Projet IMA4 - Capteur de Pollution
COMPTE-RENDU FINAL

Pierre Guigo

Introduction

Lors de ce semestre, j'ai mené un projet qui vise la réalisation d'un capteur de pollution, afin de rattraper le projet qui s'était déroulé lors du semestre 8 et qui n'avait pas été mené à terme. Ce projet, reprend les travaux qui ont été réalisés par Antoine Branquart l'année dernière et de les mener également à terme, tout en apprenant des potentielles erreurs commises.

Par ailleurs, pour la réalisation de ce projet, je me suis également basé sur un document universitaire réalisé par des chercheurs de l'Université de Gent en Belgique.

Enfin, les différentes étapes de mon projet ont été la définition d'un cahier des charges en accord avec M.Boé. Ensuite, la recherche de carte STM32 qui accueille le dispositif électronique de détection de pollution. Puis, la réalisation de la carte Potentiostat. Pour terminer, les différents tests électroniques pour s'assurer la fonctionnalité de la carte.

Mon rapport reprendra donc les différentes étapes de mon projet.

Sommaire

1 Cahier des charges

Lors d'une première entrevue avec M.Boé, nous avons définit un cahier des charges. M.Boé m'a indiqué que le projet s'inscrit dans l'internet des objets et vise donc la réalisation d'un potentiostat (dispositif électronique qui permet de "cartographier" les polluants dans un liquide). Ce potentiostat, on peut l'imaginer, se situerait près d'une source d'eau, et serait alimenté par une batterie ou une source d'énergie renouvelable comme un panneau solaire. La réalisation d'un système économique en énergie. Par ailleurs, il enverrait via une technologie sans fil, périodiquement, les rapports des tests que le dispositif aurait réalisé. Ces rapport seraient ensuite traités par un opérateur. Les synoptique suivant résume le projet :

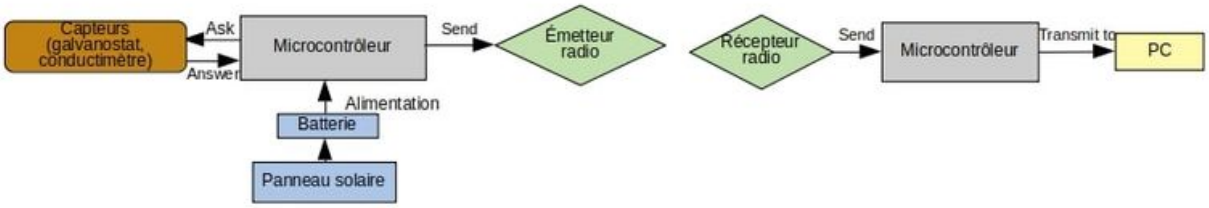

Figure 1: Synoptique initial du projet

Le potentiostat sera donc une carte de type shield à connecter directement à un microcontrôleur. Pour le choix de sélection de celui-ci, comme nous sommes sur un projet qui vise à consommer le moins d'énergie possible, M.Boé m'a orienté vers des microcontrôleur de chez STM32, car ils ont une faible consommation d'énergie. De plus, ce microcontrôleur doit être capable de communiquer via un bus SPI avec le potentiostat. Enfin, il doit pouvoir intéragir avec un module radio afin d'envoyer les rapports via une liaison sans fil.

La suite du projet fut donc les différentes recherches à propos du potentiostat ainsi que la recherche d'un microcontrôleur adéquat.

2 Potentiostat

Avant toute chose, il est nécéssaire d'expliquer ce qu'est un potentiostat, car c'est sur ce dispositif que le projet repose.

Un potentiostat est un outil capable de réaliser des expériences de voltamétrie cyclique simple. C'est une méthode d'électro-analyse basée sur la mesure du flux de courant résultant de la réduction ou de l'oxydation des composés tests présents en solution (dans de l'eau) sous l'effet d'une variation contrôlée de la différence de potentiel entre deux électrodes spécifiques.

Un potentiostat est composé de 3 électrodes :

- Un électrode de Travail "ET" (en platine)
- Une contre électrode "CE" (en platine)
- Une électrode de référence "ECS" (au calomel saturé)

Le principe est d'imposer une différnce de potentiel entre l'ET et l'ER et de mesurer le courant traversant l'ET.

Le schéma suivant permet de comprendre son fonctionnement :

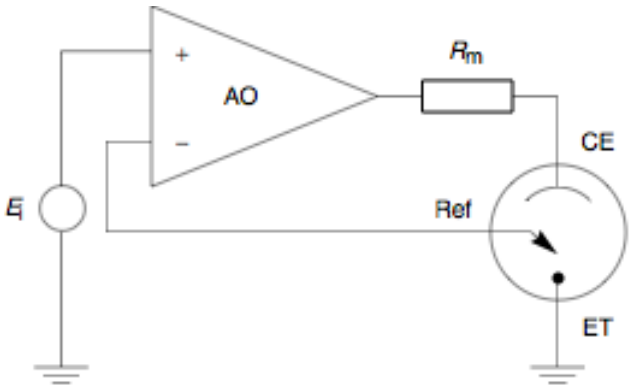

Figure 2: Schéma de principe d'un potentiostat

Sur le schéma, on peut voir que l'AOP permet de maintenir la différence de potentiel entre la référence et l'électrode de travail, aussi proche possible du potentiel d'entrée de la source (matérialisé içi par la source E_i).

3 Technologie LoRa

Par ailleurs, j'introduit içi le protocole LoRa qui sera utilisé pour envoyer des informations entre la doublette potentiostat/microcontrôleur et un recepteur qui simulera la transmission des rapports du potentiostat.

Le protocole LoRa ou LoRaWAN(Long Range Wide-area network) permet la communication à bas débit d'objet à faible consommation d'énergie facilitant l'internet des objets. C'est un protocole crée par la startup grenobloise Cycléo et rachetée par Semtech en 2012. Ce protocole utilise une technique de modulation par étalement de spectre de type Chirp spread spectrum propriétaire appelée LoRa.

Un réseau LoRaWAN est composé d'équipements sans-fils basse consommation qui interagissent avec des serveurs applicatifs au travers de passerelles. La modulation utilisée entre les équipements et les passerelles est LoRa. La communication entre les passerelles et les serveurs est établie via le protocole IP au moyen d'un réseau de collecte Ethernet ou 3G. La topologie d'un réseau LoRaWAN est en étoile car un serveur applicatif est connecté à une multitude de passerelles elles-mêmes connectées à une multitude d'équipements.

4 Recherche microcontrôleur STM32

La sélection du microcontrôleur qui allait acceuillir le potentiostat était un point important du projet. Il devait être peu énergivore, avoir à sa disposition des bus SPI pour la communication avec le potentiostat, ainsi que sa compatibilité avec un module LoRa afin d'envoyer les rapports.

Lors de mes recherches je suis arrivé sur une carte de développement (B-L072Z-LRWAN1) crée par STMicroélectronics qui intègre directement un module LoRa sur sa carte. Il existe de plus des librairies LoRa dédiés à cette carte sur la plateforme Mbed. En outre, elle embarque un microcontrôleur STM32L0, qui est la branche des microcontrôleur STM32 qui consomme le moins d'énergie. Cette carte était donc parfaitement adaptée à mon projet. Coup de chance, M.Vantroys en possédait une.

