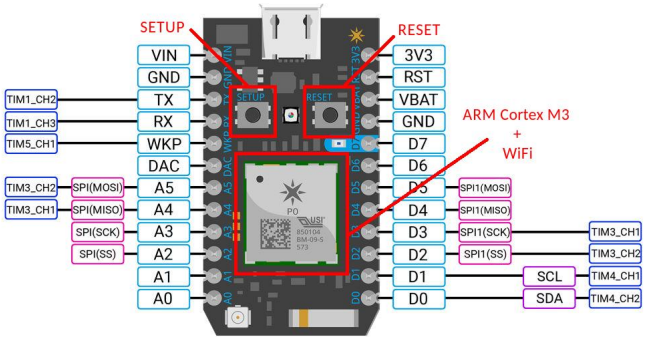


FIGURE 20: Boutons de la carte Particle Photon

Les leds :

Il y a deux leds sur le Photon. Une grosse led RGB au milieu de la carte indique l'état de la connexion Internet. L'autre est une led utilisateur connectée sur la broche D7. La led RGB peut présenter les états suivants :

Couleur	Description
Clignote en bleu	Mode Soft-AP (Point d'accès logiciel) et en attente des informations réseau.
Bleu fixe	Configuration Soft-AP complète, informations réseau disponibles.
Clignote en vert	Connexion au réseau WiFi local.
Clignote en cyan	Connexion sur le Cloud.
Flash rapide en cyan	Synchronisation Cloud (handshake).
Pulsation lente en cyan	Connecté avec succès au Cloud.
Clignote en jaune	En mode Bootloader, attente d'un nouveau code/Firmware via USB ou JTAG.
Pulsation en blanc	Démarrage, le Photon est mis sous tension ou réinitialisé.
Clignote en blanc	Réinitialisation d'usine entamé.
Blanc fixe	Réinitialisation d'usine terminé, reboot.
Clignote en magenta	Mise-à-jour du Firmware ou entre en mode « Réinitialisation d'usine sécurisé » (Factory Reset Safe).
Magenta fixe	Possibilité d'avoir perdu la connexion avec le Cloud. Presser le bouton (RESET) permet de réaliser un nouvel essai de mise-à-jour.

FIGURE 21: Tableau récapitulatif de l'état du Photon en fonction des leds

La led RGB permet également de savoir si il y a une erreur lors de l'établissement de la connexion Internet. Si la led RGB est rouge, elle signale qu'une erreur est survenue. Ces erreurs peuvent par exemple inclure l'utilisation invalide d'un pointeur ou un stack overflow.

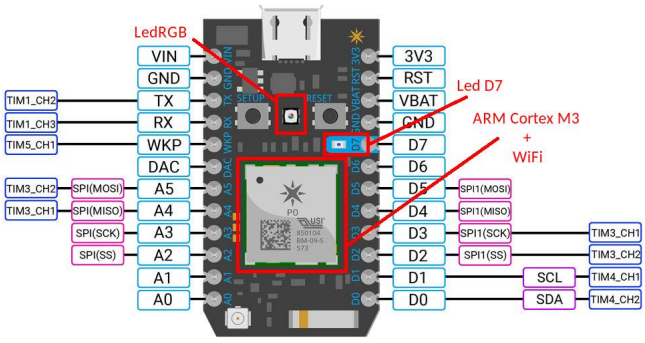


FIGURE 22: Leds de la carte Particle Photon

Particle Device Cloud API (Spark Cloud API)

Cette sous-partie a pour but de présenter l'API Particle Photon qui permet notamment de partager des données issues du Particle Photon avec le monde extérieur.

L'API du Particle Photon est une interface de programmation REST (« Representational State Transfer »). REST signifie beaucoup de choses (et nous ne rentrerons pas dans les détails ici, de plus amples informations sont disponibles via le très bon article lien 2 de l'Annexe), mais elle signifie premièrement et principalement qu'une URL est utilisée dans le sens premier de son intérêt ; c'est à dire comme une Localisation Uniforme de Ressource (« Uniform Resource Locator »).

Dans notre cas, la « Ressource » unique en question est le Particle Photon. En effet, chaque Particle Photon possède une URL, ce qui signifie qu'elle peut être utilisée avec GET pour obtenir une variable, POST pour appeler une fonction ou PUT pour placer un nouveau Firmware. Les variables et les fonctions définies dans notre code (Firmware) sont alors exposées comme des sous-ressources dans le Particle Photon. Toutes les requêtes à destination du Particle Photon sont réalisées par l'intermédiaire des Serveurs d'API de Particle en utilisant la sécurité TLS (« Transport Layer Security »).

On peut donc schématiser l'interaction entre un Client et le Particle Photon via l'intermédiaire d'un Serveur par le schéma suivant :

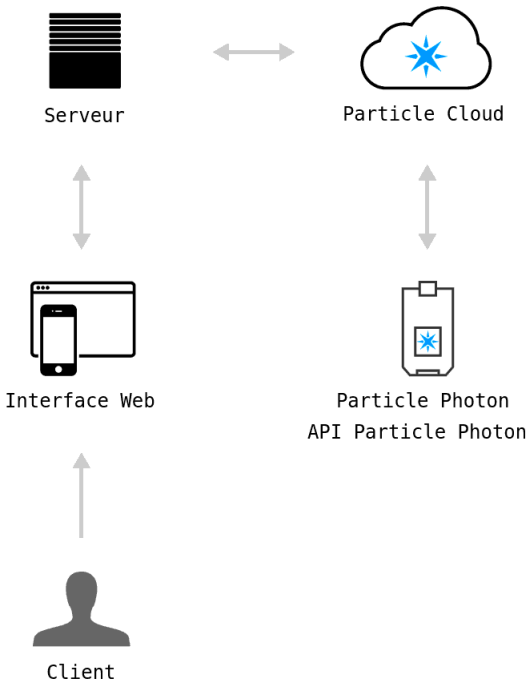


FIGURE 23: Schéma de l'intéraction Client/Particle Photon

Nous avons choisi cet exemple car il représente le cas de figure de notre projet IoT. Nous présenterons plus en détail dans la partie « 4- Affichage des données et déchiffrement : Serveur » comment s'orchestre l'interaction entre le Client et le Particle Photon afin de récupérer des données notamment.

Exemples d'appels API disponibles, rôle du Device ID et de l'Access Token

Il existe un certain nombre d'appels API disponibles. Pour obtenir les informations de base à propos d'un Photon par exemple (en incluant les variables et fonctions exposées), il est nécessaire de réaliser :

```
1  \$ GET https://api.spark.io /v1/devices/{DEVICE\_ID}?access\_token={ACCESS\_TOKEN} \
       $
```

Nous remarquons que l'URL associée au Particle Photon est composée d'un Device ID et d'un Access Token. Mais à quoi correspondent ces éléments ?

Le Device ID est une valeur permettant d'identifier le Particle Photon ; c'est un peu comme un numéro de série. En effet, en enregistrant le Particle Photon sur le Particle Cloud, nous lui donnons un nom. Dans notre cas, nous l'avons nommé PhotonWeb. Cependant, les identifiants tels que 0123456789abcdef... sont préférés pour éviter toutes les tentatives d'identifications et sont également plus faciles à stocker (dans un fichier ou une base de donnée). Le Device ID est obtenu dans le Web IDE (build) de Particle. Il suffit de se connecter à son compte Particle Cloud pour l'obtenir :

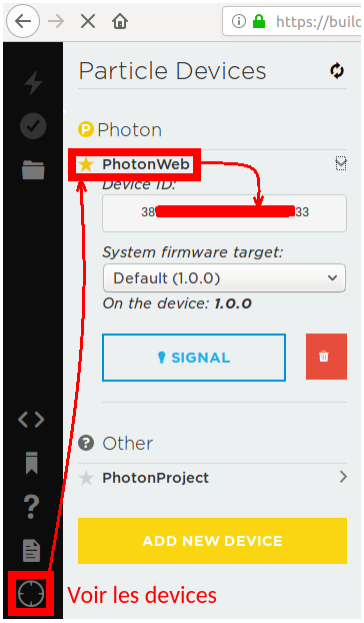


FIGURE 24: Device ID du Particle Photon

Un Access Token (ou Jeton d'Accès en Français) est un code spécial qui est lié à un Client ou à un utilisateur d'un Particle Photon. Il permet de lire les données et d'envoyer des commandes au Particle Photon de cette personne. Un paramètre API qui retourne des informations privées, ou permet le contrôle de l'appareil d'une personne nécessite un jeton d'accès. L'Access Token peut être considéré comme une clé de sécurité. Pour cette raison, l'Access Token ne doit pas être communiqué au monde extérieur ! En effet, des personnes mal intentionnées pourraient de cette façon prendre le contrôle du Particle Photon en question !

Comme pour le Device ID, l'Access Token est lié au compte Particle Cloud du propriétaire du Photon. Il faut donc se connecter à son compte Particle Cloud pour y avoir accès :

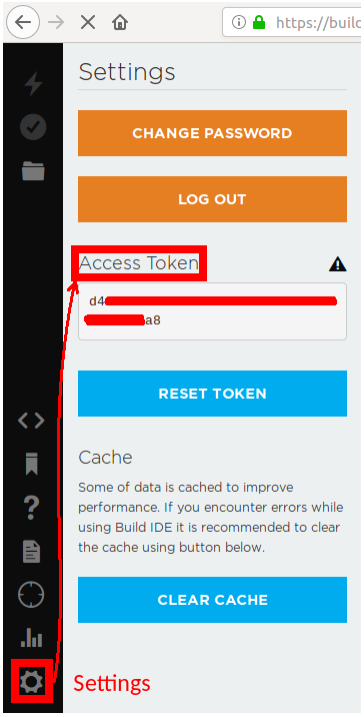


FIGURE 25: Access Token du Particle Photon

Variables et fonctions :

Depuis le Particle Photon, il est possible de déclarer des fonctions ou des variables pouvant être ensuite appelées par le Client.

Spark.function() :

« Spark.function() » permet d'exposer une fonction dans le Cloud afin qu'elle puisse être appelée avec POST. A l'aide de Curl par exemple on peut notamment réaliser un POST :

```
curl https://api.spark.io /v1/devices/{DEVICE_ID}?/NAME_FUNCTION \-d access_token={
    ACCESS_TOKEN} \ -d "args=etat"
```

Dans l'URL ci-dessus, NAME_FUNCTION correspond au paramètre API pour identifier la fonction. La déclaration dans le code du Web IDE serait donc :

```
int value = 0;

int function(String etat) {
    value = etat.toInt();
    return 0;
}

Spark.function("NAME_FUNCTION", function);
```

Ainsi, la fonction « function » est associée au paramètre API « NAME_FUNCTION ». Ici, la fonction « function » prend en paramètre une String « etat ». Ce paramètre doit donc être passé en argument du POST d'où "args=etat". Lorsque le POST a lieu, si « etat » vaut 1, la valeur de « etat » sera associé à « value » par l'intermédiaire de la fonction « function ».

Pour que l'API puisse appeler la fonction sur le Particle Photon, il faut inclure l'Access Token valide pour l'utilisateur qui est le propriétaire de l'appareil. De cette façon, l'API consulte le jeton d'accès dans la base de données de Particle ; vérifie que l'utilisateur qui a créé le jeton d'accès est le propriétaire du périphérique inclus dans la demande, et ne continuera que si le jeton a les autorisations appropriées pour appeler la fonction.

Spark.variable() :

« Spark.variable() » permet d'exposer une variable dans le Cloud pour qu'elle puisse être appelée avec GET :

```
GET https://api.spark.io /v1/devices/{DEVICE_ID}/NAME_VARIABLE?access_token={
    ACCESS_TOKEN}
```

Elle retourne une valeur de succès « vrai » lorsque la variable a été enregistrée. Dans l'URL ci-dessus, NAME_VARIABLE correspond au paramètre API pour identifier la variable. La déclaration dans le code du Web IDE serait donc :

```
int value = 10;
Particle.variable("NAME_VARIABLE", &value, INT);
```

Ici on associe à la valeur « value » de type int, le paramètre API « NAME_VARIABLE ». Ainsi, lorsque le Client souhaite récupérer la valeur de « value », il ne réalisera pas de GET directement sur le nom de la valeur, mais sur le paramètre API qui lui est associé.

La procédure de vérification est la même que pour Spark.function() concernant l'Access Token.

Il existe d'autre fonctions Cloud proposées par Particle (Spark.publish(), Spark.subscribe()...). Néanmoins, nous ne les présenterons pas ici, seules les fonctions Spark.variable() et Spark.function() nous ont été utiles dans notre projet.

Format des Demandes/Réponses :

L'API Particle Photon accepte les demandes en JSON (content type application/json) et dans un format codé (content type application/x-www-form-urlencoded). Il répond toujours en JSON (content type application/json).

Pour information, JavaScript Object Notation (JSON) est un format de données textuelles dérivé de la notation des objets du langage Javascript. Il permet de représenter de l'information de façon structurée.

Par exemple, la réponse type JSON pour récupérer la valeur de la variable NAME_VARIABLE serait :

Demande :

```
GET https://api.spark.io/v1/devices{DEVICE_ID}/NAME_VARIABLE?access_token={
    ACCESS_TOKEN}
```

Réponse :

```
HTTP/1.1 200 OK
{
  "name": "NAME_VARIABLE",
  "result": 10,
  "coreInfo": {
    "name": "PhotonWeb",
    "deviceID": "{DEVICE_ID}",
    "connected": true,
    "last_handshake_at": "2019-01-17T22:28:40.907Z",
    "last_app": ""
  }
}
```

Il est très important de comprendre comment se présente le format JSON étant donné que nous utilisons des GET et POST pour pouvoir communiquer avec notre Photon. Nous verrons dans la partie Serveur que nous récupérerons nos données bel et bien sous format JSON.

3.1.2 Photon en I2C esclave

Comme nous l'avons vu la carte Particle Photon possède un contrôleur I2C et peut être programmé à l'aide du langage Arduino.

Voici le code simplifié permettant la configuration du Photon en esclave I2C et la récupération des deux octets de la mesure de la température :

```
#include <Wire.h>

/* Constants for i2c bus configuration */
#define I2C_ADDRESS      0x08
#define SLOW_MODE        100000
#define FAST_MODE        400000
#define CLOCK_FREQ       SLOW_MODE

/* TMP3 PMOD register */
#define REGISTER_TEMP    0x01

/* Variable for register selection */
uint8_t opcode; // first transmitted byte

/* Variables for TMP3 PMOD  */
uint8_t temp_int; //integer part for the temperature measure - second transmitted
    byte
uint8_t temp_dec; //decimal part dor the temperature measure - third transmitted
    byte

float temp;

void setup() {
  Serial.begin(9600);
  Wire.setClock(CLOCK_FREQ);
  Wire.begin(I2C_ADDRESS);
  Wire.onReceive(receiveEvent);
}

void loop() {
}

void receiveEvent(int nb_bytes) {
    // Read the first byte to determine which register is concerned
    opcode = Wire.read();
    // If there are more than 1 byte, then the master is writing to the slave
    if( nb_bytes > 1 )
        if (opcode == REGISTER_TEMP) {
            // Read the two following bytes which code the temperature value
            temp_int = Wire.read();
            temp_dec = Wire.read();
        }
    }
}
```

Nous utilisons la bibliothèque *Wire.h* pour gérer la communication I2C. Dans la fonction *setup()* nous initialisons la vitesse de transmission du bus I2C à 100kbits/s et associons au Particle Photon l'adresse 0x08. De plus, la méthode *Wire.onReceive(receiveEvent)* aura pour effet l'exécution de la fonction *receiveEvent()* à chaque réception de l'octet codant l'adresse 0x08 du Photon associé au bit d'écriture ('0').

Ainsi, si la fonction *receiveEvent()* est exécutée, cela voudra dire que le maître du bus I2C voudra écrire dans le registre précisé à l'octet suivant. La première chose à faire dans cette fonction est donc de tester le premier octet (qui sera le deuxième de la transaction après celui de l'adresse du Photon) pour savoir dans quel registre écrire la donnée qui suivra.
C'est ce qui est réalisé par la ligne :

```
1  opcode = Wire.read();
```

Ensuite, s'il reste des données à lire, nous testons la valeur de l'opcode et s'il s'agit de l'adresse du registre de température alors nous lisons les deux octets qui suivent. Le premier codera la partie entière de la mesure tandis que le second codera la partie décimale.

A noter que nb_bytes en argument de la fonction receiveEvent() correspond au nombre d'octets envoyés par la maître.

3.1.3 FPGA en I2C maître

Cette partie traite de la communication I2C mise en œuvre entre le FPGA Cmod A7 et le Particle Photon afin d'envoyer les données de température précédemment récupérées. Nous abordons seulement ici comment nous avons initié la communication entre le Cmod (maître) et le Photon (esclave). Nous ne parlerons pas ici des données à envoyer au Photon. En effet, nous n'avons pas directement envoyé les deux octets correspondant à la température (MSB data et LSB data) puisque nous avons réalisé un cryptage de ces données (voir 3.2 Sécurisation de la transmission : chiffrement AES).

Concernant l'initiation de la communication entre le Cmod A7 et le Photon, nous avons utilisé notre I2C_MASTER contrôlé par notre FSM. En effet, nous pouvons câbler plusieurs esclaves à un maître. Tout comme le capteur de température, le Photon joue donc un rôle d'esclave. Afin de pouvoir communiquer avec lui depuis le Cmod, nous lui avons assigné l'adresse : 0x08

Par conséquent, une fois la récupération des données de température réalisée, nous avons commencé à échanger avec le Particle Photon. Pour cela, nous avons procédé de la même manière qu'avec le TCN75a ; en réalisant tout d'abord l'envoi sur le bus de l'adresse 0x08, soit :

1ère étape : Envoi de l'adresse 0x08 sur SDA. Dans notre FSM nous avons donc :

STOP <= '0' ;
mode <= '0' ;
reset <= '0' ;
R_or_W <= '0' ;
chip_enable <= '0' ;

Nous avons également choisi de communiquer avec le Photon à une vitesse de 100kHz (d'où mode = '0') et réaliser un reset. Nous avons ensuite spécifié que nous voulions écrire sur le bus I2C d'où R_or_W = '0'. A noter que sous forme binaire, l'octet à envoyer au Photon est :

data_in <= "00010000" soit 0x10

Les 7 premiers bits correspondant bien à l'adresse 0x08 du Photon ; le dernier bit étant positionné à 0 pour stipuler une écriture.

2ème étape : En effet, côté Photon, nous avons décidé de ranger la valeur correspondante à la température dans un registre (opcode) égal à la valeur 1. Le Cmod A7 a donc écrit cette valeur sur le bus I2C :

data_in <= "00000001"

Néanmoins avant de réaliser ceci, nous avons fait un test sur le signal ACK de la FSM afin de savoir si celui-ci est à 1 ; autrement dit pour savoir si l'étape 1 a bien été réalisée et que le Photon a bien acquitté la réception de l'octet 0x10 (00010000). Une fois l'acquittement vérifié, nous pouvons écrire la valeur de l'opcode sur SDA. On a donc toujours : R_or_W <= '0' ;

Pour vérifier le bon transfert de ces deux octets au Photon, nous avons réalisé une simulation sous Vivado, en simulant notamment l'acquittement (ACK = '1') du Photon :

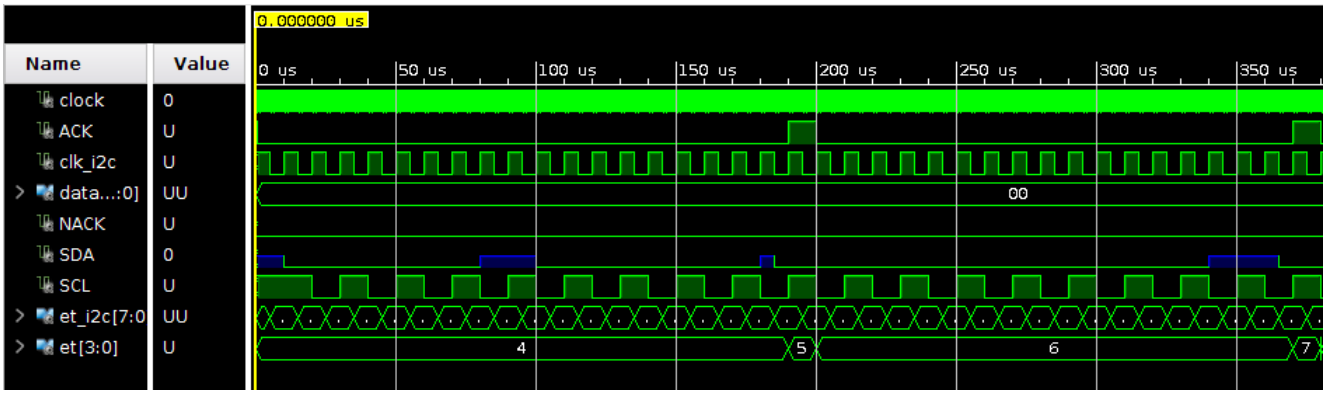


FIGURE 26: Simulation de la communication Cmod-Photon via I2C

Nous pouvons donc observer sur la figure ci-dessus la bonne transmission de l'adresse sur SDA dans un premier temps, puis l'écriture de l'opcode dans un second temps.

Si nous résumons la récupération des données de température et l'initiation de la communication avec le Photon par le Cmod nous obtenons le schéma suivant :

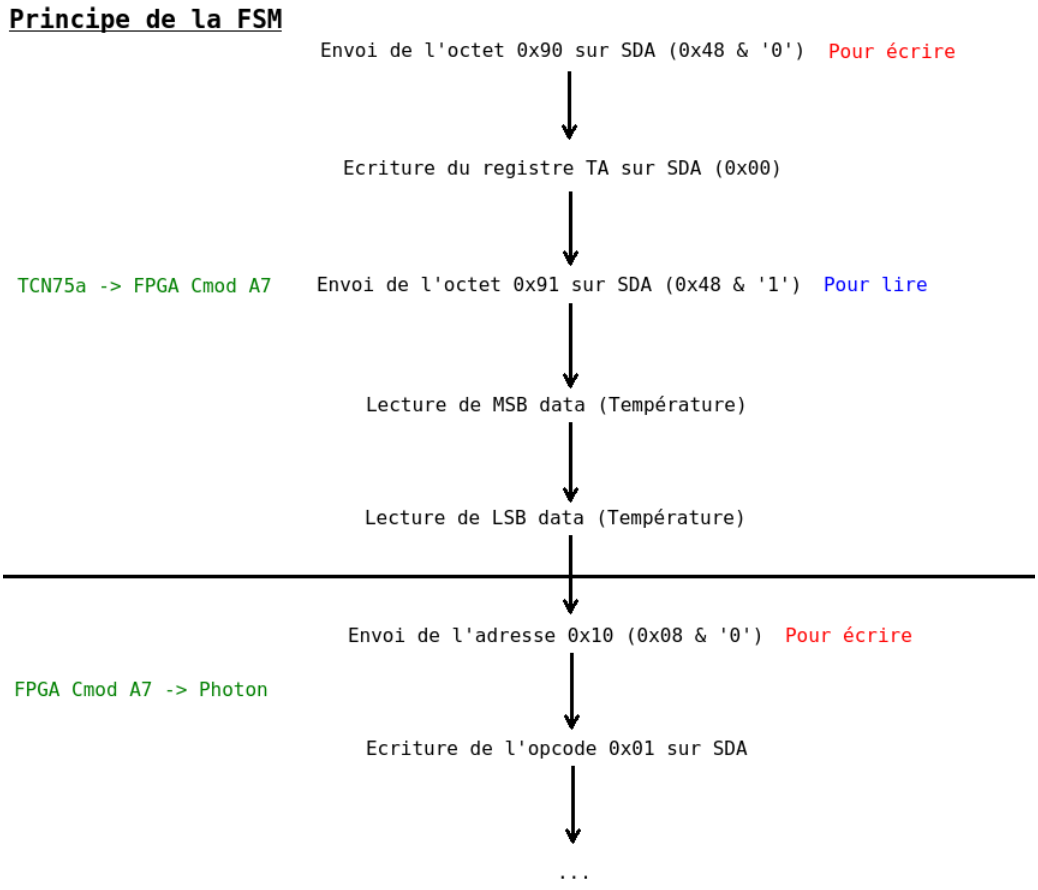


FIGURE 27: FSM implémentant la communicaton I2C FPGA/Particle Photon

3.1.4 Zynq en I2C maître

Toujours dans l'environnement XSDK et de la même manière que pour le capteur de température, nous avons développé une librairie photon_lib.h/c définissant une fonction permettant la transmission de données à la carte Particle Photon :

```
photon_send( XIicPs *Instance, u8 buffer ); /* simple wrapper de la fonction
    i2c_read() de la librairie i2c_lib.h/c permettant l écriture des données de
    température dans le registre de réception du photon. */
```

Nous ajoutons cette fonction à l'application C dont en voici une version simplifiée :

```
int config( void );
XIicPs Iic0;

int main()
{
    int Status;
    u8 tempBuffer[ 2 ];

    init_platform();

    config();

    while( 1 ) {
        temp_measure( &Iic0, tempBuffer );
        photon_send( &Iic0, tempBuffer );

        usleep(2000000);
    }

    cleanup_platform();

    return 0;
}

int config( void ) {
    i2c_config(IIC0_DEVICE_ID, &Iic0);
    temp_configMeasure( &Iic0, TEMP_RESOLUTION_12BITS );

    return XST_SUCCESS;
}
```

3.2 Sécurisation de la transmission : chiffrement AES

La communication entre les modules d'acquisition et le photon établie, nous avons souhaité sécuriser la donnée pour ne pas la transmettre en clair dans le Cloud. Pour cela nous avons mis en œuvre le chiffrement AES.

3.2.1 Présentation AES CTR 128

Sachant que la transmission des données se passe en mode sans fil (WiFi), le chiffrement des données devient nécessaire. L'algorithme que nous avons choisi pour cette tâche est l'AES-128 en mode CTR. Cet algorithme a été publié par l'Institut National des Standards et des Technologies (NIST) pour remplacer la DES qui a été prouvée vulnérable. C'est un algorithme de cryptographie à clé symétrique. Le code chiffre et déchiffre les données par blocs de 128 bits au moyen de la même clé cryptographique de 128 bits. Le chiffrement consiste à mettre le texte à chiffrer dans un tableau de deux dimensions de taille 4*4 = 16 octets qui s'appelle « state ». Dans le cas où l'entrée n'est pas sur 128 bits, un padding de zéro est ajouté. Après une opération de XOR préliminaire entre « state » et la clé « K0 », l'AES exécute dix tours des opérations suivantes :

— SubBytes, composée des transformations non-linéaires opérant indépendamment sur chaque bloc à partir d'une table dite de substitution.
— ShiftRows, opération de décalage des lignes de la manière suivante :

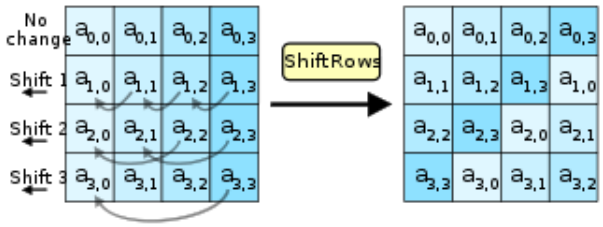


FIGURE 28: AES, opération Shift Rows

— MixColumns, une fonction qui transforme chaque octet d'entrée en une combinaison linéaire d'octets d'entrée et qui peut être exprimée mathématiquement par un produit matriciel sur le corps de Galois (2^8).
— AddRoundKey effectue un XOR entre « state » et la clé associée au tour en cours « Kr », r inclus dans [0, 10].

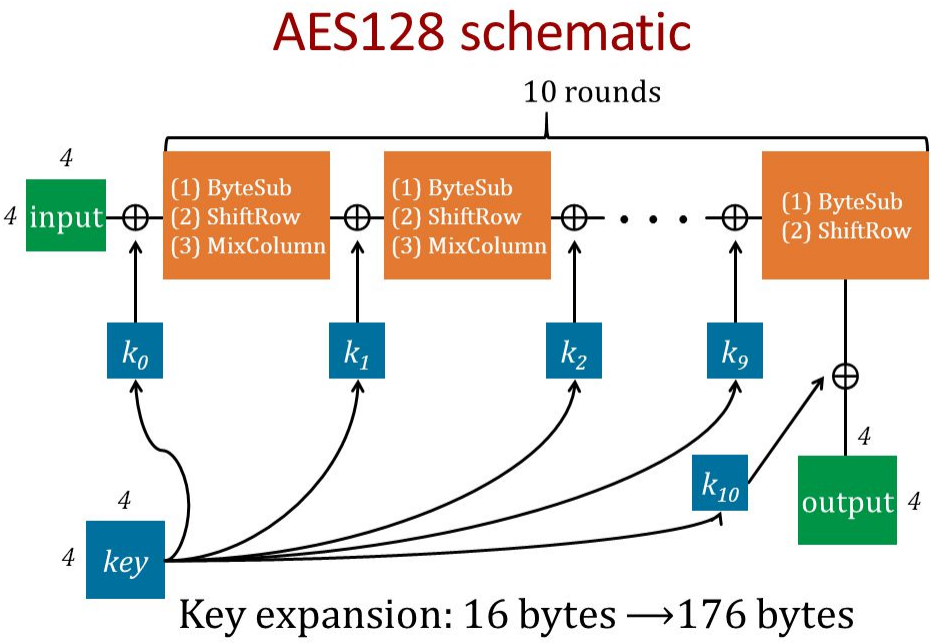


FIGURE 29: AES, opération AddRoundKey

Les dix clés sont générées par la fonction « Key expansion », qui à partir de la clé initiale, créée dix autres clés de taille 16 octets. Comme on peut voir sur le schéma précédent, à la fin de chaque tour, « state » et « Kr » passent par un XOR. Cette opération est une protection pour la clé secrète. Sachant que si on change un bit de la clé, l'erreur sera transmise dans la totalité des tours suivants, le chiffrement sera donc erroné. L'algorithme qui se cache derrière cette fonction est le suivant :

On commence par ranger la clé dans une matrice de taille 4*4 octets.

Et durant 10 tours, l'algorithme va exécuter les étapes suivantes :

- Il prend tout d'abord la dernière colonne de la clé précédente et décale les octets vers le haut.
- Les octets de cette colonne passent ensuite par une table de substitution.
- Un XOR entre cette colonne et « rcon » (constante de tour) est effectué.
- Un XOR entre la colonne et la première colonne de la clé précédente nous donne alors la première colonne de la nouvelle clé.
- Le reste des colonnes correspond au résultat du XOR de la colonne de la clé précédente avec la première colonne de la nouvelle clé.

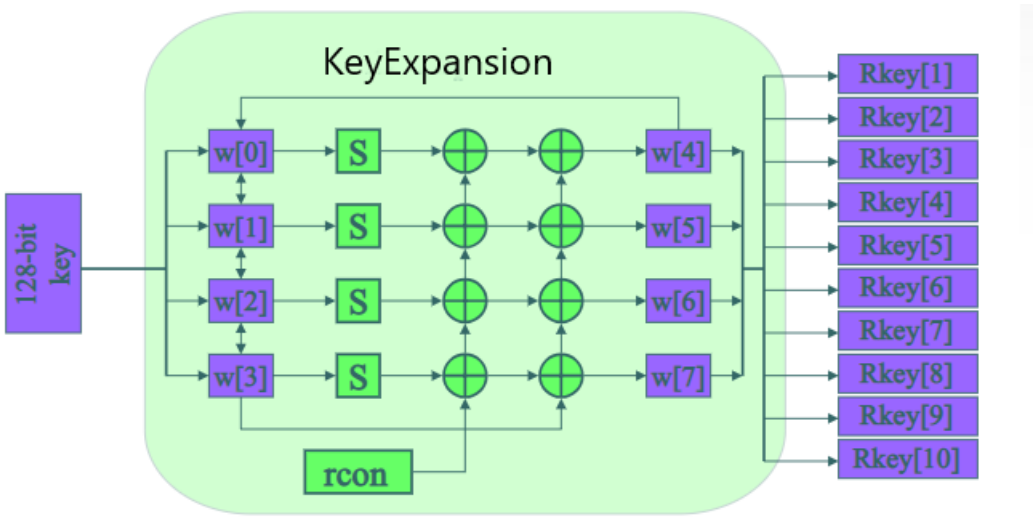


FIGURE 30: AES, Key expansion

Le mode d'AES que nous avons utilisé est le CTR (counter mode). Ce mode considère le chiffrement de bloc comme un chiffrement en flux. Le but est de faire un chiffrement rapide en parallèle. Dans ce mode, on commence par prendre un vecteur d'initialisation (IV) aussi appelé « nonce ». Cette valeur est unique et ne devrait pas être prédictible. La taille de « nonce » doit être inférieure à 128 bits (on prend 64 bits). On lui rajoute la valeur d'un compteur sur 32 bits qui est initialisé à 0 et qui est incrémenté avec chaque chiffrement d'un bloc. La concaténation de ces deux valeurs sera dans un premier temps chiffrée par l'AES. Le résultat de ce chiffrement passe ensuite par un XOR avec les données à crypter pour enfin nous donner le texte chiffré.