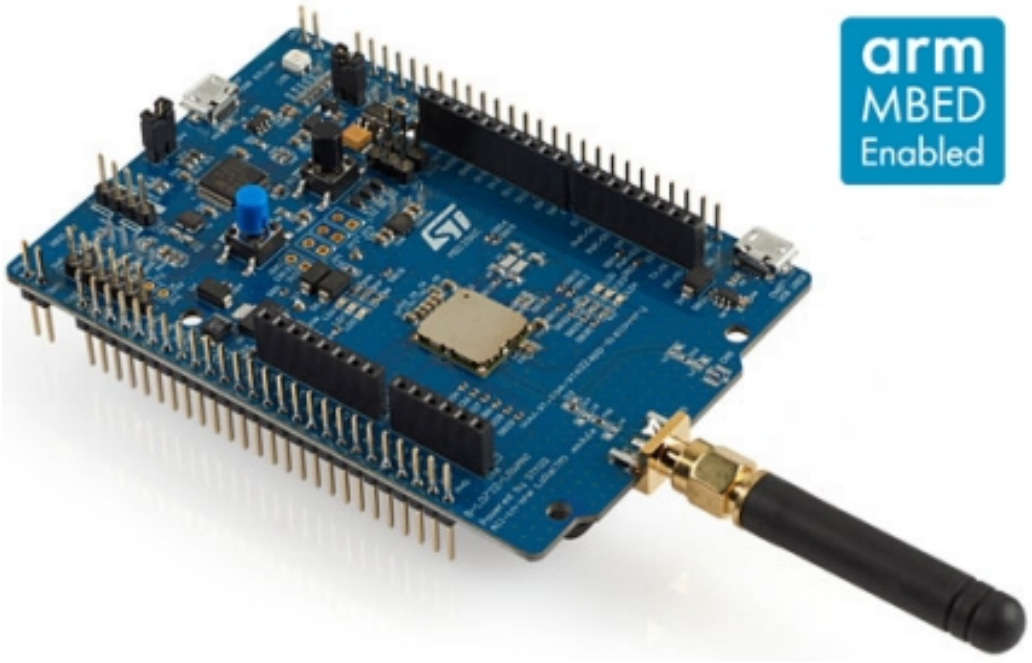

Figure 3: Carte de developpement B-L072Z-LRWAN1

5 Explication Schéma Potentiostat

Comme je l'ai indiqué dans l'introduction, je me suis basé sur les travaux d'universitaire belges, notamment du schéma qu'ils ont réalisé.

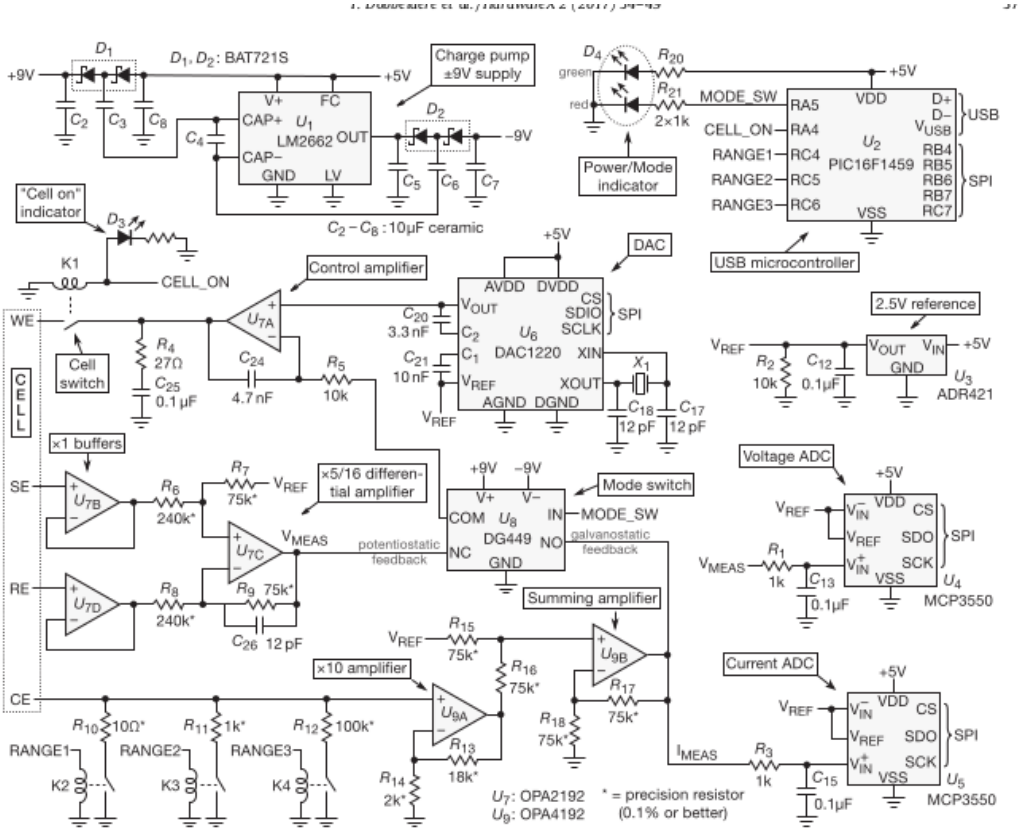

Figure 4: Schéma du potentiostat réalisé

Etudions ce schéma plus en détail.

5.1 Alimentation

Pour l'alimentation, le potentiostat utilise un LM2662 qui va transformer le 5V transmis par la carte du Micro-Contrôleur, en 9V, afin d'alimenter les circuits intégrés commes les AOP, ou le switch DG449.

De plus, Il y a un autre circuit (ADR421) qui va permettre de fixer le Vref (2.5V) qui va être utile pour le DAC et les deux ADC.

5.2 Circuit Analogique

Le circuit analogique est rendu possible grâce aux AOP, OPA2192 & OPA4192. U7a, sur le schéma, permet de comparer la tension imposée par le microcontrôleur via le DAC à la tension de retour du potentiostat ou galvanostat selon le mode, et conduit l'électrode de travail jusqu'a ce que les deux valeurs soient égales. La résistance R5 et la capacité C24 permettent de fixer la bande passante de U7a. Les valeurs donnent une fréquence d'environ 3Khz, ce qui reste largement supérieur à la constante de temps d'une mesure, et donc ne risque pas d'avoir un impact sur la mesure.

Sur le schéma, on remarque la présence d'un relais K1. Ce relais permet d'activer l'électrode de travail lorsque le microcontrôleur envoie le signal "CELL_ON", et une Led permet de signaler la mise en route de la mesure. Comme je me base sur un microcontrôleur différent, je pense remplacer le signal de commande par un interrupteur. L'interrupteur connectera le reste du circuit à du 5V. Lorsque ça sera activé, le courant passera dans la bobine ainsi que dans la led, ce qui sera équivalent au circuit antérieur, et ainsi activer la mesure via l'électrode de travail.

Le retour du potentiostat/galvanostat est récupéré par les AOP U7b, U7d et U7c. U7d et U7b permettent d'isoler les électrodes et le reste du montage grâce à leurs grandes impédances d'entrée, et donc d'avoir aucun courant entre l'électrode SE et l'électrode RE. Les tensions récupérées sont ensuite transmises à U7c qui fonctionne en amplificateur différentiel.

Avec Vref = 2.5V et R7 = 75kOmh & R6 = 240kOmh, la différence de potentiel entre l'électrode SE et l'électrode RE qui en entrée (avant U7c) varie de -8V et +8V, et en sortie de 0-5V, ce qui permet de matcher correctement avec les entrées des deux ADC qui vont permettre de transmettre les données au microcontrôleur.

Le retour galvanique est acquis grâce aux résistances de shunt qui permettent de transformer le courant de retour en une tension équivalente. Les relais sont là pour choisir la précision du galvanostat. Dans la plus grande range, le courant est compris entre -25mA et 25mA (avec R10). U9a convertit ce courant avec l'aide de la résistance associée en une tension comprise entre 0 et 5V, idéal pour les ADC ensuite. Chaque résistance divise par 10 le courant capté et permet une meilleur précision.

Comme le galvanostat n'est pas l'objectif premier de la carte, je vais enlever les relais (qui on un certain coût) et conserver seulement la résistance R10 directement connectée à la masse.

Enfin, les signaux retournés par les électrodes et traités par les AOP U7 et U9 sont envoyés vers DG449, qui est un switch analogique. Il permet de sélectionner dans quel mode se trouve la carte. Lorsque l'entrée IN est à 0, le switch met la carte en mode potentiostat. Au contraire, si IN est à 1, on est en mode galvanostat.

Comme l'entrée IN est contrôlée par le microcontrôleur, j'ai choisi de modifier cette partie. A la place, je vais mettre deux résistance de 10MOhm et 10MOhm avec un cavalier. Lorsque le cavalier sera ouvert, l'entrée IN connectée à la résistance de 10M sera forcée à 0 (la masse), et lorsque le cavalier sera fermé, le niveau logique 1 (5V) sera sur l'entrée IN, avec un courant limité à 0.5μA, ce qui est inférieur à l'ampérage maximum que l'entrée peut recevoir (selon la documentation). On pourra alors choisir entre le mode galvanostat ou potentiostat avec ce cavalier.

5.3 Conversion Analogique - Numérique

Les retours Vmes et Imes sont ensuite introduits dans les deux ADC U4 et U5 qui sont des MCP3550 qui sont d'une grande précision et permettent de filtrer le bruit d'entrée. En entrée de chaque ADC, un filtre est ajouté afin de réduire le bruit et d'améliorer la conversion.

5.4 Conversion Numérique - Analogique

Le DAC reçoit du Microcontrôleur les données numériques via le bus SPI. Lors de mes recherches, j'ai trouvé des bibliothèques sur Mbed dédiées à la carte de developpement B-L072Z-LRWAN1, ce qui facite la communication. Ensuite, le DAC envoie la tension de contrôle à U7a qui ensuite va comparer au valeur retournées par les electrodes.

6 Réalisation Carte Potentiostat

Ensuite, j'ai donc réalisé les schématiques du potentiostat sur Altium. J'ai modifié certaines parties par rapport au schéma initial. Les différents schématiques sont disponibles en annexe. Pour ce qui est des modifications, c'est l'absence des relais qui permettaient de choisir le calibre du retour galvanostatique, ainsi que du relais qui mettait la carte en fonctionnement. C'est également de pointe en metal, qui permettent à un sonde de tension de se brancher, et ainsi permettre l'observation du signal à l'oscilloscope. J'en ai mis à différents endroits sur la carte, notamment sur les signaux d'entrée des électrodes, ainsi que sur l'entrée du DAC et des ADC, tout comme pour les tensions 5V et +/-9V.

Ensuite, après avoir pris en compte les remarques pertinentes de M.Boé, j'ai réalisé le PCB de la carte. Comme l'objectif était de réaliser un shield, j'ai d'abord placé les connecteurs de la cartes à des endroits déterminés, afin que la carte puisse se connecter correctement. Ensuite, j'ai placé les composants modules par modules. C'est à dire d'abord la partie puissance, ensuite le DAC, puis la partie analogique et enfin les deux ADC avec le switch analogique. Après quelques échanges avec M.Boé pour quelques modifications, la carte a été tirée par M.Flamen. L'étape suivante fut la soudure & les tests des différents étages de la carte. Le pcb se trouve en annexe.

7 Tests

La procédure pour cette étape de mon projet a été de la manière suivante. Je soudais d'abord les composants de l'étage en question, puis des tests de continuité afin d'éviter des courts-circuit et ensuite la mise sous tension avec une alimentation de laboratoire pour voir s'il y a une consommation élevée de courant de la part de la carte. L'ordre des étages était le suivant :

- Circuit de puissance LM2662 & ADR421
- Convertisseur Numérique/Analogique DAC1220
- Circuit Analogique OPA4192 & OPA2192
- Convertisseurs Analogiques/Numériques MCP3550
- Switch Analogique DG449

Cependant pour le DAC, la partie de test fut un peu différente, je reviens en détail là dessus.

7.1 DAC1220

Pour la partie de test du DAC, j'ai d'abord soudé l'étage et après m'être assuré de l'absence de court-circuit, j'ai commencé à réaliser un code pour lui transmettre des données. En annexe, il y a des captures d'écran de sa documentation qui permet de comprendre plus en détail.

Ce qu'il faut retenir est que pour intéragir avec lui, il faut écrire dans les registres adéquats. Il existe d'abord le registre de Commande où on détermine dans quel sous-registre on va écrire et le nombre d'octet que l'on va écrire. Par exemple, si l'on souhaite écrire seulement 1 octet dans le registre DIR (soit sur la partie MSB, comme indiqué sur la documentation REGISTER), on va écrire dans le registre de commande (soit directement sur la pin MOSI) l'octet 0x02. (Documentation en annexe, figure 14)

Un autre registre important est celui de configuration. C'est le registre CR où en fonction des bits à 0 ou à 1, le DAC sera dans des modes différents. Dans la documentation, il y a une indication de mise en route du DAC. A chaque mise sous tension du DAC, il est nécéssaire de réaliser la suite de commande suivante :

- Réinitialiser le DAC via un pettern spécial sur la pin SCLK. (voir figure 13, en annexe)

- Mettre le DAC en mode calibration via le registre CMR en mettant les bits MD à 00 (MD1 à 0 et MD0 à 0). (voir figure 18, en annexe)
- Attendre la fin de la calibration et basculer en DAC en mode de conversion normal.
- Envoyer les valeurs à convertir sur le registre DIR. (voir figure 15, en annexe)