Le schéma suivant illustre ce principe :

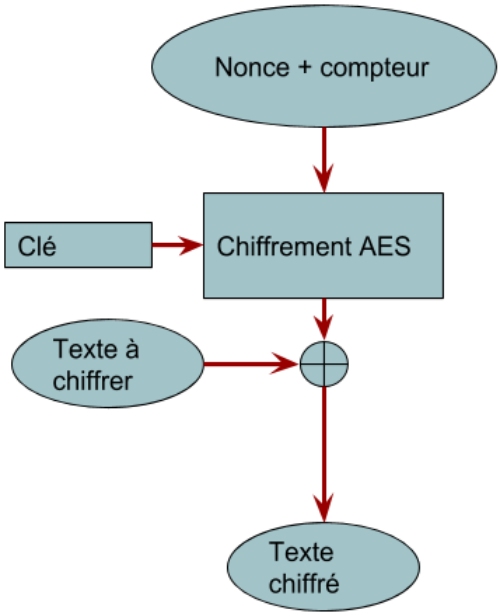


FIGURE 31: AES en mode compteur

L'implémentation sur FPGA de cet algorithme donne le module suivant :

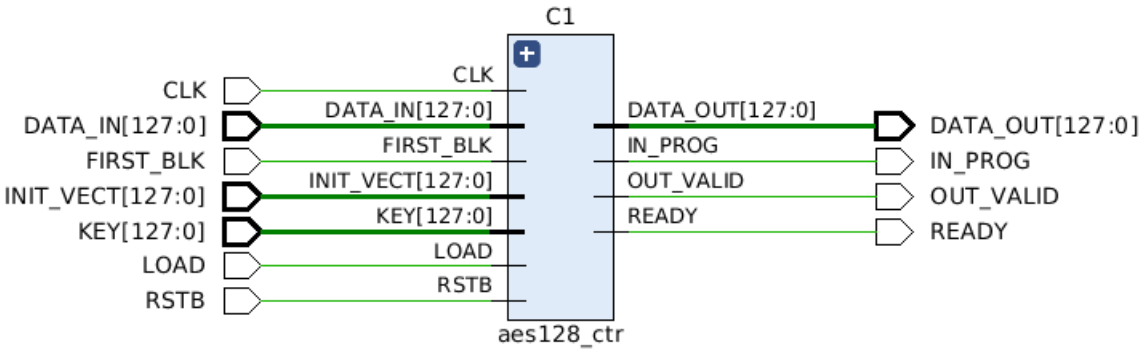


FIGURE 32: Module AES

En entrée, nous avons :

data_in : les données à chiffrer.
first_blk : le compteur du bloc AES qui est initialisé à la valeur du vecteur d'initialisation.
init_vect : la valeur « nonce » expliqué précédemment.
key : la clé de chiffrement sur 128 bits.
load : le signal de contrôle qui indique la présence d'une nouvelle donnée à chiffrer.
rst_b : le bouton de reset.

En sortie, nous avons :

data_out : les données chiffrées.
in_prog : le signal de contrôle indiquant que le chiffrement est en cours.
out_valid : un signal qui indique la fin du chiffrement.
ready : un signal qui indique que le module est disponible pour réaliser un nouveau chiffrement.

3.2.2 Implémentation FPGA

Nous avons donc procédé à l'implémentation du module AES CTR 128 présenté dans la partie précédente dans notre Top_Module qui était pour l'instant composé de l'I2C_MASTER et de la FSM. La figure page suivante présente ainsi le câblage global et fini de notre Top_Module composé de 3 sous modules :

- I2C_MASTER
- FSM
- AES_VHDL

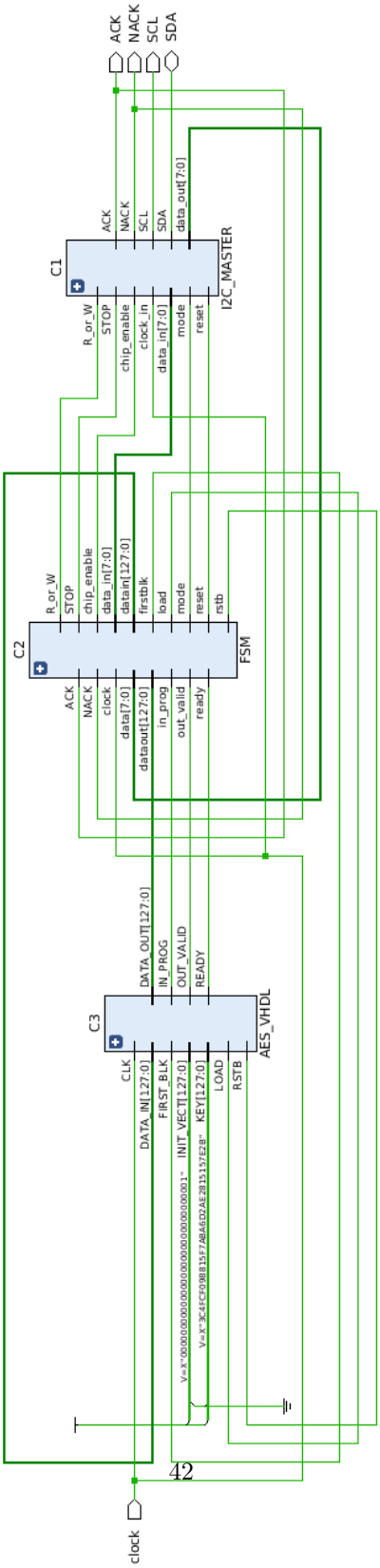

ACK
NACK
SCL
SDA
C1
I2C_MASTER
R_or_W
STOP
chip_enable
clock_in
data_in[7:0]
mode
reset
data_out[7:0]
C2
FSM
datain[127:0]
firstblk
load
rstb
clock
data[7:0]
dataout[127:0]
in_prog
out_valid
ready
C3
AES_VHDL
CLK
DATA_IN[127:0]
FIRST_BLK
INIT_VECT[127:0]
KEY[127:0]
LOAD
RSTB
DATA_OUT[127:0]
IN_PROG
OUT_VALID
READY
clock

Figure 33

Comme nous pouvons le voir, c'est donc la FSM qui s'occupe également de gérer les signaux du module AES_VHDL. Ceci est tout à fait logique car nous devons réaliser le cryptage de la température seulement quand les données (MSB data et LSB data) ont été récupérées.

Ainsi, la FSM gère 8 signaux supplémentaires :

```
1 ready : in STD_LOGIC;
2 in_prog : in STD_LOGIC;
3 data_out : in STD_LOGIC_VECTOR(127 downto 0);
4 out_valid : in STD_LOGIC;
5 load : out STD_LOGIC;
6 rstb : out STD_LOGIC;
7 data_in : out STD_LOGIC_VECTOR(127 downto 0);
8 firstblk : out STD_LOGIC;
```

Les signaux les plus importants qui nous ont permis de réaliser le cryptage sont : rstb, load, firstblk, data_in.

En effet, « rstb » correspond au reset du module AES (pour lancer un nouveau cryptage de donnée, actif niveau bas) tandis que « load » permet le chargement de la donnée « datain » à crypter. « firstblk » correspond au fait que le compteur du bloc AES est initialisé à la valeur du vecteur d'initialisation IV.

Par conséquent, nous avons fixé les valeurs de la clé (KEY en entrée du bloc AES_VHDL) et du vecteur d'initialisation (INIT_VECT en entrée du bloc AES_VHDL) :

KEY => "3c4fcf098815f7aba6d2ae2815157e2b" (en hexadécimal)
INIT_VECT => "00000000000000000000000000000001" (en hexadécimal)

Ainsi, le cryptage s'organise de la façon suivante sur le FPGA Cmod A7 :

1ère étape : Lorsque les deux octets de température sont récupérés (MSB data et LSB data), le module AES_VHDL est réinitialisé, d'où :

rstb <= '1';
load <= '1';
firstblk <= '1';

On transfert également la donnée à crypter au module. Dans notre cas, seul deux octets (MSB data et LSB data pour la température) sont à crypter. Ainsi, comme datain est codé sur 128 bits, nous avons formé le message à crypter de la façon suivante :

datain <= 120 '0' & data_MSB & data_LSB;

De cette façon, la taille de datain est de 128 bits.

Ainsi, avec l'activation de ces différents signaux, le cryptage de la donnée de température va avoir lieu. Nous allons ensuite récupérer notre valeur cryptée après initialisation de la communication avec le Photon afin de la transmettre par la suite. La valeur cryptée par le module AES_VHDL est donc renvoyée à la FSM par l'intermédiaire de dataout(127 downto 0).

2ème étape : Étant maintenant en possession de la donnée cryptée correspondante à la valeur de la température, il est nécessaire de la transmettre au Photon. Comme la donnée cryptée est codée sur 128 bits, il a fallu l'envoyer sous la forme de 16 octets consécutifs sur le bus I2C en commençant par l'octet de poids fort.

Lorsque l'acquittement de réception du Photon est reçu par le Cmod A7 pour l'octet de poids faible (par conséquent l'ensemble des 16 octets ont été envoyés), le Cmod spécifie au Photon la fin de la transmission. Le signal STOP est donc passé à 1 :

STOP <= '1' ;

Cela déclenche ainsi l'envoi d'un NACK sur le SDA, ce qui implique le passage du signal NACK à 1 pour l'I2C_MASTER : NACK <= '1' ;

NACK <= '1' ;

On passe ensuite dans un dernier état de la FSM qui correspond à un compteur permettant de ne pas recommencer le processus de récupération des données, de cryptage et d'envoi au Photon tout de suite. Ce compteur permet d'attendre 5 secondes avant de relancer une acquisition. Une fois ces 5 secondes écoulées, on recommence tout le processus décrit depuis le début du rapport sur FPGA.

Le fonctionnement global de notre FSM peut donc être représenté par la figure suivante :

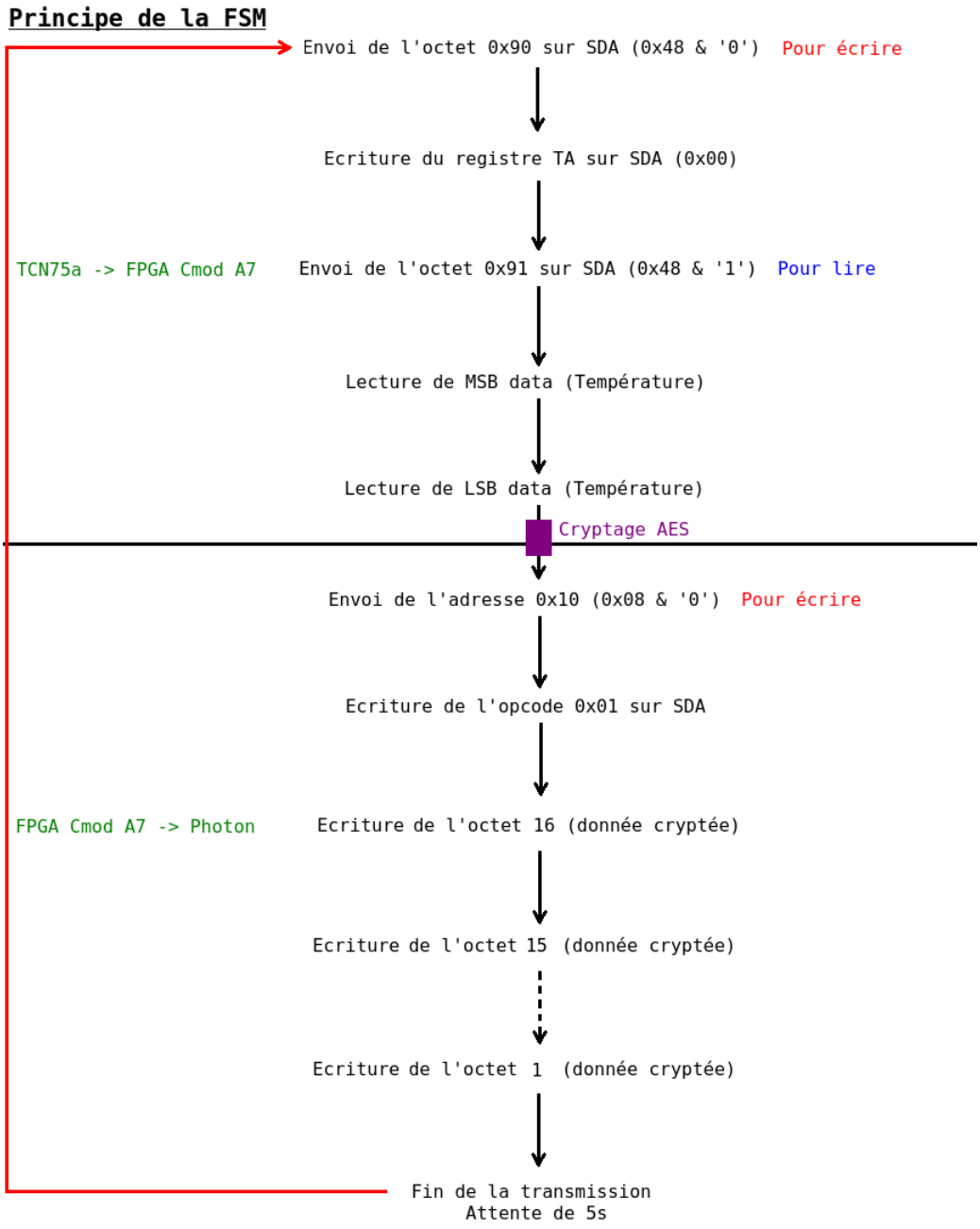


FIGURE 34: FSM mise à jour pour gérer le chiffrement AES

Cette FSM est donc celle qui a été implémentée sur le FPGA Cmod A7.

A noter que lorsque on revient dans l'état initial de la FSM (état 0), les signaux suivants du bloc AES sont remis à 0 :

```
rstb <= '0';
load <= '0';
firstblk <= '0';
```

En effet, dans l'implémentation du module AES, nous n'avons pas géré l'incrément du compteur. Celui-ci reste donc fixé à la valeur du vecteur d'initialisation INIT_VECT égal à 1 pour chaque cryptage.

A noter que nous avons réalisé la simulation sous Vivado de notre module AES_VHDL seul pour vérifier le bon cryptage des données :

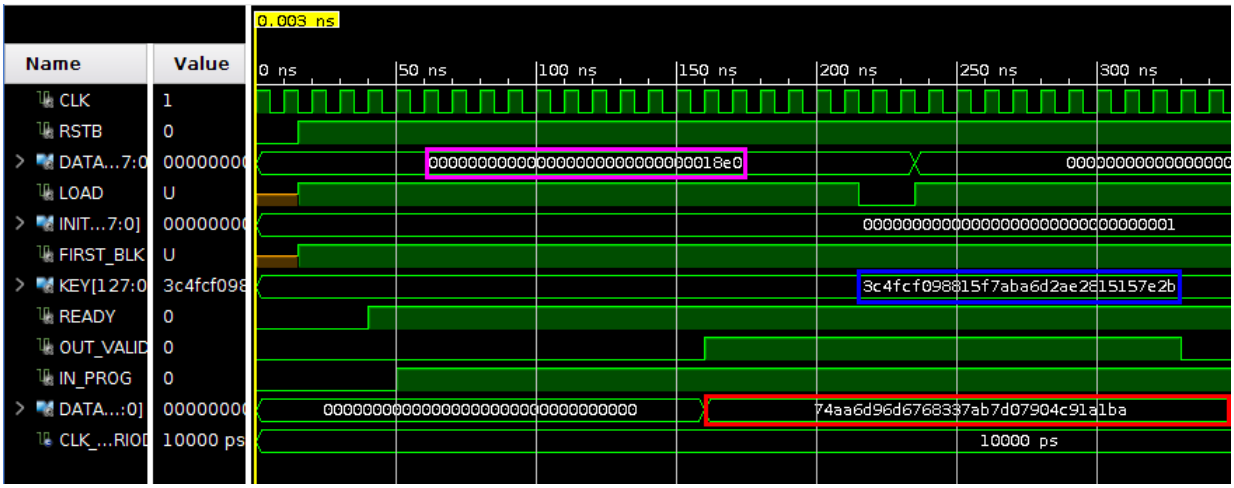


FIGURE 35: Simulation du module AES sous Vivado

Ici, la donnée entourée en violet correspond à une donnée de température simulée (MSB data = 0x18 et LSB data = 0xe0) égale à la valeur 24,875°C. La clé est entourée en bleue et la valeur cryptée en rouge.

Une fois les 16 octets de la donnée de température cryptée envoyés, il a fallu les réceptionner côté Photon. Pour cela, nous avons donc mis à jour notre fonction receiveEvent() :

```
char temp_ciphered[40] = {0};
char tmp[40] = {0};
String str_temp_ciphered;
...
void receiveEvent(int nb_bytes) {
    // Read the first byte to determine which register is concerned
    opcode = Wire.read();
    // If there are more than 1 byte, then the master is writing to the slave
    if( nb_bytes > 1 ) {
        if (opcode == REGISTER_TEMP) {
            // Read the 16 following bytes which code the encrypted data
            for( int i = 0; i < 16; i++ ) {
                temp_ciphered[ i ] =  Wire.read();
            }
            sprintf(tmp, "%02X%02X%02X%02X%02X%02X%02X%02X%02X%02X%02X%02X%02X%02X
    %02X%02X", temp_ciphered[ 0 ],temp_ciphered[ 1 ],temp_ciphered[ 2 ], ...,
    temp_ciphered[ 14 ],temp_ciphered[ 15 ]);
            str_temp_ciphered = (char*)tmp;
        }
    }
}
```

À ce stade, la valeur chiffrée de la température codée sur 128 bits est contenue dans la String *str_temp_ciphered*.

Nous présenterons dans la partie 3.3 comment nous avons ensuite fait transiter cette String au Cloud.

3.2.3 Implémentation Zynq

3.2.3.1 Création de l'IP AES, intégration au design

Il s'agit ici de réutiliser le même bloc AES et de l'intégrer au design précédent sous forme d'IP. Pour cela nous devons créer une interface permettant au processeur d'accéder à des registres en lecture/écriture via le bus AXI. Si nous reprenons le schéma du bloc AES nous pouvons déterminer le nombre de registres nécessaires à son fonctionnement :

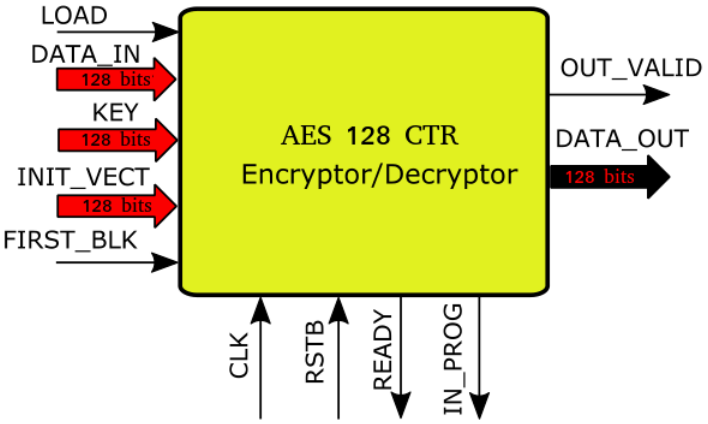


FIGURE 36: Block AES et signaux d'entrée/sortie

Il s'agit de registres de 32 bits. Nous avons choisi de les organiser de la manière suivante :

Registres d'entrée (accessibles en lecture/écriture) :

- Reg0 : bits 32-2 : unused, bit 1 : FIRST_BLK, bit 0 : LOAD
- Reg1 : DATA_IN[31 :0]
- Reg2 : DATA_IN[63 :32]
- Reg3 : DATA_IN[95 :64]
- Reg4 : DATA_IN[127 :96]
- Reg5 : KEY[31 :0]
- Reg6 : KEY[63 :32]
- Reg7 : KEY[95 :64]
- Reg8 : KEY[127 :96]
- Reg9 : INIT_VECT[31 :0]
- Reg10 : INIT_VECT[63 :32]
- Reg11 : INIT_VECT[95 :64]
- Reg12 : INIT_VECT[127 :96]

Registres de sortie (accessibles en lecture seule) :

- Reg13 : DATA_OUT[31 :0]
- Reg14 : INIT_VECT[63 :32]
- Reg15 : INIT_VECT[95 :64]
- Reg16 : INIT_VECT[127 :96]
- Reg17 : bits 31-3 : unused, bit 2 : OUT_VALID, bit 1 : IN_PROG, bit 0 : READY

Nous avons donc besoin de 17 registres de 32 bits pour interfacer le Zynq avec l'IP AES.

Nous utilisons l'utilitaire de création d'IP de Vivado afin de créer une nouvelle IP de type périphérique AXI4 que nous nommons AES128_CTR.

Nous précisons que nous voulons 17 registres de 32 bits :

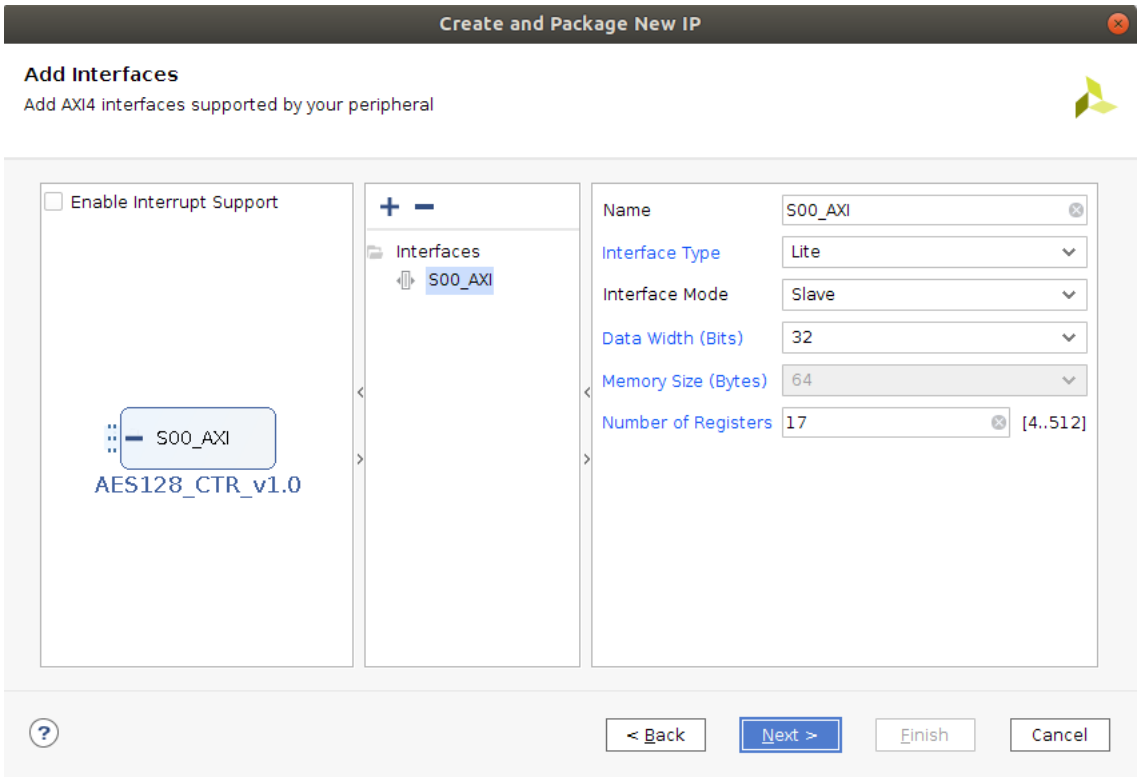


FIGURE 37: Page de création d'une nouvelle IP

Ensuite nous éditons l'IP en ajoutant les sources du bloc AES.
Nous éditons le fichier AXI128CTR_v1_0_S00_AXI.vhd qui implémente le protocole de communication avec le bus AXI4 et permet la lecture/écriture depuis/dans les registres de l'IP.

Nous déclarons 5 signaux de 32 bits pour nos 5 registres de sortie accessibles en lecture seule et nous ajoutons la déclaration du « component » de notre bloc AES dans ce fichier :

```
1  signal s_control   : std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
2  signal s_data_out0 : std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
3  signal s_data_out1 : std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
4  signal s_data_out2 : std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
5  signal s_data_out3 : std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
6
7  component aes128_ctr
8  port (
9    CLK: in std_logic;
10   RSTB: in std_logic;
11
12   DATA_IN: in std_logic_VECTOR(127 downto 0);
13   LOAD : in std_logic;
14
15   INIT_VECT: in std_logic_VECTOR(127 downto 0);
16   FIRST_BLK: in std_logic;
17   KEY : in std_logic_VECTOR(127 downto 0);
18
19   READY: out std_logic;
20   OUT_VALID: out std_logic;
21   IN_PROG: out std_logic;
22   DATA_OUT: out std_logic_VECTOR(127 downto 0));
23 end component;
```

Nous le mappons ensuite :

```
-- Add user logic here
aes128_ctr_0 : aes128_ctr
port map (
  CLK => S_AXI_ACLK,
  RSTB => S_AXI_ARESETN,
  -- read/write registers
  LOAD => slv_reg0(0),
  FIRST_BLK => slv_reg0(1),
  DATA_IN(31 downto 0) => slv_reg1,
  DATA_IN(63 downto 32) => slv_reg2,
  DATA_IN(95 downto 64) => slv_reg3,
  DATA_IN(127 downto 96) => slv_reg4,
  KEY(31 downto 0) => slv_reg5,
  KEY(63 downto 32) => slv_reg6,
  KEY(95 downto 64) => slv_reg7,
  KEY(127 downto 96) => slv_reg8,
  INIT_VECT(31 downto 0) => slv_reg9,
  INIT_VECT(63 downto 32) => slv_reg10,
  INIT_VECT(95 downto 64) => slv_reg11,
  INIT_VECT(127 downto 96) => slv_reg12,
  -- read-only registers
  DATA_OUT(31 downto 0) => s_data_out0,
  DATA_OUT(63 downto 32) => s_data_out1,
  DATA_OUT(95 downto 64) => s_data_out2,
  DATA_OUT(127 downto 96) => s_data_out3,
  READY => s_control(0),
  IN_PROG => s_control(1),
  OUT_VALID => s_control(2)
  );
-- User logic ends
```

Nous ne modifions pas le process d'écriture, de sorte que les registres correspondants à nos données de sortie soient accessibles en lecture seule par l'utilisateur. En réalité il est toujours possible d'écrire dans ces registres mais les valeurs ne seront pas transmises au bloc AES. Nous modifions ensuite le process de lecture afin de pouvoir lire les registres de sortie de l'AES.

```
process (slv_reg0, slv_reg1, slv_reg2, slv_reg3, slv_reg4, slv_reg5, slv_reg6,
    slv_reg7,
          slv_reg8, slv_reg9, slv_reg10, slv_reg11, slv_reg12, s_data_out0,
    s_data_out1,
          s_data_out2, s_data_out3, s_control, axi_araddr, S_AXI_ARESETN,
    slv_reg_rden)
variable loc_addr :std_logic_vector(OPT_MEM_ADDR_BITS downto 0);
begin
    -- Address decoding for reading registers
    loc_addr := axi_araddr(ADDR_LSB + OPT_MEM_ADDR_BITS downto ADDR_LSB);
    case loc_addr is
      when b"00000" =>
        reg_data_out <= slv_reg0;
      when b"00001" =>
        reg_data_out <= slv_reg1;
      when b"00010" =>
        reg_data_out <= slv_reg2;
      when b"00011" =>
      ...
      when b"01011" =>
        reg_data_out <= slv_reg11;
      when b"01100" =>
        reg_data_out <= slv_reg12;
      when b"01101" =>
        reg_data_out <= s_data_out0;
      when b"01110" =>
        reg_data_out <= s_data_out1;
      when b"01111" =>
        reg_data_out <= s_data_out2;
      when b"10000" =>
        reg_data_out <= s_data_out3;
      when b"10001" =>
        reg_data_out <= s_control;
      when others =>
        reg_data_out  <= (others => '0');
    end case;
end process;
```

Une fois l'IP packagée et générée nous l'ajoutons à notre design de base :

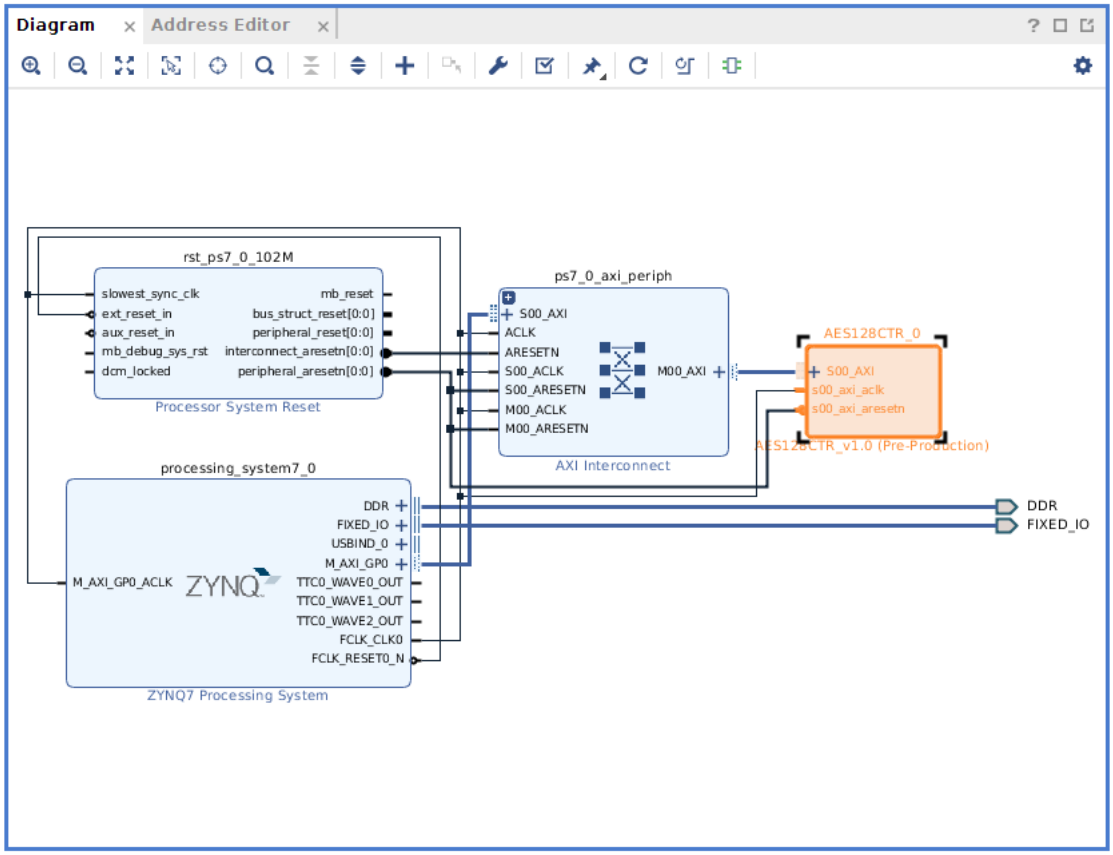


FIGURE 38: Block design, connexion AXI4 Lite Zynq/IP AES

Après génération du bitstream, nous passons à l'environnement XSDK pour développer notre application finale.

3.2.3.2 Application finale sous XSDK

Nous créons un nouveau projet et générons un Board Support Package (BSP) qui contient les couches logicielles de base permettant d'initialiser, de configurer et d'interagir avec le PS et le PL.

En particulier, nous notons la présence d'une librairie nommée AES128CTR.h/AES128CTR_selftest.c qui définit des constantes correspondant aux offsets mémoire associés aux registres du bloc AES, ainsi que 3 fonctions :

```
1 AES128CTR_mReadReg();
2 AES128CTR_mWriteReg();
3 AES128CTR_Reg_SelfTest();
```

Les deux premières fonction sont simplement des wrappers des fonctions Xil_In32() et Xil_Out32() de l'API Hardware Abstraction Layer de Xilinx qui permettent de lire/écrire des mots de 32 bits depuis/dans une adresse mémoire physique (application standalone donc pas de MMU, donc adresse mémoire = adresse physique).
La 3ème permet de réaliser un test du périphérique AES afin de s'assurer qu'il a été implémenté correctement.