Le code que j'ai donc réalisé est le suivant. Il respecte l'ordre expliqué au dessus.

```
  DigitalOut CS1(PB_12);
  DigitalOut MISO(PB_14);
  DigitalOut MOSI(PB_15);
  DigitalOut SCLK(PB_13);

  void SPIDelay()
  {
    wait_us(200);
  }

  void ClockPulse()//Horloge de 2.5kHz
  {
    SCLK = HIGH;
    SPIDelay();
    SCLK = LOW;
    SPIDelay();
  }

  void EcrireOctetSPI_DAC(uint8_t octet)//Fonction recuperee dans la documentation du
      document universitaire
  {
    uint8_t bit_counter = 8;      // set bit count for byte
    do
    {
        MOSI = (octet&0x80)?HIGH:LOW;  // output most significant bit
        ClockPulse();                        // generate a clock pulse
        octet<<= 1;                       // shift byte to the left
    } while (--bit_counter);                 // repeat until 8 bits have been
        transmitted
}


  void DAC_reset(void)//fonction de reset du DAC, envoi de la trame sur la pin SCLK
  {
    CS1=0;
    SPIDelay();
    SCLK=HIGH;
    wait_us(600);//t16
    SCLK = LOW;
    wait_us(10);//t17
    SCLK = HIGH;
    SPIDelay();
    wait_us(1100);//t18
    SCLK = LOW;
    wait_us(10);//t17
    SCLK = HIGH;
    wait_us(2100);//t19
    SCLK = LOW;
    SPIDelay();
    SCLK = HIGH;
    SPIDelay();
  }

   void DAC1220_Ecrire3octet(const uint8_t address, const uint8_t byte1, const uint8_t
       byte2, const uint8_t byte3)
  {
    CS1 = LOW;
    SPIDelay();
    EcrireOctetSPI_DAC(64+address);
    EcrireOctetSPI_DAC(byte1);
    EcrireOctetSPI_DAC(byte2);
    EcrireOctetSPI_DAC(byte3);
    CS1= HIGH;
    SPIDelay();
  }

  void DAC1220_Ecrire2octet(const uint8_t address, const uint8_t byte1, const uint8_t byte2
      )
```

```
{
  CS1 = LOW;
  SPIDelay();
  EcrireOctetSPI_DAC(32+address);
  EcrireOctetSPI_DAC(byte1);
  EcrireOctetSPI_DAC(byte2);
  CS1= HIGH;
  SPIDelay();
}

void DAC_Calibration(void)//fonction de calibration
{
  CS1=LOW;
  SPIDelay();
  EcrireOctetSPI_DAC(0x24);
  EcrireOctetSPI_DAC(0x60);
  EcrireOctetSPI_DAC(0xA1);
  CS1=HIGH;
  SPIDelay();
}

void rampe(void)//Envoi d'une rampe de Tension
{
  uint16_t i;
  uint16_t i_msb, i_lsb;
  for(i=0x0000; i<=0xFFFF; i=i+16)
  {
    i_msb = i & 0xFF00;
    i_lsb = i & 0x00FF;
    DAC1220_Ecrire2octet(0, i_msb>>8, i_lsb);
    serial.printf("%x %x\n\r", i_msb>>8, i_lsb);
    wait_ms(100);
  }
}

void DAC_Normal()
{
  CS1=LOW;
  SPIDelay();
  DAC1220_Ecrire2octet(4,0x20,0x60);//Mode 16bits
  CS1=HIGH;
  SPIDelay();
}

int main()
{
  int i;
  serial.baud(9600);//Initialisation du port serie pour voir les valeurs envoyes
  serial.format(8);
  serial.printf("Initialisation !\n\r");
  SCLK=LOW;
  wait_ms(25);
  initDAC();
  for(i=0; i<10; i++)
  ClockPulse();
  while(1)
  {
    rampe();//On envoie la rampe de tension au DAC
    wait_ms(100);
  }
}
```

En sortie du DAC, lorsque la rampe de tension était envoyée, on voyait la tension de sortie s'incrémenter au fur et à mesure.

Ensuite, pour tester le montage final, une fois que la carte était complétement soudée, j'ai réalisé un montage de type dummy-cell, qui est un filtre RC, comme la figure 4 l'indique.

Une fois ce montage réalisé et connecté à mon potentiostat, j'ai observé au fur et à mesure que la tension de sortie du DAC incrémentait, la tension aux bornes de mon montage RC se chargeait, via la constante de temps qui est dûe à la valeur de la capacité et de la résistance, comme les lois de l'électronique peuvent le démontrer. Le potentiostat est donc fontionnel.

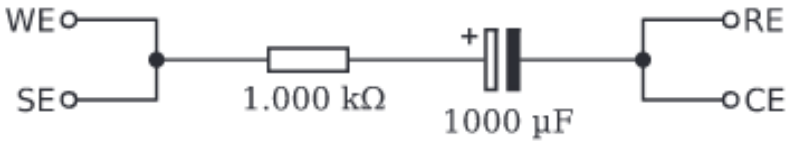

Figure 5: Montage Dummy cell

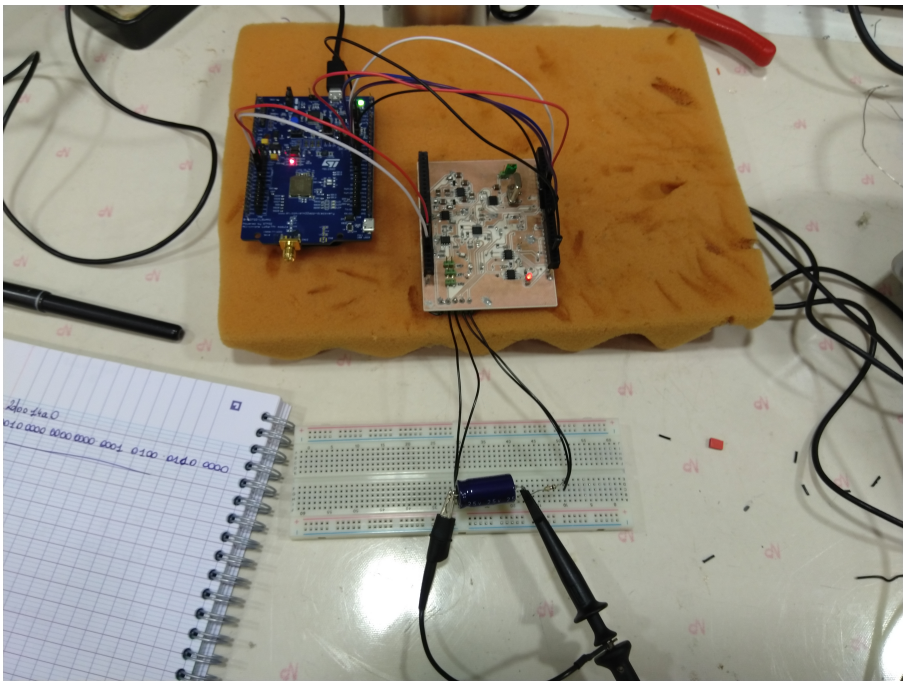

Figure 6: Montage Dummy cell avec le potentiostat

8 Problèmes rencontrés

Lors de cette partie de mon projet, j'ai recontré deux problèmes assez importants. Le premier était à propos du switch analogique. En effet, son empreinte sur la carte était de classe 6, or M.Flamen utilise une machine pour fiare les pcb qui peut faire des empreintes de classe 5 maximum. Donc, lors du tirage de la carte, l'empreinte est mal passée, et j'ai du au cutter délimiter les pattes du composants. Cependant, comme c'est assez petit, j'ai enlevé des parties de cuivre, ce qui à rendu la soudure du composant assez délicate. Au final, lors de la soudure du switch, tout le cuivre s'est arraché et il était impossible d'ajouter le switch au montage.