À l'aide de ces fonctions nous avons développé notre propre librairie permettant de contrôler le bloc AES. Nous y avons défini des fonctions/constantes permettant de configurer et utiliser le bloc AES.

Nous pouvons citer les fonctions suivantes :

```
1 init_AES_block( u8 *init\_vect, u8 *key ); /* initialise le bloc AES avec un vecteur
      d initialisation init_vect qui sera utilisé comme première valeur du compteur
     lors du chiffrement, et une clé key. */
2 AES_cipher( u8 *data_in, u8 *data_out ) /* encrypte la valeur data_in et stocke le r
    ésultat dans data_out. */
```

À noter également la valeur de l'adresse de base du périphérique AES que nous trouvons dans le fichier xparameters.h.

Finalement, nous avons développé notre application finale en ajoutant les fonctions de chiffrement. Ainsi la boucle infinie devient la suivante :

```
u8 tempBuffer[ 2 ];
u8 aes_dataIn[ 16 ] = { 0 };
u8 aes_dataOut[ 16 ];

while( 1 ) {
    temp_measure( &Iic0 , tempBuffer );

    aes_dataIn[ 0 ] = tempBuffer[ 0 ];
    aes_dataIn[ 1 ] = tempBuffer[ 1 ];

    AES_cipher( aes_dataIn , aes_dataOut );

    photon_send( &Iic0 , aes_dataOut );

    usleep(2000000);
}
```

Les 2 octets codant la température (partie entière et partie décimale) sont stockés sur les octets de poids faible du tableau aes_dataIn.

Et nous ajoutons la fonction init_AES_block() à la fonction config() :

```
int config( void ) {

    i2c_config(IIC0_DEVICE_ID, &Iic0);
    temp_setResolution( &Iic0 , TEMP_RESOLUTION_11BITS );
    init_AES_block( init_vect , key );

    return XST_SUCCESS;
}
```

3.3 Transmission au Cloud

Comme nous l'avons présenté précédemment, les données de température sont transmises via I2C au Photon que ce soit depuis le Cmod FPGA ou depuis le Zynq. Cette partie présente comment s'organise l'envoi des données à destination du Cloud.

Pour envoyer les données à destination du Cloud (afin qu'un Client puisse ensuite les visualiser via notre Serveur), nous avons utilisé Spark.variable().

En effet, nous rappelons que « Spark.variable() » permet d'exposer une variable dans le Cloud pour qu'elle puisse être appelée avec GET.

Nous avons donc utilisé Spark.variable() de la façon suivante :

```
1 Spark.variable("Temp", &str_temp_ciphered, STRING);
```

Ici, nous avons associé au paramètre API « Temp » la variable « str_temp_ciphered » de type String qui contient les 16 octets codant la donnée chiffrée. Pour construire cette chaîne de caractères, voici comment nous avons procédé :

```
1 sprintf(tmp, "%02X%02X%02X%02X%02X%02X%02X%02X%02X%02X%02X%02X%02X%02X%02X%02X",
      temp_ciphered[0],... temp_ciphered[15]);
2 str_temp_ciphered = (char*)tmp;
```

Tout d'abord, nous avons « rangé » les 16 octets de la valeur cryptée de température (récupérés via l'I2C et stockés dans temp_ciphered[]) dans un tableau de caractères tmp[] à l'aide de sprintf() (dans la fonction receiveEvent()). Nous avons fait en sorte que les octets apparaissent sous le format hexadécimal sur deux digits. Ainsi, la valeur 7 en hexadécimal est codée comme 07. Ceci est important car nous souhaitons obtenir une chaîne contenant exactement 32 caractères. Une fois ceci réalisé, un cast de type (char*) a lieu sur tmp[] afin d'obtenir la chaîne de caractères str_temp_ciphered. Cette chaîne de caractères possède donc exactement 32 caractères et représente la valeur cryptée de la température, cryptage réalisé par le module AES intégré sur le Cmod FPGA ou le Zynq.

Afin de vérifier la bonne réception et la bonne valeur de la donnée cryptée sur le Photon, nous l'avons faite affichée sur le « Particle Serial Monitor ». En effet, grâce à Particle CLI (Interface en Ligne de Commande), nous avons pu lancer sous un Terminal Linux la commande « particle serial monitor » afin d'ouvrir le moniteur série :

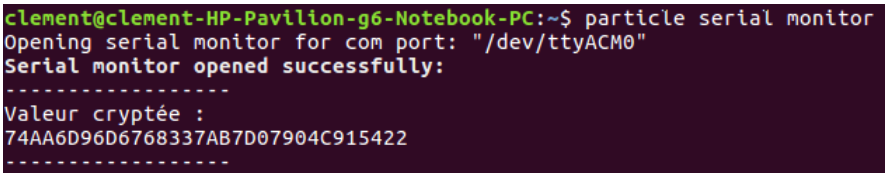


FIGURE 39: Réception de la donnée cryptée sur le Photon

Nous observons bien la donnée de température cryptée. C'est donc cette variable que nous envoyons au Cloud. Le décryptage de cette variable ne sera alors réalisé que sur le Serveur sur lequel le Client se connectera.

4 Affichage des données et déchiffrement : Serveur

4.1 Mise en œuvre du Serveur et de l'Application Web Client

Premièrement, nous allons expliquer les choix que nous avons effectués pour mettre en œuvre notre Serveur et notre Application Web côté Client. Cette partie présente également les différents langages ou notions mis en jeu.

4.1.1 Sites statiques vs sites dynamiques

Il existe deux types de sites web : les sites statiques et les sites dynamiques.

Les sites statiques : ce sont des sites réalisés uniquement à l'aide des langages HTML et CSS. Ils fonctionnent très bien mais leur contenu ne peut pas être mis à jour automatiquement : il faut que le propriétaire du site modifie le code source pour y ajouter des nouveautés. Les sites statiques sont donc bien adaptés pour réaliser des sites « vitrines » (présenter son entreprise par exemple) mais sans aller plus loin. Ce type de site se fait de plus en plus rare aujourd'hui, car dès que nous rajoutons un élément d'interaction (formulaire à remplir par exemple), on ne parle plus de site statique mais de site dynamique. Lorsque le site est statique, le schéma d'interaction Client-Serveur est très simple. La figure suivante illustre le principe :

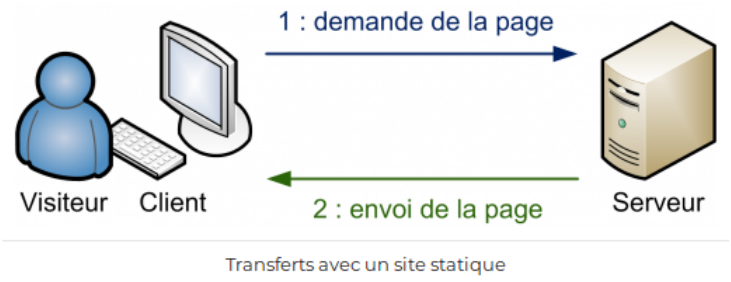


FIGURE 40: Transferts avec un site statique

1) Le Client demande au Serveur à voir une page web.
2) Le Serveur lui répond en lui envoyant la page réclamée.

Sur un site statique, il ne se passe donc rien d'autre. Le Serveur stocke des pages web et les envoie aux Clients qui les demandent sans les modifier. Le navigateur du Client reçoit alors le fichier HTML, interprète le code HTML et Javascript qu'il contient et l'affiche à l'écran. Les sites dynamiques : plus complexes, ils utilisent d'autres langages en plus de HTML et CSS, tels que PHP et MySQL. Le contenu de ces sites web est dit « dynamique » parce qu'il peut changer sans l'intervention du propriétaire du site. La plupart des sites web que nous visitons aujourd'hui sont d'ailleurs des sites dynamiques. Lorsque le site est dynamique, il y a une étape intermédiaire : la page est générée :

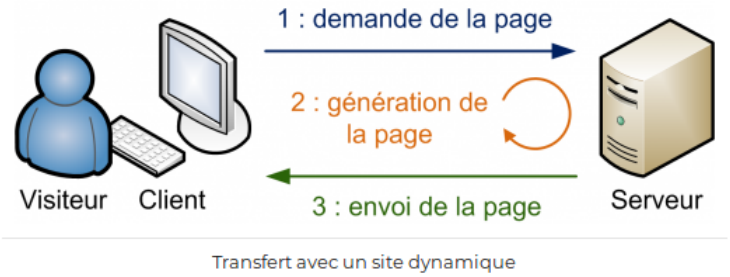


FIGURE 41: Transferts avec un site dynamique

1) Le Client demande au Serveur à voir une page web.
2) Le Serveur prépare la page spécialement pour le Client.
3) Le Serveur lui envoie la page qu'il vient de générer.

La page web est donc générée à chaque fois qu'un Client la réclame. C'est précisément ce qui rend les sites dynamiques vivants : le contenu d'une même page peut changer d'un instant à l'autre. En effet, dans le cas d'une page PHP, le Serveur a un rôle plus actif et ne fait pas simplement office de facteur comme pour un site statique. Le Client envoie toujours et de la même manière une requête au Serveur (par exemple http ://www.nom_du_site.com/index.php). Le Serveur ouvre le fichier index.php. Ici l'extension de la page est différente (.php) pour montrer au Serveur qu'il a une action supplémentaire à faire. Le fichier index.php est un script, c'est-à-dire une succession d'instructions, que le Serveur doit exécuter. Le résultat de ce script est un fichier contenant du code HTML et Javascript qui est ensuite renvoyé au navigateur du Client. L'objectif du PHP est donc de générer du code HTML (et éventuellement du Javascript) que le navigateur du Client peut lire, interpréter et afficher. Ici, il y a donc une étape supplémentaire, un ensemble de traitements exécutés par le Serveur, entre la requête et l'envoi de la page. Cela signifie aussi que le code PHP n'est pas visible sur la page générée.

Nous sommes donc partis du principe de réaliser un site web dynamique pour notre projet afin de pouvoir découvrir comment mettre en œuvre un type de Serveur gérant du PHP et pouvant être qualifié « d'actif ».

4.1.2 Les langages du Web

Pour cela, nous avons donc du utiliser différents langages pour réaliser notre Serveur. Les langages mis en jeu sont résumés par la figure suivante :

FIGURE 42: Transferts avec un site dynamique, langages impliqués

Côté Serveur donc, le langage PHP est utilisé pour générer le code HTML, CSS et/ou Javascript affiché (pour le HTML et CSS) et exécuté (pour Javascript) côté Client. Voici donc une description de ces différents langages :

HTML : L'HyperText Markup Language est le langage de balisage conçu pour représenter les pages web. C'est un langage permettant d'écrire de l'hypertexte, d'où son nom. HTML permet également de structurer sémantiquement et logiquement et de mettre en forme le contenu des pages, d'inclure des ressources multimédias dont des images, des formulaires de saisie et des programmes informatiques. Il est souvent utilisé conjointement avec le langage de programmation Javascript et des feuilles de style en cascade (CSS).

CSS : CSS (de l'Anglais Cascading Style Sheets) est le langage de mise en forme des sites web. Alors que le HTML permet d'écrire le contenu des pages web et de les structurer, le langage CSS s'occupe de la mise en forme et de la mise en page. C'est en CSS que l'on choisit notamment la couleur et/ou la taille des menus.

Javascript : Le Javascript est un langage de programmation de scripts orienté objet. Le Javascript est majoritairement utilisé sur Internet, conjointement avec les pages web HTML. Le Javascript s'inclut directement dans la page web (ou dans un fichier externe) et permet de dynamiser une page HTML, en ajoutant des interactions avec l'utilisateur, des animations, de l'aide à la navigation, comme par exemple :

- Afficher/masquer du texte ;
- Faire défiler des images ;
- Créer un diaporama avec un aperçu « en grand » des images ;
- Créer des infobulles.

Le JavaScript est un langage dit « Client-side », c'est-à-dire que les scripts sont exécutés par le navigateur chez le Client. Cela diffère des langages de scripts dits « Server-side » qui sont exécutés par le Serveur web (comme PHP).

PHP : PHP est donc un langage que seuls les Serveurs comprennent et qui permet de rendre un site dynamique. C'est lui qui « génère » la page web envoyée au Client. Le Client ne reçoit alors que le résultat du script, sans aucun moyen d'avoir accès au code qui a produit ce résultat.

4.1.3 Le concept d'Ajax

Que signifie Ajax et à quoi sert-il ?

Ajax n'est ni une technologie ni un langage de programmation. Ajax est un concept de programmation web reposant sur plusieurs technologies comme le JavaScript et le XML (d'où son nom Ajax). L'idée même d'Ajax est de faire communiquer une page web avec un Serveur web sans occasionner le rechargement de la page. C'est la raison pour laquelle Javascript est utilisé, car c'est lui qui va se charger d'établir la connexion entre la page web et le Serveur. Afin de mieux comprendre le principe de l'Ajax, voici une comparaison du fonctionnement d'un site web dit traditionnel (dynamique) et d'une application web mettant en œuvre Ajax.

Comme nous l'avons vu précédemment pour un site dynamique, le Client envoie une requête au Serveur par l'intermédiaire de son navigateur (via une URL). Le Serveur répond en renvoyant au navigateur le code HTML de la page ainsi que tout ce qui lui est associé comme les scripts Javascript. Par conséquent, la réponse du Serveur est beaucoup plus volumineuse que la requête. Le navigateur affiche la page et l'utilisateur peut la parcourir quelques instants avant de cliquer sur un lien hypertexte qui enverra une nouvelle requête au Serveur qui lui-même renverra le HTML correspondant et ainsi de suite. D'un point de vue purement pratique, c'est extrêmement inefficace et coûteux puisque le Serveur va à chaque fois tout renvoyer, ce qui prend du temps et ce qui occasionne une charge pour le Serveur. Ce principe est illustré par la figure suivante.

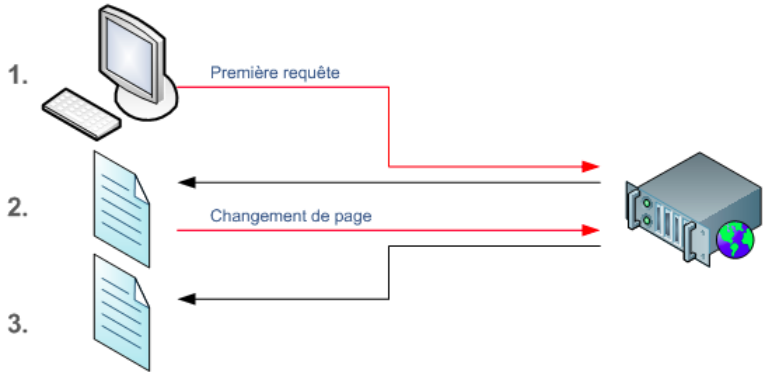


FIGURE 43: Fonctionnement d'un site web dynamique

Voyons ce même schéma du point de vue d'Ajax maintenant. Quand on utilise le concept d'Ajax dans une page web, on parle d'Application Web. La première requête est la même. La différence va résider dans le fait que quand le Client cliquera sur un lien (ou un autre élément cliquable) la page ne se rechargera pas et le navigateur enverra une requête au Serveur, lequel renverra les données demandées dans un format léger dans le format JSON. Dans ce cas, le Serveur n'aura renvoyé qu'un minimum de données ce qui est beaucoup plus léger et donc plus rapide. Le navigateur, par le biais de Javascript, peut alors mettre à jour une petite partie de la page avec les données reçues du Serveur. Le serveur et le navigateur travaillent pour proposer une solution optimale. Ajax doit donc être utilisé pour charger/modifier de petites parties d'une page comme l'illustre la figure suivante :

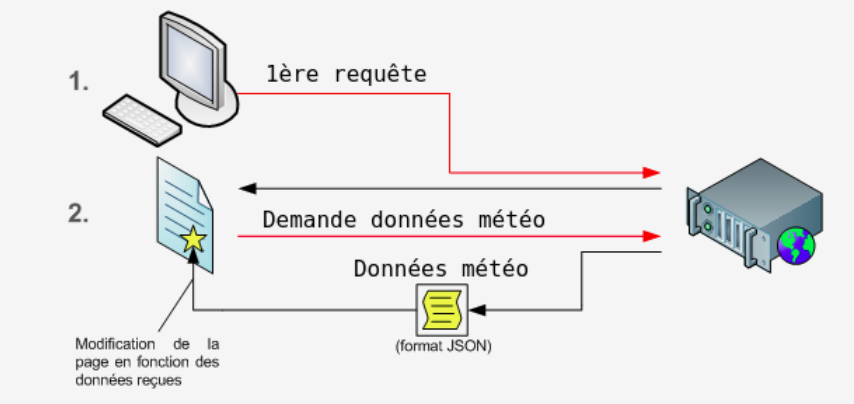


FIGURE 44: Rôle de l'Ajax

Le JSON est le format travaillant de paire avec Ajax quand il s'agit de recevoir des données classées et structurées, car c'est une manière de structurer l'information en utilisant la syntaxe objet de JavaScript.