Pour régler ce soucis, j'ai donc shunter toute la partie galvanostat de la carte, et j'ai directement relié le retour potentiostatique à la tension de comparaison en sortie du DAC, sur la broche de U7a, au niveau de R5.

Par ailleurs, j'ai également eu des difficultés pour récupérer les valeurs en sortie des ADC. J'arrivais à obtenir des valeurs, cependant je n'étais pas en capacité de les comprendre, car elles étaient trop éloignés de celle qui sont attendues dans la documentation.

9 Communication LoRa

L'autre partie de mon projet était de réaliser la communication entre deux cartes, en utilisant le protocole LoRa. J'ai donc utilisé deux cartes LRWAN1, ainsi que la plateforme Mbed Os, qui permet de réaliser des codes et de compiler en ligne puis de télécharger le binaire pour le téléverser directement sur la carte. Le code se divise en plusieurs parties.

Tout d'abord, on définit les différentes variables du module LoRa :

```
#define RF_FREQUENCY            868000000   // Hz
#define TX_OUTPUT_POWER         14                  // 14 dBm
#define LORA_BANDWIDTH          125000   // LoRa default, details in SX1276::BandwidthMap
```

```
#define LORA_SPREADING_FACTOR   LORA_SF7
#define LORA_CODINGRATE         LORA_ERROR_CODING_RATE_4_5
#define LORA_PREAMBLE_LENGTH    6        // Same for Tx and Rx
#define LORA_SYMBOL_TIMEOUT     10        // Symbols
#define LORA_FIX_LENGTH_PAYLOAD_ON  false
#define LORA_FHSS_ENABLED       false
#define LORA_NB_SYMB_HOP        4
#define LORA_IQ_INVERSION_ON    false
#define LORA_CRC_ENABLED        true
#define RX_TIMEOUT_VALUE    3500    // in ms
#define BUFFER_SIZE       32        // Define the payload size here
```

Ces variables vont nous permettre de déterminer la fréquence des paquets LoRa, la bande passante, etc...

Ensuite, nous avons les différentes initialisations :

```
typedef enum {
  SLEEP = 0,
  RX_INIT,
  RX_INIT_WAIT,
  RX_ENUM,
  RX_ENUM_WAIT,
  TX_PING,
  RX_TIMEOUT,
  RX_ERROR,
  TX_INIT,
  TX_INIT_WAIT,
  TX_ENUM,
  TX_ENUM_WAIT,
  CAD,
  CAD_DONE
} AppStates_t;//Objet necessaire pour l'initialisation du module LoRa ensuite
DigitalOut myled1(LED1);
static RadioEvents_t RadioEvents;// Initialisation d'un objet de classe RadioEvents
Serial serial(SERIAL_TX, SERIAL_RX);// Initialisation d'un objet serial de classe Serial
SX1276Generic *Radio;// Initialisation d'un pointeur sur un objet de classe SX1276
const uint8_t PongMsg[] = { 0xFF, 0xFF, 0xFF, 0xFF};// Message a envoyer;
```

Puis, dans une fonction void init(), on initialise nos différents objets :

```
//On initialise notre Objet radio
Radio = new SX1276Generic(NULL, MURATA_SX1276,
          LORA_SPI_MOSI, LORA_SPI_MISO, LORA_SPI_SCLK, LORA_CS, LORA_RESET,
          LORA_DIO0, LORA_DIO1, LORA_DIO2, LORA_DIO3, LORA_DIO4, LORA_DIO5,
          LORA_ANT_RX, LORA_ANT_TX, LORA_ANT_BOOST, LORA_TCXO);

          //Initialisation du driver Radio

 RadioEvents.TxDone = OnTxDone;
 RadioEvents.RxDone = OnRxDone;
 RadioEvents.RxError = OnRxError;
 RadioEvents.TxTimeout = OnTxTimeout;
 RadioEvents.RxTimeout = OnRxTimeout;
 if (Radio->Init( &RadioEvents ) == false) {
   while(1) {
     serial.printf("Radio could not be detected!");
     wait( 1 );
   }
 }

 Radio->SetChannel(RF_FREQUENCY );

 Radio->SetTxConfig( MODEM_LORA, TX_OUTPUT_POWER, 0, LORA_BANDWIDTH,
 LORA_SPREADING_FACTOR, LORA_CODINGRATE,
 LORA_PREAMBLE_LENGTH, LORA_FIX_LENGTH_PAYLOAD_ON,
 LORA_CRC_ENABLED, LORA_FHSS_ENABLED, LORA_NB_SYMB_HOP,
 LORA_IQ_INVERSION_ON, 2000 );

 Radio->SetRxConfig( MODEM_LORA, LORA_BANDWIDTH, LORA_SPREADING_FACTOR,
 LORA_CODINGRATE, 0, LORA_PREAMBLE_LENGTH,
 LORA_SYMBOL_TIMEOUT, LORA_FIX_LENGTH_PAYLOAD_ON, 0,
 LORA_CRC_ENABLED, LORA_FHSS_ENABLED, LORA_NB_SYMB_HOP,
 LORA_IQ_INVERSION_ON, true );
```

Les fonctions de RadioEvents sont importantes, car c'est ce qui permet de savoir si un paquet est correctement reçu ou envoyé.

```
void OnTxDone(void *radio, void *a, void *b)
{
  Radio->Sleep( );
  serial.printf( "> OnTxDone\n\r" );
}

void OnRxDone(void *radio, void *a, void *b, uint8_t *payload, uint16_t size, int16_t
    rssi, int8_t snr)
{
  Radio->Sleep( );
  serial.printf( "> OnRxDone\n\r" );
}

void OnTxTimeout( void *radio, void *a, void *b)
{
  Radio->Sleep( );
  serial.printf( "> OnTxTimeout\n\r" );
}

void OnRxTimeout( void *radio, void *a, void *b)
{
  Radio->Sleep( );
  serial.printf( "> OnRxTimeout\n\r" );
}

void OnRxError( void *radio, void *a, void *b )
{
  Radio->Sleep( );
  serial.printf( "> OnRxError\n\r" );
}
```

Par exemple, la fonction void OnTxDone(void *radio, void *a, void *b) envoie sur le port série le messag ”>OnTxDone” lorsque l'envoie est correctement effectué.

C'est dans ces fonctions que les programmes d'envoi de paquet et de réception de paquet vont différentier.