Nous avons donc également mis en place de l'Ajax pour ne renvoyer qu'un minimum de données au Client une fois la première requête envoyée. Pour cela, nous avons fait appel à la bibliothèque Javascript jQuery qui permet de mettre en œuvre l'Ajax. Nous verrons dans la suite que nous avons notamment utilisé dans notre script Javascript :

- $.ajax() qui permet d'envoyer une requête de type POST ou GET au Serveur (nécessité de spécifier le type de la requête).
- $.get() et $.post() qui sont des raccourcis de $.ajax() et qui permettent directement de réaliser une requête GET ou POST.
- $.getJSON() qui permet d'envoyer une requête GET. Le Client souhaite récupérer la réponse sous le format JSON.

Comme nous l'avons vu précédemment dans le rapport, l'API Particle Photon accepte les demandes en JSON et répond toujours en JSON. Par conséquent, les fonctions présentées ci-dessus seront celles utilisées pour « communiquer avec le Photon ». Nous présenterons la mise en œuvre technique dans la suite du rapport ; partie 4.4 ; Fonctionnement du Serveur et Application Web côté Client.

Ainsi, nous utilisons tous les concepts/langages présentés précédemment : PHP, HTML, CSS, Javascript et Ajax afin de réaliser notre Serveur et l'Application Web Client associée. Dans la prochaine partie, nous exposons comment nous avons mis en place notre Serveur.

4.2 Xampp : Mise en place d'un Serveur Web local

Le PHP s'exécute donc sur le Serveur et son rôle est de générer des pages web. Cependant, seul un Serveur peut lire du PHP et nos ordinateurs ne sont pas des Serveurs. Nous avons donc temporairement transformé un de nos ordinateurs en Serveur pour que nous puissions exécuter du PHP et travailler sur notre site dynamique. Il a donc fallu simplement installer les mêmes programmes que ceux que l'on trouve sur les Serveurs qui délivrent les sites web aux internautes. Pour cela, nous avons utiliser Xampp sur Linux.

Xampp signifie X Apache MySQL Perl PHP. Il sert à utiliser les langages Apache, Perl et PHP mais aussi à stocker des données sous forme de bases de données grâce à MySQL. Dans notre cas, il va nous servir à installer un Serveur de développement en PHP sur notre ordinateur. En effet, Xampp n'est simplement qu'un regroupement de différents logiciels indispensables au bon fonctionnement d'un Serveur. De plus, il est entièrement gratuit et se veut facilement installable sous Linux. Pour ces différentes raisons, nous avons donc décidé d'utiliser Xampp pour créer un Serveur local sur une de nos machines.

Après installation, pour démarrer Xampp (et donc PHP notamment), il est nécessaire d'exécuter la commande suivante dans un Terminal :

/opt/lampp/lampp start (stop pour l'arrêter)

Il faut cependant ne pas oublier d'être en « root » lorsque nous démarrons ou arrêtons Xampp.

Lorsque Xampp est démarré, nous pouvons alors ouvrir un navigateur et taper l'adresse suivante :

`http://localhost`

La page principale de configuration de Xampp s'affiche ensuite. Sous Linux, les fichiers PHP qui sont créés doivent être placés dans le répertoire /opt/lampp/htdocs. Dans notre cas, nous avons donc créé un dossier STATION_METEO qui comporte différents fichiers/dossiers et qui composent tout simplement notre Serveur. En voici la liste :

- Un dossier « img » qui contient les images que le Serveur peut utiliser pour construire la ou les page(s) qu'il doit renvoyer au Client.
- Un fichier « page1.php » qui représente la page à laquelle le Client doit se connecter pour avoir accès au relevé de la température. Cette page PHP permet de protéger la page « page2.php ». En effet, un identifiant et un mot de passe doivent être soumis sur la page 1 pour pouvoir accéder à la page 2. Si l'un ou l'autre n'est pas correct, l'accès à la page 2 est refusé. Les langages utilisés sur cette page sont : HTML et PHP.
- Un fichier « page2.php » qui représente réellement l'Application Web du Client. En effet, lorsque le Client obtient cette page (identifiant et mot de passe corrects), elle lui permet de récupérer la température issue du capteur TCN75a (pour la chaîne FPGA comme pour la chaîne Zynq) avec une mise à jour toutes les 5 secondes (Javascript + Ajax). Les langages utilisés sur cette page sont : PHP, HTML et Javascript. Bien évidemment, c'est ici que l'Ajax prend son intérêt et est utilisé pour réaliser des requêtes (POST ou GET) à notre Serveur. La partie 3.4 Fonctionnement du Serveur et Application Web côté Client explique plus en détail les différents échanges Client-Serveur-Cloud Particle.
- Un fichier « stylepage1.css » qui permet simplement la mise en forme de la page « page1.php ».
- Un fichier « stylepage2.css » qui permet simplement la mise en forme de la page « page2.php ».
- Deux fichiers « proxy.php » et « decryptAES.php » que nous présentons plus spécifiquement dans la partie suivante.

Ainsi, pour nous connecter à notre Serveur depuis un navigateur, il suffit de taper l'adresse suivante :

`http://localhost/STATION_METEO/page1.php`

A noter que si un Client tente directement de se connecter à la page 2 (`http://localhost/STATION_METEO/page2.php`) sans appeler la page 1, il obtiendra un message d'erreur comme s'il s'était trompé d'identifiant ou de mot de passe.

4.3 proxy.php et decrypt.php

Ces deux fichiers PHP sont très importants dans le fonctionnement de notre Serveur.

proxy.php :
Comme son l'indique ce script PHP est un proxy réseau. Par définition, un proxy réseau est une fonction informatique Client-Serveur qui a pour fonction de relayer des requêtes entre une fonction Cliente et une fonction Serveur (couches 5 à 7 du modèle OSI). Les Serveurs proxys sont notamment utilisés pour assurer les fonctions suivantes :

- Accélération de la navigation : mémoire cache, compression de données, filtrage des publicités ou des contenus lourds (java, flash).
- La journalisation des requêtes (historique).
- La sécurité du réseau local.
- Le filtrage et l'anonymat.

On peut représenter de façon simplifiée le fonctionnement d'un proxy réseau via la figure suivante :

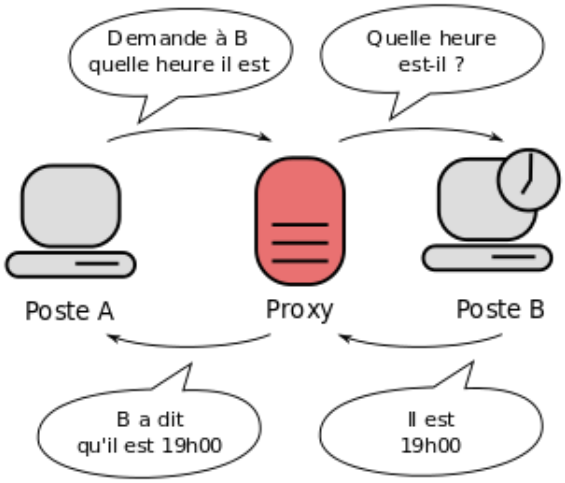


FIGURE 45: Principe de fonctionnement du proxy

C'est exactement pour ce dernier cas que nous avons mis en œuvre le proxy, pour l'anonymat. En effet, nous rappelons que lorsque nous faisons une requête au Cloud Particle (grâce à son API) ; pour obtenir la valeur de la donnée de température (qui est cryptée) ; nous sommes obligés de stipuler le Device ID et l'Access Token dans l'URL de la requête GET :

```
https:
//api.spark.io/v1/devices{DEVICE_ID}/NAME\_VARIABLE?access\_token={ACCESS_TOKEN}
```

Comme nous l'avons précédemment stipulé, il n'est pas possible de faire afficher directement l'Access Token en clair sur le réseau sous peine de se faire pirater son Particle Photon. C'est ici qu'intervient le proxy.

En effet, avec lui, lorsque le Client va réaliser une requête de type GET (via Ajax) toutes les 5s pour obtenir la nouvelle valeur de température, l'URL de la requête du Client sera :

'proxy.php?'+'/NAME_VARIABLE'

En effet, le Client va d'abord contacter notre Serveur ; Serveur qui par l'intermédiaire de notre proxy se chargera de contacter le Cloud Particle. Les données sensibles (Device ID et Access Token) sont ainsi stockées du côté de notre Serveur et le Client ne peut ainsi pas avoir accès à l'URL complète (en visualisant le code source de la page notamment).

Dans notre cas, lorsque le Particle Cloud aura renvoyé la réponse à la requête à notre Serveur ; notre Serveur renverra ensuite la réponse au Client en utilisant le proxy et bien évidemment le principe d'Ajax.

L'intérêt d'avoir défini un tel proxy est donc purement sécuritaire.

Note : Seul le nom de la Spark.variable() utilisée est donc visible dans l'URL envoyée à notre Serveur.

decryptAES.php :
Comme son nom l'indique ici aussi, ce script a pour but de réaliser le déchiffrement de la donnée de température cryptée (par le FPGA ou le Zynq) du côté de notre Serveur. De cette façon également, les données sensibles (clé de déchiffrement et vecteur d'initialisation) sont définies du côté Serveur et ne sont donc pas consultables par le Client.

Le fonctionnement est donc simple. Lorsque le Serveur renvoie la réponse à la requête pour obtenir la donnée de température cryptée ; une requête Client de type POST est renvoyée au Serveur en lui stipulant l'URL decryptAES.php (requête Ajax). Ce script se charge alors par l'intermédiaire de la fonction « decrypt » de réaliser le déchiffrement. Nous avons notamment utilisé la fonction PHP :

openssl_decrypt

Cette fonction prend une chaîne brute encodée et la décrypte en utilisant la méthode et la clé qui sont passées. Dans notre cas, la méthode est bien évidemment « aes-128-ctr ». Ce qui nous donne le code ci-dessous :

```
1  function decrypt($data)
2  {
3      $form_data_str = $data['myParams']['term'];
4      $algorithm = 'aes-128-ctr';
5      $sKey = pack("H*", "3c4fcf098815f7aba6d2ae2815157e2d");
6      $iv = pack("H*", "00000000000000000000000000000001");
7      $decrypted_data = openssl_decrypt(pack('H*', $form_data_str), $algorithm, $sKey,
       OPENSSL_RAW_DATA, $iv);
8      $TempLSB = substr($decrypted_data, -1);
9      $TempMSB = substr($decrypted_data, -2, -1);
10     $TempL   = ord($TempLSB);
11     $TempM   = ord($TempMSB);
12     $Temperature = $TempM + ($TempL/256);
13     echo $Temperature
14 }
```

Nous avons ensuite réalisé plusieurs manipulations afin de retourner la valeur de la température sous son format réel (24,875°C par exemple). En effet, après déchiffrement, nous obtenons toujours une String avec les deux premiers octets correspondant à MSB data et LSB data. Nous les avons donc soustraits à la String générale grâce à la fonction substr(). Nous avons ensuite utilisé ord() pour convertir la valeur hexadécimale obtenue sous forme entière. Enfin, nous avons procédé au calcul en utilisant MSB data et LSB data pour obtenir la température réelle. Soit :

$$\text{Température} = \text{MSB data} + (\text{LSB data} \,/\, 256)$$

La valeur ainsi obtenue est ensuite renvoyée au Client pour être affichée sur l'Application Web (réponse Ajax).

4.4 Fonctionnement du Serveur et Application Web côté Client

Cette partie a pour but de présenter le fonctionnement global de notre Serveur permettant au Client de profiter d'une Application Web en temps réel.

Un grand schéma aura pour but de récapituler les différentes interactions entre le Client (l'Application Web), notre Serveur et le Cloud du Particle Photon. Nous allons donc présenter étape par étape le fonctionnement de notre Serveur :

1ère étape : Un Client souhaite avoir accès à l'Application Web et tente de se connecter sur la « page1.php ». Il envoie donc une requête à notre Serveur qui lui répond en lui envoyant la page ci-dessous ; page sur laquelle il doit renseigner un identifiant et un mot de passe (nous avons choisi projets9 comme identifiant et mot de passe pour information) :

FIGURE 46: Page de connexion, page1.php

Deux cas de figure s'offrent à lui dans ce cas :

— Il possède l'identifiant et le mot de passe et il arrive à accéder à la page « page2.php ».

— Il se trompe ou ne possède pas tous les éléments pour se connecter (identifiant et/ou mot de passe erronés) et arrive ainsi sur la page d'erreur suivante :

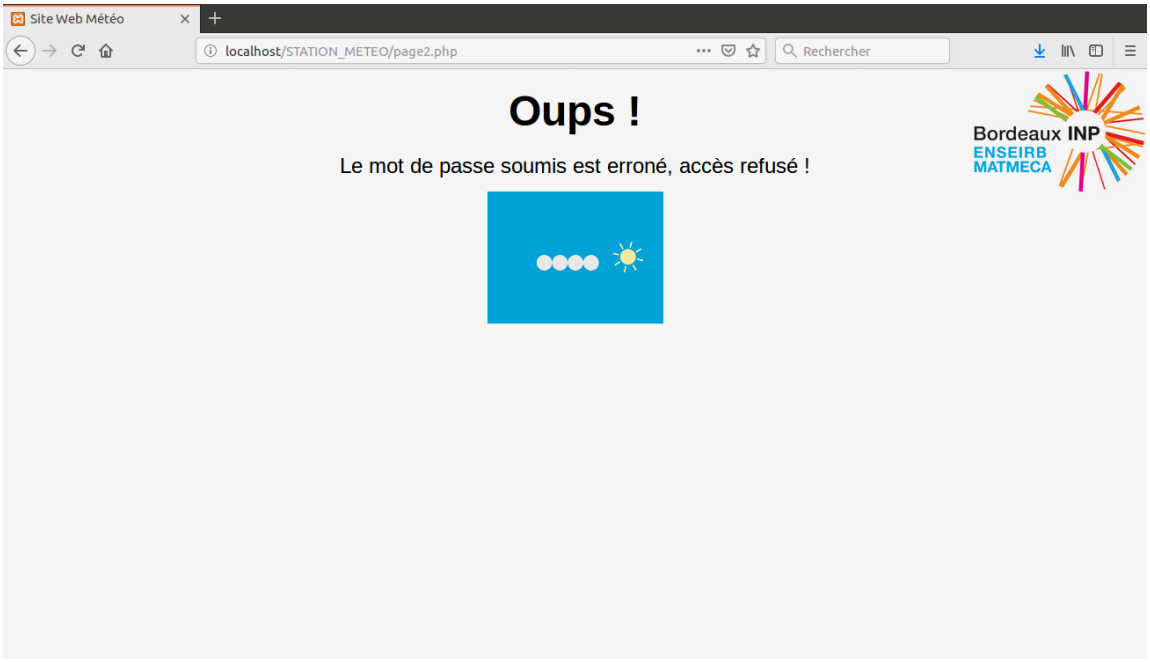


FIGURE 47: Page d'erreur, page2.php

2ème étape : Si le Client a réussi à s'identifier correctement, notre Serveur répond positivement à la requête du Client et donne accès au contenu de la page « page2.php ». Cela va ainsi lui permettre de récupérer la donnée de température. Il arrive donc sur la page ci-dessous :

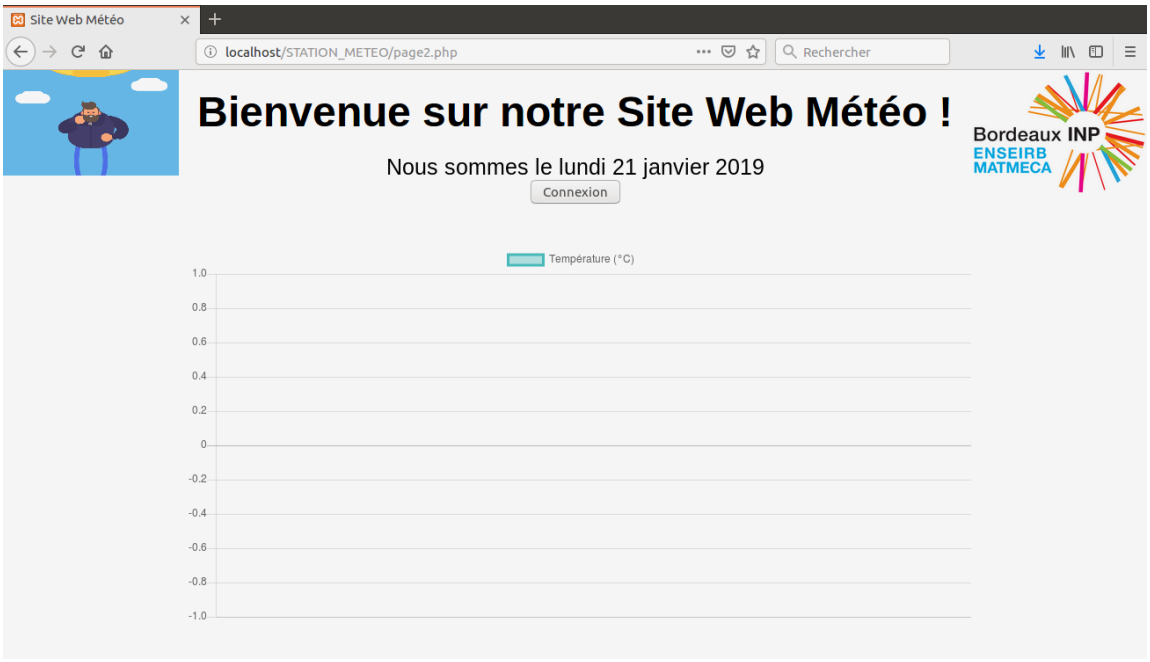


FIGURE 48: Page de visualisation en attente de connexion, page2.php

Cette page correspond à l'Application Web du Client. Derrière cette page se cache le code Javascript couplé à l'Ajax. Le Client souhaite visualiser la température mesurée par notre station météo : il clique sur le bouton « Connexion ». C'est à ce moment que le Javascript codé entre en jeu.

3ème étape : En effet, dès lors que le Client clique sur le bouton « Connexion », une succession d'étapes se produit jusqu'à la réception de la donnée de température. En effet, le code Javascript est exécuté à partir du moment où le bouton « Connexion » est cliqué.

Notre code Javascript se compose de 7 fonctions, toutes liées les unes aux autres. Nous les présentons en suivant :

updateBtn() : Cette fonction est exécutée lorsque le bouton « Connexion » ou « Déconnexion » est appuyé (c'est le même bouton). Cette fonction permet de changer le nom du bouton. Si le Client se connecte, le bouton prendra le nom « Déconnexion ». Au contraire, si le Client est déconnecté (c'est son cas de base), le bouton prendra le nom « Connexion ». Lorsque le bouton est cliqué et que « Connexion » est affiché dessus, la fonction connection() est appelée. Au contraire, si le bouton est cliqué lorsque « Déconnexion » est affiché dessus, la fonction deconnection() est appelée.

connection() : Elle affiche le message "Connexion..." et appelle ensuite la fonction receiveData().

receiveData() : C'est la fonction principale de notre Application Web ! Lorsqu'elle est appelée (et tant que le bouton « Déconnexion » n'est pas appuyée), cette fonction va tourner en boucle par l'intermédiaire notamment de window.setInterval() qui va permettre de réaliser des requêtes Ajax GET et POST toutes les 5 secondes :

- La requête GET aura pour objectif de récupérer la valeur de la variable « Temp » stockée dans le Cloud Photon (en passant par notre Serveur) et qui correspond à la température cryptée : c'est donc ici que le proxy.php est utilisé :
  $.getJSON('proxy.php ?'+'/Temp', function(json)
- La requête POST est utilisée pour demander à notre Serveur de bien vouloir décrypter la donnée de température cryptée (récupérée précédemment avec la requête GET) en appelant decryptAES.php (et plus précisément sa fonction decrypt()) :

```
$;ajax({
    url:  'decryptAES.php',
    type: 'POST',
    data:
    {
        myFunction: 'decrypt',
        myParams: {
            term:temperature
        }
    },
    success: function(data)
    {
        decryptedTemp = data;
    }
});
```

Lorsque la donnée est décryptée, le Serveur répond alors au Client en lui renvoyant la donnée réelle de la température. Il s'ensuit l'affichage textuel de la température avec la date et l'affichage graphique via la fonction adddatagraph(). Ce processus s'opère donc toutes les 5 secondes.

addatagraph() : Comme son nom l'indique, cette fonction permet simplement de réaliser l'affichage de la température sur un graphique (temps en abscisse, température en ordonnée). Elle affiche les 20 dernières valeurs reçues (par pas de 5 secondes).

deconnection() : Elle affiche le message "Déconnexion..." et appelle ensuite la fonction noreceiveData().

noreceiveData() : Cette fonction permet donc de mettre fin à la réception de la donnée de température lorsque le bouton « Déconnexion » est cliqué. Elle réalise un window.clearInterval() qui a pour objectif de stopper le window.setInterval() défini dans receiveData() ; ce qui a pour conséquence de mettre fin à l'envoi de requêtes à destination du Cloud du Photon notamment.

sayError() : Cette fonction est une fonction pour prévenir le Client. Si lors de la phase de connexion au Cloud, la connexion n'est pas établie au bout d'une minute (le Photon n'est pas connecté à Internet par exemple), une fenêtre s'affichera avec le message :

« Impossible de se connecter à la station météo. »
« Vérifier si la station météo est alimentée ! »

Si tout se passe bien et que le Client est connecté, l'acquisition de la donnée de température doit donner le résultat graphique suivant :

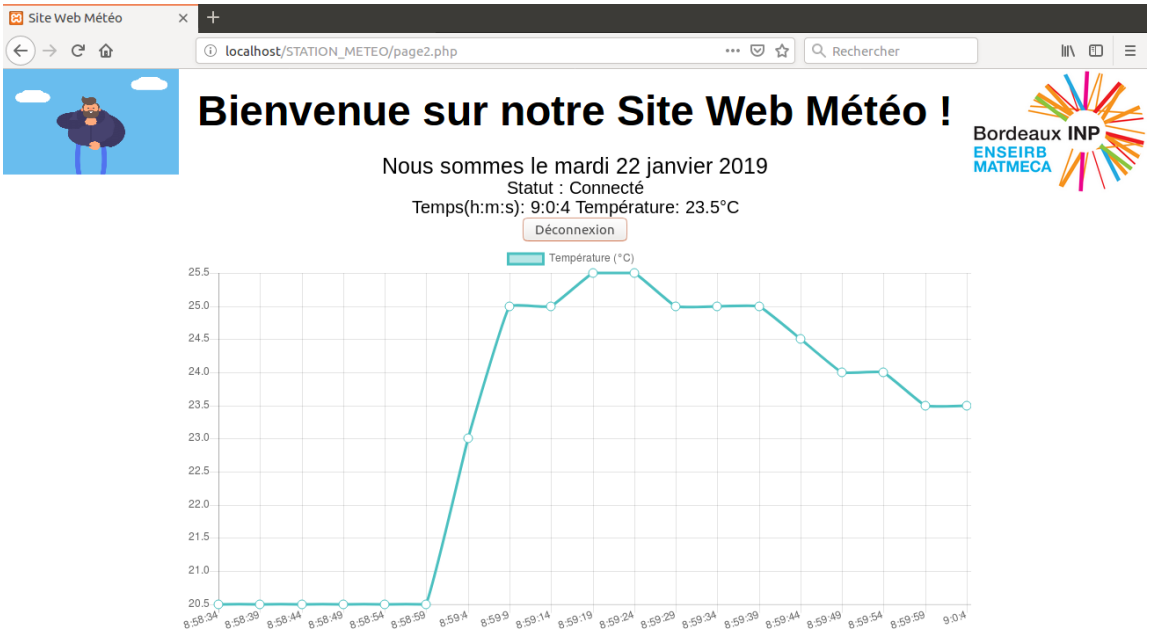


FIGURE 49: Page de visualisation, page2.php

Comme nous l'avons introduit dans cette partie, le graphique suivant illustre les interactions Client-Serveur-Particle Photon :

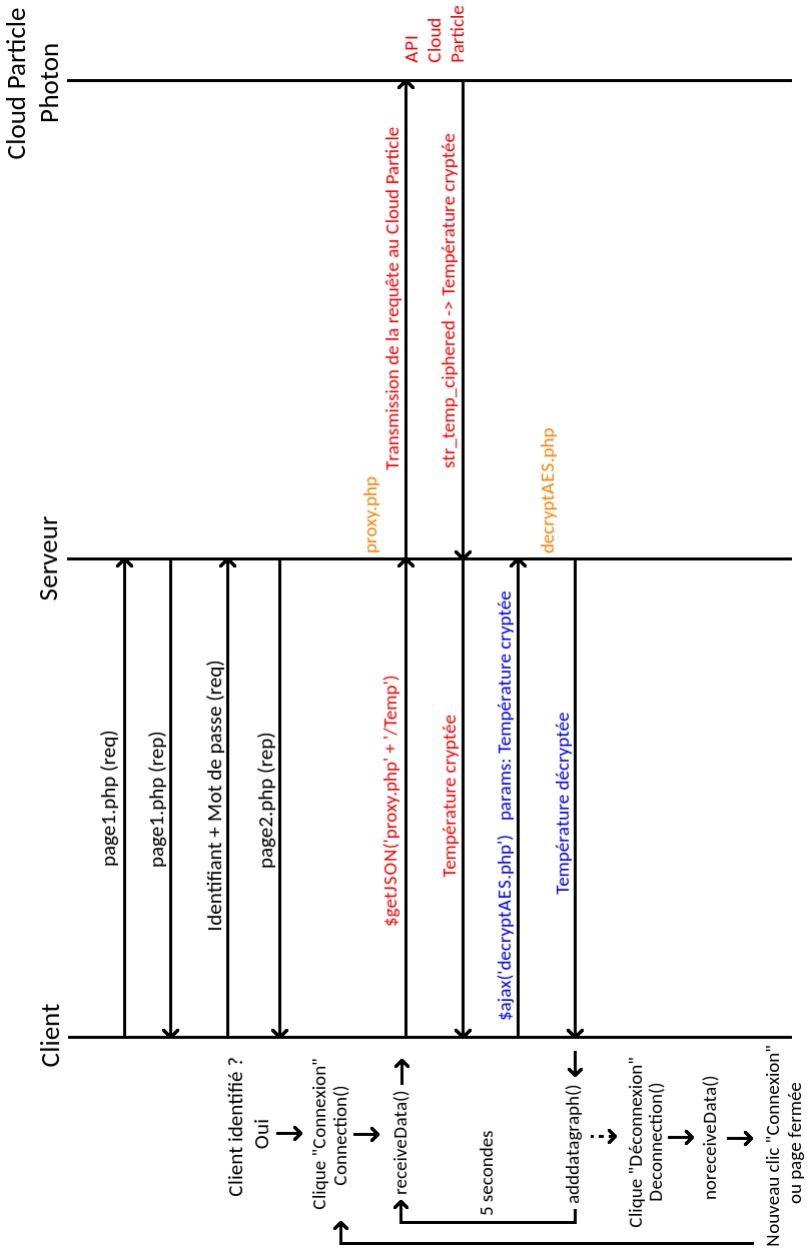


FIGURE 50: Diagramme de séquence de l'acquisition de données depuis le Cloud

A noter que lorsque le Client se déconnecte (il ne reçoit plus la dernière valeur de température), il peut toujours se reconnecter en cliquant de nouveau sur le bouton « Connexion ». Il n'a pas besoin de ressaisir identifiant et mot de passe ; il reste donc sur la « page2.php ». Dans ce cas, le graphique gardera les anciennes données affichées et affichera les nouvelles à la suite. S'il souhaite quitter définitivement l'Application Web, il lui suffit simplement de fermer la fenêtre du navigateur utilisé.

5 Optimisation énergétique de la plateforme

Dans un souci d'économie d'énergie, nous avons cherché à ne pas réaliser de mesures de température inutiles.
En effet, nous souhaitons que la partie acquisition/transmission de la plateforme IoT ne soit active uniquement qu'en cas de connexion d'un Client sur le Serveur.

L'objectif de l'implémentation Zynq est de pouvoir réveiller (ou de laisser « endormi ») le Zynq en fonction de la connectivité d'un Client sur le Serveur. En effet, si le Client se connecte via le bouton « Connexion » de la page 2, cela réveillera le Zynq qui réalisera l'acquisition de la donnée de température, l'enverra au Photon qui lui-même stockera la donnée dans le Cloud. Dans le cas contraire, si le Client n'est pas connecté sur le Serveur et qu'il n'a pas souhaité recevoir la température de la station météo, le Zynq restera endormi.

5.1 Récupération du statut de connexion du serveur depuis le Photon

Tout d'abord nous devons récupérer l'information de connection depuis le Cloud.
Pour réaliser cela, nous avons donc eu besoin d'utiliser une Spark.function() :

```
1 Spark.function("senddata", sendData);
```

En effet, « Spark.function() » permet d'exposer une fonction dans le Cloud afin qu'elle puisse être appelée avec POST. C'est donc ce que nous avons mis en œuvre avec la fonction requeteState(printData) codée du côté de notre Serveur. En effet, lorsque le Client appuie sur le bouton « Connexion » de l'Application Web, une requête de type POST est envoyée à notre Serveur (utilisation du proxy) qui transmet ensuite au Cloud du Photon pour lier la variable « printData » à la fonction sendData(String etat) codée sur le Photon.

Lors de l'appui sur « Connexion » de la part du Client, « printData » est positionné à 1 et est envoyée au Cloud du Photon par l'intermédiaire de la fonction requeteState(printData). L'argument « etat » de la fonction sendData() correspond ainsi à « printData » :

```
1 int sendData(String etat) {
2     state = etat.toInt();
3     return 0;
4 }
```

5.2 Transfert de l'information du Photon au Zynq

Ainsi, lorsque le Photon détecte la connexion d'un Client au Serveur, il le signifiera au Zynq en émettant un front montant sur sa sortie digitale D2.

Lorsque « printData » passe à 1, « etat » passe à 1 ; ce qui occasionne le passage de « state » à 1 pour le Photon. Nous testons ensuite cette variable « state » pour émettre un front montant sur la sortie digitale D2 :

```
1 if(state != newstate) {
2     digitalWrite(interruptPin, HIGH);
3     delay(5);
4     digitalWrite(interruptPin, LOW);
5     state = newstate;
6 }
```

Le principe est strictement identique pour la « Déconnexion » hormis que la variable « printData » sera dans ce cas là fixée à 0.

Le Zynq possède 54 pins externes d'entrée/sortie qui sont pilotées depuis l'interface MIO (Multiplexed I/O).
Cette interface permet aux processeurs ARM d'avoir un accès direct à ces pins externes sans passer par le FPGA (nous n'avons pas besoin de modifier le design précédemment créé).
La documentation de la Zedboard nous a permis de voir que la pin 13 du MIO est connectée à la pin JE1 du connecteur PMOD JE (voir Figure 14).
La pin D2 du Photon sera donc reliée à la pin JE1 de la Zedboard. Nous devons donc configurer le Zynq pour générer une interruption lors de la réception d'un front montant sur la pin 13 du MIO.

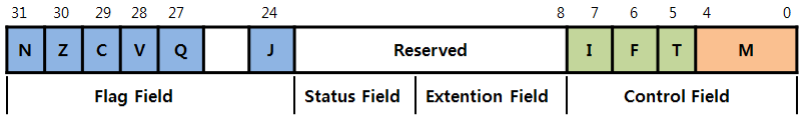
Cela se fait en plusieurs étapes successives :
- toutes les interruptions doivent être désactivées,
- le contrôleur d'interruption (General Interrupt Controller : GIC) doit être configuré en identifiant toutes les sources potentielles d'interruption et en leur assignant un identifiant unique et une priorité,
- chaque périphérique devant générer une interruption doit être configuré pour envoyer une requête d'interruption au GIC,
- toutes les interruptions doivent être réactivées.