Pour le programme de réception, la fonction OnRxDone est celle qui va afficher le message reçu :

```
void OnRxDone(void *radio, void *a, void *b, uint8_t *payload, uint16_t size, int16_t rssi,
     int8_t snr)
{
   serial.printf("0x%x, %d bytes\n\r", payload, size);// Affiche la valeur en hexa de la
       variable recue
   serial.printf("> OnRxDone ");
   Radio->Sleep( );
   BufferSize = size;
   memcpy(Buffer, payload, BufferSize );// On copie la variable recue dans la variable
       Buffer
   serial.printf("RssiValue=%d dBm, SnrValue=%d, size=%d\n\r", rssi, snr, size);
   serial.printf("Buffer : %s\n\r", Buffer);// On affiche la chaine de caractere
   if(size != BUFFER_SIZE) {
   Radio->Sleep( );
   serial.printf("> Payload error");
   return;}
}
```

Enfin, les deux mains des deux différents programmes sont :

```
  while(1) {
       memcpy(Buffer, chaine, sizeof(chaine)); //On copie la chaine de caractere chaine
           dans buffer
       wait_ms( 10 );
       Radio->Send( Buffer, BufferSize );//On envoie le buffer
       serial.printf("%s\n\r", Buffer);//Afficher la chaine de caractere a envoyer
       invert();   // inverser etat led1
   }//Main boucle envoie
```

```
   while(1) {
       Radio->Rx(23000);//Passer la carte en mode reception pendant un temps determine
       invert();//Inverser Led
   }//Main boucle reception
```

Lorsque je scrute le port série de mon ordinateur pour déterminer si l'émission et la réception des paquets fonctionne, j'obtiens ceci :

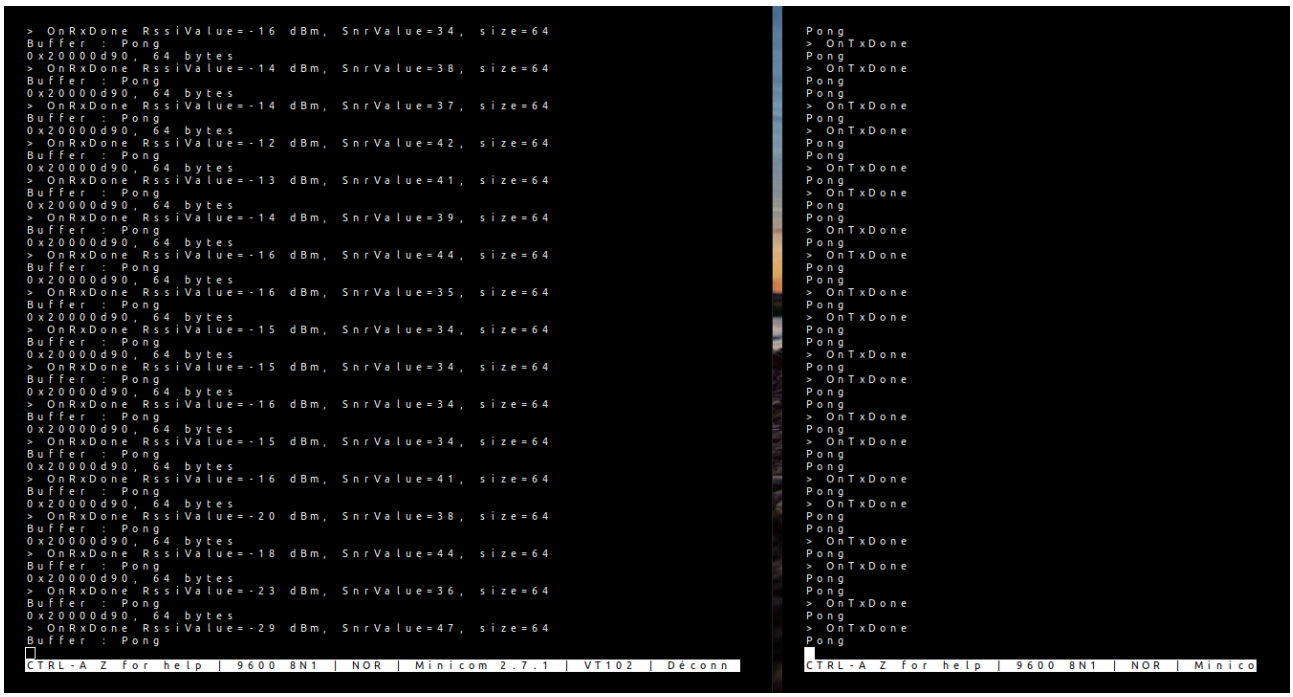

Figure 7: Schematique du Potentiostat 1

Cela fonctionne correctement !

10 Conclusion

Pour conclure sur ce projet, je peux affirmer que j'y ai pris beaucoup de plaisir, et qu'il m'a conforté dans mon choix de travailler dans le domaine de l'électronique. J'ai refait des tâches que je réalisais lors de mon stage au sein de Melexis cet été, notamment la réalisation de PCB. Par ailleurs, la phase de test est un moment assez intense, car c'est l'application de mes connaissances acquises lors de mon DUT et de ces 2-3 ans à Polytech.

Enfin, je souhaiterais remercier M.Boé pour son aide tout au long du projet, M.Flamen pour son écoute et sa patience pour toutes les questions que je lui ai posé. Enfin, M.Vantroys & M.Redon pour le prêt de matériel, notaments les cartes de developpement ST.

11 Annexe

A Schématique & PCB

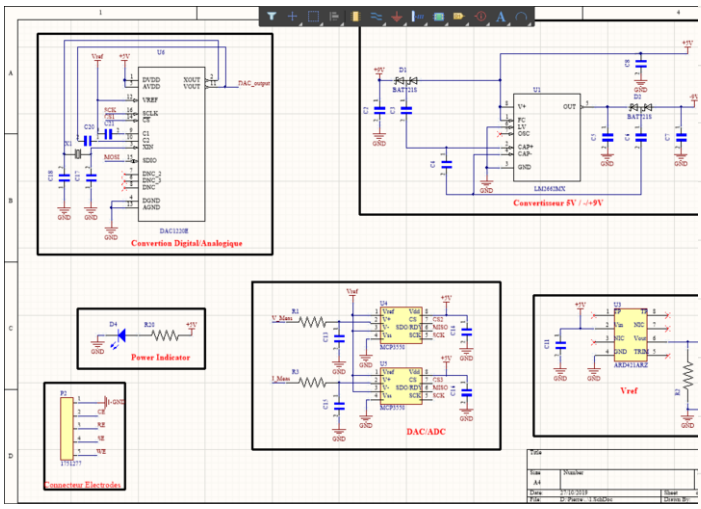

Figure 8: Schematique du Potentiostat 1

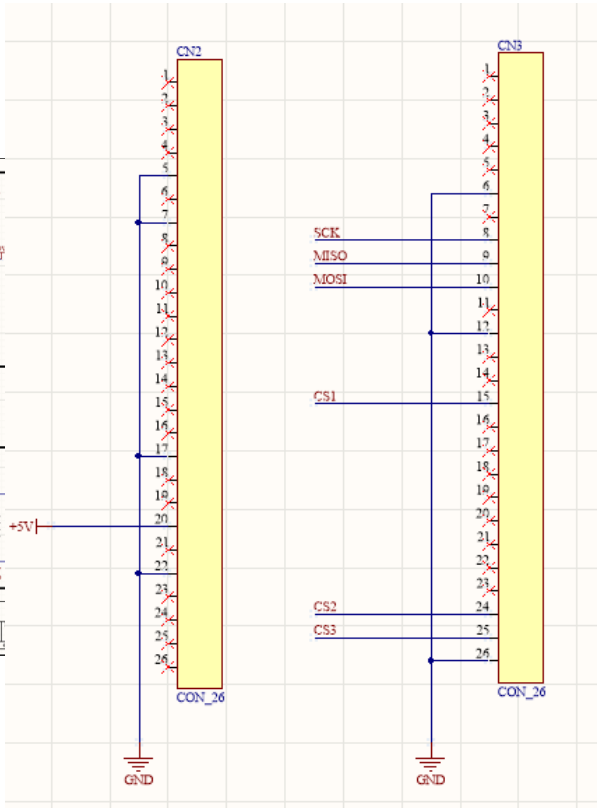

Figure 9: Schematique du Potentiostat 1

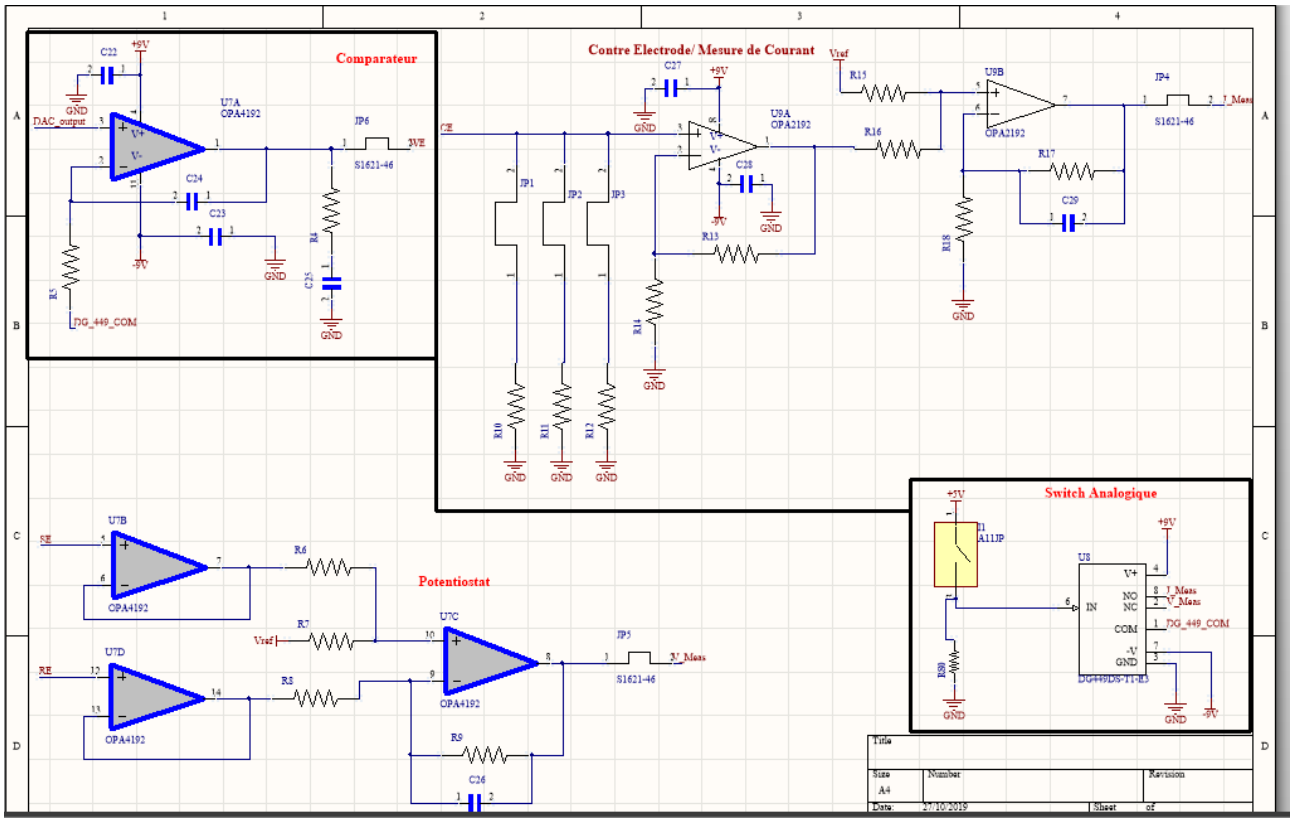

Figure 10: Schematique du Potentiostat 2

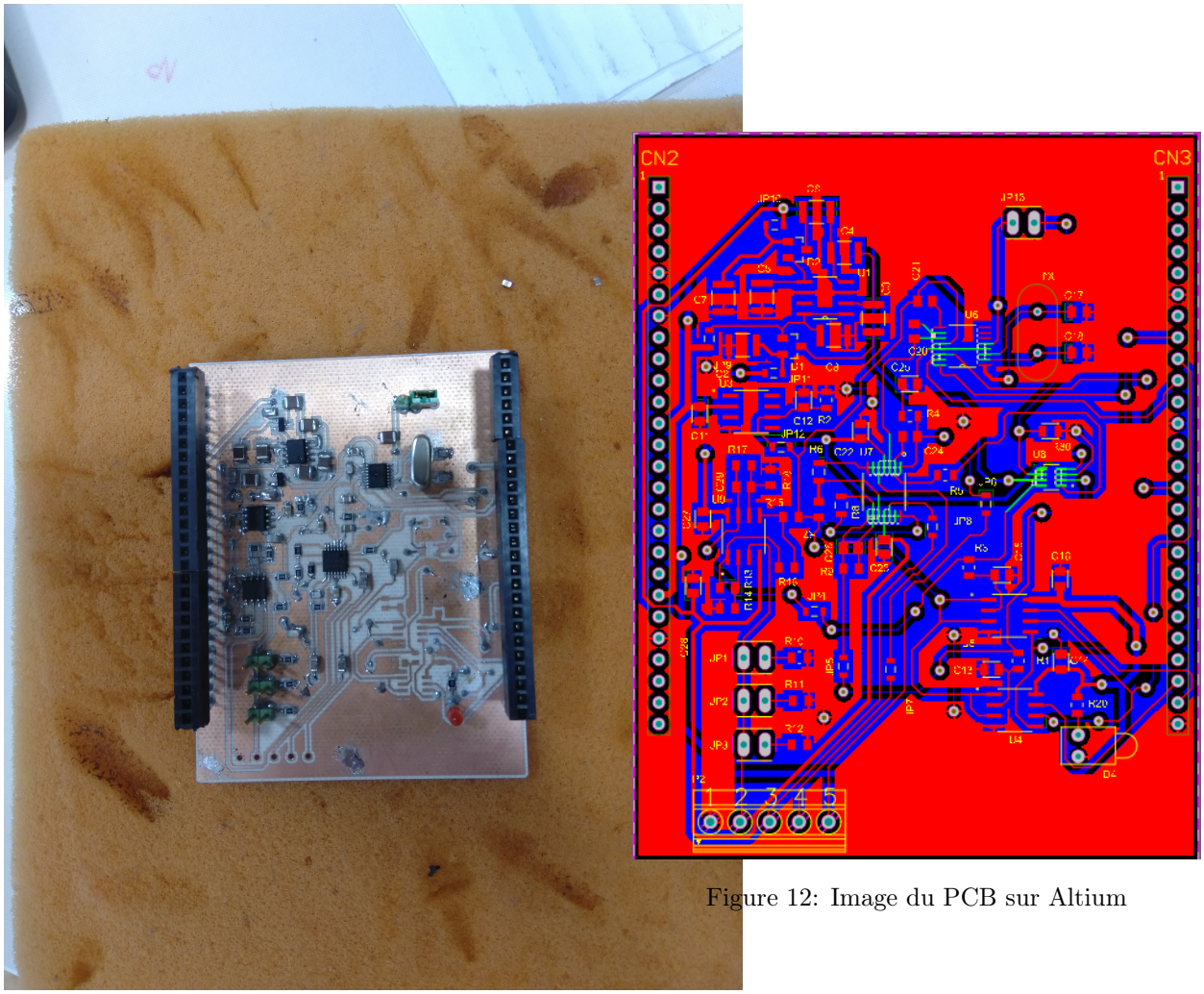

Figure 12: Image du PCB sur Altium

Figure 11: PCB lors de la soudure des composants

B Documentation DAC1220

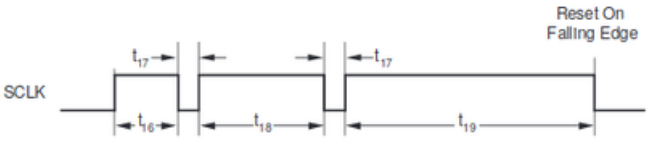

Figure 12. Resetting the DAC1220

Table 5. Reset Timing Characteristics

SYMBOL	DESCRIPTION	MIN	NOM	MAX	UNITS
t_{16}	First high period	512 × t_{XIN}		800 × t_{XIN}	ns
t_{17}	Low period	10 × t_{XIN}			ns
t_{18}	Second high period	1024 × t_{XIN}		1800 × t_{XIN}	ns
t_{19}	Third high period	2048 × t_{XIN}		2400 × t_{XIN}	ns

Figure 13: Pattern de Reset à envoyer sur la broche SCLK

Table 6. Command Byte Format

7	6	5	4	3	2	1	0
R/$\overline{W}$	MB		0	ADR			

Table 7. Command Byte Bits

BIT(S)	NAME	VALUE	DESCRIPTION
7	R/$\overline{W}$	0	Write to register map
		1	Read from register map
6–5	MB		Number of bytes to read or write
		00b	1 byte
		01b	2 bytes
		10b	3 bytes
		11b	Reserved; do not use
3–0	ADR	0–15	Start address in register map

Figure 14: Registre CR de configuration

Table 8. Transfer Functions

DATA FORMAT	20-BIT MODE	16-BIT MODE
Offset two's complement	$V_{OUT} = 2V_{REF}\frac{code + 2^{19}}{2^{20}}$	$V_{OUT} = 2V_{REF}\frac{code + 2^{15}}{2^{16}}$
Straight binary	$V_{OUT} = 2V_{REF}\frac{code}{2^{20}}$	$V_{OUT} = 2V_{REF}\frac{code}{2^{16}}$

Table 9. Example Output Voltages

APPROXIMATE OUTPUT VOLTAGE	RESOLUTION	DATA FORMAT	CODE	DIR CONTENT (1)
0V	16-bit	Two's complement	8000h	8000xxh
		Straight binary	0000h	0000xxh
	20-bit	Two's complement	8000h	80000xh
		Straight binary	0000h	00000xh
2.5V	16-bit	Two's complement	0000h	0000xxh
		Straight binary	8000h	8000xxh
	20-bit	Two's complement	0000h	00000xh
		Straight binary	8000h	80000xh
5V	16-bit	Two's complement	7FFFh	7FFFxxh
		Straight binary	FFFFh	FFFFxxh
	20-bit	Two's complement	7FFFFh	7FFFFxh
		Straight binary	FFFFFh	FFFFFxh

(1) x = Do not care

Figure 15: Exemple de valeur de conversion pour le registre DIR

REGISTERS

The register map is shown in Table 11.

Table 11. Register Memory Map

ADDRESS	CONTENT
0	DIR byte 2 (MSB)
1	DIR byte 1
2	DIR byte 0 (LSB)
3	Reserved
4	CMR byte 1 (MSB)
5	CMR byte 0 (LSB)
6	Reserved
7	Reserved
8	OCR byte 2 (MSB)
9	OCR byte 1
10	OCR byte 0 (LSB)
11	Reserved
12	FCR byte 2 (MSB)
13	FCR byte 1
14	FCR byte 0 (LSB)
15	Reserved

Figure 16: Tableau avec les adresses des différents registres

Table 12. Command Register

15	14	13	12	11	10	9	8
ADPT	CALPIN	Reserved	Reserved	Reserved	Reserved	CRST	Reserved
R/W-0	R/W-0	R-1(1)	R-0	R-1	R-0	R/W-0	R-0