Nous avons créé la bibliothèque gpio_interrupt_lib.h/c qui contient les 4 fonctions correspondants à ces 4 étapes. Une 5ème fonction y est définie et permet la configuration de la pin MIO 13 en tant que GPIO (General Purpose Input/Output).

À cela il faut ajouter la création de la fonction spéciale (IRQ Exception Handler) qui sera appelée lors de la réception d'une interruption. Nous détaillons ci-après les fonctions de la bibliothèque gpio_interrupt_lib.h/c.

5.2.1 Désactivation des interruptions

Nous devons écrire dans le registre CPSR (Current Program Status Register).



Flag Field	
N	Negative result from ALU
Z	Zero result from ALU
C	ALU operation caused Carry
V	ALU operation oVerflowed
Q	ALU operation saturated
J	Java Byte Code Execution

Control bits	
I	1: disables IRQ
F	1: disables FIQ
T	1: Thumb, 0: ARM

Mode bits M[4:0]	
0b10000	User
0b11111	System
0b10001	FIQ
0b10010	IRQ
0b10011	SVC(Supervisor)
0b10111	Abort
0b11011	Undefined

FIGURE 51: Registre CPSR

Les 5 bits de poids faible de ce registre définissent le mode du processeur. Afin que les ressources protégées soient accessibles, il est nécessaire de passer en mode « système », ce qui correspond à « 11111 ». On positionne également à 1 les bits F et I afin de désactiver les interruptions (IRQ et FIQ).
Cependant, la plupart des instructions ne peuvent pas modifier le CPSR (pour des raisons de sécurité et d'intégrité du système).
Les instructions assembleur spéciales MRS (pour Move to Register from Special) et MSR (pour Move to Special Register from Register) font partie des rares instructions permettant de lire/écrire dans le CPSR. Elles sont donc à manipuler avec précaution.
Nous utilisons la fonction suivante afin d'écrire la valeur 0xDF dans le CPSR :

```
1  void disable_ARM_A9_interrupts(){
2      uint32_t mode = 0xDF; // System mode [4:0] and IRQ disabled [7]
3      uint32_t read_cpsr=0; // used to read previous CPSR value
4      uint32_t bit_mask = 0xFF; // used to clear bottom 8 bits
5      __asm__ __volatile__("mrs %0, cpsr\n"  : "=r" (read_cpsr) );
6      __asm__ __volatile__("msr cpsr,%0\n" : : "r" ((read_cpsr & (~bit_mask))| mode));
7      return;
8  }
```

5.2.2 Configuration du General Interrupt Controller (GIC)

Étape 1 : Afin de configurer le GIC nous devons d'abord désactiver les interruptions GPIO. Cela se fait en écrivant les valeurs appropriées dans les registres ICDIPTR et ICDICER qui permettent de désactiver le suivi d'une interruption vers l'interface CPU.

Étape 2 : Ensuite, le registre de distribution des interruptions ICDDCR doit être désactivé afin de pouvoir modifier d'autres registres.

Étape 3 : Nous devons assigner un niveau de priorité à l'interruption en écrivant une valeur comprise entre 0 (plus forte priorité) et 0xFF (plus faible priorité) dans le registre ICDIPR (nous ne comptons pas utiliser d'autres sources d'interruption donc la valeur a peu d'importance).

Étape 4 : Nous pouvons alors réactiver le suivi d'interruption vers le processeur.

Étape 5 : Nous définissons la sensibilité de l'interruption (front montant).

Étape 6 : Nous activons l'interruption via le registre ICDISER (registre pendant de ICDICER).

Étape 7 : Nous définissons le niveau de priorité du processeur via le registre ICCPMR. Ce niveau doit simplement être inférieur (en priorité donc supérieur en valeur) à celui de notre interruption.

Étape 8 : Activation des interruptions via le registre de contrôle de l'interface CPU ICCICR.

Étape 9 : Ré-activation du distributeur d'interruption (ICDDCR).

Voici la fonction complète :

```
1  void configure_GIC(){
2      *((uint32_t*) ICDIPTR_BASEADDR+0x34/4) = 0x00000000; // Etape 1
3      *((uint32_t*) ICDICER_BASEADDR+0x04/4) = 0x00000000; // Etape 1
4      *((uint32_t*) ICDDCR_BASEADDR) = 0x0; // Etape 2
5      *((uint32_t*) ICDIPR_BASEADDR+0x34/4) = 0x000000A0; // Etape 3
6      *((uint32_t*) ICDIPTR_BASEADDR+0x0000034/4) = 0x00000001; // Etape 4
7      *((uint32_t*) ICDICFR_BASEADDR+0x00000C/4) = 0x00000100; // Etape 5
8      *((uint32_t *) ICDISER_BASEADDR+0x4/4) = 0x00100000; // Etape 6
9      *((uint32_t*) ICCPMR_BASEADDR) = 0xFF; // Etape 7
10     *((uint32_t*) ICCICR_BASEADDR) = 0x3; // Etape 8
11     *((uint32_t*) ICDDCR_BASEADDR) = 0x1; // Etape 9
12     return;
13 }
```

5.2.3 Configuration de l'interruption du GPIO

Maintenant que le GIC est configuré pour rediriger les interruptions du GPIO vers le CPU, nous devons configurer le GPIO afin qu'il génère une interruption.

Étape 1 : Désactivation des interruptions GPIO (registres GPIO_INT_DIS_0 et GPIO_INT_DIS_1).

Étape 2 : Effacer le registre de statut des interruptions (registre GPIO_INT_STAT_0).

Étape 3 : Définition du type d'interruption (front ou niveau), via le registre GPIO_INT_TYPE_0. Dans notre cas il s'agira du type front.

Étape 4 : Définition de la sensibilité d'interruption (montant/haut ou descendant/bas), via le registre GPIO_INT_POL_0. Dans notre cas il s'agira de front montant.

Étape 5 : Dans le cas où nous voudrions qu'une interruption soit déclenchée quel que soit le type/sensibilité, tous les bits du registre GPIO_INT_ANY_0 doivent être positionnés à 1 et la configuration précédemment définie (via GPIO_INT_TYPE_0 et GPIO_INT_POL_0) est ignorée. Ce n'est pas ce que nous voulons, nous lui attribuons donc la valeur 0.

Étape 6 : Activation de l'interruption du MIO13 (pin sur laquelle est branchée le signal du Photon). Nous devons positionner à 1 le 13ème bit du registre GPIO_INT_EN_0.

Voici la fonction complète :

```
1  void Initialize_GPIO_Interrupts(){
2      *((uint32_t*) GPIO_INT_DIS_0) = 0xFFFFFFFF; // Etape 1
3      *((uint32_t*) GPIO_INT_DIS_1) = 0xFFFFFFFF; // Etape 1
4      *((uint32_t*) GPIO_INT_STAT_0) = 0xFFFFFFFF; // Etape 2
5      *((uint32_t*) GPIO_INT_TYPE_0) = 0x02000; // Etape 3
6      *((uint32_t*) GPIO_INT_POL_0) = 0x002000; // Etape 4
7      *((uint32_t*) GPIO_INT_ANY_0) = 0x000000; // Etape 5
8      *((uint32_t*) GPIO_INT_EN_0) = 0x002000; // Etape 6
9  }
```

5.2.4 Réactivation des interruptions

Cette fonction positionne le bit IRQ à 1 dans le registre CPSR afin d'activer les interruptions :

```
1  void enable_ARM_A9_interrupts(){
2      uint32_t read_cpsr=0; // used to read previous CPSR value
3      uint32_t mode = 0x5F; // System mode [4:0] and IRQ enabled [7]
4      uint32_t bit_mask = 0xFF; // used to clear bottom 8 bits
5      __asm__ __volatile__("mrs    %0, cpsr\n"  : "=r" (read_cpsr) );
6      __asm__ __volatile__("msr    cpsr,%0\n" : : "r" ((read_cpsr & (~bit_mask))| mode)
       );
7      return;
8  }
```

5.2.5 Configuration de la pin MIO

La dernière fonction de la librairie gpio_interrupt_lib.h/c ne concerne pas l'interruption mais simplement la configuration de la pin MIO13 en entrée (registre GPIO_DIRM_0) et configure également la désactivation de résistance de PULL UP et le type de buffer (LVCMOS33) via le registre MIO_PIN_13.
Avant ces écritures il est cependant nécessaire d'écrire une clé spéciale (valeur 0xDF0D) dans le registre SLCR_UNLOCK afin de permettre l'écriture dans les registres SLCR dont MIO_PIN_13 fait partie.
Voici donc la fonction :

```
1  void configure_MIO_pin() {
2      /*
3       * Write unlock code to enable writing
4       * into System Level Control Unlock Register
5       */
6      *((uint32_t *) 0xF8000000+0x8/4) = 0x0000DF0D;
7      // Configure MIO pins
8      *((uint32_t*) MIO_PIN_13) = 0x00000600;
9      *((uint32_t*) GPIO_DIRM_0) = 0x00000000;
10 }
```

5.2.6 Traitement de l'interruption

Maintenant que notre interruption est configurée, nous pouvons écrire notre gestionnaire d'interruption IRQ_Handler().
Nous lions cette fonction à la réception d'interruption à l'aide de la fonction Xil_ExceptionRegisterHandler() fournie par la librairie Xilinx dans le fichier « xil_exception.h ».
De cette manière, la fonction IRQ_Handler() sera exécutée à chaque fois que le CPU recevra une interruption.

Voici la fonction IRQ_Handler() :

```
1  void IRQ_Handler(void *data){
2      uint32_t GPIO_INT = *((uint32_t*)GPIO_INT_STAT_0);
3      uint32_t interrupt_ID = *((uint32_t*)ICCIAR_BASEADDR);
4      uint32_t MIO_ID = 0x02000 & GPIO_INT;
5      if ( ( interrupt_ID == 52 ) && MIO_ID ) { // check if interrupt is from the GPIO
        MIO13
6          clientConnected = !clientConnected;
7      }
8      *((uint32_t*)ICCEOIR_BASEADDR) = interrupt_ID;
9      *((uint32_t*)GPIO_INT_STAT_0) = 0xFFFFFF;
10 }
```

Dans un premier temps, nous identifions la source d'interruption en lisant le registre ICCIAR. S'il s'agit bien d'une interruption provenant du Photon, nous changeons la valeur du booléen clientConnected qui sera testée dans le programme principal afin de déterminer si la mesure doit être effectuée.

Tout ce qu'il reste à faire est de signifier que l'interruption a bien été traitée. Pour cela nous devons effacer le registre de statut du GPIO en positionnant à 1 tous les bits du registre GPIO_INT_STAT_0, signifier la fin de l'interruption en écrivant l'identifiant de l'interruption dans le registre ICCEOIR.

Finalement nous avons écrit un code qui s'appuie sur les librairies précédemment définies et qui permet l'acquisition et la transmission de données uniquement lorsque le Photon le signifie.
Vous trouverez page suivante une version simplifiée.

5.3 Application finale

```
int config( void );
void IRQ_Handler(void *data);

XIicPs Iic0;
int clientConnected = 0;

int main()
{
    int Status;
    u8 tempBuffer[ 2 ];

    init_platform();

    config();

    while( 1 ) {
        if( clientConnected ) {
            temp_measure( &Iic0, tempBuffer );
            photon_send( &Iic0, tempBuffer );
        }
        usleep(2000000);
    }

    cleanup_platform();
    return 0;
}

int config( void ) {
    configure_MIO_pin();
    /*
     * GPIO interrupt config
     */
    disable_ARM_A9_interrupts();
    configure_GIC();
    Initialize_GPIO_Interrupts();
    Xil_ExceptionRegisterHandler(5, IRQ_Handler, NULL);
    enable_ARM_A9_interrupts();
    /*
     * I2C config
     */
    i2c_config(IIC0_DEVICE_ID, &Iic0);
    /*
     * Temperature sensor config
     */
    temp_configMeasure( &Iic0, TEMP_RESOLUTION_12BITS );

    return XST_SUCCESS;
}

void IRQ_Handler(void *data){
    uint32_t GPIO_INT = *((uint32_t*)GPIO_INT_STAT_0);
    uint32_t interrupt_ID = *((uint32_t*)ICCIAR_BASEADDR);
    uint32_t MIO_ID = 0x02000 & GPIO_INT;
    if ( ( interrupt_ID == 52 ) && MIO_ID ) { // check if interrupt is from the GPIO
     MIO13
```

```
55              clientConnected = !clientConnected;
56          }
57          *((uint32_t*)ICCEOIR_BASEADDR) = interrupt_ID;
58          *((uint32_t*)GPIO_INT_STAT_0) = 0xFFFFFF;
59      }
```

Critique :

Cette méthode ne s'avère cependant pas suffisante en terme d'économie d'énergie puisque dans la librairie Xilinx la fonction usleep() est implémentée pour l'architectire ARM Cortex A9 comme une boucle active et ne permet pas d'endormir réellement le processeur.
Il aurait fallu pour cela mettre en place la procédure d'entrée en mode « sleep » décrite dans le Zynq-7000 SoC Technical Reference Manual (`https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf`) page 684 et utiliser l'instruction WFI (Wait For Interrupt) qui permet au Zynq endormi d'être réveillé par une interruption.

6 Conclusion

Nous obtenons finalement une plateforme IoT fonctionnelle de l'acquisition de données, à la visualisation en passant par la transmission.

Récapitulons les 3 étapes de notre développement :

- Acquisition de données du capteur de température PMOD TMP3 via le protocole I2C sur architecture :
  - FPGA : implémentation d'un contrôleur I2C et d'une machine d'états permettant l'acquisition de la température.
  - Zynq : configuration et utilisation d'un contrôleur I2C, développement d'une application standalone en C permettant la configuration et l'acquisition de la mesure.
- Transmission de données au Particle Photon via le protocole I2C sur architecture :
  - FPGA : évolution de la machine d'états permettant la transmission des données précédemment acquises.
  - Zynq : enrichissement de l'application précédente pour gérer la transmission de données au Photon.
- Visualisation des données sur un serveur Web :
  Plusieurs concepts ont été abordés :
  - Mise en place d'une Application Web et d'un Serveur Web Local en PHP.
  - Utilisation de Xampp sur Linux.
  - Mise en place d'un proxy.
  - Utilisation d'Ajax.
  - Création d'un script de déchiffrement de données côté Serveur.
  - Utilisation d'une bibliothèque graphique Javascript.

À ces 3 étapes élémentaires nous avons ajouté une dimension de sécurité, avec la gestion du chiffrement AES sur architecture :

- FPGA : intégration d'un module de cryptographie symétrique AES 128 CTR, modification de la machine d'états permettant l'envoi des données au Photon après chiffrement.
- Zynq : Export du module AES en IP de type AXI4, modification de l'application permettant l'envoi des données au Photon après chiffrement.

Nous avons également tenté de limiter la consommation d'énergie sur l'architecture Zynq en ne réalisant des mesures que lorsque le client est connecté au serveur Web.

Les perspectives d'amélioration du projet sont les suivantes :

- Mise en place d'un module de cryptographie asymétrique de type RSA permettant l'échange de la clé symétrique entre le serveur Web et le module d'acquisition.
- Mise en œuvre de l'endormissement réel du Zynq et utilisation de l'instruction WFI (Wait For Interrupt) pour optimiser l'économie énergétique.
- Évaluation de la consommation de chaque plateforme d'acquisition/transmission (FPGA et Zynq) à corréler avec le temps de développement.
- Intégration de nouveaux capteurs.
- Hébergement du serveur.

7 Bibliographie

1) http://ww1.microchip.com/downloads/en/DeviceDoc/21935D.pdf
2) https://blog.nicolashachet.com/niveaux/confirme/larchitecture-rest-expliquee-en-5-regles/
3) https://docs.particle.io/reference/device-cloud/api/
4) http://www.fpgadeveloper.com/2014/08/creating-a-custom-ip-block-in-vivado.html
5) https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf
6) https://www.xilinx.com/support/documentation/user_guides/ug1181-zynq-7000-architecture-porting.pdf
7) https://www.xilinx.com/support/documentation/university/XUP%20Boards/XUPZedBoard/documentation/ZedBoard_RevC.1_Schematic_130129.pdf
8) http://zedboard.org/sites/default/files/documentations/ZedBoard_HW_UG_v2_2.pdf
9) http://www.markharvey.info/des/aesctr_128_hls/aesctr_128_hls.html
10) https://www.realdigital.org/doc/99a6a9e6108fd00fa7700d5817cd8e0b
11) https://github.com/Xilinx/embeddedsw/tree/master/XilinxProcessorIPLib/drivers