(1) In early versions of the DAC1220, this bit was rw-0. See the *Calibration* section for details.

7	6	5	4	3	2	1	0
RES	CLR	DF	DISF	BD	MSB	MD	
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-10b	

LEGEND: R = Read, W = Write

Figure 17: Registre CR

Table 13. Command Register Bits

BIT(S)	NAME	VALUE	DESCRIPTION
15	ADPT		Controls adaptive filtering. if DISF is set, this bit has no effect.
		0	Adaptive filtering enabled (default).
		1	Adaptive filtering disabled.
14	CALPIN	0	Output is disconnected (high impedance) during calibration (default).
		1	Output is connected during calibration.
13	Reserved		Write '1' to this bit. On early versions of the device, this bit is writable and defaults to zero, but still should be set to '1'. On current devices this bit is read-only and always reads '1'. See the *Calibration* section for details.
12	Reserved		Read-only. Always '0'.
11	Reserved		Read-only. Always '0'.
10	Reserved		Read-only. Always '0'.
9	CRST		In Normal mode, writing '1' to this bit resets the calibration registers, setting OCR to 000000h and FCR to 800000h. In Normal mode, this bit always reads '0'. In Sleep mode, this bit is read/write, and has no effect. Writing '1' to this bit and switching to Normal mode at the same time will reset the calibration registers.
		0	Do not clear calibration registers.
		1	Clear calibration registers.
8	Reserved		Read-only. Always '0'.
7	RES		Selects resolution.
		0	16-bit resolution (default).
		1	20-bit resolution.
6	CLR		In Normal mode, writing '1' to this bit writes 0 to the data register. In Sleep mode, this bit is read/write, and has no effect. Writing '1' to this bit and switching to Normal mode at the same time will reset the data register. The actual voltage that the DAC1220 will output on setting this bit depends on the data format selected by DF. If DF is 1, zero gives 0V; if DF is 0, zero gives V_{REF} (mid-scale).
		0	Do not clear calibration registers.
		1	Clear calibration registers.
5	DF		Selects binary number format of the data register.
		0	Offset two's complement (default).
		1	Straight binary.
4	DISF		Can be used to inhibit fast settling and/or adaptive filtering. See text for details.
		0	Fast settling and/or adaptive filtering enabled (default).
		1	Fast settling disabled; filter always at default cutoff.
3	BD		Selects address increment or decrement when reading or writing multiple bytes, except when writing to the command register. The command register is always written to in increment mode (most significant byte first). Reads from the command register are according to this bit.
		0	Address is incremented after each byte (default).
		1	Address is decremented after each byte.

Figure 18: Détail Registre CR